Chapter 2 Handle vector data using the sf package | R as GIS for Economists

Taro Mieno

2020-05-15

* [R as GIS for Economists](./)
* [Welcome](index.html)
* [Preface](preface.html)
  + [Why R as GIS for Economists?](preface.html#why-r-as-gis-for-economists)
    - [R vs Python](preface.html#r-vs-python)
    - [R vs ArcGIS or QGIS](preface.html#r-vs-arcgis-or-qgis)
    - [Summary](preface.html#summary)
  + [How is this book different from other online books and resources?](preface.html#how-is-this-book-different-from-other-online-books-and-resources)
  + [What is going to be covered in this book?](preface.html#what-is-going-to-be-covered-in-this-book)
  + [Conventions of the book and some notes](preface.html#conventions-of-the-book-and-some-notes)
    - [Texts in gray boxes](preface.html#texts-in-gray-boxes)
    - [Colored Boxes](preface.html#colored-boxes)
    - [Parentheses around codes](preface.html#parentheses-around-codes)
    - [Footnotes](preface.html#footnotes)
  + [Session Information](preface.html#session-information)
* [**1** R as GIS: Demonstrations](demo.html)
  + [Before you start](demo.html#before-you-start)
    - [Target Audience](demo.html#target-audience)
    - [Direction for replication](demo.html#direction-for-replication)
  + [**1.1** Demonstration 1: The impact of groundwater pumping on depth to water table](demo.html#Demo1)
    - [**1.1.1** Project Overview](demo.html#project-overview)
    - [**1.1.2** Project Demonstration](demo.html#project-demonstration)
  + [**1.2** Demonstration 2: Precision Agriculture](demo.html#demonstration-2-precision-agriculture)
    - [**1.2.1** Project Overview](demo.html#project-overview-1)
    - [**1.2.2** Project Demonstration](demo.html#project-demonstration-1)
  + [**1.3** Demonstration 3: Land Use and Weather](demo.html#demonstration-3-land-use-and-weather)
    - [**1.3.1** Project Overview](demo.html#project-overview-2)
    - [**1.3.2** Project Demonstration](demo.html#project-demonstration-2)
  + [**1.4** Demonstration 4: The Impact of Railroad Presence on Corn Planted Acreage](demo.html#demo4)
    - [**1.4.1** Project Overview](demo.html#project-overview-3)
    - [**1.4.2** Project Demonstration](demo.html#project-demonstration-3)
  + [**1.5** Demonstration 5: Groundwater use for agricultural irrigation](demo.html#demonstration-5-groundwater-use-for-agricultural-irrigation)
    - [**1.5.1** Project Overview](demo.html#project-overview-4)
    - [**1.5.2** Project Demonstration](demo.html#project-demonstration-4)
* [**2** Handle vector data using the sf package](vector-basics.html)
  + [Before you start](vector-basics.html#before-you-start-1)
    - [sf or sp?](vector-basics.html#sf-or-sp)
  + [**2.1** Spatial Data Structure](vector-basics.html#spatial-data-structure)
  + [**2.2** Simple feature geometry, simple feature geometry list-column, and simple feature](vector-basics.html#simple-feature-geometry-simple-feature-geometry-list-column-and-simple-feature)
    - [**2.2.1** Simple feature geometry (sfg)](vector-basics.html#simple-feature-geometry-sfg)
    - [**2.2.2** Create simple feature geometry list-column (sfc) and simple feature (sf) from scratch](vector-basics.html#create-simple-feature-geometry-list-column-sfc-and-simple-feature-sf-from-scratch)
  + [**2.3** Reading and writing vector data](vector-basics.html#reading-and-writing-vector-data)
    - [**2.3.1** Reading a shapefile](vector-basics.html#reading-a-shapefile)
    - [**2.3.2** Writing to a shapefile](vector-basics.html#writing-to-a-shapefile)
    - [**2.3.3** Better alternatives](vector-basics.html#better-alternatives)
  + [**2.4** Projection with a different Coordinate Reference Systems](vector-basics.html#projection-with-a-different-coordinate-reference-systems)
  + [**2.5** Non-spatial transformation of sf](vector-basics.html#non-spatial-transformation-of-sf)
  + [**2.6** Turning a data.frame of points into an sf](vector-basics.html#turning-a-data.frame-of-points-into-an-sf)
  + [**2.7** Conversion to and from sp](vector-basics.html#conversion-to-and-from-sp)
  + [**2.8** Geometrical operations](vector-basics.html#geometrical-operations)
    - [**2.8.1** st\_buffer](vector-basics.html#st_buffer)
    - [**2.8.2** st\_area](vector-basics.html#st_area)
    - [**2.8.3** st\_centroid](vector-basics.html#st_centroid)
    - [**2.8.4** st\_length](vector-basics.html#st_length)
* [**3** Spatial Interactions of Vector Data: Subsetting and Joining](spatial-interactions-of-vector-data-subsetting-and-joining.html)
  + [**3.1** Topological relations](spatial-interactions-of-vector-data-subsetting-and-joining.html#topological-relations)
    - [**3.1.1** st\_intersects()](spatial-interactions-of-vector-data-subsetting-and-joining.html#st_intersects)
    - [**3.1.2** st\_intersection()](spatial-interactions-of-vector-data-subsetting-and-joining.html#st_intersection)
    - [**3.1.3** st\_is\_within\_distance()](spatial-interactions-of-vector-data-subsetting-and-joining.html#st_is_within_distance)
  + [**3.2** Spatial Subsetting (or Flagging)](spatial-interactions-of-vector-data-subsetting-and-joining.html#spatial-subsetting-or-flagging)
    - [**3.2.1** polygons vs polygons](spatial-interactions-of-vector-data-subsetting-and-joining.html#polygons-vs-polygons)
    - [**3.2.2** points vs polygons](spatial-interactions-of-vector-data-subsetting-and-joining.html#points-vs-polygons)
    - [**3.2.3** lines vs polygons](spatial-interactions-of-vector-data-subsetting-and-joining.html#lines-vs-polygons)
    - [**3.2.4** Flagging instead of subsetting](spatial-interactions-of-vector-data-subsetting-and-joining.html#flagging-instead-of-subsetting)
  + [**3.3** Spatial Join](spatial-interactions-of-vector-data-subsetting-and-joining.html#spatial-join)
    - [**3.3.1** Case 1: points (target) vs polygons (source)](spatial-interactions-of-vector-data-subsetting-and-joining.html#case-1-points-target-vs-polygons-source)
    - [**3.3.2** Case 2: polygons (target) vs points (source)](spatial-interactions-of-vector-data-subsetting-and-joining.html#case-2-polygons-target-vs-points-source)
    - [**3.3.3** Case 3: polygons (target) vs polygons (source)](spatial-interactions-of-vector-data-subsetting-and-joining.html#polygon-polygon)
* [Published with bookdown](https://github.com/rstudio/bookdown)

# [R as GIS for Economists](./)

# Chapter 2 Handle vector data using the sf package

## Before you start

In this chapter we learn how to use the sf package to handle and operate on spatial datasets. The sf package uses the class of simple feature (sf)[36](#fn36) for spatial objects in R. We first learn how sf objects store and represent spatial datasets. We then move on to the following practical topics:

* read and write a shapefile and spatial data in other formats (and why you might not want to use the shapefile system any more, but use other alternative formats)
* project and reproject spatial objects
* convert sf objects into sp objective, vice versa
* confirm that dplyr works well with sf objects
* implement non-interactive (does not involve two sf objects) geometric operations on sf objects
  + create buffers
  + find the area of polygons
  + find the centroid of polygons
  + calculate the length of lines

### sf or sp?

The sf package was designed to replace the sp package, which has been one of the most popular and powerful spatial packages in R for more than a decade. It has been about four years since sf package was first registered on CRAN. A couple of years back, many other spatial packages did not have support for the package yet. In this [blog post](https://www.r-bloggers.com/should-i-learn-sf-or-sp-for-spatial-r-programming/) that asked the question of whether one should learn sp of sf, the author said:

"That’s a tough question. If you have time, I would say, learn to use both. sf is pretty new, so a lot of packages that depend on spatial classes still rely on sp. So you will need to know sp if you want to do any integration with many other packages, including raster (as of March 2018).

However, in the future we should see an increasing shift toward the sf package and greater use of sf classes in other packages. I also think that sf is easier to learn to use than sp."

The future has come, and it’s not a tough question anymore. I cannot think of any major spatial packages that do not support sf package, and sf has largely becomes the standard for handling vector data in \(R\). Thus, this lecture note does not cover how to use sp at all.[37](#fn37)

sf has several advantages over sp package (Pebesma [2018](#ref-pebesma2018simple)).[38](#fn38) First, it cut off the tie that sp had with ESRI shapefile system, which has somewhat loose way of representing spatial data. Instead, it uses *simple feature access*, which is an open standard supported by Open Geospatial Consortium (OGC). Another important benefit is its compatibility with the tidyverse package, which include widely popular packages like ggplot2 and dplyr. Consequently, map-making with ggplot() and data wrangling with a family of dplyr functions come very natural to many \(R\) users. sp objects have different slots for spatial information and attributes data, and they are not amenable to dplyr way of data transformation.

## 2.1 Spatial Data Structure

Here, we learn how the sf package stores spatial data, along with the definition of three key sf object classes: simple feature geometry (sfg), simple feature geometry list-column (sfc), and simple feature (sf). The sf package provides a simply way of storing geographic information and the attributes of the geographic units in a single dataset. This special type of dataset is called simple feature (sf). It is best to take a look at an example to see how this is achieved. We use North Carolina county boundaries with county attributes (Figure [2.1](vector-basics.html#fig:nc-county)).

#--- a dataset that comes with the sf package ---#  
nc <- st\_read(system.file("shape/nc.shp", package="sf"))

Reading layer `nc' from data source `/Library/Frameworks/R.framework/Versions/4.0/Resources/library/sf/shape/nc.shp' using driver `ESRI Shapefile'  
Simple feature collection with 100 features and 14 fields  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965  
CRS: 4267

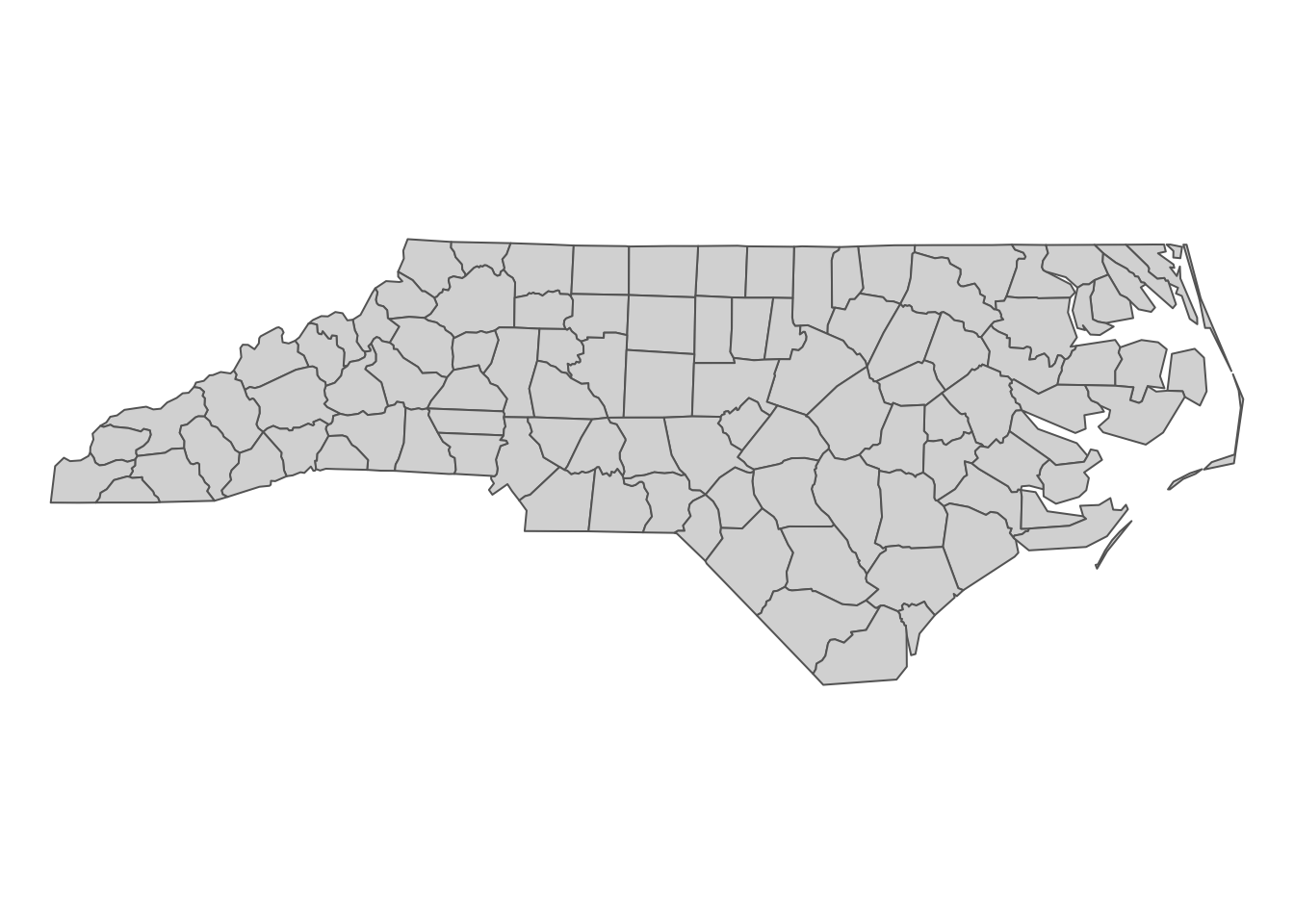


Figure 2.1: North Carolina county boundary

As you can see below, this dataset is of class sf (and data.frame at the same time).

class(nc)

[1] "sf" "data.frame"

Now, let’s take a look inside of nc.

#--- take a look at the data ---#  
head(nc)

Simple feature collection with 6 features and 14 fields  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: -81.74107 ymin: 36.07282 xmax: -75.77316 ymax: 36.58965  
CRS: 4267  
 AREA PERIMETER CNTY\_ CNTY\_ID NAME FIPS FIPSNO CRESS\_ID BIR74 SID74  
1 0.114 1.442 1825 1825 Ashe 37009 37009 5 1091 1  
2 0.061 1.231 1827 1827 Alleghany 37005 37005 3 487 0  
3 0.143 1.630 1828 1828 Surry 37171 37171 86 3188 5  
4 0.070 2.968 1831 1831 Currituck 37053 37053 27 508 1  
5 0.153 2.206 1832 1832 Northampton 37131 37131 66 1421 9  
6 0.097 1.670 1833 1833 Hertford 37091 37091 46 1452 7  
 NWBIR74 BIR79 SID79 NWBIR79 geometry  
1 10 1364 0 19 MULTIPOLYGON (((-81.47276 3...  
2 10 542 3 12 MULTIPOLYGON (((-81.23989 3...  
3 208 3616 6 260 MULTIPOLYGON (((-80.45634 3...  
4 123 830 2 145 MULTIPOLYGON (((-76.00897 3...  
5 1066 1606 3 1197 MULTIPOLYGON (((-77.21767 3...  
6 954 1838 5 1237 MULTIPOLYGON (((-76.74506 3...

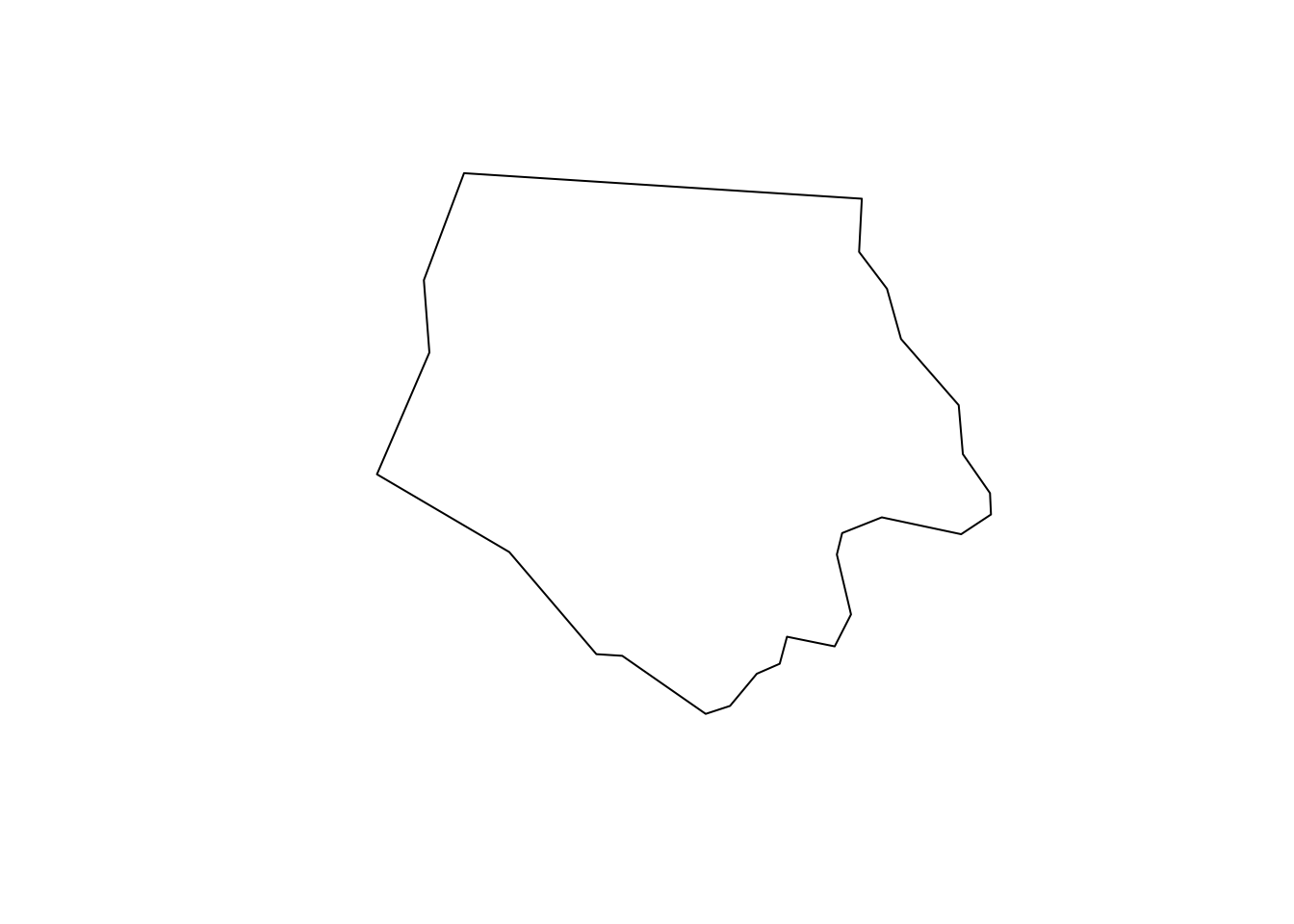
Just like a regular data.frame, you see a number of variables (attributes) except that you have a variable called geometry at the end. Each row represents a single geographic unit (here, county). Ashe County (1st row) has area of \(0.114\), FIPS code of \(37009\), and so on. And the entry in geometry column at the first row represents the geographic information of Ashe County. An entry in the geometry column is a simple feature geometry (sfg), which is an \(R\) object that represents the geographic information of a single geometric feature (county in this example). There are different types of sfgs (POINT, LINESTRING, POLYGON, MULTIPOLYGON, etc). Here, sfgs representing counties in NC are of type MULTIPOLYGON. Let’s take a look inside the sfg for Ashe County using st\_geometry().

st\_geometry(nc[1, ])[[1]][[1]]

[[1]]  
 [,1] [,2]  
 [1,] -81.47276 36.23436  
 [2,] -81.54084 36.27251  
 [3,] -81.56198 36.27359  
 [4,] -81.63306 36.34069  
 [5,] -81.74107 36.39178  
 [6,] -81.69828 36.47178  
 [7,] -81.70280 36.51934  
 [8,] -81.67000 36.58965  
 [9,] -81.34530 36.57286  
[10,] -81.34754 36.53791  
[11,] -81.32478 36.51368  
[12,] -81.31332 36.48070  
[13,] -81.26624 36.43721  
[14,] -81.26284 36.40504  
[15,] -81.24069 36.37942  
[16,] -81.23989 36.36536  
[17,] -81.26424 36.35241  
[18,] -81.32899 36.36350  
[19,] -81.36137 36.35316  
[20,] -81.36569 36.33905  
[21,] -81.35413 36.29972  
[22,] -81.36745 36.27870  
[23,] -81.40639 36.28505  
[24,] -81.41233 36.26729  
[25,] -81.43104 36.26072  
[26,] -81.45289 36.23959  
[27,] -81.47276 36.23436

As you can see, the sfg consists of a number of points (pairs of two numbers). Connecting the points in the order they are stored delineate the Ashe County boundary.

plot(st\_geometry(nc[1, ]))



We will take a closer look at different types of sfg in the next section.

Finally, the geometry variable is a list of individual sfgs, called simple feature geometry list-column (sfc).

dplyr::select(nc, geometry)

Simple feature collection with 100 features and 0 fields  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965  
CRS: 4267  
First 10 features:  
 geometry  
1 MULTIPOLYGON (((-81.47276 3...  
2 MULTIPOLYGON (((-81.23989 3...  
3 MULTIPOLYGON (((-80.45634 3...  
4 MULTIPOLYGON (((-76.00897 3...  
5 MULTIPOLYGON (((-77.21767 3...  
6 MULTIPOLYGON (((-76.74506 3...  
7 MULTIPOLYGON (((-76.00897 3...  
8 MULTIPOLYGON (((-76.56251 3...  
9 MULTIPOLYGON (((-78.30876 3...  
10 MULTIPOLYGON (((-80.02567 3...

Elements of a geometry list-column are allowed to be different in nature from other elements[39](#fn39). In the nc data, all the elements (sfgs) in geometry column are MULTIPOLYGON. However, you could also have LINESTRING or POINT objects mixed with MULTIPOLYGONS objects in a single sf object if you would like.

## 2.2 Simple feature geometry, simple feature geometry list-column, and simple feature

Here, we learn how different types of sfg are constructed. We also learn how to create sfc and sf from sfg from scratch.[40](#fn40)

### 2.2.1 Simple feature geometry (sfg)

The sf package uses a class of sfg (simple feature geometry) objects to represent a geometry of a single geometric feature (say, a city as a point, a river as a line, county and school district as polygons). There are different types of sfgs. Here are some example feature types that we commonly encounter as an economist[41](#fn41):

* POINT: area-less feature that represents a point (e.g., well, city, farmland)
* LINESTRING: (e.g., a tributary of a river)
* MULTILINESTRING: (e.g., river with more than one tributaries)
* POLYGON: geometry with a positive area (e.g., county, state, country)
* MULTIPOLYGON: collection of polygons to represent a single object (e.g., countries with islands: U.S., Japan)

POINT is the simplest geometry type, and is represented by a vector of two[42](#fn42) numeric values. An example below shows how a POINT feature can be made from scratch[43](#fn43):

#--- create a POINT ---#  
a\_point <- st\_point(c(2,1))

The st\_point() function creates a POINT object when supplied with a vector of two numeric values. If you check the class of the newly created object,

#--- check the class of the object ---#  
class(a\_point)

[1] "XY" "POINT" "sfg"

You can see that it’s indeed a POINT object. But, it’s also an sfg object. So, a\_point is an sfg object of type POINT.

A LINESTRING objects are represented by a sequence of points:

#--- collection of points in a matrix form ---#  
s1 <- rbind(c(2,3),c(3,4),c(3,5),c(1,5))  
  
#--- see what s1 looks like ---#  
s1

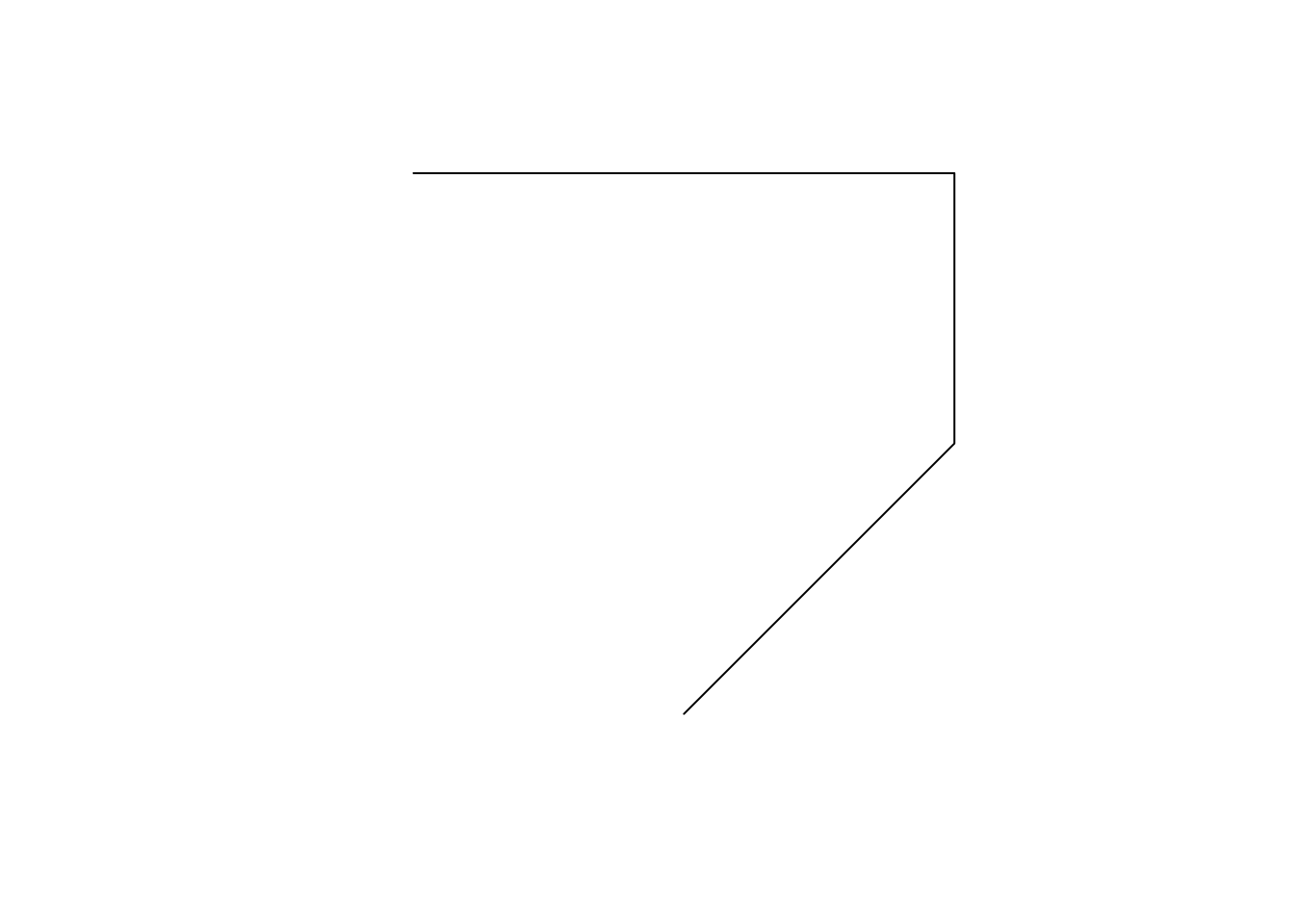
[,1] [,2]  
[1,] 2 3  
[2,] 3 4  
[3,] 3 5  
[4,] 1 5

#--- create a "LINESTRING" ---#  
a\_linestring <- st\_linestring(s1)  
  
#--- check the class ---#  
class(a\_linestring)

[1] "XY" "LINESTRING" "sfg"

s1 is a matrix where each row represents a point. By applying st\_linestring() function to s1, you create a LINESTRING object. Let’s see what the line looks like.

plot(a\_linestring)



As you can see, each pair of consecutive points in the matrix are connected by a straight line to form a line.

A POLYGON is very similar to LINESTRING in the manner it is represented.

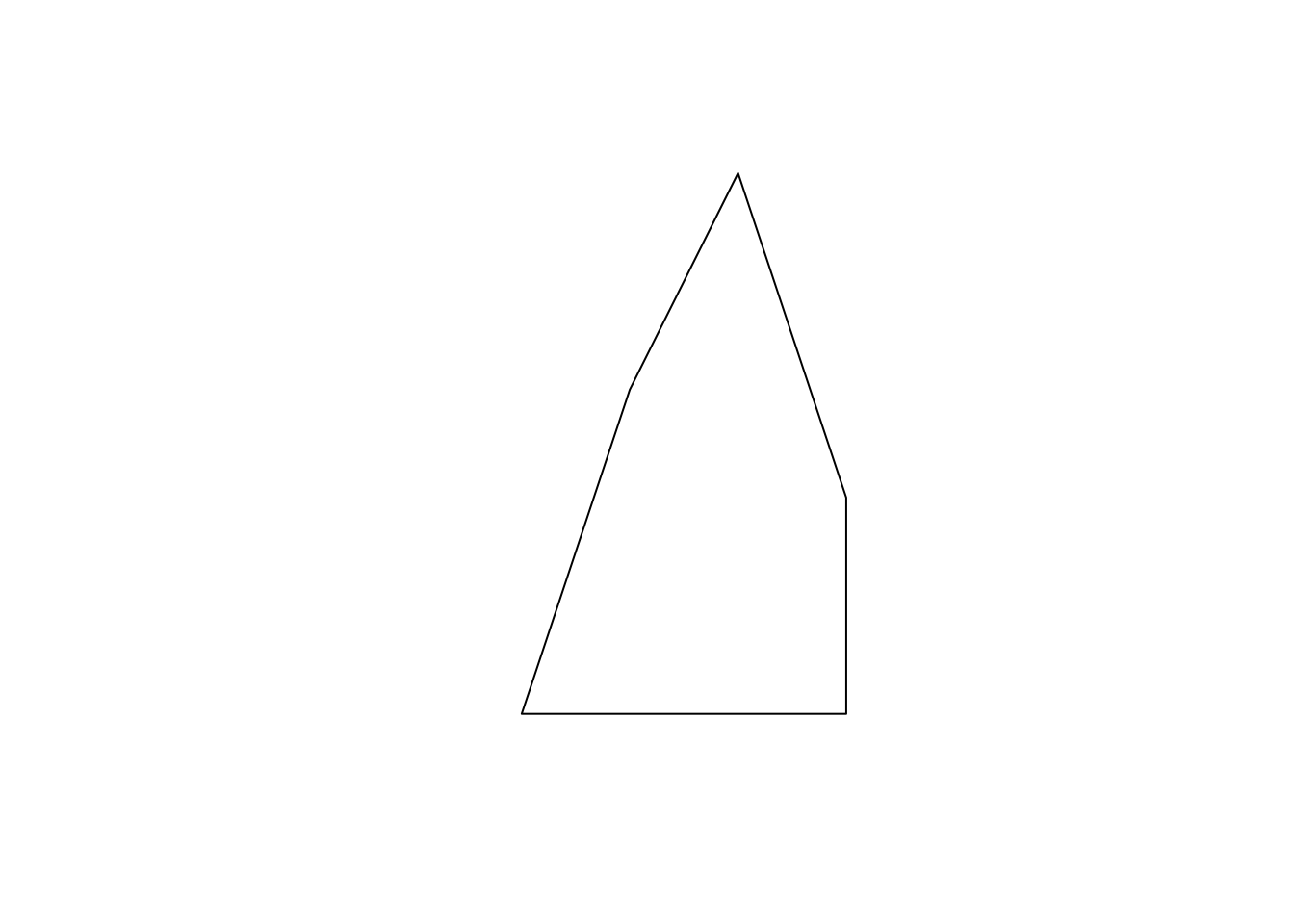
#--- collection of points in a matrix form ---#  
p1 <- rbind(c(0,0), c(3,0), c(3,2), c(2,5), c(1,3), c(0,0))  
  
#--- see what s1 looks like ---#  
p1

[,1] [,2]  
[1,] 0 0  
[2,] 3 0  
[3,] 3 2  
[4,] 2 5  
[5,] 1 3  
[6,] 0 0

#--- create a "LINESTRING" ---#  
a\_polygon <- st\_polygon(list(p1))  
  
#--- check the class ---#  
class(a\_polygon)

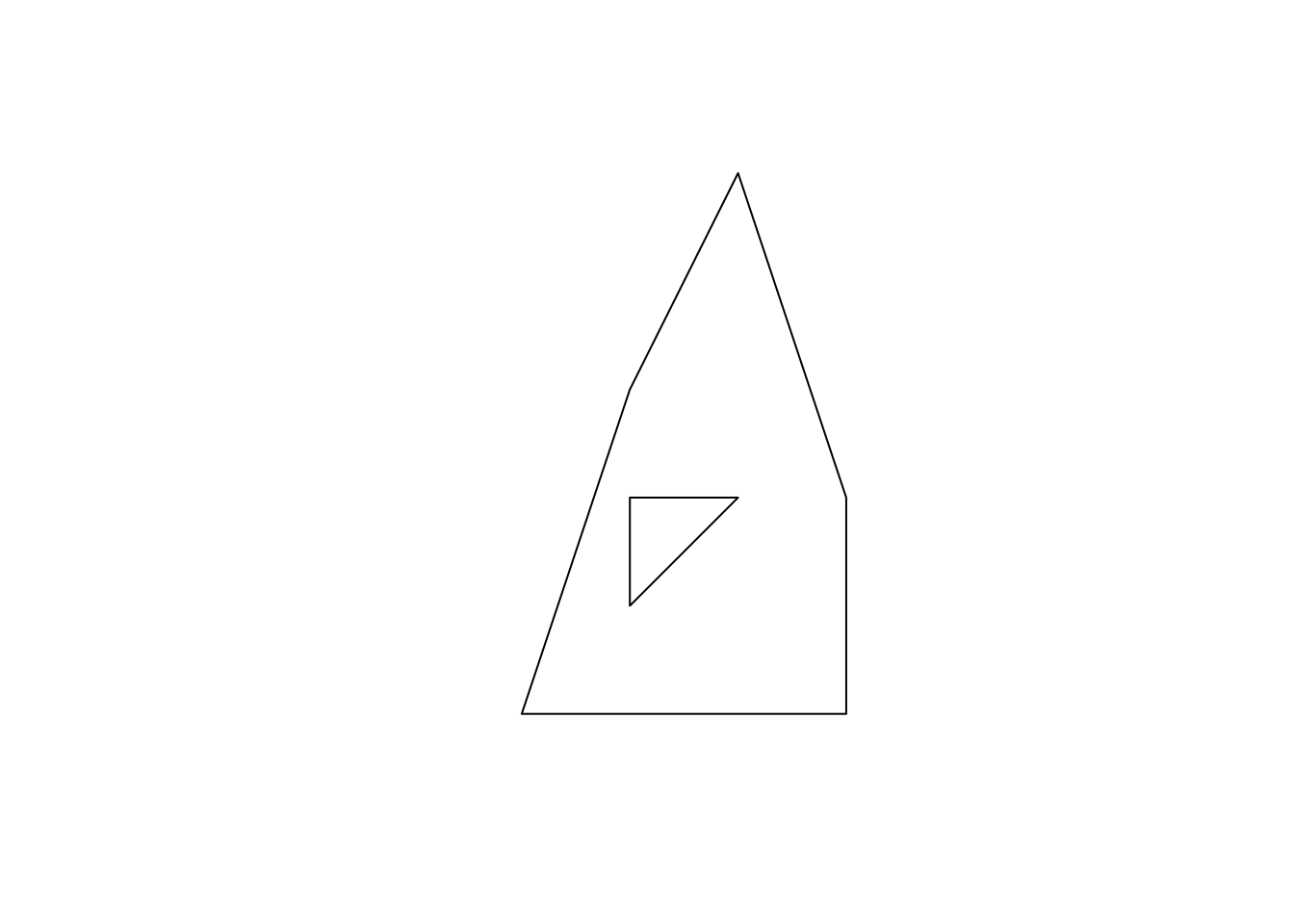
[1] "XY" "POLYGON" "sfg"

#--- see what it looks like ---#  
plot(a\_polygon)



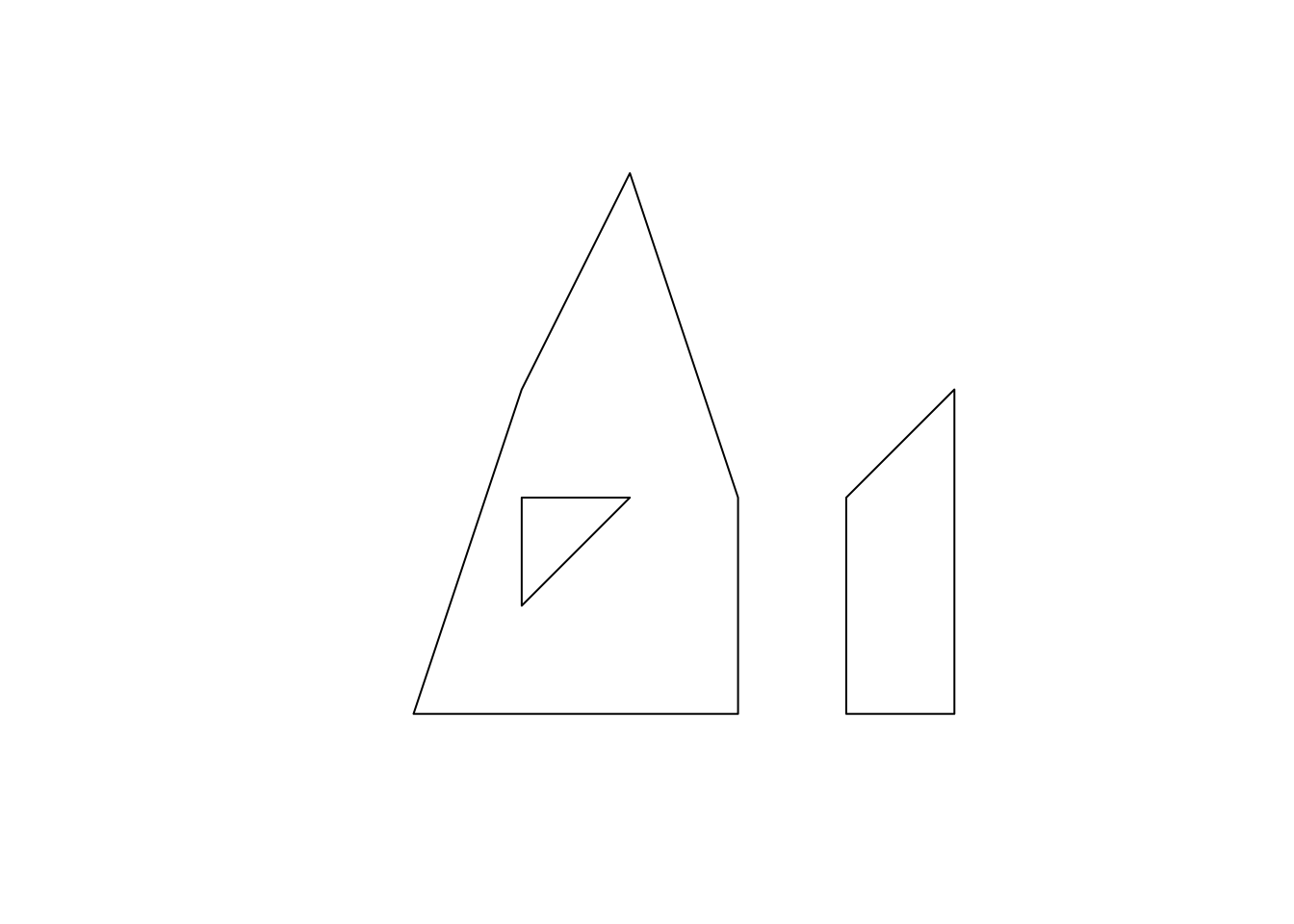
Just like the LINESTRING object we created earlier, a POLYGON is represented by a collection of points. The biggest difference between them is that we need to have some positive area enclosed by lines connecting the points. To do that, you have the the same point for the first and last points to close the loop: here, it’s c(0,0). A POLYGON can have a hole in it. The first matrix of a list becomes the exterior ring, and all the subsequent matrices will be holes within the exterior ring.

#--- a hole within p1 ---#  
p2 <- rbind(c(1,1), c(1,2), c(2,2), c(1,1))  
  
#--- create a polygon with hole ---#  
a\_plygon\_with\_a\_hole <- st\_polygon(list(p1,p2))  
  
#--- see what it looks like ---#  
plot(a\_plygon\_with\_a\_hole)



You can create a MULTIPOLYGON object in a similar manner. The only difference is that you supply a list of lists of matrices, with each inner list representing a polygon. An example below:

#--- second polygon ---#  
p3 <- rbind(c(4,0), c(5,0), c(5,3), c(4,2), c(4,0))   
  
#--- create a multipolygon ---#  
a\_multipolygon <- st\_multipolygon(list(list(p1,p2), list(p3)))  
  
#--- see what it looks like ---#  
plot(a\_multipolygon)



Each of list(p1,p2), list(p3,p4), list(p5) represents a polygon. You supply a list of these lists to the st\_multipolygon() function to make a MULTIPOLYGON object.

### 2.2.2 Create simple feature geometry list-column (sfc) and simple feature (sf) from scratch

To make a simple feature geometry list-column (sfc), you can simply supply a list of sfg to the st\_sfc() function as follows:

#--- create an sfc ---#  
sfc\_ex <- st\_sfc(list(a\_point,a\_linestring,a\_polygon,a\_multipolygon))

To create an sf object, you first add an sfc as a column to a data.frame.

#--- create a data.frame ---#  
df\_ex <- data.frame(  
 name=c('A','B','C','D')  
)  
  
#--- add the sfc as a column ---#  
df\_ex$geometry <- sfc\_ex   
  
#--- take a look ---#  
df\_ex

name geometry  
1 A POINT (2 1)  
2 B LINESTRING (2 3, 3 4, 3 5, ...  
3 C POLYGON ((0 0, 3 0, 3 2, 2 ...  
4 D MULTIPOLYGON (((0 0, 3 0, 3...

At this point, it is not recognized as an sf by R yet.

#--- see what it looks like (this is not an sf object yet) ---#  
class(df\_ex)

[1] "data.frame"

You can register it as an sf object using st\_as\_sf().

#--- let R recognize the data frame as sf ---#  
sf\_ex <- st\_as\_sf(df\_ex)  
  
#--- see what it looks like ---#  
sf\_ex

Simple feature collection with 4 features and 1 field  
geometry type: GEOMETRY  
dimension: XY  
bbox: xmin: 0 ymin: 0 xmax: 5 ymax: 5  
CRS: NA  
 name geometry  
1 A POINT (2 1)  
2 B LINESTRING (2 3, 3 4, 3 5, ...  
3 C POLYGON ((0 0, 3 0, 3 2, 2 ...  
4 D MULTIPOLYGON (((0 0, 3 0, 3...

As you can see sf\_ex is now recognized also as an sf object.

#--- check the class ---#  
class(sf\_ex)

[1] "sf" "data.frame"

## 2.3 Reading and writing vector data

I claimed that you do not need ArcGIS \(99\%\) of your work as an economist. However, the vast majority of people still use ArcGIS to handle spatial data, which has its own system of storing spatial data[44](#fn44) called shapefile. So, chances are that your collaborators still use shapefiles. Moreover, there are many GIS data online that are available only as shapefiles. So, it is important to learn how to read and write shapefiles.

### 2.3.1 Reading a shapefile

We can use st\_read() function to read a shapefile. It reads in a shapefile and then turn the data into an sf object. Let’s take a look at an example.

#--- read a NE county boundary shapefile ---#  
nc\_loaded <- st\_read(dsn = "./Data", "nc")

Reading layer `nc' from data source `/Users/tmieno2/Box/Teaching/AAEA R/GIS/Data' using driver `ESRI Shapefile'  
Simple feature collection with 100 features and 1 field  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965  
CRS: 4267

Typically, you have two arguments to specify for st\_read(). The first one is dsn, which is basically the path to the shapefile you want to import. The second one is the name of the shapefile. Notice that you do not add .shp extension to the file name: *NE\_county*, not *NE\_county.shp*.[45](#fn45).

### 2.3.2 Writing to a shapefile

Writing an sf object as a shapefile is just as easy. You use the st\_write() function, with the first argument being the name of the sf object you are exporting, and the second being the name of the new shapefile. For example, the code below will export an sf object called NE\_county as NE\_county\_2.shp (along with other supporting files).

st\_write(nc\_loaded, dsn="./Data", "nc", driver="ESRI Shapefile", append = FALSE)

append = FALSE forces writing the data when there already exists a file with the same name. Without the option, this happens.

st\_write(nc\_loaded, dsn="./Data", "nc", driver="ESRI Shapefile")

Layer nc in dataset ./Data already exists:  
use either append=TRUE to append to layer or append=FALSE to overwrite layer

Error in CPL\_write\_ogr(obj, dsn, layer, driver, as.character(dataset\_options), : Dataset already exists.

### 2.3.3 Better alternatives

Now, if your collaborator is using ArcGIS and demanding that he/she needs a shapefile for his/her work, sure you can use the above command to write a shapefile. But, there is really no need to work with the shapefile system. One of the alternative data formats that are considered superior to the shapefile system is GeoPackage[46](#fn46), which overcomes various limitations associated with shapefile[47](#fn47). Unlike the shapefile system, it produces only a single file with .gpkg extension. Note that GeoPackage file can also be easily read into ArcGIS. So, it might be worthwhile to convince your collaborators to stop using shapefiles and start using GeoPackage.

#--- write as a gpkg file ---#  
st\_write(nc, dsn = "./Data/nc.gpkg")  
  
#--- read a gpkg file ---#  
nc <- st\_read("./Data/nc.gpkg")

Or better yet, if your collaborator uses R (or if it is only you who is going to use the data), then just save it as an rds file using saveRDS(), which can be of course read using readRDS().

#--- save as an rds ---#  
saveRDS(nc, "/Users/tmieno2/Box/Teaching/AAEA R/GIS/nc\_county.rds")  
  
#--- read an rds ---#  
nc <- readRDS("/Users/tmieno2/Box/Teaching/AAEA R/GIS/nc\_county.rds")

The use of rds files can be particularly attractive when the dataset is large because rds files are typically more memory efficient than shapefiles, eating up less of your disk memory.

## 2.4 Projection with a different Coordinate Reference Systems

You often need to reproject an sf using a different coordinate reference system (CRS) because you need it to have the same CRS as an sf object that you are interacting it with (spatial join) or mapping it with. In order to check the current CRS for an sf object, you can use the st\_crs() function.

st\_crs(nc)

Coordinate Reference System:  
 User input: 4267   
 wkt:  
GEOGCS["NAD27",  
 DATUM["North\_American\_Datum\_1927",  
 SPHEROID["Clarke 1866",6378206.4,294.9786982138982,  
 AUTHORITY["EPSG","7008"]],  
 AUTHORITY["EPSG","6267"]],  
 PRIMEM["Greenwich",0,  
 AUTHORITY["EPSG","8901"]],  
 UNIT["degree",0.0174532925199433,  
 AUTHORITY["EPSG","9122"]],  
 AUTHORITY["EPSG","4267"]]

wkt stands for **Well Known Text**[48](#fn48), which is one of many many formats to store CRS information.[49](#fn49) 4267 is the SRID (Spatial Reference System Identifier) defined by the European Petroleum Survey Group (EPSG) for the CRS[50](#fn50).

When you transform your sf using a different CRS, you can use its EPSG number if the CRS has an EPSG number.[51](#fn51) Let’s transform the sf to WGS 84 (another commonly used GCS), whose EPSG number is 4326. We can use the st\_transform() function to achieve that, with the first argument being the sf object you are transforming and the second being the EPSG number of the new CRS.

#--- transform ---#  
nc\_wgs84 <- st\_transform(nc, 4326)  
  
#--- check if the transformation was successful ---#  
st\_crs(nc\_wgs84)

Coordinate Reference System:  
 User input: EPSG:4326   
 wkt:  
GEOGCS["WGS 84",  
 DATUM["WGS\_1984",  
 SPHEROID["WGS 84",6378137,298.257223563,  
 AUTHORITY["EPSG","7030"]],  
 AUTHORITY["EPSG","6326"]],  
 PRIMEM["Greenwich",0,  
 AUTHORITY["EPSG","8901"]],  
 UNIT["degree",0.0174532925199433,  
 AUTHORITY["EPSG","9122"]],  
 AUTHORITY["EPSG","4326"]]

Notice that wkt was also altered accordingly to reflect the change in CRS: datum was changed to WGS 84. Now, let’s transform (reproject) the data using NAD83 / UTM zone 17N CRS. Its EPSG number is \(26917\).[52](#fn52) So, the following code does the job.

#--- transform ---#  
nc\_utm17N <- st\_transform(nc\_wgs84, 26917)  
  
#--- check if the transformation was successful ---#  
st\_crs(nc\_utm17N)

Coordinate Reference System:  
 User input: EPSG:26917   
 wkt:  
PROJCS["NAD83 / UTM zone 17N",  
 GEOGCS["NAD83",  
 DATUM["North\_American\_Datum\_1983",  
 SPHEROID["GRS 1980",6378137,298.257222101,  
 AUTHORITY["EPSG","7019"]],  
 TOWGS84[0,0,0,0,0,0,0],  
 AUTHORITY["EPSG","6269"]],  
 PRIMEM["Greenwich",0,  
 AUTHORITY["EPSG","8901"]],  
 UNIT["degree",0.0174532925199433,  
 AUTHORITY["EPSG","9122"]],  
 AUTHORITY["EPSG","4269"]],  
 PROJECTION["Transverse\_Mercator"],  
 PARAMETER["latitude\_of\_origin",0],  
 PARAMETER["central\_meridian",-81],  
 PARAMETER["scale\_factor",0.9996],  
 PARAMETER["false\_easting",500000],  
 PARAMETER["false\_northing",0],  
 UNIT["metre",1,  
 AUTHORITY["EPSG","9001"]],  
 AXIS["Easting",EAST],  
 AXIS["Northing",NORTH],  
 AUTHORITY["EPSG","26917"]]

As you can see in its CRS information, the projection system is now UTM zone 17N.

You often need to change the CRS of an sf object when you interact[53](#fn53) it with another sf object. In such a case, you can extract the CRS of the other sf object using st\_crs() and use it for transformation[54](#fn54).

#--- transform ---#  
nc\_utm17N\_2 <- st\_transform(nc\_wgs84, st\_crs(nc\_utm17N))  
  
#--- check if the transformation was successful ---#  
st\_crs(nc\_utm17N\_2)

Coordinate Reference System:  
 User input: EPSG:26917   
 wkt:  
PROJCS["NAD83 / UTM zone 17N",  
 GEOGCS["NAD83",  
 DATUM["North\_American\_Datum\_1983",  
 SPHEROID["GRS 1980",6378137,298.257222101,  
 AUTHORITY["EPSG","7019"]],  
 TOWGS84[0,0,0,0,0,0,0],  
 AUTHORITY["EPSG","6269"]],  
 PRIMEM["Greenwich",0,  
 AUTHORITY["EPSG","8901"]],  
 UNIT["degree",0.0174532925199433,  
 AUTHORITY["EPSG","9122"]],  
 AUTHORITY["EPSG","4269"]],  
 PROJECTION["Transverse\_Mercator"],  
 PARAMETER["latitude\_of\_origin",0],  
 PARAMETER["central\_meridian",-81],  
 PARAMETER["scale\_factor",0.9996],  
 PARAMETER["false\_easting",500000],  
 PARAMETER["false\_northing",0],  
 UNIT["metre",1,  
 AUTHORITY["EPSG","9001"]],  
 AXIS["Easting",EAST],  
 AXIS["Northing",NORTH],  
 AUTHORITY["EPSG","26917"]]

## 2.5 Non-spatial transformation of sf

An important feature of an sf object is that it is basically a data.frame with geometric information stored as a variable (column). This means that transforming an sf object works just like transforming a data.frame. Basically, everything you can do to a data.frame, you can do to an sf as well. The code below just provides an example of basic operations including select(), filter(), and mutate() in action with an sf object to just confirm that dplyr operations works with an sf object just like a data.frame.

#--- here is what the data looks like ---#  
dplyr::select(wells\_sf, wellid, nrdname, acres, regdate, nrdname)

Simple feature collection with 105822 features and 4 fields  
geometry type: POINT  
dimension: XY  
bbox: xmin: -104.0531 ymin: 40.00161 xmax: -96.87681 ymax: 41.85942  
CRS: unknown  
First 10 features:  
 wellid nrdname acres regdate geometry  
1 2 Central Platte 160 12/30/55 POINT (-99.58401 40.69825)  
2 3 South Platte 46 4/29/31 POINT (-102.6249 41.11699)  
3 4 South Platte 46 4/29/31 POINT (-102.6249 41.11699)  
4 5 South Platte 46 4/29/31 POINT (-102.6249 41.11699)  
5 6 Central Platte 160 8/29/32 POINT (-99.6258 40.73268)  
6 7 Central Platte 120 2/15/35 POINT (-99.64524 40.73164)  
7 8 South Platte 113 8/7/37 POINT (-103.5257 41.24492)  
8 10 South Platte 160 5/4/38 POINT (-103.0284 41.13243)  
9 11 Middle Republican 807 5/6/38 POINT (-101.1193 40.3527)  
10 12 Middle Republican 148 11/29/77 POINT (-101.1146 40.35631)

#--- do some transformations ---#  
wells\_sf %>%   
 #--- select variables (geometry will always remain after select) ---#  
 dplyr::select(wellid, nrdname, acres, regdate, nrdname) %>%   
 #--- removes observations with acre < 30 ---#  
 filter(acres > 30) %>%   
 #--- hectare instead of acre ---#  
 mutate(hectare = acres \* 0.404686)

Simple feature collection with 63271 features and 5 fields  
geometry type: POINT  
dimension: XY  
bbox: xmin: -104.0529 ymin: 40.00161 xmax: -96.87681 ymax: 41.73599  
CRS: unknown  
First 10 features:  
 wellid nrdname acres regdate geometry hectare  
1 2 Central Platte 160 12/30/55 POINT (-99.58401 40.69825) 64.74976  
2 3 South Platte 46 4/29/31 POINT (-102.6249 41.11699) 18.61556  
3 4 South Platte 46 4/29/31 POINT (-102.6249 41.11699) 18.61556  
4 5 South Platte 46 4/29/31 POINT (-102.6249 41.11699) 18.61556  
5 6 Central Platte 160 8/29/32 POINT (-99.6258 40.73268) 64.74976  
6 7 Central Platte 120 2/15/35 POINT (-99.64524 40.73164) 48.56232  
7 8 South Platte 113 8/7/37 POINT (-103.5257 41.24492) 45.72952  
8 10 South Platte 160 5/4/38 POINT (-103.0284 41.13243) 64.74976  
9 11 Middle Republican 807 5/6/38 POINT (-101.1193 40.3527) 326.58160  
10 12 Middle Republican 148 11/29/77 POINT (-101.1146 40.35631) 59.89353

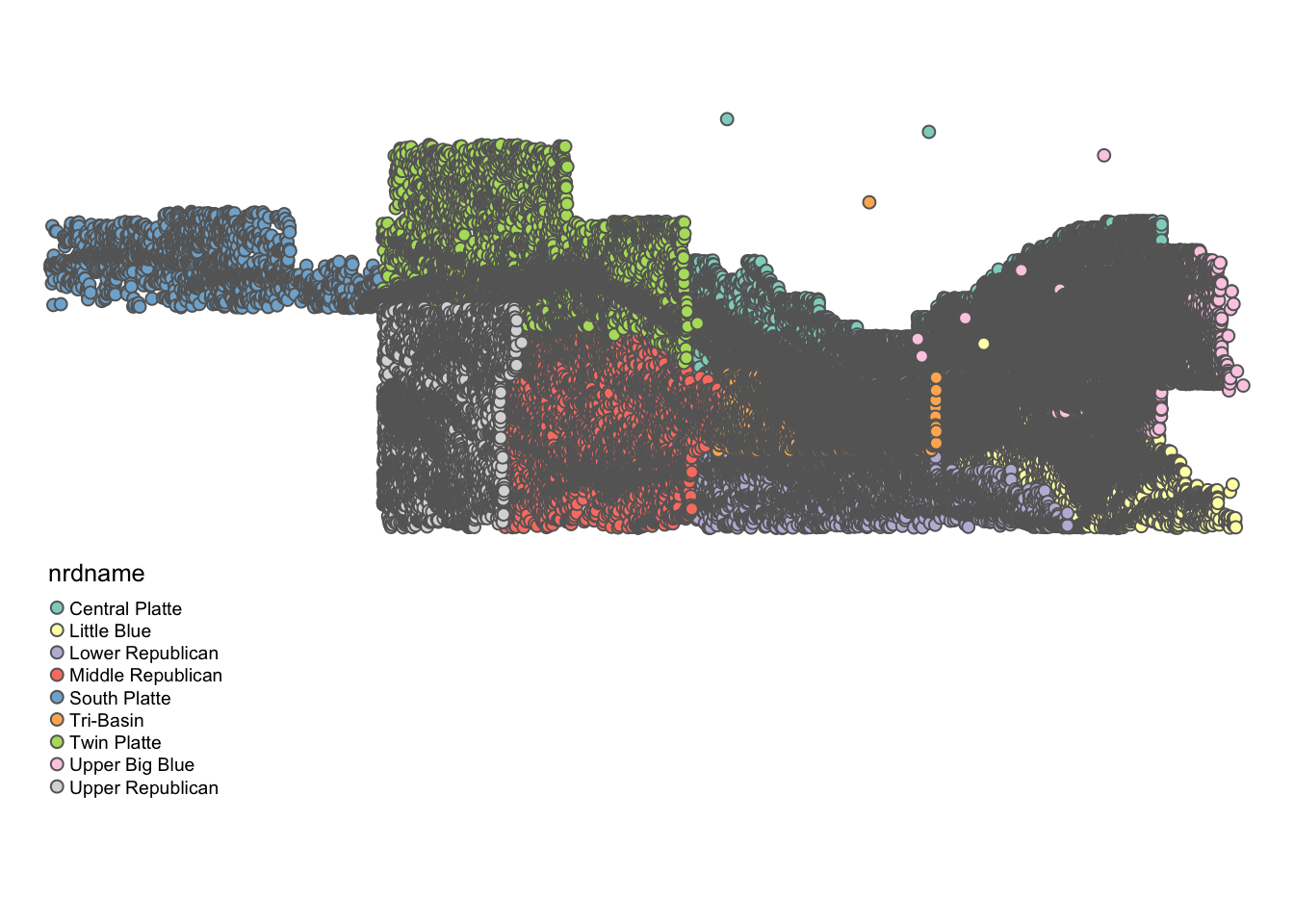
Now, let’s try to get a summary of a variable by group using the group\_by() and summarize() functions.

#--- summary by group ---#  
wells\_by\_nrd <- wells\_sf %>%   
 #--- group by nrdname ---#  
 group\_by(nrdname) %>%   
 #--- summarize ---#  
 summarize(tot\_acres = sum(acres, na.rm = TRUE))  
  
#--- take a look ---#  
wells\_by\_nrd

Simple feature collection with 9 features and 2 fields  
geometry type: MULTIPOINT  
dimension: XY  
bbox: xmin: -104.0531 ymin: 40.00161 xmax: -96.87681 ymax: 41.85942  
CRS: unknown  
# A tibble: 9 x 3  
 nrdname tot\_acres geometry  
 <chr> <dbl> <MULTIPOINT [°]>  
1 Central Plat… 1890918. ((-100.2329 41.14385), (-100.2328 41.05678), (-100.23…  
2 Little Blue 995900. ((-98.72659 40.30463), (-98.72434 40.68021), (-98.724…  
3 Lower Republ… 543079. ((-100.1968 40.32314), (-100.196 40.33553), (-100.195…  
4 Middle Repub… 443472. ((-101.3691 40.1208), (-101.3448 40.64638), (-101.344…  
5 South Platte 216109. ((-104.0531 41.18248), (-104.053 41.19347), (-104.052…  
6 Tri-Basin 847058. ((-100.0927 40.42312), (-100.0904 40.43158), (-100.08…  
7 Twin Platte 452678. ((-102.0557 41.05204), (-102.0556 41.05488), (-102.05…  
8 Upper Big Bl… 1804782. ((-98.83619 40.85932), (-98.81149 40.78093), (-98.549…  
9 Upper Republ… 551906. ((-102.0516 40.24644), (-102.0515 40.6287), (-102.051…

So, we got total acres by NRD as we expected. One interesting change that happened is geometry variable. Each NRD now has multipoint sfg, which is the combination of all the wells (points) located inside the NRD as you can see below.

tm\_shape(wells\_by\_nrd) +   
 tm\_symbols(col = "nrdname", size = 0.2) +  
 tm\_layout(  
 frame = NA,  
 legend.outside = TRUE,  
 legend.outside.position = "bottom"  
 )



This feature is unlikely to be of much use to us. If you would like to drop a geometry column, you can use the st\_drop\_geometry() function:

#--- remove geometry ---#  
wells\_no\_longer\_sf <- st\_drop\_geometry(wells\_by\_nrd)  
  
#--- take a look ---#  
wells\_no\_longer\_sf

# A tibble: 9 x 2  
 nrdname tot\_acres  
\* <chr> <dbl>  
1 Central Platte 1890918.  
2 Little Blue 995900.  
3 Lower Republican 543079.  
4 Middle Republican 443472.  
5 South Platte 216109.  
6 Tri-Basin 847058.  
7 Twin Platte 452678.  
8 Upper Big Blue 1804782.  
9 Upper Republican 551906.

Finally, data.table does not work as well with sf objects as dplyr does.

#--- convert an sf to data.table ---#  
wells\_by\_nrd\_dt <- data.table(wells\_by\_nrd)  
  
#--- take a look ---#  
wells\_by\_nrd\_dt

nrdname tot\_acres geometry  
1: Central Platte 1890918.2 MULTIPOINT ((-100.2329 41.1...,...  
2: Little Blue 995900.3 MULTIPOINT ((-98.72659 40.3...,...  
3: Lower Republican 543079.2 MULTIPOINT ((-100.1968 40.3...,...  
4: Middle Republican 443472.2 MULTIPOINT ((-101.3691 40.1...,...  
5: South Platte 216109.0 MULTIPOINT ((-104.0531 41.1...,...  
6: Tri-Basin 847058.4 MULTIPOINT ((-100.0927 40.4...,...  
7: Twin Platte 452677.6 MULTIPOINT ((-102.0557 41.0...,...  
8: Upper Big Blue 1804781.5 MULTIPOINT ((-98.83619 40.8...,...  
9: Upper Republican 551906.2 MULTIPOINT ((-102.0516 40.2...,...

#--- check the class ---#  
class(wells\_by\_nrd\_dt)

[1] "data.table" "data.frame"

You see that wells\_by\_nrd\_dt is no longer an sf object even though geometry still remains in the data. If you try to run sf operations on it, it will of course give you an error. Like this:

st\_buffer(wells\_by\_nrd\_dt, dist = 2)

Error in UseMethod("st\_buffer"): no applicable method for 'st\_buffer' applied to an object of class "c('data.table', 'data.frame')"

But, it is easy to revert it back to an sf object again by using the st\_as\_sf() function[55](#fn55).

wells\_by\_nrd\_sf\_again <- st\_as\_sf(wells\_by\_nrd\_dt)

So, this means that if you need fast data transformation, you can first turn an sf to a data.table, transform the data using the data.table functionality, and then revert back to sf[56](#fn56). However, for most economists, the geometry variable itself is not of interest in the sense that it never enters econometric models. For most of us, the geographic information contained in the geometry variable is just a glue to tie two datasets together by geographic referencing. Once we get values of spatial variables of interest, then there is no point in keeping your data an sf object. Personally, whenever I no longer need to carry around the geometry variable, I immediately turn an sf object into a data.table for fast data transformation especially when the data is large.

## 2.6 Turning a data.frame of points into an sf

Often times, you have a dataset with geographic coordinates as variables in a csv or other formats, which would not be recognized as a spatial dataset by R immediately when it is read into R. In this case, you need to identify which variables represent the geographic coordinates from the data set, and create an sf yourself. Fortunately, it is easy to do so using the st\_as\_sf() function.

#--- read well registration data ---#  
wells <- readRDS('./Data/registration.rds')   
  
#--- recognize it as an sf ---#  
wells\_sf <- st\_as\_sf(wells, coords = c("longdd","latdd"))  
  
#--- take a look at the data ---#  
head(wells\_sf[,1:5])

Simple feature collection with 6 features and 5 fields  
geometry type: POINT  
dimension: XY  
bbox: xmin: -102.6249 ymin: 40.69824 xmax: -99.58401 ymax: 41.11699  
CRS: NA  
 wellid long\_utm lat\_utm useid ownerid geometry  
1 2 450660.7 4505424 I 106106 POINT (-99.58401 40.69825)  
2 3 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
3 4 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
4 5 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
5 6 447157.4 4509271 I 15837 POINT (-99.6258 40.73268)  
6 7 445515.0 4509168 I 90248 POINT (-99.64524 40.73164)

Note that the CRS of wells\_sf is NA. Obviously, \(R\) does not know the reference system without you telling it. We know[57](#fn57) that the geographic coordinates in the wells data is NAD 83 (\(epsg=4269\)) for this dataset. So, we can assign the right CRS using either st\_set\_crs() or st\_crs().

#--- set CRS ---#  
wells\_sf <- st\_set\_crs(wells\_sf, 4269)   
  
#--- or this ---#  
st\_crs(wells\_sf) <- 4269  
  
#--- see the change ---#  
head(wells\_sf[,1:5])

Simple feature collection with 6 features and 5 fields  
geometry type: POINT  
dimension: XY  
bbox: xmin: -102.6249 ymin: 40.69824 xmax: -99.58401 ymax: 41.11699  
CRS: EPSG:4269  
 wellid long\_utm lat\_utm useid ownerid geometry  
1 2 450660.7 4505424 I 106106 POINT (-99.58401 40.69825)  
2 3 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
3 4 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
4 5 195648.1 4558080 I 14133 POINT (-102.6249 41.11699)  
5 6 447157.4 4509271 I 15837 POINT (-99.6258 40.73268)  
6 7 445515.0 4509168 I 90248 POINT (-99.64524 40.73164)

## 2.7 Conversion to and from sp

Though unlikely, you may find instances where sp objects are necessary or desirable.[58](#fn58) In that case, it is good to know how to convert an sf object to an sp object, vice versa. You can convert an sf object to its sp counterpart using as(sf\_object, "Spatial"):

#--- conversion ---#  
wells\_sp <- as(wells\_sf,"Spatial")  
  
#--- check the class of NE\_sp ---#  
class(wells\_sp)

[1] "SpatialPointsDataFrame"  
attr(,"package")  
[1] "sp"

As you can see NE\_sp is a class of SpatialPointsDataFrame, polygons with data frame supported by the sp package. The above syntax works for converting an sf of polygons into SpatialPolygonsDataFrame as well[59](#fn59).

You can revert NE\_sp back to an sf object using the st\_as\_sf() function, as follows:

#--- revert back to sf ---#  
wells\_sf <- st\_as\_sf(wells\_sp)  
  
#--- check the class ---#  
class(wells\_sf)

[1] "sf" "data.frame"

We do not cover how to use the sp package as the benefit of learning it has become marginal compared to when sf was just introduced a few years back[60](#fn60).

## 2.8 Geometrical operations

There are various geometrical operations that are particularly useful for economists. Here, some of the most commonly used geometrical operations are introduced[61](#fn61). You can see the practical use of some of these functions in Chapter [1](demo.html#demo).

### 2.8.1 st\_buffer

st\_buffer() creates a buffer around points, lines, or the border of polygons.

Let’s create buffers around points. First, we read well locations data.

#--- read wells location data ---#  
urnrd\_wells\_sf <- readRDS("./Data/urnrd\_wells.rds") %>%   
 #--- project to UTM 14N WGS 84 ---#  
 st\_transform(32614)

Here is the spatial distribution of the wells (Figure [2.2](vector-basics.html#fig:urnrd-wells)).

tm\_shape(urnrd\_wells\_sf) +  
 tm\_symbols(col = "red", size = 0.1) +  
 tm\_layout(frame = NA)

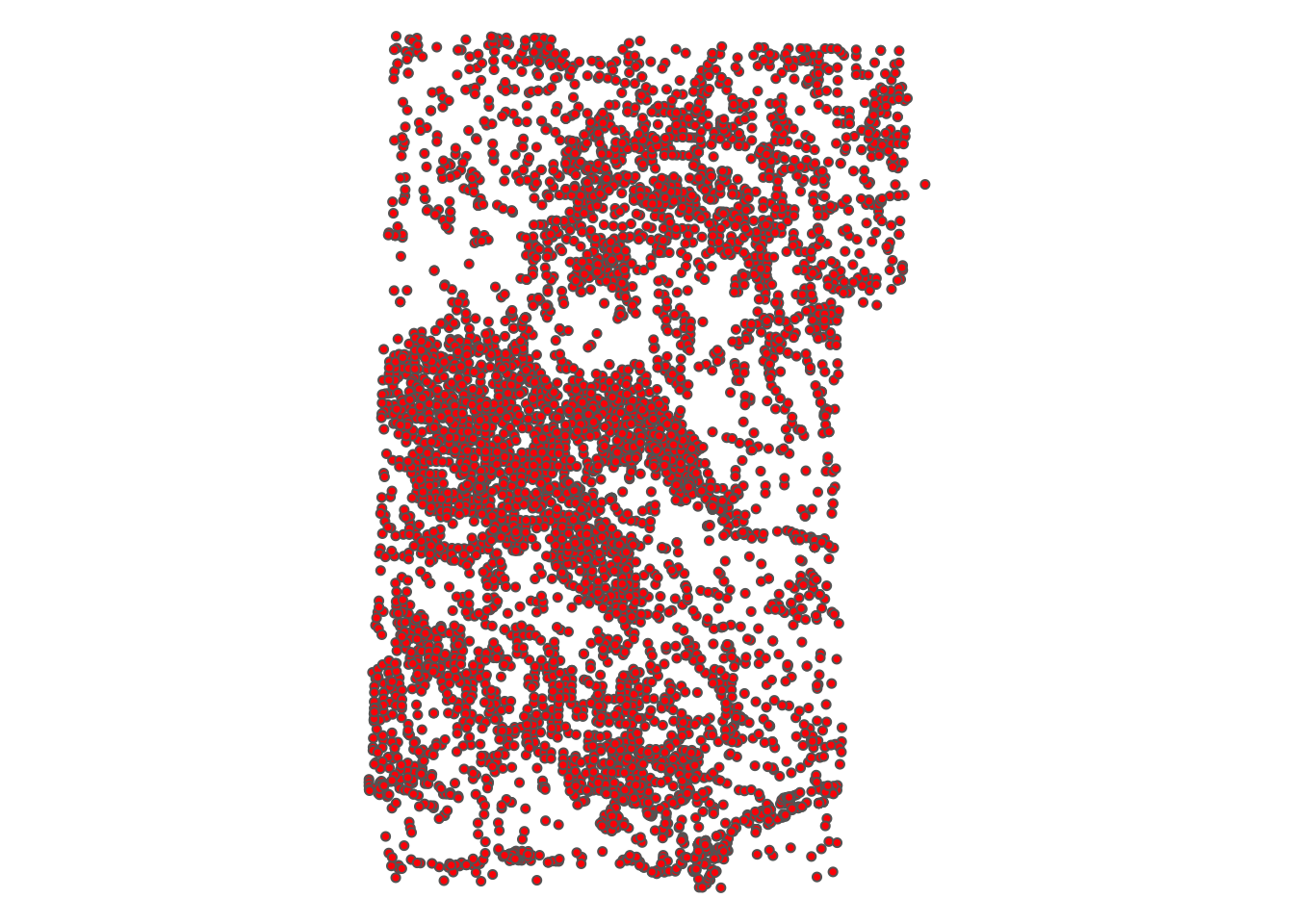


Figure 2.2: Map of the wells

Let’s create buffers around the wells.

#--- create a one-mile buffer around the wells ---#  
wells\_buffer <- st\_buffer(urnrd\_wells\_sf, dist = 1600)

As you can see, you see bunch of circles around wells with the radius of \(1,600\) meters (Figure [2.3](vector-basics.html#fig:buffer-points-map)).

tm\_shape(wells\_buffer) +  
 tm\_polygons(alpha = 0) +  
tm\_shape(urnrd\_wells\_sf) +  
 tm\_symbols(col = "red", size = 0.1) +  
 tm\_layout(frame = NA)

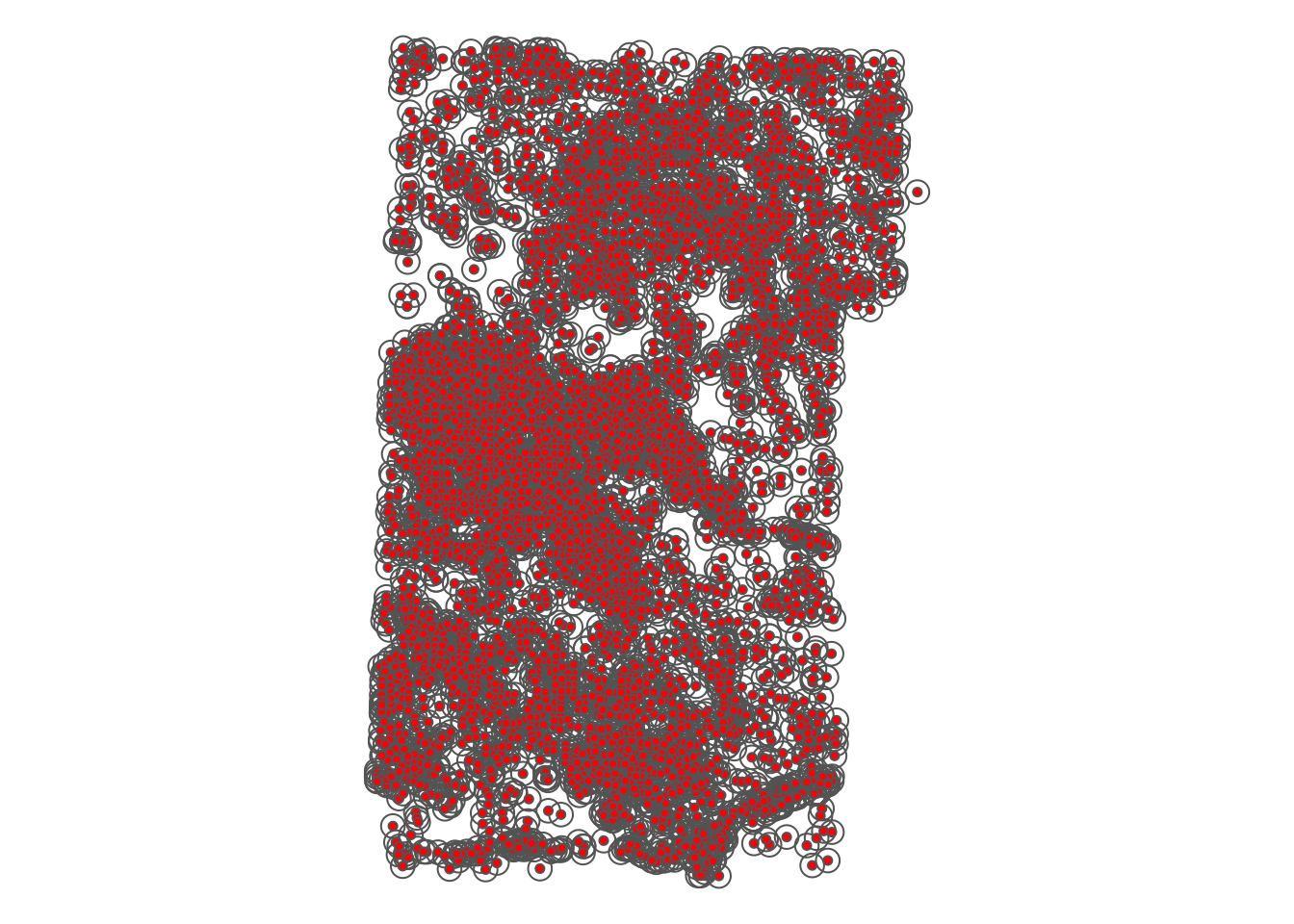


Figure 2.3: Buffers around wells

A practical application of buffer creation will be seen in Chapter [1.1](demo.html#Demo1).

We now create buffers around polygons. First, read NE county boundary data and select three counties (Chase, Dundy, and Perkins).

Here is what they look like (Figure [2.4](vector-basics.html#fig:map-three-counties)):

tm\_shape(NE\_counties) +  
 tm\_polygons('NAME', palette="RdYlGn", contrast=.3, title="County") +  
 tm\_layout(frame = NA)

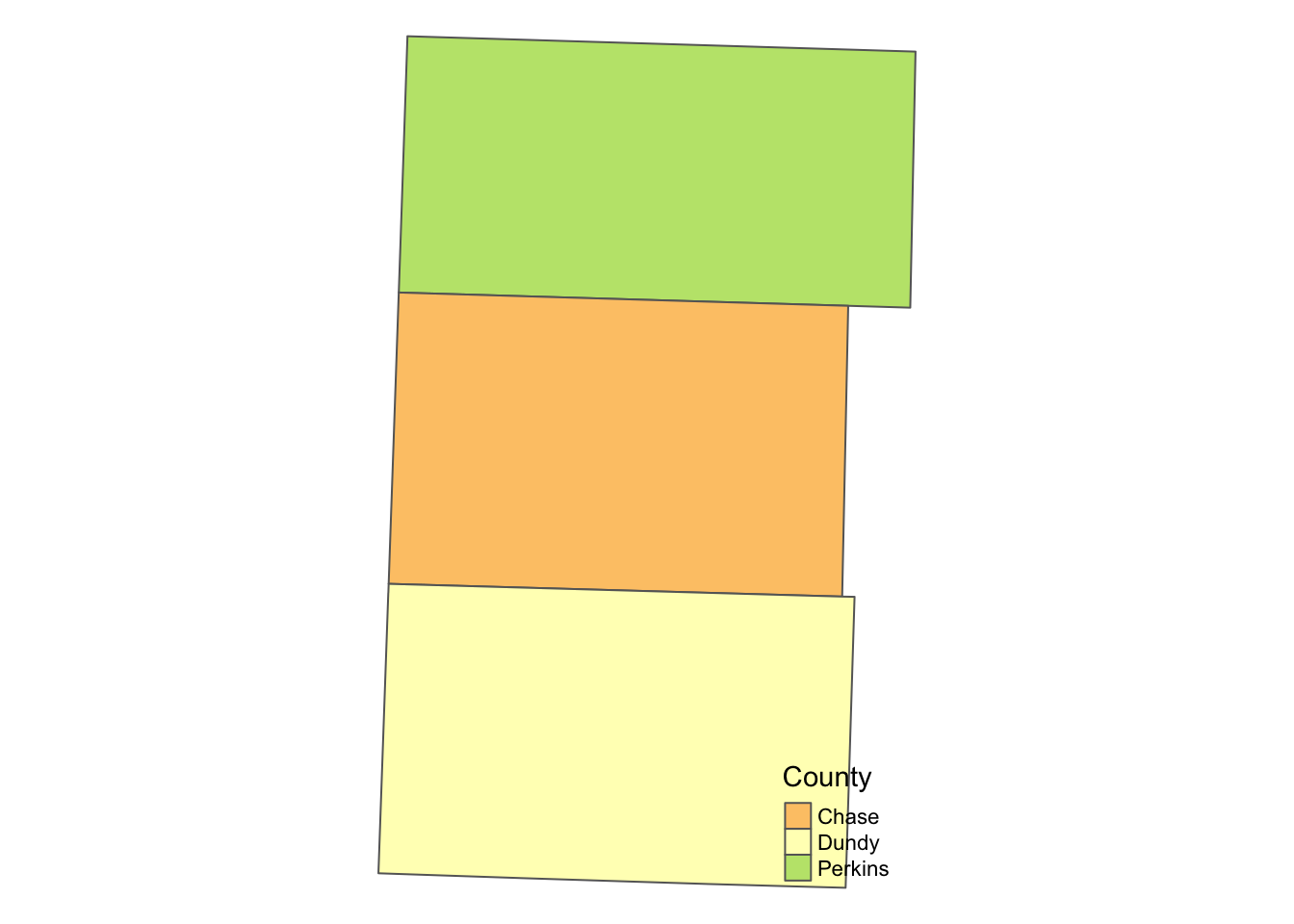


Figure 2.4: Map of the three counties

The following code creates buffers around polygons (see the results in Figure [2.5](vector-basics.html#fig:buffer-county)):

NE\_buffer <- st\_buffer(NE\_counties, dist = 2000)

tm\_shape(NE\_buffer) +  
 tm\_polygons(col='blue',alpha=0.2) +  
tm\_shape(NE\_counties) +  
 tm\_polygons('NAME', palette="RdYlGn", contrast=.3, title="County") +   
 tm\_layout(  
 legend.outside=TRUE,  
 frame=FALSE  
 )

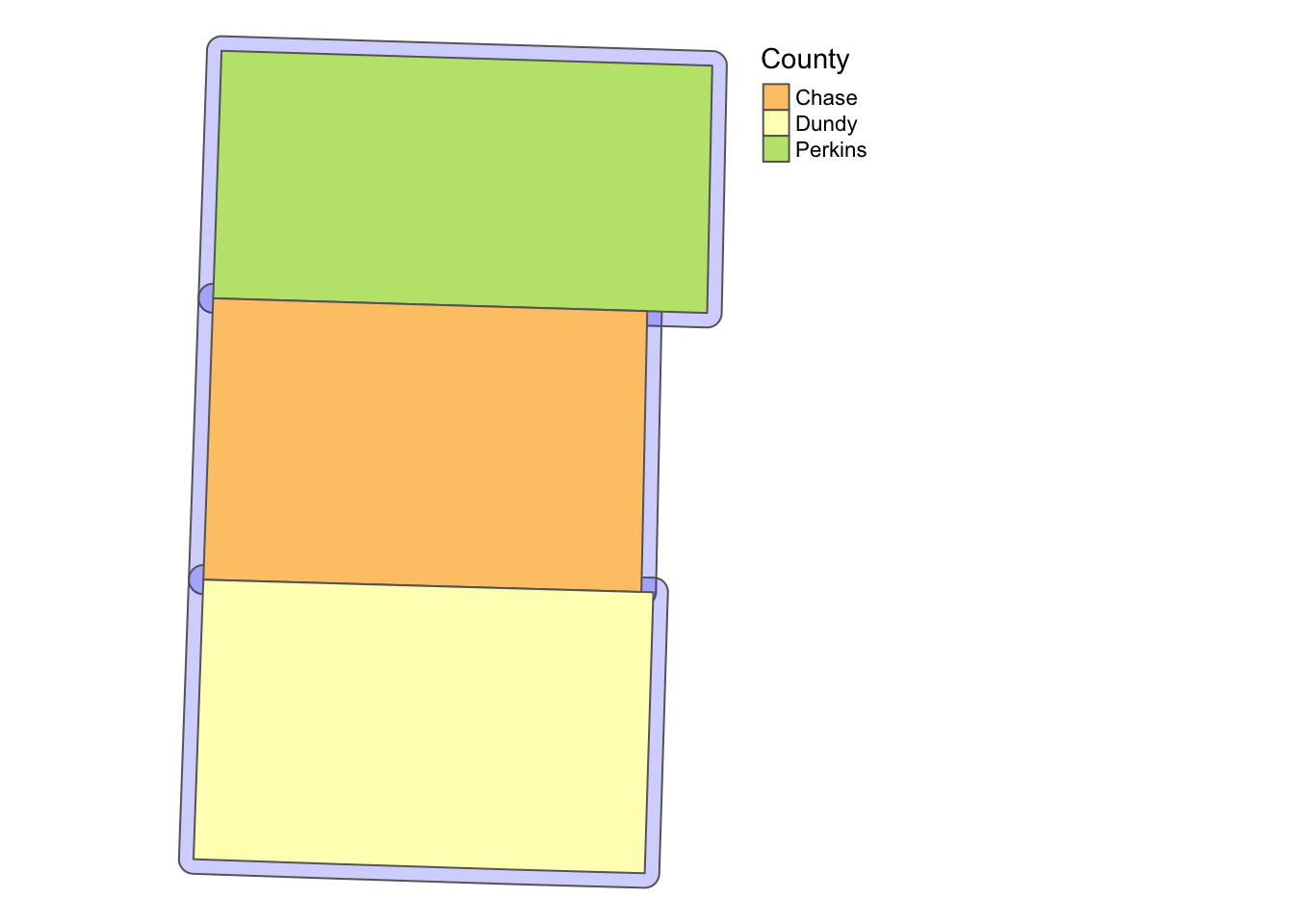


Figure 2.5: Buffers around the three counties

For example, this can use useful to identify observations are close enough to the border of political boundaries when you want to take advantage of spatial discontinuity of policies across adjacent political boundaries.

### 2.8.2 st\_area

The st\_area() function calculates the area of polygons.

#--- generate area by polygon ---#  
(  
 NE\_counties <- mutate(NE\_counties, area = st\_area(NE\_counties))  
)

Simple feature collection with 3 features and 10 fields  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: 239494.1 ymin: 4430632 xmax: 310778.1 ymax: 4543676  
CRS: EPSG:32614  
 STATEFP COUNTYFP COUNTYNS AFFGEOID GEOID NAME LSAD ALAND  
1 31 135 00835889 0500000US31135 31135 Perkins 06 2287828025  
2 31 029 00835836 0500000US31029 31029 Chase 06 2316533447  
3 31 057 00835850 0500000US31057 31057 Dundy 06 2381956151  
 AWATER geometry area  
1 2840176 MULTIPOLYGON (((243340.2 45... 2302174854 [m^2]  
2 7978172 MULTIPOLYGON (((241201.4 44... 2316908196 [m^2]  
3 3046331 MULTIPOLYGON (((240811.3 44... 2389890531 [m^2]

Now, as you can see below, the default class of the results of st\_area() is units, which does not accept numerical operations.

class(NE\_counties$area)

[1] "units"

So, let’s turn it into double.

(  
NE\_counties <- mutate(NE\_counties, area = as.numeric(area))  
)

Simple feature collection with 3 features and 10 fields  
geometry type: MULTIPOLYGON  
dimension: XY  
bbox: xmin: 239494.1 ymin: 4430632 xmax: 310778.1 ymax: 4543676  
CRS: EPSG:32614  
 STATEFP COUNTYFP COUNTYNS AFFGEOID GEOID NAME LSAD ALAND  
1 31 135 00835889 0500000US31135 31135 Perkins 06 2287828025  
2 31 029 00835836 0500000US31029 31029 Chase 06 2316533447  
3 31 057 00835850 0500000US31057 31057 Dundy 06 2381956151  
 AWATER geometry area  
1 2840176 MULTIPOLYGON (((243340.2 45... 2302174854  
2 7978172 MULTIPOLYGON (((241201.4 44... 2316908196  
3 3046331 MULTIPOLYGON (((240811.3 44... 2389890531

st\_area() is useful when you want to find area-weighted average of characteristics after spatially joining two polygon layers using the st\_intersection() function (See Chapter [3.3.3](spatial-interactions-of-vector-data-subsetting-and-joining.html#polygon-polygon)).

### 2.8.3 st\_centroid

The st\_centroid() function finds the centroid of each polygon.

#--- create centroids ---#  
(  
NE\_centroids <- st\_centroid(NE\_counties)  
)

Simple feature collection with 3 features and 10 fields  
geometry type: POINT  
dimension: XY  
bbox: xmin: 271156.7 ymin: 4450826 xmax: 276594.1 ymax: 4525635  
CRS: EPSG:32614  
 STATEFP COUNTYFP COUNTYNS AFFGEOID GEOID NAME LSAD ALAND  
1 31 135 00835889 0500000US31135 31135 Perkins 06 2287828025  
2 31 029 00835836 0500000US31029 31029 Chase 06 2316533447  
3 31 057 00835850 0500000US31057 31057 Dundy 06 2381956151  
 AWATER geometry area  
1 2840176 POINT (276594.1 4525635) 2302174854  
2 7978172 POINT (271469.9 4489429) 2316908196  
3 3046331 POINT (271156.7 4450826) 2389890531

Here’s what the map of the output (Figure [2.6](vector-basics.html#fig:map-centroids)).

tm\_shape(NE\_counties) +  
 tm\_polygons() +  
tm\_shape(NE\_centroids)+  
 tm\_symbols(size=0.5) +  
tm\_layout(  
 legend.outside=TRUE,  
 frame=FALSE  
 )

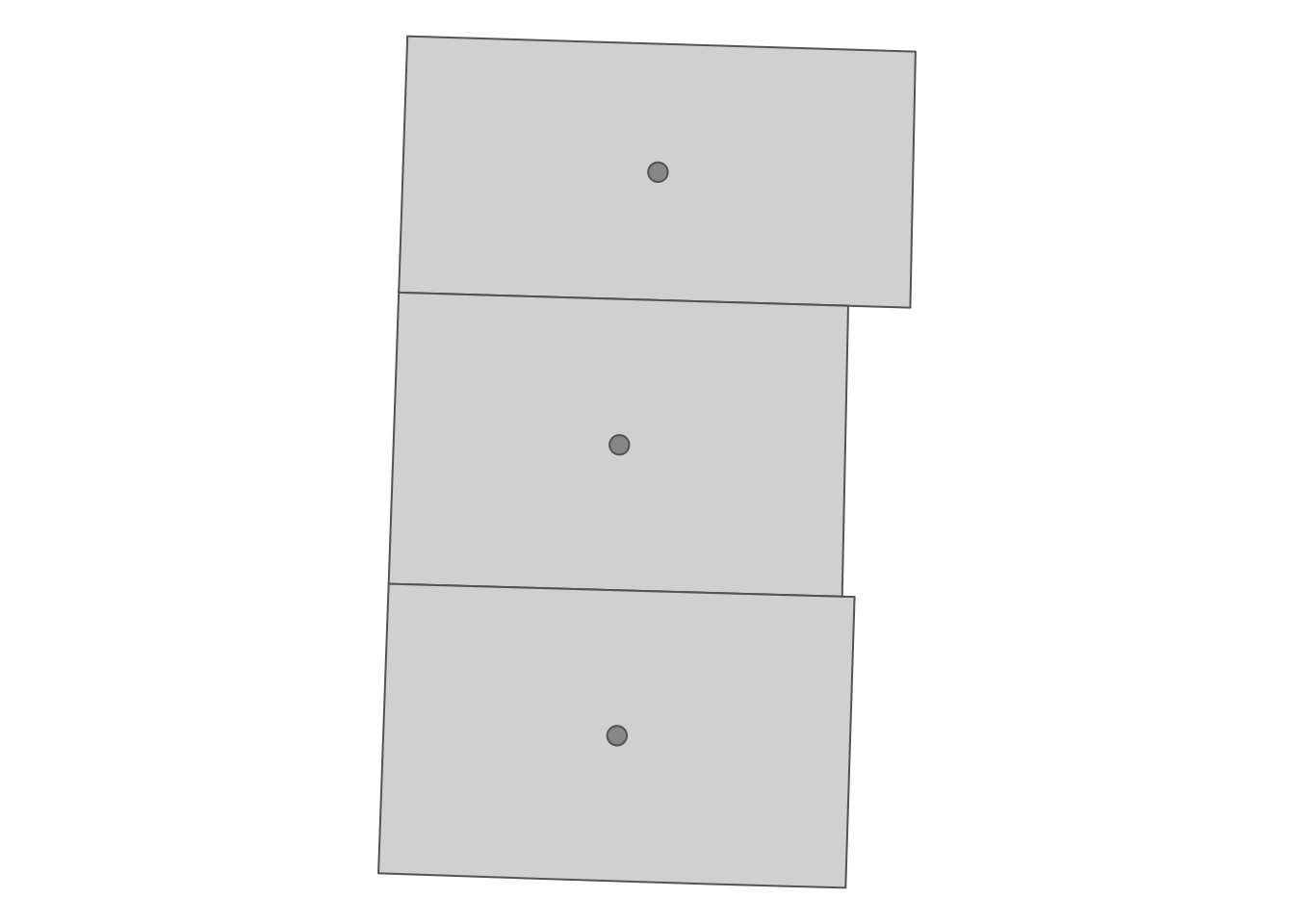


Figure 2.6: The centroids of the polygons

It can be useful when creating a map with labels because the centroid of polygons tend to be a good place to place labels at like this (Figure [2.7](vector-basics.html#fig:cent-label)).[62](#fn62)

tm\_shape(NE\_counties) +  
 tm\_polygons() +  
tm\_shape(NE\_centroids)+  
 tm\_text("NAME") +  
tm\_layout(  
 legend.outside=TRUE,  
 frame=FALSE  
 )

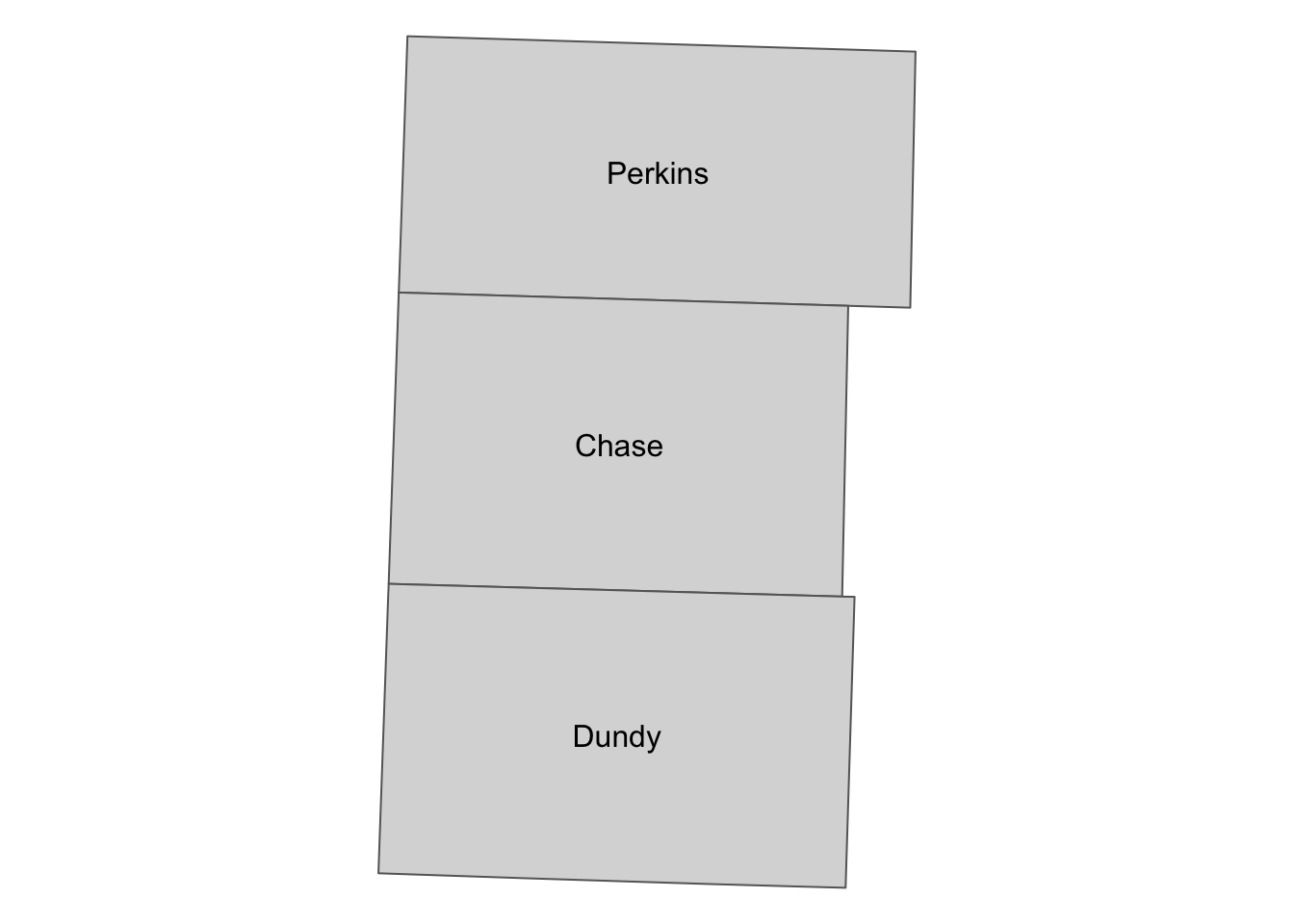


Figure 2.7: County names placed at the centroids of the counties

It may be also useful when you somehow need to calculate the “distance” between polygons.

### 2.8.4 st\_length

We can use st\_length() to calculate great circle distances[63](#fn63) of LINESTRING and MULTILINESTRING when they are represented in geodetic coordinates. On the other hand, if they are projected and use a Cartesian coordinate system, it will calculate Euclidean distance. We use U.S. railroad data for a demonstration.

#--- import US railroad data and take only the first 10 of it ---#  
(  
a\_railroad <- rail\_roads <- st\_read(dsn = "./Data/", layer = "tl\_2015\_us\_rails")[1:10, ]  
)

Reading layer `tl\_2015\_us\_rails' from data source `/Users/tmieno2/Box/Teaching/AAEA R/GIS/Data' using driver `ESRI Shapefile'  
Simple feature collection with 180958 features and 3 fields  
geometry type: MULTILINESTRING  
dimension: XY  
bbox: xmin: -165.4011 ymin: 17.95174 xmax: -65.74931 ymax: 65.00006  
CRS: 4269

Simple feature collection with 10 features and 3 fields  
geometry type: MULTILINESTRING  
dimension: XY  
bbox: xmin: -79.74031 ymin: 35.0571 xmax: -79.2377 ymax: 35.51776  
CRS: 4269  
 LINEARID FULLNAME MTFCC geometry  
1 11020239500 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.47058...  
2 11020239501 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.46687...  
3 11020239502 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.66819...  
4 11020239503 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.46687...  
5 11020239504 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.74031...  
6 11020239575 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.43695...  
7 11020239576 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.47852...  
8 11020239577 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.43695...  
9 11020239589 Aberdeen and Rockfish RR R1011 MULTILINESTRING ((-79.38736...  
10 11020239591 Aberdeen and Briar Patch RR R1011 MULTILINESTRING ((-79.53848...

#--- check CRS ---#  
st\_crs(a\_railroad)

Coordinate Reference System:  
 User input: 4269   
 wkt:  
GEOGCS["NAD83",  
 DATUM["North\_American\_Datum\_1983",  
 SPHEROID["GRS 1980",6378137,298.257222101,  
 AUTHORITY["EPSG","7019"]],  
 TOWGS84[0,0,0,0,0,0,0],  
 AUTHORITY["EPSG","6269"]],  
 PRIMEM["Greenwich",0,  
 AUTHORITY["EPSG","8901"]],  
 UNIT["degree",0.0174532925199433,  
 AUTHORITY["EPSG","9122"]],  
 AUTHORITY["EPSG","4269"]]

It uses geodetic coordinate system. Let’s calculate the great circle distance of the lines (Chapter [1.4](demo.html#demo4) for a practical use case of this function).

(  
a\_railroad <- mutate(a\_railroad, length = st\_length(a\_railroad))  
)

Simple feature collection with 10 features and 4 fields  
geometry type: MULTILINESTRING  
dimension: XY  
bbox: xmin: -79.74031 ymin: 35.0571 xmax: -79.2377 ymax: 35.51776  
CRS: 4269  
 LINEARID FULLNAME MTFCC geometry  
1 11020239500 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.47058...  
2 11020239501 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.46687...  
3 11020239502 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.66819...  
4 11020239503 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.46687...  
5 11020239504 Norfolk Southern Rlwy R1011 MULTILINESTRING ((-79.74031...  
6 11020239575 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.43695...  
7 11020239576 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.47852...  
8 11020239577 Seaboard Coast Line RR R1011 MULTILINESTRING ((-79.43695...  
9 11020239589 Aberdeen and Rockfish RR R1011 MULTILINESTRING ((-79.38736...  
10 11020239591 Aberdeen and Briar Patch RR R1011 MULTILINESTRING ((-79.53848...  
 length  
1 661.3381 [m]  
2 657.4261 [m]  
3 19982.5998 [m]  
4 13888.3385 [m]  
5 7194.7745 [m]  
6 1061.2335 [m]  
7 7824.0945 [m]  
8 31756.9803 [m]  
9 4547.1970 [m]  
10 17103.0691 [m]

### References

Pebesma, Edzer. 2018. “Simple Features for R: Standardized Support for Spatial Vector Data.” *R Journal* 10 (1).

* yes, it is the same as the package name[↩](vector-basics.html#fnref36)
* except we learn how to convert back and forth between sf object ans sp objects just in case you need sp objects.[↩](vector-basics.html#fnref37)
* There are cases where sp is faster completing the same task than sf. For example, see the answer to [this question](https://gis.stackexchange.com/questions/324952/spover-vs-sfst-intersection-in-r). But, the I doubt the difference between the two is practically negligible even with a bigger data than the test data.[↩](vector-basics.html#fnref38)
* just like a regular list object can contain mixed types of elements: numeric, character, etc[↩](vector-basics.html#fnref39)
* I must say that creating spatial objects from scratch yourself is an unnecessary skill for many of us as an economist. But, it is still good to know the underlying structure of the data. Also, occasionally the need arises. For example, I had to construct spatial objects from scratch when I designed on-farm randomized nitrogen trials. In such cases, it is of course necessary to understand how different types of sfg are constructed, create sfc from a collection of sfgs, and then create an sf from an sfc.[↩](vector-basics.html#fnref40)
* You will hardly see the other geometry types: MULTIPOINT and GEOMETRYCOLLECTION. You may see GEOMETRYCOLLECTION after intersecting two spatial objects. You can see [here](https://r-spatial.github.io/sf/articles/sf1.html#sfg-simple-feature-geometry-1) if you are interested in learning what they are.[↩](vector-basics.html#fnref41)
* or three to represent a point in the three-dimensional space[↩](vector-basics.html#fnref42)
* we will learn how to make sfg objects from scratch because it helps to better understand how the spatial data is stored.[↩](vector-basics.html#fnref43)
* See here for other various formats spatial data are stored.[↩](vector-basics.html#fnref44)
* When storing a spatial dataset, ArcGIS divides the information into separate files. All of them have the same prefix, but have different extensions. We typically say we read a shapefile, but we really are importing all these files including the shapefile with the .shp extension. When you read those data, you just refer to the common prefix because you really are importing all the files, not just a .shp file.[↩](vector-basics.html#fnref45)
* Link here[↩](vector-basics.html#fnref46)
* see the last paragraph of [chapter 7.5 of this book](https://csgillespie.github.io/efficientR/data-carpentry.html#data-processing-with-data.table), [this blogpost](https://carto.com/blog/fgdb-gpkg/), or [this](http://switchfromshapefile.org/)[↩](vector-basics.html#fnref47)
* sf versions prior to 0.9 provides CRS information in the form of proj4string. The newer version of sf presents CRS in the form of wtk (see [this slide](https://nowosad.github.io/whyr_webinar004/#25)). You can find the reason behind this change in the same slide, starting from [here](https://nowosad.github.io/whyr_webinar004/#18).[↩](vector-basics.html#fnref48)
* See [here](https://spatialreference.org/ref/epsg/nad27/) for numerous other formats that represent the same CRS.[↩](vector-basics.html#fnref49)
* You can find the CRS-EPSG number correspondence [here](http://spatialreference.org/ref/epsg/).[↩](vector-basics.html#fnref50)
* Potential pool of CRS is infinite. Only the commonly-used CRS have been assigned EPSG SRID.[↩](vector-basics.html#fnref51)
* (see [here](http://spatialreference.org/ref/epsg/nad83-utm-zone-14n/))[↩](vector-basics.html#fnref52)
* e.g., spatial subsetting, joining, etc[↩](vector-basics.html#fnref53)
* In this example, we are using the same data with two different CRS. But, you get the point.[↩](vector-basics.html#fnref54)
* This was not the case before. Turning an sf object to a data.table object used to replace sfg with NA.[↩](vector-basics.html#fnref55)
* Remember that conversions between sp and sf also take time.[↩](vector-basics.html#fnref56)
* Yes, YOU need to know the CRS of your data.[↩](vector-basics.html#fnref57)
* For example, those who run spatial econometric methods using spdep, creating neighbors from polygons is a bit faster using sp objects than using sf objects).[↩](vector-basics.html#fnref58)
* The function does not work for an sf object that consists of different geometry types (e.g., POINT and POLYGON). This is because sp objects do not allow to have different types of geometries in the single sp object. For example, SpatialPointsDataFrame consists only of points data.[↩](vector-basics.html#fnref59)
* For those interested in learning the sp package, [this website](https://rspatial.org/) is a good resource.[↩](vector-basics.html#fnref60)
* For the complete list of available geometrical operations under the sf package, see [here](https://cran.r-project.org/web/packages/sf/vignettes/sf1.html).[↩](vector-basics.html#fnref61)
* When creating maps with the ggplot2 package, you can use geom\_sf\_text() or geom\_sf\_label(), which automatically finds where to put texts. See some examples [here](https://yutani.rbind.io/post/geom-sf-text-and-geom-sf-label-are-coming/).[↩](vector-basics.html#fnref62)
* Great circle distance is the shortest distance between two points on the surface of a sphere (earth)[↩](vector-basics.html#fnref63)