Matching points to polygons

This document demonstrates approaches for determining the (multi)polygon each geographic point falls into. The process relies heavily on the **sf** package (Pebesma, 2018).

Specifically, we will determine which zip code tabulation area (ZCTA) 960 points spread throughout the state of Colorado fall into.

The materials for this demonstration are available at [https://github.com/jfrench/code_demonstration/tree/main/match_points_to_polygons]

Interested parties will need to:

- Install and load the **sf** package.
- Download `usgs_data_series_520_clean.csv` into their current working directory.
- Unzip `co_zcta.zip` into their current working directory. I assumed the zip file is unzipped into the `co_zcta` folder.

```
library(sf) # needed for spatial analysis
# download csv file to working directly
download.file("https://raw.githubusercontent.com/jfrench/code_demonstration/main/match_points_to_polygon
              destfile = "usgs_data_series_520_clean.csv")
# download zip file to working directly
download.file("https://github.com/jfrench/code_demonstration/raw/main/match_points_to_polygons/co_zcta.
              "co_zcta.zip")
# unzip co_zcta.zip into working directory
unzip("co_zcta.zip")
```

First, we use `sf::st_read` to import the shapefile describing the ZCTAs. The `sf::st_read` function will import the shapefile as an `sf` object with `MULTIPOLYGON` geometry type. We assign the name `co_zcta` to this object.

```
# read multipolygon object
co_zcta <- st_read("./co_zcta/Colorado_ZIP_Code_Tabulation_Areas_ZCTA.shp")
```

```
## Reading layer `Colorado_ZIP_Code_Tabulation_Areas_ZCTA' from data source
##   `C:\Users\frencjos\Documents\OneDrive - The University of Colorado Denver\GitHub\code_demonstratio
##   using driver `ESRI Shapefile'
## Simple feature collection with 526 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -109.2712 ymin: 36.84912 xmax: -101.9768 ymax: 41.12812
## Geodetic CRS:  NAD83
```

We first visualize `co_zcta` by using the `sf::st_geometry` function to extract its geometry object and then plot it using the `plot` function.

```
plot(st_geometry(co_zcta))
```

Next, we read in a `csv` file containing heavy metal data from the U.S. Geological Survey [https://pubs.usgs.gov/ds/520/] (Smith et al., 2010) using the `read.csv` function and assign it the name `co_points`. `co_points` is a data frame contains the measurements for 960 locations in Colorado; the data frame includes longitude/latitude coordinates for each measurement.

```r
# read data frame
co_points <- read.csv("usgs_data_series_520_clean.csv")
class(co_points)
```

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

```r
# column names
names(co_points)
```

```
##  [1] "laboratory_number" "field_number"      "latitude"
##  [4] "longitude"         "date_collected"    "land_use"
##  [7] "hg_mg_kg"          "al_pct"            "ca_pct"
## [10] "fe_pct"            "k_pct"             "mg_pct"
## [13] "na_pct"            "s_pct"             "ti_pct"
## [16] "ag_mg_kg"          "as_mg_kg"          "ba_mg_kg"
## [19] "be_mg_kg"          "bi_mg_kg"          "cd_mg_kg"
## [22] "ce_mg_kg"          "co_mg_kg"          "cr_mg_kg"
## [25] "cs_mg_kg"          "cu_mg_kg"          "ga_mg_kg"
## [28] "in_mg_kg"          "la_mg_kg"          "li_mg_kg"
## [31] "mn_mg_kg"          "mo_mg_kg"          "nb_mg_kg"
## [34] "ni_mg_kg"          "p_mg_kg"           "pb_mg_kg"
## [37] "rb_mg_kg"          "sb_mg_kg"          "sc_mg_kg"
## [40] "sn_mg_kg"          "sr_mg_kg"          "te_mg_kg"
## [43] "th_mg_kg"          "tl_mg_kg"          "u_mg_kg"
## [46] "v_mg_kg"           "w_mg_kg"           "y_mg_kg"
## [49] "zn_mg_kg"          "se_mg_kg"
```

We need to convert the `co_points` data frame to an `sf` object, which can be easily done using `sf::st_as_sf`. The first argument is the object to be converted and the `coords` argument is the column names describing the spatial locations associated with each row of the data frame.

```r
# convert to sf object
co_points <- sf::st_as_sf(co_points, coords = c("longitude", "latitude"))
class(co_points)
```

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

Unfortunately, the coordinate reference system (CRS) of the `co_zcta` and `co_points` do not match. We can see this using the `sf::st_crs` function.

```r
st_crs(co_zcta)
```

```
## Coordinate Reference System:
##   User input: NAD83
##   wkt:
## GEOGCRS["NAD83",
##     DATUM["North American Datum 1983",
##         ELLIPSOID["GRS 1980",6378137,298.257222101,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["latitude",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["longitude",east,
##             ORDER[2],
```

```
##             ANGLEUNIT["degree",0.0174532925199433]],
##     ID["EPSG",4269]]
```

```r
st_crs(co_points)
```

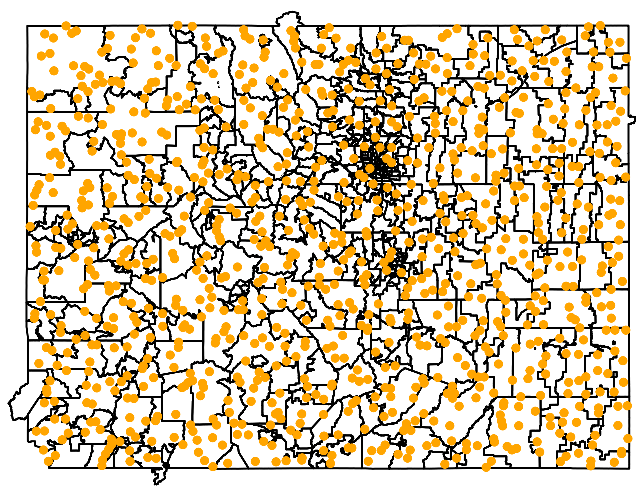
```
## Coordinate Reference System: NA
```

The geometry objects for both `co_zcta` and `co_points` are both defined in terms of longitude/latitude coordinates, but have differenct CRSs. When `sf` objects have different CRSs, we can use `sf::st_transform` to transform one of the objects to the same CRS as the other object. However, since the CRS of `co_points` is `NA`, we will simply set the CRS of `co_points` to the CRS of `co_zcta` using `sf::st_set_crs`.

```r
# set crs of co_points to same crs as co_zcta
co_points <- sf::st_set_crs(co_points, sf::st_crs(co_zcta))
```

To check whether there are any obvious problems with our two `sf` objects, we plot the geometry of both objects in one graphic. Note that we use `add = TRUE` in the second `plot` call to draw the geometry of `co_points` on the plot of `co_zcta`. The object seems to mesh together nicely

```r
# plot geometry
plot(st_geometry(co_zcta))
# plot points on polygons
plot(st_geometry(co_points), add = TRUE, pch = 19, col = "orange", cex = 0.5)
```



To determine the points of `co_points` that are in the multipolygons contained in `co_zcta`, we can use `sf::st_contains`. The first argument is the object whose geometries *contain* the geometries of the second argument. We store the results in `sf::st_contains` returns a `list`; each element of the list indicates the points in `co_points` that are contained in the associated multipolygon of `co_zcta`. e.g., After assigning the list returned by `sf::st_contains` the name `region_points_list`, we see that the 200th point in `co_points` is contained in the 4th multipolygon of `co_zcta`.

```
# determine which points are in which polygon of co_zcta
region_points_list  <- sf::st_contains(co_zcta, co_points)
region_points_list[4]
```

```
## [[1]]
## [1] 200
```

Once we have the list of indices of the points contained in each multipolygon, we can use `sapply` and `length` to count the number of points in each multipolygon.

```
# convert list of points to count
region_count <- sapply(region_points_list, length)
head(region_count)
```

```
## [1]  0  0  1  1 11  1
```

We check the length of the geometry of `co_zcta` and the length of the counts to ensure they are the same length.

```
# make sure number of regions matches number of counts
length(st_geometry(co_zcta))
```

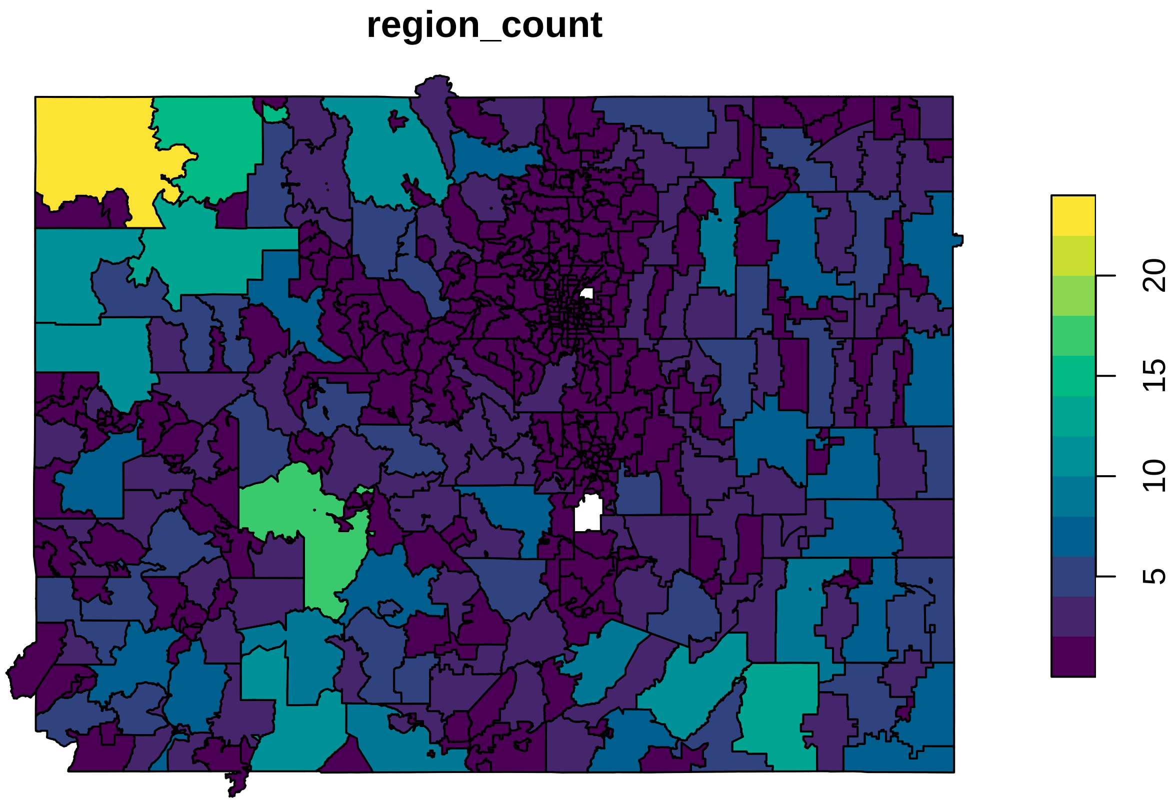
```
## [1] 526
```

```
length(region_count)
```

```
## [1] 526
```

Since the lengths are the same, we add the `region_count` variable to the `co_zcta` `sf` object and plot the result.

```
# add region_count to co_zcta
co_zcta$region_count <- region_count
# plot region_count variable
plot(co_zcta["region_count"], pal = hcl.colors)
```



Alternatively, you may wish to identify which multipolygon of `co_zcta` contains each point in `co_points`. The `sf::st_within` function will perform this task. The first argument is the object whose geometries are in the second argument; in this case, the points in `co_points` that are *within* the multipolygons of `co_zcta`. We assign the list of results the name `points_region_list`.

```
# determine the multipolygon of co_zcta each point is in
points_region_list  <- sf::st_within(co_points, co_zcta)
```

Each point should only be in no more than a single multipolygon. We compute the `length` of each element of the list using the `sapply` function and then check the range of the results. Each point is in 0 or 1 regions (there are some points outside the multipolygons in the bottom left of the graphic above.)

```
# double-check that each point is in at most 1 region
range(sapply(points_region_list, length))
```

```
## [1] 0 1
```

Since each point is only in a single multiplygon, we use `as.numeric` to convert the list to a vector.

```
# since the point should only intersect a single region, convert to number
within <- as.numeric(points_region_list)
```

To double-check that we have performed the desired task correctly, we check that the length of `within` is the same as the number of points in `co_points`.

```
# double-check that length(within) matches number of points
# add within column to co_points
length(within)
```

```
## [1] 960
```

```r
length(st_geometry(co_points))
```

```
## [1] 960
```

Since the lengths do match, we add `within` as a column to `co_points`.

```r
# add within column to co_points data frame
co_points$within  <- within
```

References

Smith, D.B., Ellefsen, K.J., and Kilburn, J.E., 2010. Geochemical data for Colorado soils—Results from the 2006 state-scale geochemical survey: U.S. Geological Survey, Data Series 520, 9 p.

Pebesma, E., 2018. Simple Features for R: Standardized Support for Spatial Vector Data. The R Journal 10 (1), 439-446, [https://doi.org/10.32614/RJ-2018-009]