Using Spatial Data with R

Claudia A Engel

Last updated: February 20, 2024

Contents

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`.
- Open up a new R Script file and call it `R-spatial.R` for the code you'll create during the workshop.
- 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 and extract them in the `data` folder.

- Install and load the following libraries:
  - `sf`
  - `terra`
  - `tidyverse`
- For the mapping section install and load these additional libraries:
  - `classInt`
  - `RColorBrewer`
  - `ggplot2`
  - `tmap`
  - `leaflet`(On Mac installing binary version is ok)

References

Lovelace, R., Nowosad, J., Muenchow. J. (2024): Geocomputation with R

Pebesma, E. Bivand, R. (2023): Spatial Data Science

Gimond, M (2023): Intro to GIS and Spatial Analysis

Spatial Data Analysis and Modeling with R and *terra*

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

- Read table with geo coordinates into `sf` object
- Read shapefiles into `sf` object
- Examine `sf` objects
- Use base plot with `sf` objects and attribute data
- Read GeoTiff single and multiband into a `SpatRaster` object
- Examine `SpatRaster` objects

1.1 The `sf` package

The `sf`[1] package was first released on CRAN in late October 2016, and has in the mean time superseded the original R Package for storing and manipulating spatial data, `sp`, which was first released in 2005. `sp` is still actively maintained, but less often used now, so you should be aware of it, but we will not teach it here.

`sf` 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.

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

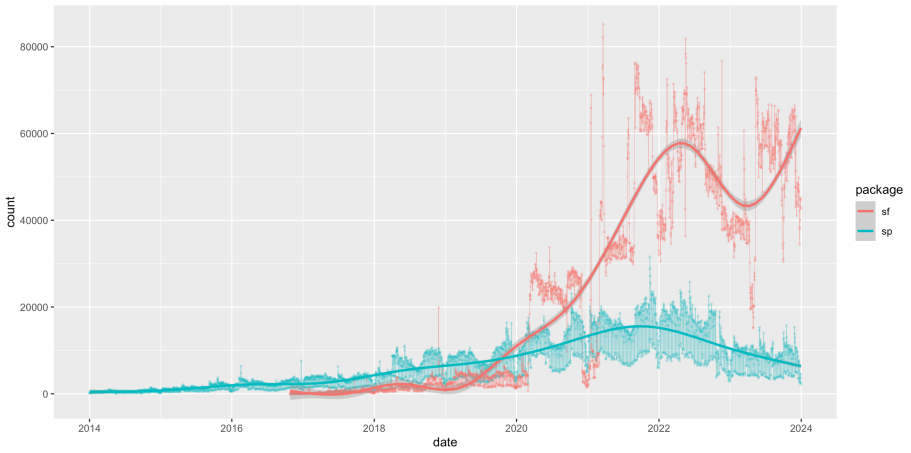


Figure 1.1: sf vs sp downloads on CRAN

If you work with PostGis or GeoJSON you may have come across the WKT (well-known text) format (Fig 1.1 and 1.2)

Geometry primitives (2D)

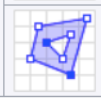
Type	Examples	
Point		`POINT (30 10)`
LineString		`LINESTRING (30 10, 10 30, 40 40)`
Polygon		`POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))`
		`POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))`

Figure 1.2: Well-Known-Text Geometry primitives (wikipedia)

`sf` implements this standard natively in R. In `sf` spatial objects are stored as a tabular format (data frame) with a special column that contains the information for the geometry coordinates. That special column holds 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.

`sf` objects are built up using the following structures:

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 1.3: Well-Known-Text Multipart geometries (wikipedia)

1. `sfg` - simple feature geometry (one feature)
2. `sfc` - simple feature collection (a collection of `sfg`)
3. `sf` - simple feature object (`sfc` with data attributes)

So 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 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.

To create a network of highways 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))
```

```
#> Geometry set for 2 features
#> Geometry type: LINESTRING
#> Dimension:     XY
#> Bounding box:  xmin: 0.235151 ymin: 0.00839848 xmax: 0.9894396 ymax: 0.6266413
#> CRS:           NA
```

```
#> LINESTRING (0.8592813 0.00839848, 0.9273943 0.3...
```

```
#> LINESTRING (0.235151 0.2268521, 0.9894396 0.350...
```

And lastly create a data frame from above to generate the `sf` object:

```
dfr <- data.frame(id = c("hwy1", "hwy2"),
                  cars_per_hour = c(78, 22))
(lnstr_sf <- st_sf(dfr , lnstr_sfc))
```

```
#> Simple feature collection with 2 features and 2 fields
#> Geometry type: LINESTRING
#> Dimension:     XY
#> Bounding box:  xmin: 0.235151 ymin: 0.00839848 xmax: 0.9894396 ymax: 0.6266413
#> CRS:           NA
#>     id cars_per_hour                      lnstr_sfc
#> 1 hwy1            78 LINESTRING (0.8592813 0.008...
#> 2 hwy2            22 LINESTRING (0.235151 0.2268...
```

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

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

```
#>   [1] [                           [[<-
#>   [3] [<-                         $<-
#>   [5] aggregate                   anti_join
#>   [7] arrange                     as.data.frame
#>   [9] cbind                       coerce
#>  [11] crs                         dbDataType
#>  [13] dbWriteTable                distance
#>  [15] distinct                    dplyr_reconstruct
#>  [17] drop_na                     duplicated
#>  [19] ext                         extract
#>  [21] filter                      full_join
#>  [23] gather                      group_by
#>  [25] group_split                 identify
#>  [27] initialize                  inner_join
#>  [29] left_join                   lines
```

```
#>  [31] mask                           merge
#>  [33] mutate                         nest
#>  [35] pivot_longer                   pivot_wider
#>  [37] plot                           points
#>  [39] polys                          print
#>  [41] rasterize                      rbind
#>  [43] rename_with                    rename
#>  [45] right_join                     rowwise
#>  [47] sample_frac                    sample_n
#>  [49] select                         semi_join
#>  [51] separate_rows                  separate
#>  [53] show                           slice
#>  [55] slotsFromS3                    spread
#>  [57] st_agr                         st_agr<-
#>  [59] st_area                        st_as_s2
#>  [61] st_as_sf                       st_as_sfc
#>  [63] st_bbox                        st_boundary
#>  [65] st_break_antimeridian          st_buffer
#>  [67] st_cast                        st_centroid
#>  [69] st_collection_extract          st_concave_hull
#>  [71] st_convex_hull                 st_coordinates
#>  [73] st_crop                        st_crs
#>  [75] st_crs<-                       st_difference
#>  [77] st_drop_geometry               st_filter
#>  [79] st_geometry                    st_geometry<-
#>  [81] st_inscribed_circle            st_interpolate_aw
#>  [83] st_intersection                st_intersects
#>  [85] st_is_valid                    st_is
#>  [87] st_join                        st_line_merge
#>  [89] st_m_range                     st_make_valid
#>  [91] st_minimum_rotated_rectangle   st_nearest_points
#>  [93] st_node                        st_normalize
#>  [95] st_point_on_surface            st_polygonize
#>  [97] st_precision                   st_reverse
#>  [99] st_sample                      st_segmentize
#> [101] st_set_precision               st_shift_longitude
#> [103] st_simplify                    st_snap
#> [105] st_sym_difference              st_transform
#> [107] st_triangulate_constrained     st_triangulate
#> [109] st_union                       st_voronoi
#> [111] st_wrap_dateline               st_write
#> [113] st_z_range                     st_zm
#> [115] summarise                      svc
#> [117] transform                      transmute
#> [119] ungroup                        unite
#> [121] unnest                         vect
```

```
#> 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`

Usually you can find functions that convert objects to and from these formats.

Challenge

Generate an `sf` point object.

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` 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 tabular format (tibble) with `dplyr::read_table` or `dplyr::read_csv`. We can then very easily convert the table into a spatial object in R.

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 tibble in R and name it `philly_homicides_df`.

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

```
#> Rows: 3883 Columns: 10
#> -- Column specification --------------------------------------------------------
#> Delimiter: ","
#> chr  (3): SECTOR, LOCATION_BLOCK, TEXT_GENERAL_CODE
#> dbl  (5): DC_DIST, UCR_GENERAL, OBJ_ID, POINT_X, POINT_Y
#> date (1): DISPATCH_DATE
#> time (1): DISPATCH_TIME
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

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

```
library(sf)

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

```
#> [1] "DC_DIST"           "SECTOR"            "DISPATCH_DATE"
#> [4] "DISPATCH_TIME"     "LOCATION_BLOCK"    "UCR_GENERAL"
#> [7] "OBJ_ID"            "TEXT_GENERAL_CODE" "geometry"
```

Note the additional **geometry** list-column which now holds the simple feature collection with the coordinates of all the points. We now use `st_crs()` to check on the projection.

```
st_crs(philly_homicides_sf)
```

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

To make it a complete geographical object we use `st_crs()` to assign the WGS84 projection, which has the EPSG code 4326:

```
st_crs(philly_homicides_sf) <- 4326
st_crs(philly_homicides_sf)
```

```
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCRS["WGS 84",
#>     ENSEMBLE["World Geodetic System 1984 ensemble",
```

```
#>         MEMBER["World Geodetic System 1984 (Transit)"],
#>         MEMBER["World Geodetic System 1984 (G730)"],
#>         MEMBER["World Geodetic System 1984 (G873)"],
#>         MEMBER["World Geodetic System 1984 (G1150)"],
#>         MEMBER["World Geodetic System 1984 (G1674)"],
#>         MEMBER["World Geodetic System 1984 (G1762)"],
#>         MEMBER["World Geodetic System 1984 (G2139)"],
#>         ELLIPSOID["WGS 84",6378137,298.257223563,
#>             LENGTHUNIT["metre",1]],
#>         ENSEMBLEACCURACY[2.0]],
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["geodetic latitude (Lat)",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["geodetic longitude (Lon)",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     USAGE[
#>         SCOPE["Horizontal component of 3D system."],
#>         AREA["World."],
#>         BBOX[-90,-180,90,180]],
#>     ID["EPSG",4326]]
```

Wow this is long. It is usually more helpful to just retrieve the proj4string:

```
st_crs(philly_homicides_sf)$proj4string
```

```
#> [1] "+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
```

1.3 Loading shape files into R

`sf` relies on the powerful GDAL library, which is automatically linked in when loading `sf`. The GDAL provides the functionality to read and write spatial files of many formats. For shape files 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/data/Philly'
#>   using driver `ESRI Shapefile'
#> Simple feature collection with 384 features and 17 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: 1739497 ymin: 457343.7 xmax: 1764030 ymax: 490544.9
#> Projected CRS: Albers
```

```
# 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 : chr  "42" "42" "42" "42" ...
#>  $ COUNTYFP10: chr  "101" "101" "101" "101" ...
#>  $ TRACTCE10 : chr  "036301" "036400" "036600" "034803" ...
#>  $ GEOID10   : num  4.21e+10 4.21e+10 4.21e+10 4.21e+10 4.21e+10 ...
#>  $ NAME10    : chr  "363.01" "364" "366" "348.03" ...
#>  $ NAMELSAD10: chr  "Census Tract 363.01" "Census Tract 364" "Census Tract 366" "Census Tract
#>  $ MTFCC10   : chr  "G5020" "G5020" "G5020" "G5020" ...
#>  $ FUNCSTAT10: chr  "S" "S" "S" "S" ...
#>  $ ALAND10   : num  2322732 4501110 1004313 1271533 1016206 ...
#>  $ AWATER10  : num  66075 8014 1426278 8021 0 ...
#>  $ INTPTLAT10: chr  "+40.0895349" "+40.1127747" "+39.9470272" "+40.0619427" ...
#>  $ INTPTLON10: chr  "-074.9667387" "-074.9789137" "-075.1404472" "-075.0023705" ...
#>  $ GISJOIN   : chr  "G4201010036301" "G4201010036400" "G4201010036600" "G4201010034803" ...
#>  $ 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 [1:3] "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 NA
#>   ..- attr(*, "names")= chr [1:17] "STATEFP10" "COUNTYFP10" "TRACTCE10" "GEOID10" ...
```

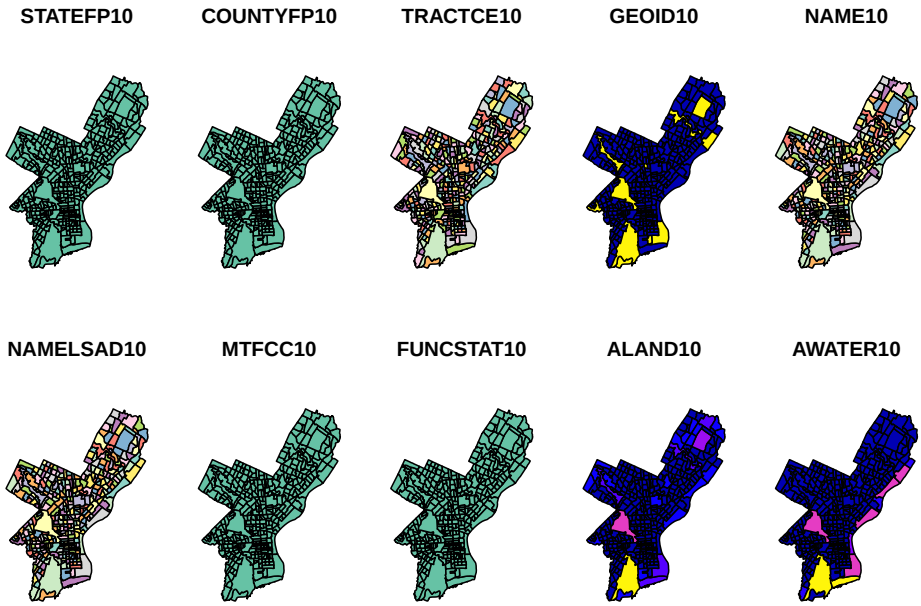
Two more words about the geometry column: Though it is not recommended, you can name this column any way you wish. Secondly, you can remove this column and revert to a regular, non-spatial data frame at any dime with `st_drop_geometry()`.

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
```

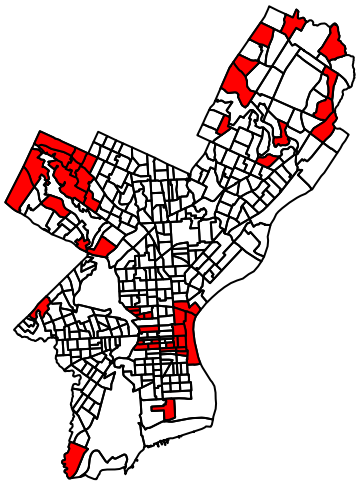


In order to only plot the polygon boundaries we need to directly use the geometry column. We use the `st_geometry()` function. It extracts the `sfc` object (the simple feature geometry list column):

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

Let's add a subset of polygons with only the census tracts where the median houshold income (*medHHinc*) is more than $60,000. We can extract elements from an `sf` object based on attributes using the dplyr filter function (base R subsetting also works) and add the census tracts to the plot in a different color.

```
plot(st_geometry(philly_sf))
philly_sf %>%
  filter(medHHinc > 60000) %>% # filter for high income
  st_geometry() %>% # extract the geometry for plotting
  plot(col="red", add=T) # add to the plot
```



1.4 Raster data in R

Raster files, as you might know, have a 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.

`terra` was first released in 2020 and now replaces the `raster` package which was first released in 2010. `terra` has greater functionality, is faster and easier to use.

The `terra` package has functions for creating, reading, manipulating, and writing raster data. The package also implements raster algebra and many other

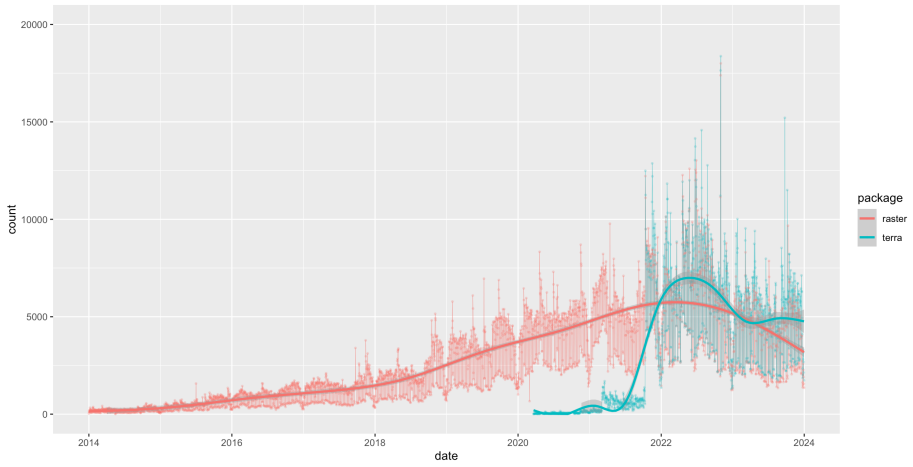


Figure 1.4: raster vs terra downloads on CRAN

functions for raster data manipulation.

The package works with `SpatRaster` objects. The `rast()` function is used to create these objects. For example, to create a raster object from scratch we would do the following:

```
library(terra)
r <- rast(nrows=20, ncols=20, # number of cells in x and y dimension
          xmin=0, xmax=360) # min and max x coordinates (left-right borders)
r
```

```
#> class       : SpatRaster
#> dimensions  : 20, 20, 1  (nrow, ncol, nlyr)
#> resolution  : 18, 9  (x, y)
#> extent      : 0, 360, -90, 90  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84
```

From the output above we know that:

- the object is of class `SpatRaster`
- its dimensions are 20x20 cells
- it has one layer (band)
- the extent of the raster
- it **has a CRS defined!** If the crs argument is missing when creating the SpatRaster object, if 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.

There are functions to look at individual properties of the raster object. For

examle for the number of cells:

```
ncell(r)
```

```
#> [1] 400
```

Or we can retrieve just the number of bands using the `nlyr()` function.

```
nlyr(r)
```

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

We can also find out about the Coordinate Reference System (CRS) with the `crs` function. The default output looks a little messy:

```
crs(r)
```

```
#> [1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOID[\"WGS 8
```

We can make this easier to read by setting the proj argument:

```
crs(r, proj = TRUE) # return the PROJ-string notation
```

```
#> [1] "+proj=longlat +datum=WGS84 +no_defs"
```

Let's try and plot this.

```
plot(r)
```

The canvas is empty! This is because even though we have a layer, the cells do not have any values.

```
values(r)
```

To add some random values to the cells we can take advantage of the `ncells()` function and do this:

```
values(r) <- runif(ncell(r))
r
```

```
#> class       : SpatRaster
#> dimensions  : 20, 20, 1  (nrow, ncol, nlyr)
#> resolution  : 18, 9  (x, y)
#> extent      : 0, 360, -90, 90  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84
#> source(s)   : memory
#> name        :        lyr.1
#> min value   : 0.0003101979
#> max value   : 0.9925648135
```

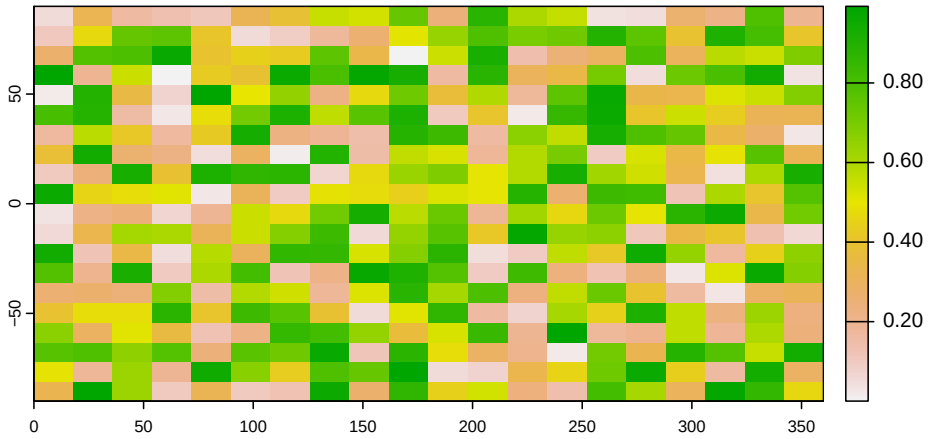
In addition to the output above, we now see:

- the source, which indicates where the cell values are stored (here they are in memory)

- the range of the cell values (min value adn max value) now added.
- the name, of the layer which is by default lyr.1.

This now plots successfully:

```
plot(r)
```



(The `rasterVis` package provides a set of methods for enhanced visualization and interaction for more advanced plotting of raster objects.)

`SpatRaster` objects can also be created from a matrix.

```
class(volcano)
```

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

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

```
#> [1] "SpatRaster"
#> attr(,"package")
#> [1] "terra"
```

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

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

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

```
HARV
```

```
#> class       : SpatRaster
#> dimensions  : 2317, 3073, 3  (nrow, ncol, nlyr)
#> resolution  : 0.25, 0.25  (x, y)
#> extent      : 731998.5, 732766.8, 4712956, 4713536  (xmin, xmax, ymin, ymax)
#> coord. ref. : WGS 84 / UTM zone 18N (EPSG:32618)
```

```
#> source      : HARV_RGB_Ortho.tif
#> names       : HARV_RGB_Ortho_1, HARV_RGB_Ortho_2, HARV_RGB_Ortho_3
#> min values  :                 0,                 0,                 0
#> max values  :               255,               255,               255
```

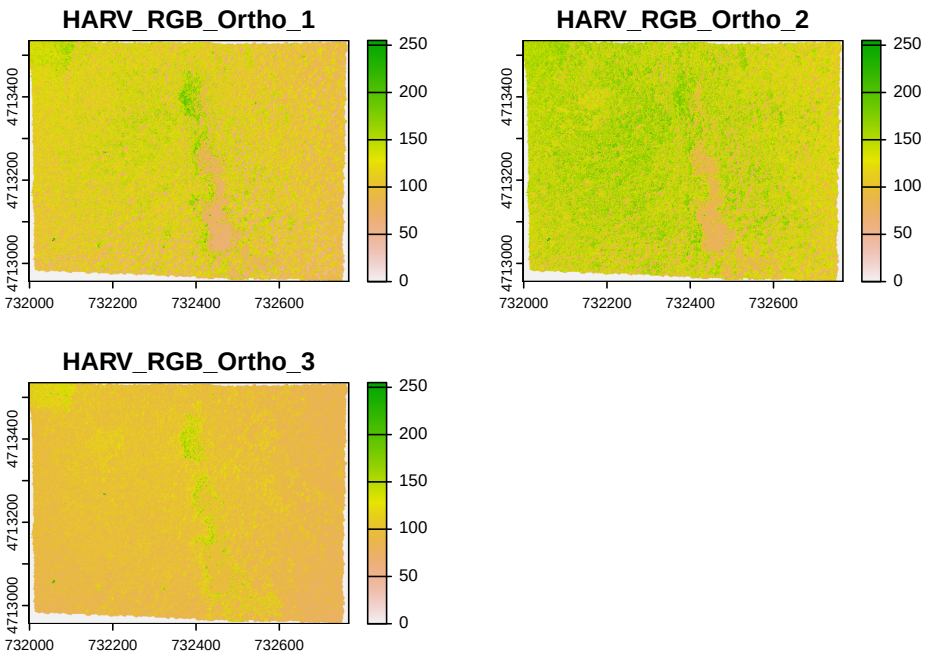
Challenge

Based on the output above answer the following questions:

1. How many bands?
2. What are the names of the bands)?
3. Where are the cell values stored?
4. What is the CRS?

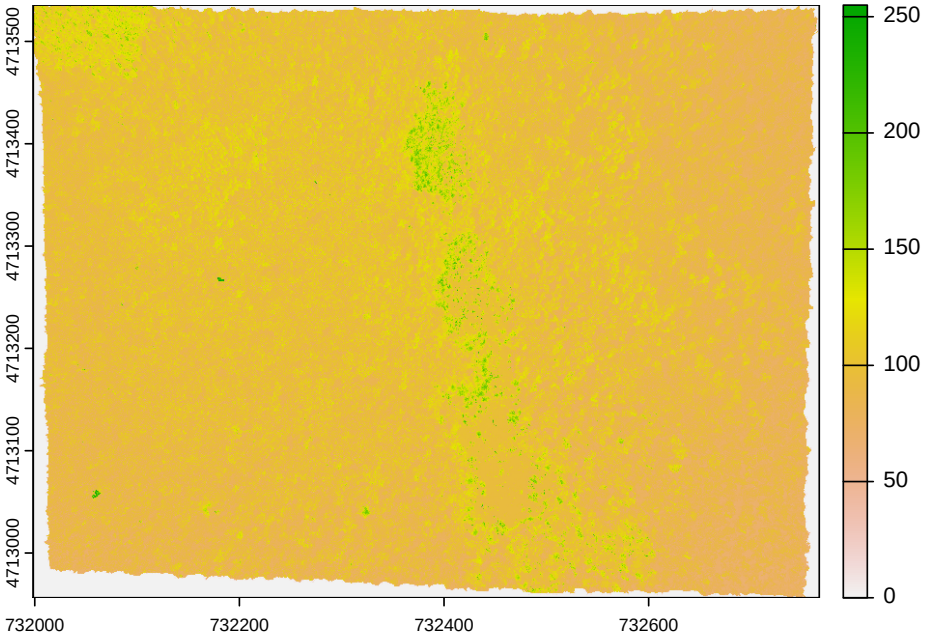
We can plot the object like this:

```
plot(HARV)
```



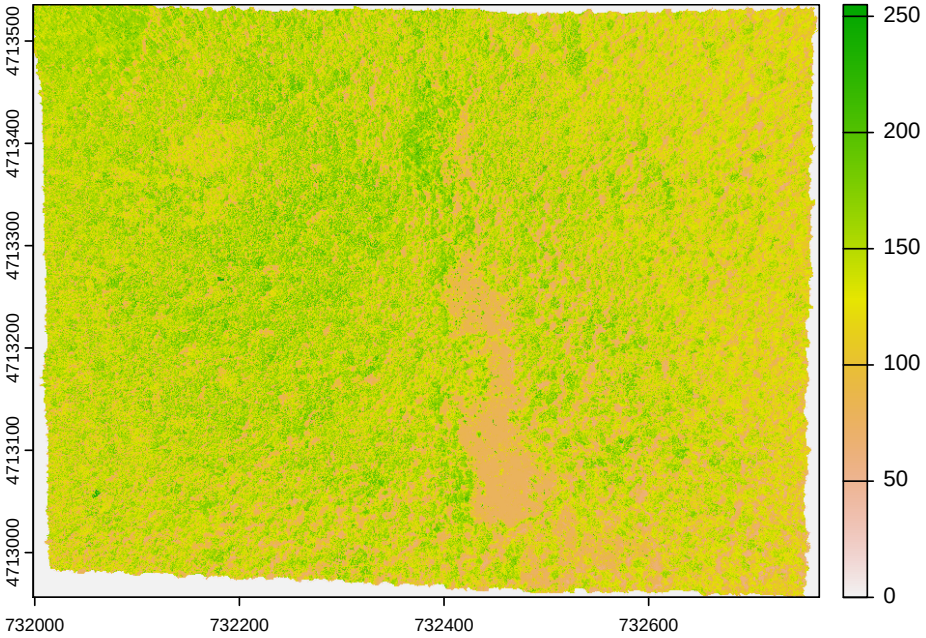
Or to plot a single band:

```
plot(HARV, 3)
```



We can also use the `rast()` function to import one single band:

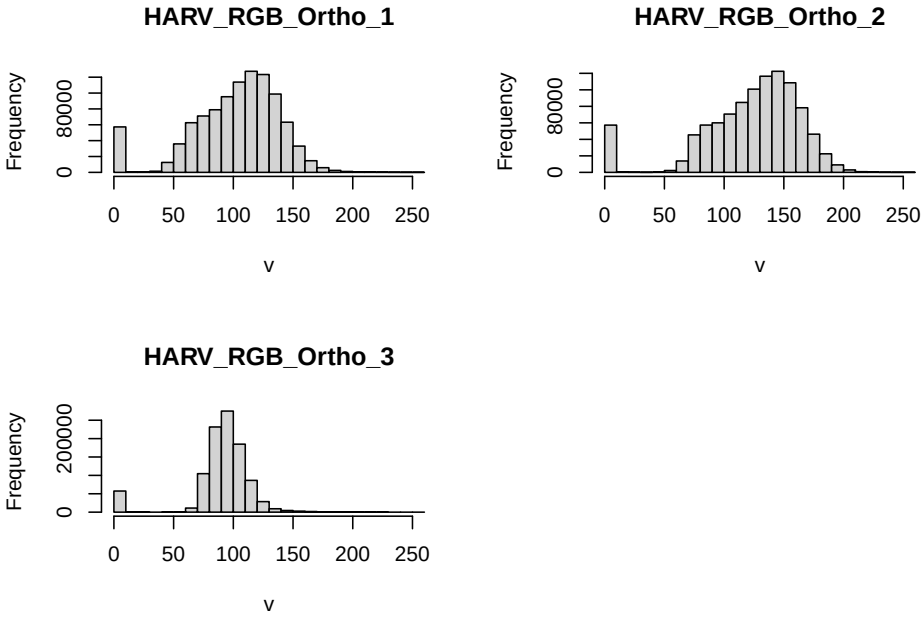
```
HARV_Band2 <- rast("data/HARV_RGB_Ortho.tif", lyrs = 2)
plot(HARV_Band2)
```



Let's now 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: [hist] a sample of 14% of the cells was used

#> Warning: [hist] a sample of 14% of the cells was used

#> Warning: [hist] a sample of 14% of the cells was used
```

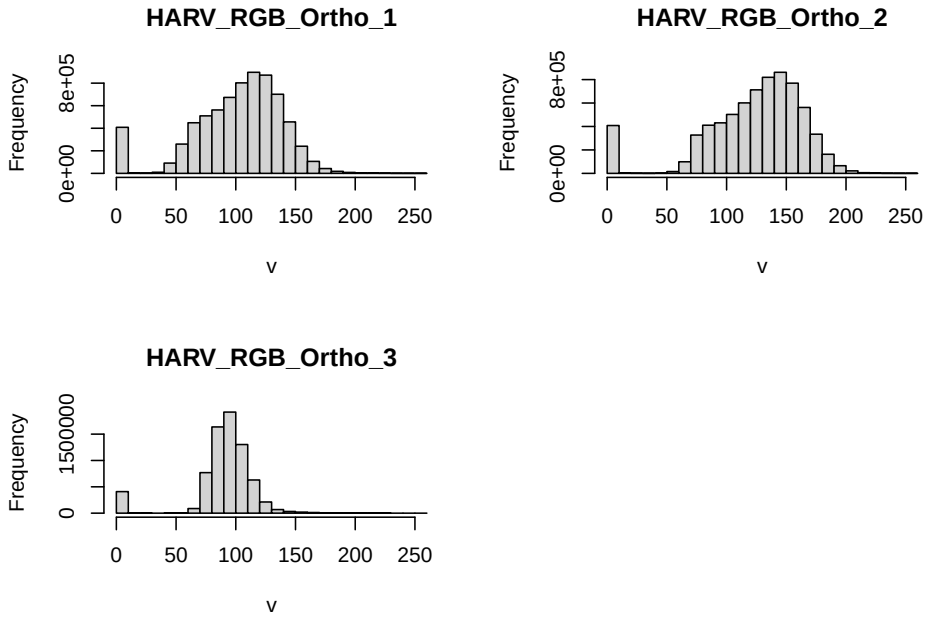


Notice that a warning message is produced when R creates the histogram. By default the maximum cells processed per band is 1,000,000. This maximum value is to ensure processing efficiency as our data become larger. We can force the `hist` function to use all cell values.

```
ncell(HARV)
```

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

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



At times it may be useful to explore raster metadata before loading them into R. This can be done with the function `describe()`

```
describe("data/HARV_RGB_Ortho.tif")
```

```
#>  [1] "Driver: GTiff/GeoTIFF"
#>  [2] "Files: data/HARV_RGB_Ortho.tif"
#>  [3] "Size is 3073, 2317"
#>  [4] "Coordinate System is:"
#>  [5] "PROJCRS[\"WGS 84 / UTM zone 18N\","
#>  [6] "    BASEGEOGCRS[\"WGS 84\","
#>  [7] "        DATUM[\"World Geodetic System 1984\","
#>  [8] "            ELLIPSOID[\"WGS 84\",6378137,298.257223563,"
#>  [9] "                LENGTHUNIT[\"metre\",1]]],"
#> [10] "        PRIMEM[\"Greenwich\",0,"
#> [11] "            ANGLEUNIT[\"degree\",0.0174532925199433]],"
#> [12] "        ID[\"EPSG\",4326]],"
#> [13] "    CONVERSION[\"UTM zone 18N\","
#> [14] "        METHOD[\"Transverse Mercator\","
#> [15] "            ID[\"EPSG\",9807]],"
#> [16] "        PARAMETER[\"Latitude of natural origin\",0,"
#> [17] "            ANGLEUNIT[\"degree\",0.0174532925199433],"
#> [18] "            ID[\"EPSG\",8801]],"
#> [19] "        PARAMETER[\"Longitude of natural origin\",-75,"
#> [20] "            ANGLEUNIT[\"degree\",0.0174532925199433],"
#> [21] "            ID[\"EPSG\",8802]],"
#> [22] "        PARAMETER[\"Scale factor at natural origin\",0.9996,"
```

```
#> [23] "            SCALEUNIT[\"unity\",1],"
#> [24] "            ID[\"EPSG\",8805]],"
#> [25] "        PARAMETER[\"False easting\",500000,"
#> [26] "            LENGTHUNIT[\"metre\",1],"
#> [27] "            ID[\"EPSG\",8806]],"
#> [28] "        PARAMETER[\"False northing\",0,"
#> [29] "            LENGTHUNIT[\"metre\",1],"
#> [30] "            ID[\"EPSG\",8807]]],"
#> [31] "    CS[Cartesian,2],"
#> [32] "        AXIS[\"(E)\",east,"
#> [33] "            ORDER[1],"
#> [34] "            LENGTHUNIT[\"metre\",1]],"
#> [35] "        AXIS[\"(N)\",north,"
#> [36] "            ORDER[2],"
#> [37] "            LENGTHUNIT[\"metre\",1]],"
#> [38] "    USAGE["
#> [39] "        SCOPE[\"Engineering survey, topographic mapping.\"],"
#> [40] "        AREA[\"Between 78°W and 72°W, northern hemisphere between equator and 84°N, onsh
#> [41] "        BBOX[0,-78,84,-72]],"
#> [42] "    ID[\"EPSG\",32618]]"
#> [43] "Data axis to CRS axis mapping: 1,2"
#> [44] "Origin = (731998.500000000000000,4713535.500000000000000)"
#> [45] "Pixel Size = (0.250000000000000,-0.250000000000000)"
#> [46] "Metadata:"
#> [47] "  AREA_OR_POINT=Area"
#> [48] "Image Structure Metadata:"
#> [49] "  COMPRESSION=LZW"
#> [50] "  INTERLEAVE=PIXEL"
#> [51] "Corner Coordinates:"
#> [52] "Upper Left  (  731998.500, 4713535.500) ( 72d10'29.27\"W, 42d32'21.80\"N)"
#> [53] "Lower Left  (  731998.500, 4712956.250) ( 72d10'30.11\"W, 42d32' 3.04\"N)"
#> [54] "Upper Right (  732766.750, 4713535.500) ( 72d 9'55.63\"W, 42d32'20.97\"N)"
#> [55] "Lower Right (  732766.750, 4712956.250) ( 72d 9'56.48\"W, 42d32' 2.21\"N)"
#> [56] "Center      (  732382.625, 4713245.875) ( 72d10'12.87\"W, 42d32'12.00\"N)"
#> [57] "Band 1 Block=3073x1 Type=Float64, ColorInterp=Gray"
#> [58] "  Min=0.000 Max=255.000 "
#> [59] "  Minimum=0.000, Maximum=255.000, Mean=nan, StdDev=nan"
#> [60] "  NoData Value=-1.7e+308"
#> [61] "  Metadata:"
#> [62] "    STATISTICS_MAXIMUM=255"
#> [63] "    STATISTICS_MEAN=nan"
#> [64] "    STATISTICS_MINIMUM=0"
#> [65] "    STATISTICS_STDDEV=nan"
#> [66] "Band 2 Block=3073x1 Type=Float64, ColorInterp=Undefined"
#> [67] "  Min=0.000 Max=255.000 "
#> [68] "  Minimum=0.000, Maximum=255.000, Mean=nan, StdDev=nan"
```

```
#> [69] "  NoData Value=-1.7e+308"
#> [70] "  Metadata:"
#> [71] "    STATISTICS_MAXIMUM=255"
#> [72] "    STATISTICS_MEAN=nan"
#> [73] "    STATISTICS_MINIMUM=0"
#> [74] "    STATISTICS_STDDEV=nan"
#> [75] "Band 3 Block=3073x1 Type=Float64, ColorInterp=Undefined"
#> [76] "  Min=0.000 Max=255.000 "
#> [77] "  Minimum=0.000, Maximum=255.000, Mean=nan, StdDev=nan"
#> [78] "  NoData Value=-1.7e+308"
#> [79] "  Metadata:"
#> [80] "    STATISTICS_MAXIMUM=255"
#> [81] "    STATISTICS_MEAN=nan"
#> [82] "    STATISTICS_MINIMUM=0"
#> [83] "    STATISTICS_STDDEV=nan"
```

For the many functions available for working with such an object see:

```
methods(class="SpatRaster")
```

```
#>   [1] !                     [                     [[
#>   [4] [[<-                  [<-                   %in%
#>   [7] $                     $<-                   activeCat
#>  [10] activeCat<-           add<-                 addCats
#>  [13] adjacent              aggregate             align
#>  [16] all.equal             allNA                 animate
#>  [19] anyNA                 app                   approximate
#>  [22] area                  Arith                 as.array
#>  [25] as.bool               as.character          as.contour
#>  [28] as.data.frame         as.factor             as.int
#>  [31] as.integer            as.lines              as.list
#>  [34] as.logical            as.matrix             as.numeric
#>  [37] as.points             as.polygons           as.raster
#>  [40] atan_2                atan2                 autocor
#>  [43] barplot               blocks                boundaries
#>  [46] boxplot               buffer                c
#>  [49] catalyze              categories            cats
#>  [52] cellFromRowCol        cellFromRowColCombine cellFromXY
#>  [55] cells                 cellSize              clamp_ts
#>  [58] clamp                 classify              click
#>  [61] coerce                colFromCell           colFromX
#>  [64] colorize              coltab                coltab<-
#>  [67] Compare               compare               compareGeom
#>  [70] concats               contour               costDist
#>  [73] countNA               cover                 crds
#>  [76] crop                  crosstab              crs
#>  [79] crs<-                 datatype              deepcopy
```

```
#>  [82] density              depth                depth<-
#>  [85] diff                 dim                  dim<-
#>  [88] direction            disagg               distance
#>  [91] droplevels           expanse              ext
#>  [94] ext<-                extend               extract
#>  [97] extractRange         fillTime             flip
#> [100] focal                focal3D              focalCor
#> [103] focalCpp             focalPairs           focalReg
#> [106] focalValues          freq                 getTileExtents
#> [109] global               gridDist             gridDistance
#> [112] has.colors           has.RGB              has.time
#> [115] hasMinMax            hasValues            head
#> [118] hist                 identical            ifel
#> [121] image                init                 inMemory
#> [124] inset                interpIDW            interpNear
#> [127] interpolate          intersect            is.bool
#> [130] is.factor            is.finite            is.infinite
#> [133] is.int               is.lonlat            is.na
#> [136] is.nan               is.related           is.rotated
#> [139] isFALSE              isTRUE               k_means
#> [142] lapp                 layerCor             levels
#> [145] levels<-             linearUnits          lines
#> [148] log                  Logic                logic
#> [151] longnames            longnames<-          makeTiles
#> [154] mask                 match                math
#> [157] Math                 Math2                mean
#> [160] median               merge                meta
#> [163] metags               metags<-             minmax
#> [166] modal                mosaic               NAflag
#> [169] NAflag<-             names                names<-
#> [172] ncell                ncol                 ncol<-
#> [175] nlyr                 nlyr<-               noNA
#> [178] not.na               nrow                 nrow<-
#> [181] nsrc                 origin               origin<-
#> [184] pairs                panel                patches
#> [187] persp                plet                 plot
#> [190] plotRGB              points               polys
#> [193] prcomp               predict              princomp
#> [196] project              quantile             rangeFill
#> [199] rapp                 rast                 rasterize
#> [202] rasterizeGeom        rasterizeWin         rcl
#> [205] readStart            readStop             readValues
#> [208] rectify              regress              relate
#> [211] rep                  res                  res<-
#> [214] resample             rescale              rev
#> [217] RGB                  RGB<-                roll
```

```
#> [220] rotate              rowColCombine       rowColFromCell
#> [223] rowFromCell         rowFromY            sapp
#> [226] saveRDS             scale               scoff
#> [229] scoff<-             sds                 segregate
#> [232] sel                 selectHighest       selectRange
#> [235] serialize           set.cats            set.crs
#> [238] set.ext             set.names           set.RGB
#> [241] set.values          setMinMax           setValues
#> [244] shift               show                sieve
#> [247] size                sort                sources
#> [250] spatSample          split               sprc
#> [253] st_bbox             st_crs              stdev
#> [256] str                 stretch             subset
#> [259] subst               summary             Summary
#> [262] t                   tail                tapp
#> [265] terrain             text                tighten
#> [268] time                time<-              timeInfo
#> [271] trans               trim                unique
#> [274] units               units<-             update
#> [277] values              values<-            varnames
#> [280] varnames<-          viewshed            weighted.mean
#> [283] where.max           where.min           which.lyr
#> [286] which.max           which.min           window
#> [289] window<-            wrap                wrapCache
#> [292] writeCDF            writeRaster         writeStart
#> [295] writeStop           writeValues         xapp
#> [298] xFromCell           xFromCol            xmax
#> [301] xmax<-              xmin                xmin<-
#> [304] xres                xyFromCell          yFromCell
#> [307] yFromRow            ymax                ymax<-
#> [310] ymin                ymin<-              yres
#> [313] zonal               zoom
#> see '?methods' for accessing help and source code
```

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
- Reproject a raster
- Perform a raster calculation

There are a wide variety of spatial, topological, and attribute data operations you can perform with R. Lovelace et al's recent publication[1] goes into great depth about this and is highly recommended.

In this section we will look at 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).

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.

[1] Lovelace, R., Nowosad, J., & Muenchow, J. (2024). Geocomputation with R. CRC Press.

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")
```

```
#> Rows: 384 Columns: 13
#> -- Column specification --------------------------------------------------------
#> Delimiter: ","
#> chr  (1): NAME
#> dbl (12): GEOID, fem_bachelor, fem_doctorate, fem_highschool, fem_noschool, ...
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
names(ph_edu)
```

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.

```
# if you need to read in again:
# 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 <- left_join(philly_sf, ph_edu, by = c("GEOID10" = "GEOID"))
names(philly_sf_merged)
```

```
#>  [1] "STATEFP10"       "COUNTYFP10"      "TRACTCE10"       "GEOID10"
#>  [5] "NAME10"          "NAMELSAD10"      "MTFCC10"         "FUNCSTAT10"
#>  [9] "ALAND10"         "AWATER10"        "INTPTLAT10"      "INTPTLON10"
#> [13] "GISJOIN"         "Shape_area"      "Shape_len"       "medHHinc"
#> [17] "totalPop"        "NAME"            "fem_bachelor"    "fem_doctorate"
#> [21] "fem_highschool"  "fem_noschool"    "fem_ovr_25"      "male_bachelor"
#> [25] "male_doctorate"  "male_highschool" "male_noschool"   "male_ovr_25"
#> [29] "pop_ovr_25"      "geometry"
```

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

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

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 also in the *USA Contiguous Albers Equal Area Conic* projected CRS, which is the same as CRS as `philly_sf`.

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, `scf`.

```
# 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 = st_crs(philly_sf))
```

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 <- st_buffer(philly_ctr_sfc, 2000)
```

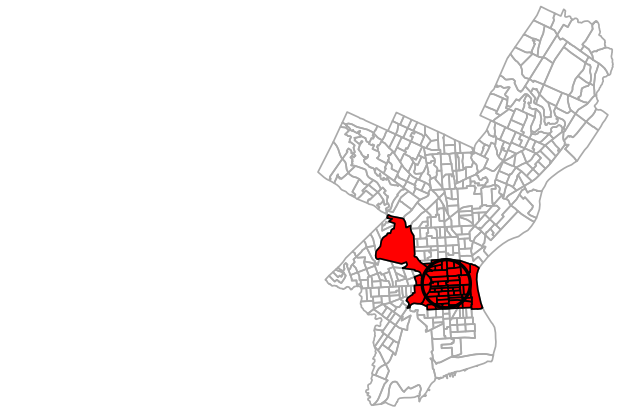
Ok. Now we can use that buffer to select all census tract polygons that intersect with the center buffer. In order to determine the polygons we use `st_intersects`, a geometric binary which returns a vector of indices of the polygons that intersect with the buffer.

```
philly_intersects <- st_intersects(philly_buf, philly_sf)
philly_intersects
```

```
#> Sparse geometry binary predicate list of length 1, where the predicate
#> was `intersects'
#>  1: 3, 10, 11, 12, 13, 19, 20, 21, 50, 51, ...
```

We have created a `sgbp` object, which is a "Sparse Geometry 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<- philly_sf[philly_intersects[[1]],]

# plot
plot(st_geometry(philly_sf), border="#aaaaaa", main="Census tracts that overlap with 2k
plot(st_geometry(philly_sel), add=T, col="red")
plot(st_geometry(philly_buf), add=T, lwd = 2)
```



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 like cookie a cutter:

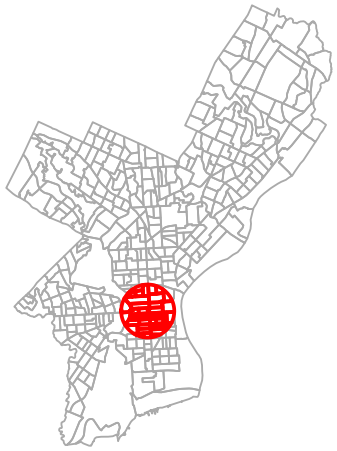
```
philly_intersection <- st_intersection(philly_buf, philly_sf)
philly_intersection
```

```
#> Geometry set for 46 features
#> Geometry type: GEOMETRY
#> Dimension:     XY
#> Bounding box:  xmin: 1748160 ymin: 465499.9 xmax: 1752160 ymax: 469499.9
#> Projected CRS: Albers
#> First 5 geometries:

#> POLYGON ((1752157 467395.2, 1752153 467339.7, 1...

#> POLYGON ((1752149 467290.8, 1752135 467187, 175...

#> POLYGON ((1751347 466573.5, 1751321 467037.6, 1...
```

```
#> POLYGON ((1750298 467616.7, 1750148 467615.2, 1...
#> POLYGON ((1748853 467255, 1748874 467605.3, 174...
```

```
plot(st_geometry(philly_sf), border="#aaaaaa", main="Census tracts around city center, clipped by
plot(philly_intersection, add=T, lwd = 2, border = "red")
```

Census tracts around city center, clipped by 2km buffer



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 `SpatRaster` object with `project()`

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 a `SpatRaster` 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)$proj4string
```

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

```
st_crs(philly_homicides_sf)$proj4string
```

```
#> [1] "+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 `str_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))
```

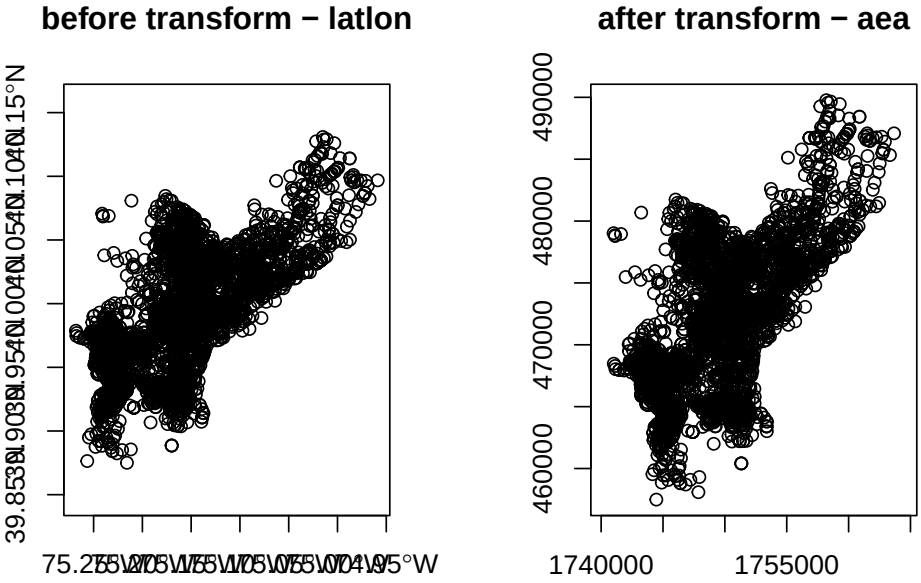
```
#> [1] -75.26809  40.13086
```

```
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 Raster reprojection

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

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

```
#> [1] "+proj=utm +zone=18 +datum=WGS84 +units=m +no_defs"
```

```
HARV_WGS84 <- project(HARV, "+init=epsg:4326")
crs(HARV_WGS84, proj = TRUE)
```

```
#> [1] "+proj=longlat +datum=WGS84 +no_defs"
```

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

```
ext(HARV)
```

```
#> SpatExtent : 731998.5, 732766.75, 4712956.25, 4713535.5 (xmin, xmax, ymin, ymax)
```

```
ext(HARV_WGS84)
```

```
#> SpatExtent : -72.1750316832584, -72.1654516638594, 42.5339461813323, 42.5393881837189 (xmin, x
```

Due to the reprojection the number of cells has also changed:

```
ncell(HARV)
```

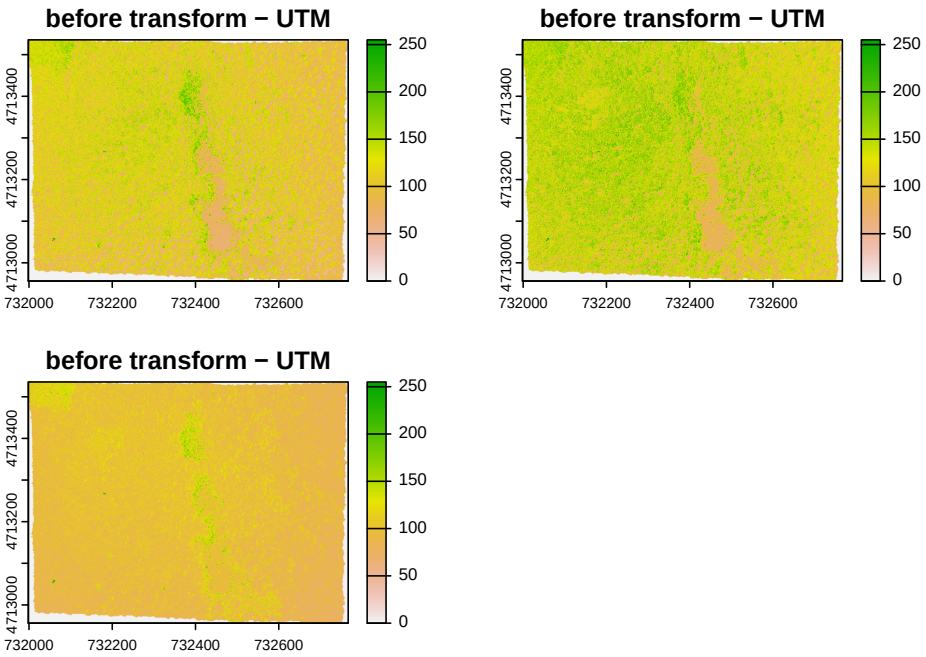
```
#> [1] 7120141
```

```
ncell(HARV_WGS84)
```

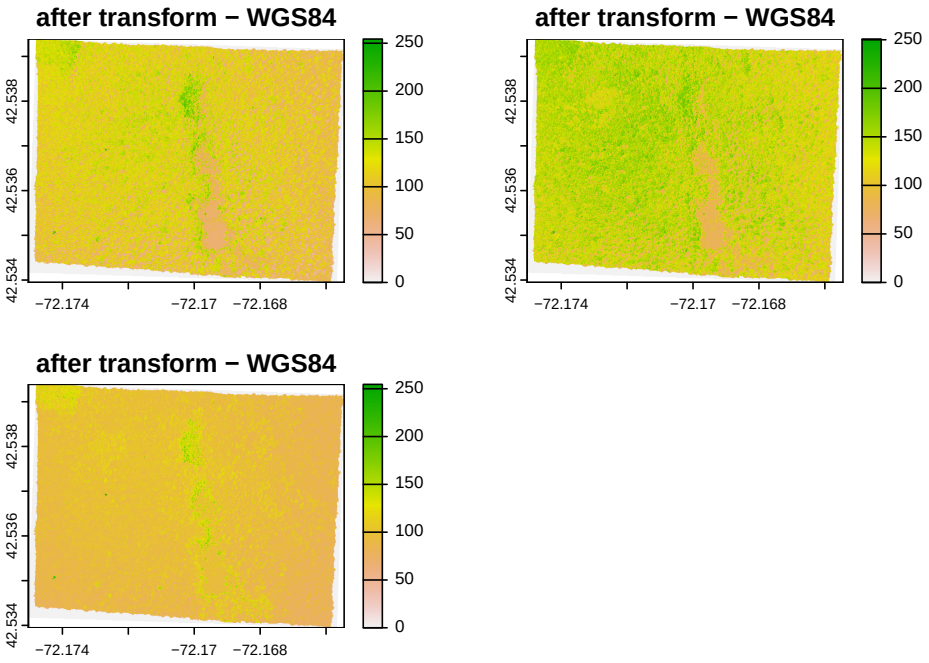
```
#> [1] 6859650
```

And here is the visual proof:

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




```
plot(HARV_WGS84, main = "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: $\frac{homicides}{area}$

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.

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 uiquely 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 we 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 `philly_crimes_sf`.

```
philly_crimes_sf <- 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 = as.numeric(1e6 * (n_homic/tract_area)))
```

Finally, we write this out for later:

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

2.5 Raster calculations with `terra`

We often want to perform calculations on two or more rasters to create a new output raster. For example, if we are interested in mapping the heights of trees across an entire field site, we might want to calculate the difference between the Digital Surface Model (DSM, tops of trees) and the Digital Terrain Model (DTM, ground level). The resulting dataset is referred to as a Canopy Height Model (CHM) and represents the actual height of trees, buildings, etc. with the influence of ground elevation removed.

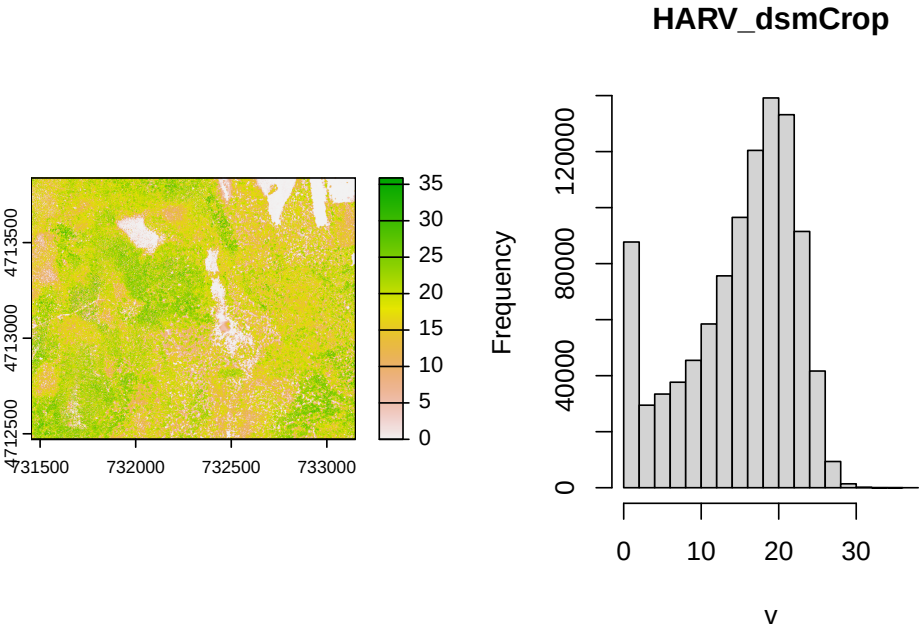
First let's read in the two datasets.

```
HARV_DTM <- rast("data/HARV_dtmCrop.tif")
HARV_DSM <- rast("data/HARV_dsmCrop.tif")
```

Now we can subtract the DTM from the DSM to create a Canopy Height Model. It will for each CHM pixel calculate the difference of the respective DTM and DSM pixels.

```
HARV_CHM <- HARV_DSM - HARV_DTM
par(mfrow = c(1, 2))
plot(HARV_CHM)
hist(HARV_CHM)
```

```
#> Warning: [hist] a sample of 43% of the cells was used (of which 0% was NA)
```



This works fine for the small rasters in this tutorial. However, the calculation above becomes less efficient when computations are more complex or file sizes become large.

Thet `terra` package contains a function called `lapp()`function to make processing more efficient. It takes two or more rasters and applies a function to them. The generic syntax is:

```
outputRaster <- lapp(x, fun)
```

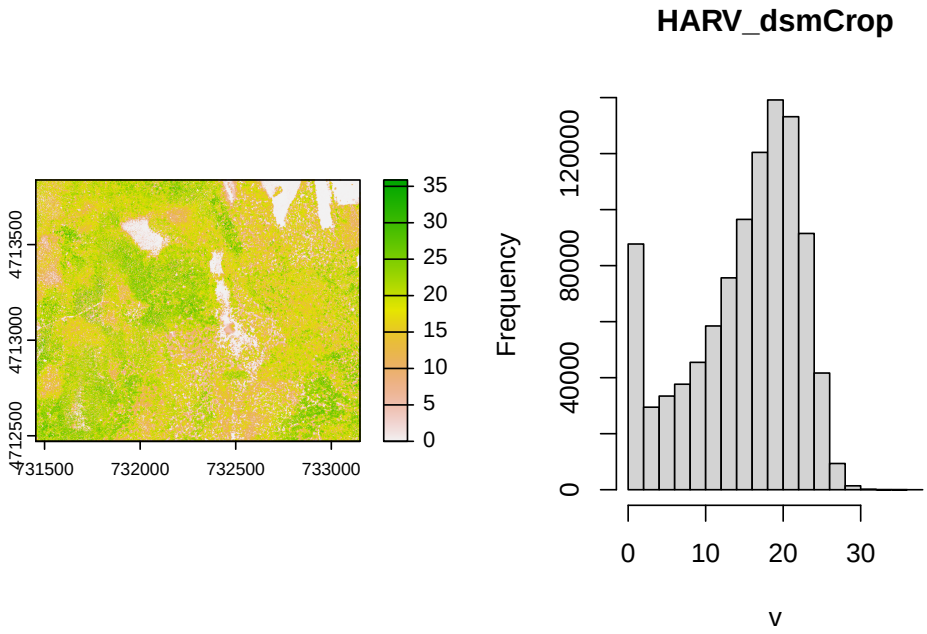
where x is a `SpatRasterDataset` and fun is a custom function for the operation you want to perform.

```
CHM_ov_HARV <- lapp(sds(list(HARV_DSM, HARV_DTM)),
                    fun = function(r1, r2) {
                      return( r1 - r2)
                      })
```

As arguments for our `lapp` operation we use the `sds()` function and provide it with the list of rasters that we want to operate on. As custom function we provide the function with two arguments (`r1` and `r1`) that subtracts the second (`r2`) from the first (`r1`) and returns the difference. The output of `lapp` is a `SpatRaster` and we assign it to a new variable `CHM_ov_HARV`.

```
par(mfrow = c(1, 2))
plot(CHM_ov_HARV)
hist(CHM_ov_HARV)
```

```
#> Warning: [hist] a sample of 43% of the cells was used (of which 0% was NA)
```


HARV_dsmCrop
4713500
4713000
4712500
731500
732000
732500
733000
35
30
25
20
15
10
5
0
Frequency
120000
80000
40000
0
0
10
20
30
v

Chapter 3

Making Maps in R

Learning Objectives

- plot an `sf` object
- create a choropleth map with `ggplot`
- plot a raster map with `ggplot`
- use `RColorBrewer` to improve legend colors
- use `classInt`to improve legend breaks
- create a choropleth map with `tmap`
- plot a raster 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.

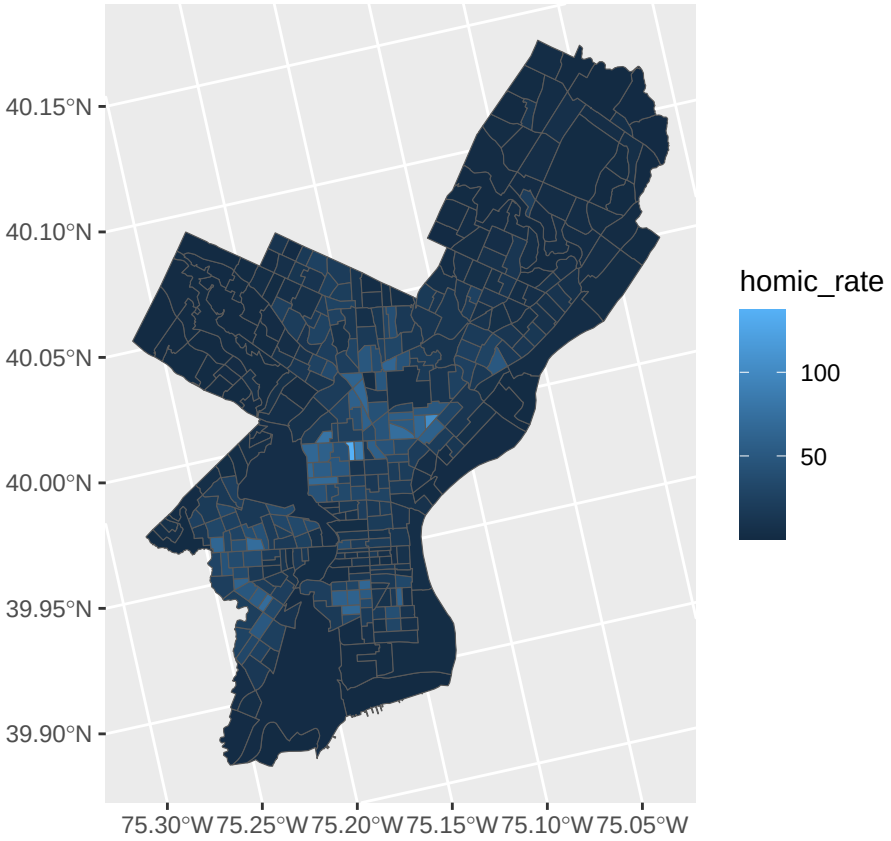
3.1 Choropleth Mapping with `ggplot2`

`ggplot2` is a widely used and powerful plotting library for R. It is not specifically geared towards mapping, it is possible to create quite nice maps.

For an introduction to `ggplot` check out this site for more pointers.

`ggplot` can plot `sf` objects directly by using the geom `geom_sf`. So all we have to do is:

```
library(ggplot2)
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 continous variable into a categoriacal 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 continious vaule 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_r

str(breaks_qt)
```

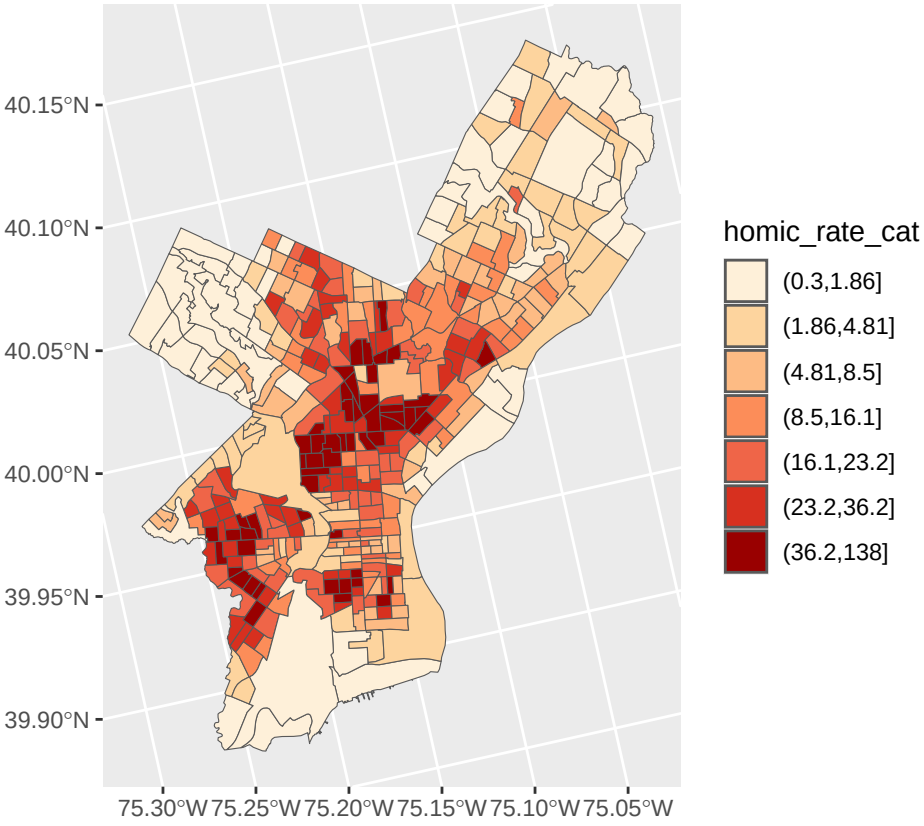
```
#> List of 2
#>  $ var : num [1:385] 0.3 14.2 10.5 12.7 38.9 ...
#>  $ brks: num [1:8] 0.3 1.86 4.81 8.5 16.14 ...
#>  - attr(*, "style")= chr "quantile"
#>  - attr(*, "nobs")= int 385
#>  - attr(*, "call")= language classIntervals(var = c(min(philly_crimes_sf$homic_rate) - 1e-05,
#>  - attr(*, "intervalClosure")= chr "left"
#>  - attr(*, "class")= chr "classIntervals"
```

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

We use `cut` to divicde `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")
```



3.2 Raster and ggplot

To visualize raster data using `ggplot2`, we will use the raster with the values for the digital terrain model (DTM).

If you need to read it in again:

```
HARV_DTM <- rast("data/HARV_dtmCrop.tif")
```

Before using ggplot we need to convert this to a dataframe. The `terra` package has an built-in function for conversion to a plotable dataframe.

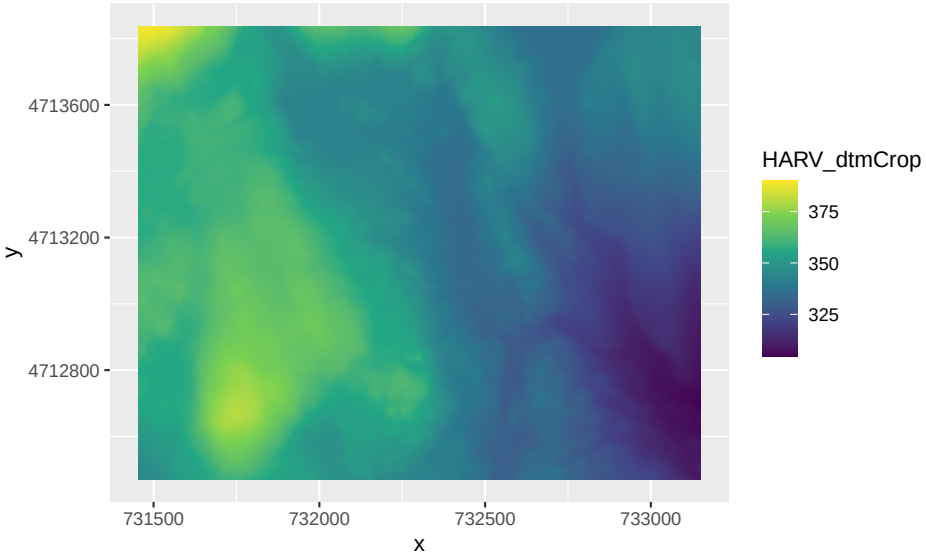
```
HARV_DTM_df <- as.data.frame(HARV_DTM, xy = TRUE)
str(HARV_DTM_df)
```

```
#> 'data.frame':    2319798 obs. of  3 variables:
#>  $ x           : num  731454 731454 731456 731456 731458 ...
#>  $ y           : num  4713838 4713838 4713838 4713838 4713838 ...
#>  $ HARV_dtmCrop: num  389 390 389 389 389 ...
```

We can now use `ggplot()` to plot this data frame. We will set the color scale to

`scale_fill_viridis_c` which is a color-blindness friendly color scale. Here is more about the viridis color maps. We will also use the `coord_fixed()` function with the default, ratio = 1, which ensures that one unit on the x-axis is the same length as one unit on the y-axis.

```
ggplot() +
    geom_raster(data = HARV_DTM_df , aes(x = x, y = y, fill = HARV_dtmCrop)) +
    scale_fill_viridis_c() +
    coord_fixed()
```

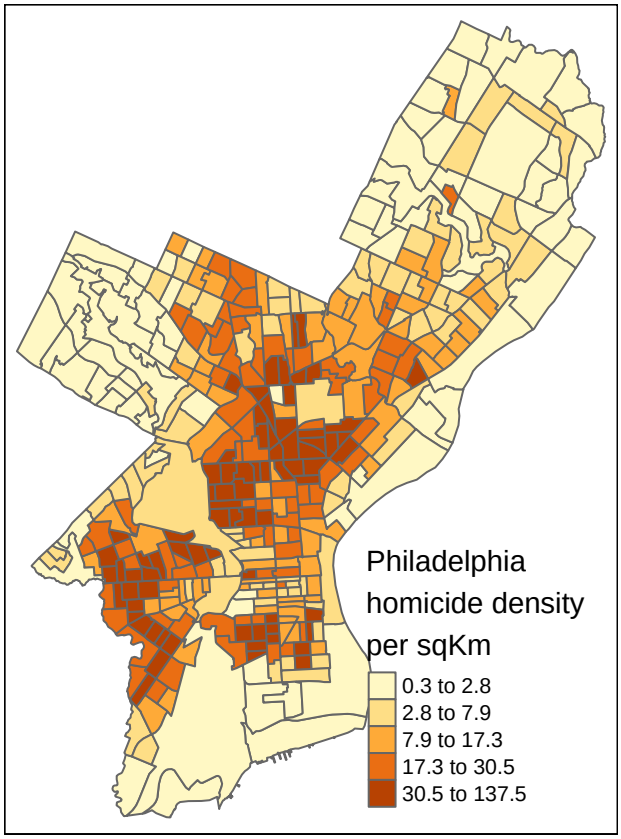


3.3 Choropleth with tmap

`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
- `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("homic_rate",
              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")
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")`.

3.4 Raster with `tmap`

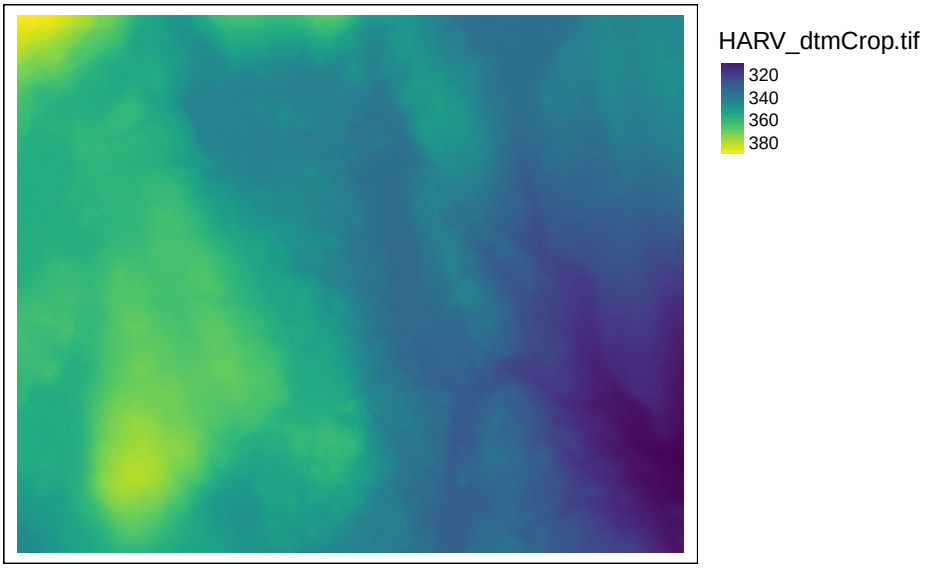
`tmap` can also plot raster files natively, for example:

```
tmap_mode("plot")
```

```
#> tmap mode set to plotting
```

```
tm_shape(HARV_DTM)+
    tm_raster(style = "cont", palette = "viridis")+
    tm_layout(legend.outside = TRUE)
```

```
#> stars object downsampled to 1114 by 897 cells. See tm_shape manual (argument raster.downsample
```



See Elegant and informative maps with `tmap` for more options.

3.5 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 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_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 Rate: </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 va
    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 breaks[1].

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(
    stroke = FALSE,
```

[1]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**)."

```
    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
- rayshader: Create Maps and Visualize Data in 2D and 3D

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