Geographic Data and Mapping*

Chapter 7

Anna Ziff

R Workflow for Economists

Contents

*Please contact anna.ziff@duke.edu if there are errors.

Here are the libraries you will need for this chapter.

```
# Geographic packages
library(sf)
library(spData)
library(raster)
library(tmap)

# Optional! Only if you are interested in rasters
# Install with remotes::install_github("Nowosad/spDataLarge")
library(spDataLarge)

# Other packages
library(dplyr)
library(ggplot2)
library(stringr)
library(tidyr)
```

Geographic Data Vocabulary

Geographic datasets have particular features that are useful to understand before using them for analysis and graphing. There are two models of geographic data: the vector data model and the raster data model. The vector data model uses points, lines, and polygons to represent geographic areas. The raster data model uses equally-sized cells to represent geographic areas. Vector data is usually adequate for maps in economics, as it is able to represent human-defined areas precisely. That being said, raster data is needed for some contexts (e.g., environmental studies), and can provide richness and context to maps.

Vector Data Model

Vector data require a coordinate reference system (CRS). There are many different options for the CRS, with different countries and regions using their own systems. The differences between the different systems include the reference point (where is $(0, 0)$ located) and the units of the distances (e.g., km, degrees).

The package `sf` contains functions to handle different types of vector data. The trio of packages `sp`, `rgdal`, and `rgeos` used to be the go-to packages for vector data in R, but have been superseded by `sf`. It has efficiency advantages and the data can be accessed more conveniently than in the other packages. You may still see examples and StackExchange forums with these other packages, however. If needed, the following code can be used to convert an `sf` object to a Spatial object used in the package `sp`, and back to an `sf` object.

```
library(sp)
example_sp <- as(example, Class = "Spatial")
example_sf <- st_as_sf(example_sp)
```

Manage Geographic Data

Geographic data in R are stored in a data frame (or tibble) with a column for geographic data. These are called spatial data frames (or sf object). This column is called `geom` or `geometry`. Here is an example with the `world` dataset from `spData`.

```
data("world")
names(world)
```

```
##  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
##  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap" "geom"
```

The last variable, `geom`, is a list column. If you inspect the column using `View(world$geom)`, you will see that each element is a list of varying lengths, corresponding to the vector data of the country. It is possible to interact with this sf object as one would a non-geographic tibble or data frame.

```
summary(world)
```

```
##     iso_a2           name_long          continent          region_un
##  Length:177         Length:177         Length:177         Length:177
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##   subregion             type              area_km2             pop
##  Length:177         Length:177         Min.   :    2417   Min.   :5.630e+04
##  Class :character   Class :character   1st Qu.:   46185   1st Qu.:3.755e+06
##  Mode  :character   Mode  :character   Median :  185004   Median :1.040e+07
##                                        Mean   :  832558   Mean   :4.282e+07
##                                        3rd Qu.:  621860   3rd Qu.:3.075e+07
##                                        Max.   :17018507   Max.   :1.364e+09
##                                                           NA's   :10
##     lifeExp        gdpPercap                    geom
##  Min.   :50.62   Min.   :   597.1   MULTIPOLYGON :177
##  1st Qu.:64.96   1st Qu.:  3752.4   epsg:4326    :  0
##  Median :72.87   Median : 10734.1   +proj=long...:  0
##  Mean   :70.85   Mean   : 17106.0
##  3rd Qu.:76.78   3rd Qu.: 24232.7
##  Max.   :83.59   Max.   :120860.1
##  NA's   :10      NA's   :17
```

```
attributes(world)
```

```
## $names
##  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
##  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap" "geom"
##
## $row.names
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
## [127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
## [145] 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
## [163] 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
##
## $sf_column
## [1] "geom"
##
## $agr
##    iso_a2 name_long continent region_un subregion      type  area_km2       pop
##      <NA>      <NA>      <NA>      <NA>      <NA>      <NA>      <NA>      <NA>
```

```
##   lifeExp gdpPercap
##      <NA>      <NA>
## Levels: constant aggregate identity
##
## $class
## [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

```
world
```

```
## Simple feature collection with 177 features and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -89.9 xmax: 180 ymax: 83.64513
## Geodetic CRS:  WGS 84
## # A tibble: 177 x 11
##    iso_a2 name_long continent region_un subregion type  area_km2     pop lifeExp
##  * <chr>  <chr>     <chr>     <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>
##  1 FJ     Fiji      Oceania   Oceania   Melanesia Sove~   1.93e4  8.86e5    70.0
##  2 TZ     Tanzania  Africa    Africa    Eastern ~ Sove~   9.33e5  5.22e7    64.2
##  3 EH     Western ~ Africa    Africa    Northern~ Inde~   9.63e4 NA         NA
##  4 CA     Canada    North Am~ Americas  Northern~ Sove~   1.00e7  3.55e7    82.0
##  5 US     United S~ North Am~ Americas  Northern~ Coun~   9.51e6  3.19e8    78.8
##  6 KZ     Kazakhst~ Asia      Asia      Central ~ Sove~   2.73e6  1.73e7    71.6
##  7 UZ     Uzbekist~ Asia      Asia      Central ~ Sove~   4.61e5  3.08e7    71.0
##  8 PG     Papua Ne~ Oceania   Oceania   Melanesia Sove~   4.65e5  7.76e6    65.2
##  9 ID     Indonesia Asia      Asia      South-Ea~ Sove~   1.82e6  2.55e8    68.9
## 10 AR     Argentina South Am~ Americas  South Am~ Sove~   2.78e6  4.30e7    76.3
## # ... with 167 more rows, and 2 more variables: gdpPercap <dbl>,
## #   geom <MULTIPOLYGON [°]>
```

The attributes of `world` show the variable and row names, the name of the `sf` column, the attribute-geometry-relationship (agr), and the information about the object itself (class). Notably, there are some geographic characteristics attached to the data: the geometry type (most commonly, `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`, `GEOMETRYCOLLECTION`), the dimension, the bbox (limits of the plot), the CRS, and the name of the geographic column.

Load Spatial Data

As seen above, it is possible to load geographic data built-in to R or other packages using `data()`. There are other packages with more complete data, rather than small examples. For example, the package `osmdata` connects you to the OpenStreetMap API, the package `rnoaa` imports NOAA climate data, and `rWBclimate` imports World Bank data. If you need any geographic data that might be collected by a governmental agency, check to see if there is an R package before downloading it. There may be some efficiency advantages to using the package.

You can also import geographic data into R directly from your computer. These will usually be stored in spatial databases, often with file extensions that one would also use in ArcGIS or a related program. The most popular format is ESRI Shapefile (`.shp`). The function to import data is `st_read()`. To see what types of files can be imported with that function, check `st_drivers()`.

```
sf_drivers <- st_drivers()
head(sf_drivers)
```

```
unzip("Major_Streets.zip")
chi_streets <- st_read("Major_Streets.shp")
```

```
## Reading layer `Major_Streets' from data source
```

```
##   `/Users/annaziff/Desktop/Duke/projects/R-Workflow-for-Economists/Data/Major_Streets/Major_Streets.s
##   using driver `ESRI Shapefile'
## Simple feature collection with 16065 features and 65 fields
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: 1093384 ymin: 1813892 xmax: 1205147 ymax: 1951666
## Projected CRS: NAD83 / Illinois East (ftUS)
```

```
head(chi_streets[, 1:10], 3)
```

```
## Simple feature collection with 3 features and 10 fields
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: 1163267 ymin: 1887115 xmax: 1180175 ymax: 1904103
## Projected CRS: NAD83 / Illinois East (ftUS)
##   OBJECTID FNODE_ TNODE_ LPOLY_ RPOLY_    LENGTH TRANS_ TRANS_ID SOURCE_ID
## 1        1  16580  16726      0      0 519.31945  52842   115733     15741
## 2        2  18237  18363      0      0 432.44697  45410   149406     49489
## 3        3  12874  12840      0      0  80.97939  23831   149515     49599
##   OLD_TRANS_                       geometry
## 1     118419 LINESTRING (1163267 1893234...
## 2     160784 LINESTRING (1176733 1887541...
## 3     152590 LINESTRING (1180163 1904023...
```

See `st_write()` to output a shapefile, or other type of spatial data file. The function `saveRDS()` is very useful here as well, as it compresses the files.

Attribute Data

The non-spatial (or attribute) data can be treated as you would treat any other data frame or tibble. You can use the `tidyverse` or built-in functions to clean, summarize, and otherwise manage the attribute data.

There is are a few details that can prevent frustration.

1. Both `raster` and `dplyr` have the function `select()`. If you have `raster` loaded, make sure you are specifying `dplyr::select()` when you want the `tidyverse` function.
2. If you want to drop the spatial element of the dataset, this can be done with `st_drop_geometry()`. If you are not using the data for its spatial elements, you should drop the geometry as it the list column can take up a lot of memory.

```
world_tib <- st_drop_geometry(world)
names(world_tib)
```

```
##  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
##  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap"
```

CRS

The CRS can be accessed with an ESPG code (`espg`) or a projection (`proj4string`). The ESPG is usually shorter, but less flexible than the analogous projection. To inspect the CRS of an sf object, use the `st_crs()` function.

```
st_crs(world)
```

The CRS can either be geographic (i.e., latitude and longitude with degrees) or projected. Many of the functions used for sf objects assume that there is some CRS, and it may be necessary to set one. Operations involving distances depend heavily on the projection, and may not work with geographic CRSs.

```r
london <- tibble(lon = -0.1, lat = 51.5) %>%
  st_as_sf(coords = c("lon", "lat"))
st_is_longlat(london) # NA means that there is no set CRS
```

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

To set the CRS, use the `st_set_crs()` function.

```r
london <- st_set_crs(london, 4326)
st_is_longlat(london)
```

```
## [1] TRUE
```

```r
st_crs(london)
```

```
## Coordinate Reference System:
##   User input: EPSG:4326
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     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]]
```

If you are changing the CRS rather than setting one, use the `st_transform()` function. This is necessary when comparing two sf objects with different projections (a common occurrence).

```r
london_27700 <- st_transform(london, 27700)
st_distance(london, london_27700)
```

```
## Error in st_distance(london, london_27700): st_crs(x) == st_crs(y) is not TRUE
```

The most common geographic CRS is **WGS84**, or EPSG code 4326. When in doubt, this may be a good place to start. Selecting a projected CRS requires more context of the specific data. Different sources will use different projections. See chapter 6.3 of Lovelace, Nowosad, and Muenchow (2021) and the EPSG repsitory for more information on this.

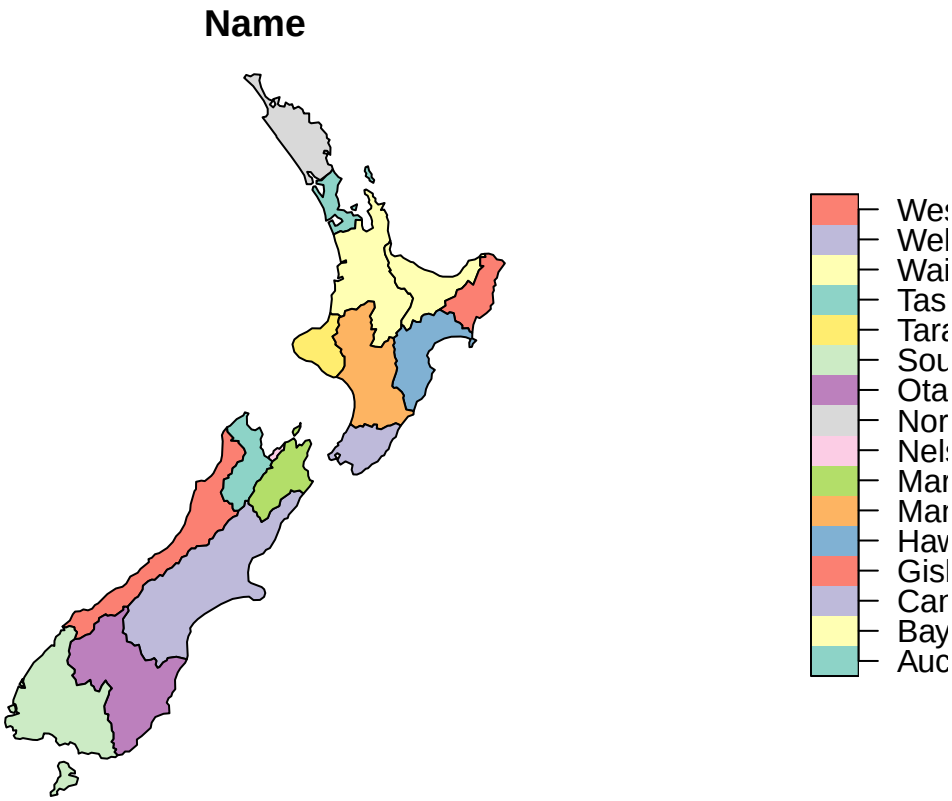
Spatial Operations

Spatial subsetting involves selecting features based on if they relate to other objects. It is analogous to attribute subsetting. As an example, we will use the New Zealand data.

```r
# Map of New Zealand and demographic data
data("nz")
names(nz)
```

```
## [1] "Name"          "Island"        "Land_area"     "Population"
## [5] "Median_income" "Sex_ratio"     "geom"
```

```
plot(nz[1])
```




```
# 101 highest points in new Zealand
data("nz_height")
names(nz_height)
```

```
## [1] "t50_fid"   "elevation" "geometry"
```

Whereas in attribute subsetting, with the general format `x[y, ]`, the `y` would be a logical value, integer, or character string. In spatial subsetting, the `y` is an sf object itself.

```
 canterbury <- nz %>%
  filter(Name == "Canterbury")

canterbury_height <- nz_height[canterbury, ]
```

There are different options for operators for subsetting. Intersects is the default, and is quite general. For example, if the object touches, crosses, or is within, the object will also intersect. Here are some examples of specifying the operator using the `op` argument.

```
nz_height[canterbury, , op = st_intersects]
```

```
## Simple feature collection with 70 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1365809 ymin: 5158491 xmax: 1654899 ymax: 5350463
## Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
## First 10 features:
```

```
##    t50_fid elevation                geometry
## 5  2362630      2749 POINT (1378170 5158491)
## 6  2362814      2822 POINT (1389460 5168749)
## 7  2362817      2778 POINT (1390166 5169466)
## 8  2363991      3004 POINT (1372357 5172729)
## 9  2363993      3114 POINT (1372062 5173236)
## 10 2363994      2882 POINT (1372810 5173419)
## 11 2363995      2796 POINT (1372579 5173989)
## 13 2363997      3070 POINT (1373796 5174144)
## 14 2363998      3061 POINT (1373955 5174231)
## 15 2363999      3077 POINT (1373984 5175228)
```

```
nz_height[canterbury, , op = st_disjoint]
```

```
## Simple feature collection with 31 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1204143 ymin: 5048309 xmax: 1822492 ymax: 5650492
## Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
## First 10 features:
##    t50_fid elevation                geometry
## 1  2353944      2723 POINT (1204143 5049971)
## 2  2354404      2820 POINT (1234725 5048309)
## 3  2354405      2830 POINT (1235915 5048745)
## 4  2369113      3033 POINT (1259702 5076570)
## 12 2363996      2759 POINT (1373264 5175442)
## 25 2364028      2756 POINT (1374183 5177165)
## 26 2364029      2800 POINT (1374469 5176966)
## 27 2364031      2788 POINT (1375422 5177253)
## 46 2364166      2782 POINT (1383006 5181085)
## 47 2364167      2905 POINT (1383486 5181270)
```

```
nz_height[canterbury, , op = st_within]
```

```
## Simple feature collection with 70 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1365809 ymin: 5158491 xmax: 1654899 ymax: 5350463
## Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
## First 10 features:
##    t50_fid elevation                geometry
## 5  2362630      2749 POINT (1378170 5158491)
## 6  2362814      2822 POINT (1389460 5168749)
## 7  2362817      2778 POINT (1390166 5169466)
## 8  2363991      3004 POINT (1372357 5172729)
## 9  2363993      3114 POINT (1372062 5173236)
## 10 2363994      2882 POINT (1372810 5173419)
## 11 2363995      2796 POINT (1372579 5173989)
## 13 2363997      3070 POINT (1373796 5174144)
## 14 2363998      3061 POINT (1373955 5174231)
## 15 2363999      3077 POINT (1373984 5175228)
```

```
nz_height[canterbury, , op = st_touches]
```

```
## Simple feature collection with 0 features and 2 fields
## Bounding box:  xmin: NA ymin: NA xmax: NA ymax: NA
```

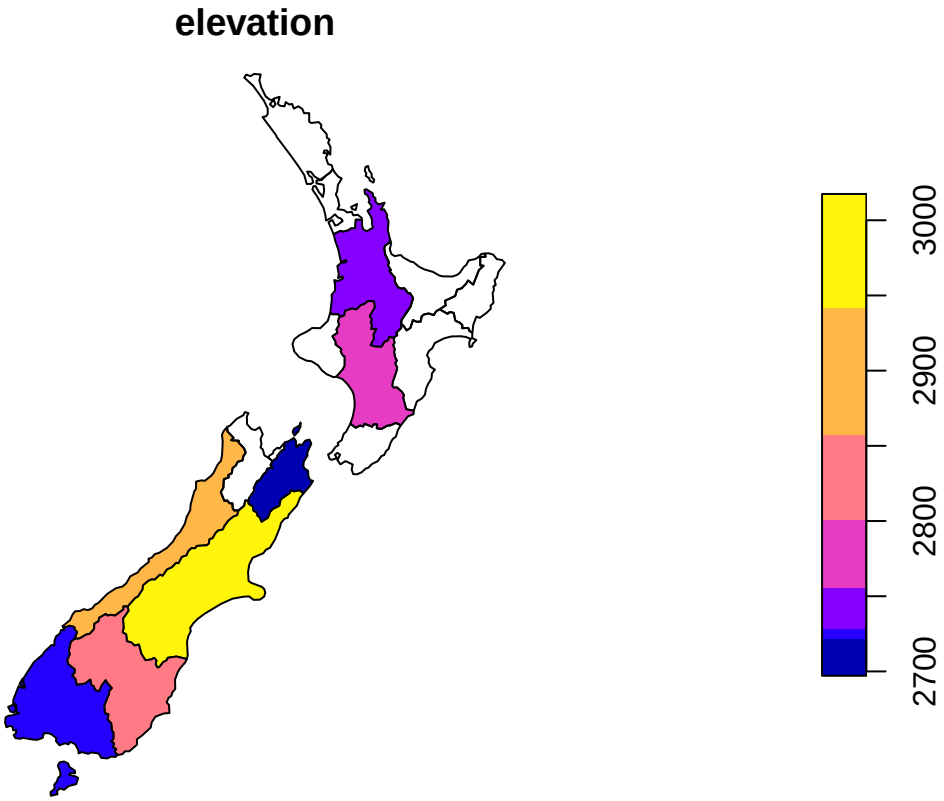
```
## Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
## [1] t50_fid   elevation geometry
## <0 rows> (or 0-length row.names)
```

Grouping for creating summary statistics or other calculations can be done using built-in functions (`aggregate`) or `dplyr` functions. In `aggregate`, the `by` argument is the grouping source and the `x` argument is the target output.

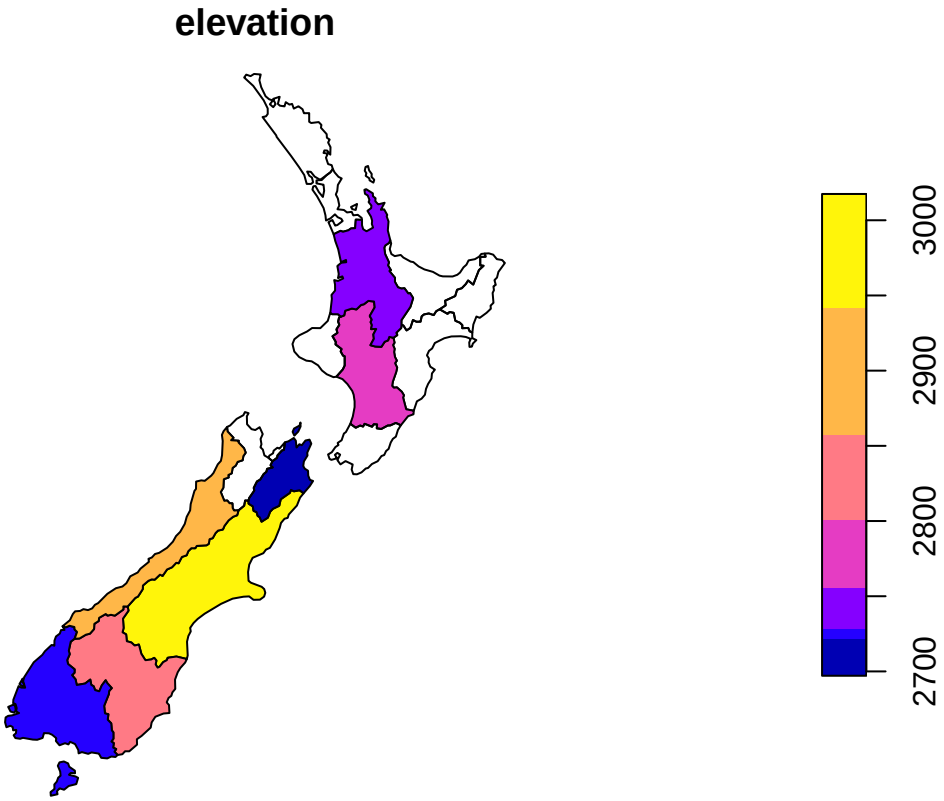
```
# Built-in
nz_avgheight <- aggregate(x = nz_height, by = nz, FUN = mean)
nz_avgheight
```

```
## Simple feature collection with 16 features and 2 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 1090144 ymin: 4748537 xmax: 2089533 ymax: 6191874
## Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
## First 10 features:
##    t50_fid elevation                       geometry
## 1       NA        NA MULTIPOLYGON (((1745493 600...
## 2       NA        NA MULTIPOLYGON (((1803822 590...
## 3  2408405  2734.333 MULTIPOLYGON (((1860345 585...
## 4       NA        NA MULTIPOLYGON (((2049387 583...
## 5       NA        NA MULTIPOLYGON (((2024489 567...
## 6       NA        NA MULTIPOLYGON (((2024489 567...
## 7       NA        NA MULTIPOLYGON (((1740438 571...
## 8  2408394  2777.000 MULTIPOLYGON (((1866732 566...
## 9       NA        NA MULTIPOLYGON (((1881590 548...
## 10 2368390  2889.455 MULTIPOLYGON (((1557042 531...
```

```
plot(nz_avgheight["elevation"])
```




```
# dplyr
nz_avgheight <- nz %>%
  st_join(nz_height) %>%
  group_by(Name) %>%
  summarise(elevation = mean(elevation, na.rm = TRUE))
plot(nz_avgheight["elevation"])
```



It is possible to measure the geographic distance between spatial objects using `st_distance()`. Notice that it returns the units!

```
# Get the highest point in New Zealand
nz_highest <- nz_height %>%
  slice_max(order_by = elevation)

# Get the centroid of Canterbury
canterbury_centroid <- nz %>%
  filter(Name == "Canterbury") %>%
  st_centroid()
```

```
## Warning in st_centroid.sf(.): st_centroid assumes attributes are constant over
## geometries of x
```

```
# Calculate the distance between these two points
st_distance(nz_highest, canterbury_centroid)
```

```
## Units: [m]
##          [,1]
## [1,] 115540
```

The function `st_distance()` can also be used to calculate distance matrices.

```
# Get the 3 highest points in New Zealand
nz_3highest <- nz_height %>%
  arrange(desc(elevation)) %>%
  slice_head(n = 3)

# Get the centroids of all states
all_centroid <- nz %>%
```

```
  st_centroid()
```

```
## Warning in st_centroid.sf(.): st_centroid assumes attributes are constant over
## geometries of x
```

```
# Calculate the distance matrix with centroids
st_distance(nz_3highest, all_centroid)
```

```
## Units: [m]
##          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
## [1,] 951233.0 856625.2 765070.3 825245.2 881491.0 723575.1 593453.8 621157.3
## [2,] 952019.6 857339.3 765680.7 825764.2 881937.4 724002.7 594057.0 621634.9
## [3,] 951563.1 856927.5 765332.0 825470.7 881687.6 723764.2 593712.6 621366.3
##          [,9]    [,10]    [,11]    [,12]    [,13]    [,14]    [,15]    [,16]
## [1,] 520471.9 108717.6 115540.0 191499.2 294665.7 315036.6 373697.0 348221.3
## [2,] 520789.4 109366.2 115390.2 190666.6 294032.1 315602.5 374195.6 348620.9
## [3,] 520616.6 108993.9 115493.6 191151.6 294394.6 315280.6 373914.3 348399.0
```

```
# Without centroids
st_distance(nz_3highest, nz)
```

```
## Units: [m]
##          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
## [1,] 865750.4 797431.4 659107.4 730881.6 807936.8 630429.8 551289.0 522987.7
## [2,] 866509.0 798128.8 659627.6 731367.7 808407.5 630769.4 551922.6 523389.0
## [3,] 866069.8 797727.2 659333.3 731094.0 808143.1 630583.5 551559.8 523166.2
##          [,9]    [,10] [,11]    [,12]    [,13]    [,14]    [,15]    [,16]
## [1,] 447927.2 2184.049     0 54305.52 163297.5 232290.5 355518.1 263784.7
## [2,] 448273.4 2991.754     0 53735.24 163011.5 232811.0 356027.2 264131.4
## [3,] 448083.5 2507.071     0 54058.45 163164.7 232516.4 355739.6 263941.1
```

Other geographic measurements include `st_area()` and `st_length()`.

```
# Area
nz %>%
  group_by(Name) %>%
  st_area()
```

```
## Units: [m^2]
##  [1] 12890576439  4911565037 24588819863 12271015945  8364554416 14242517871
##  [7]  7313990927 22239039580  8149895963 23409347790 45326559431 31903561583
## [13] 32154160601  9594917765   408075351 10464846864
```

```
# Length
seine %>%
  group_by(name) %>%
  st_length()
```

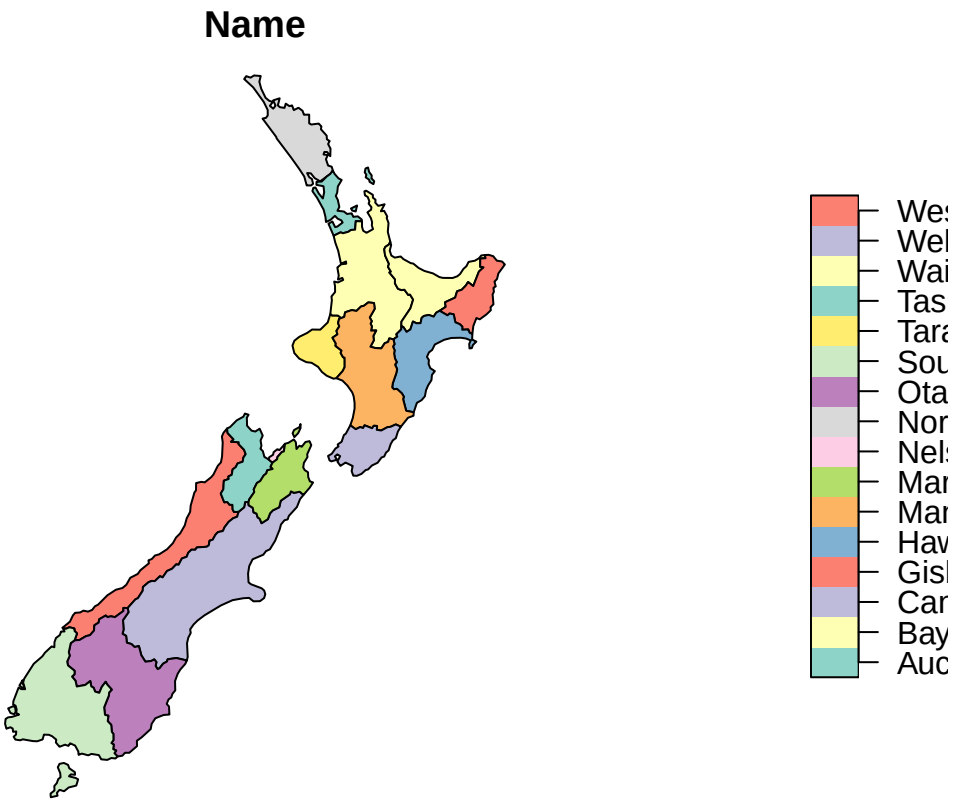
```
## Units: [m]
## [1] 363610.5 635526.7 219244.6
```

Geometry Operations

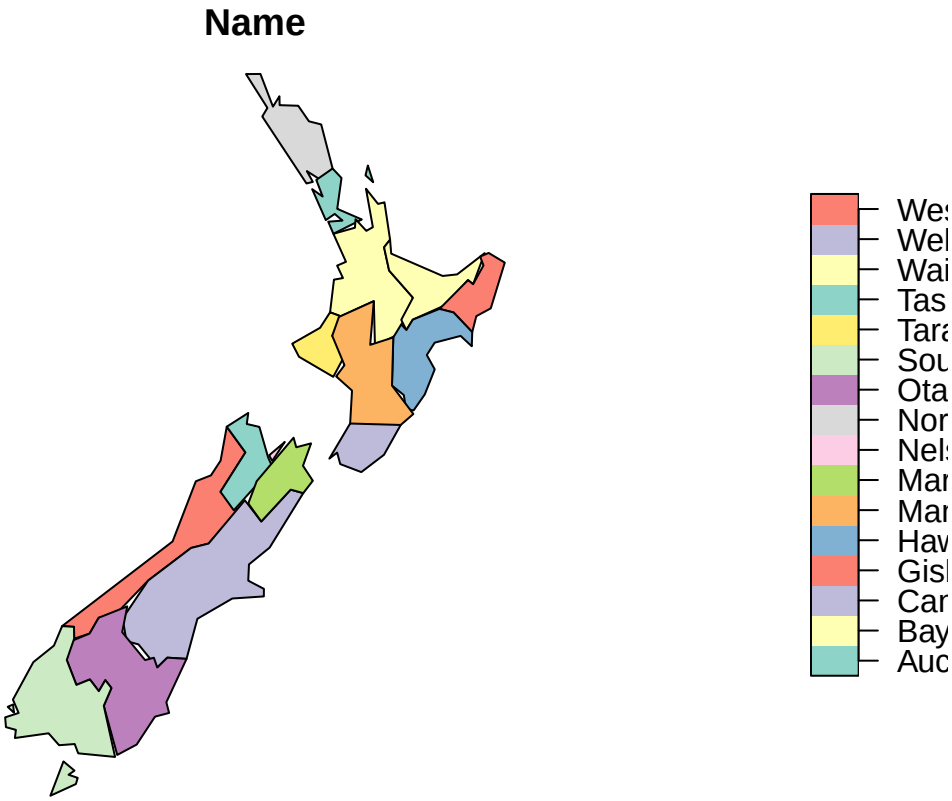
The functions in this section interact with the `geom` variable.

The function `st_simplify()` reduces the number of verticies in a spatial object. This results in a "smoothing" of the geography as well as an object that takes up less memory. Use `ms_simplify()` from the `rmapshaper` to avoid spacing issues.

```
plot(nz[1])
```




```
plot(st_simplify(nz[1], dTolerance = 10000)) #Smooth by 10 km
```

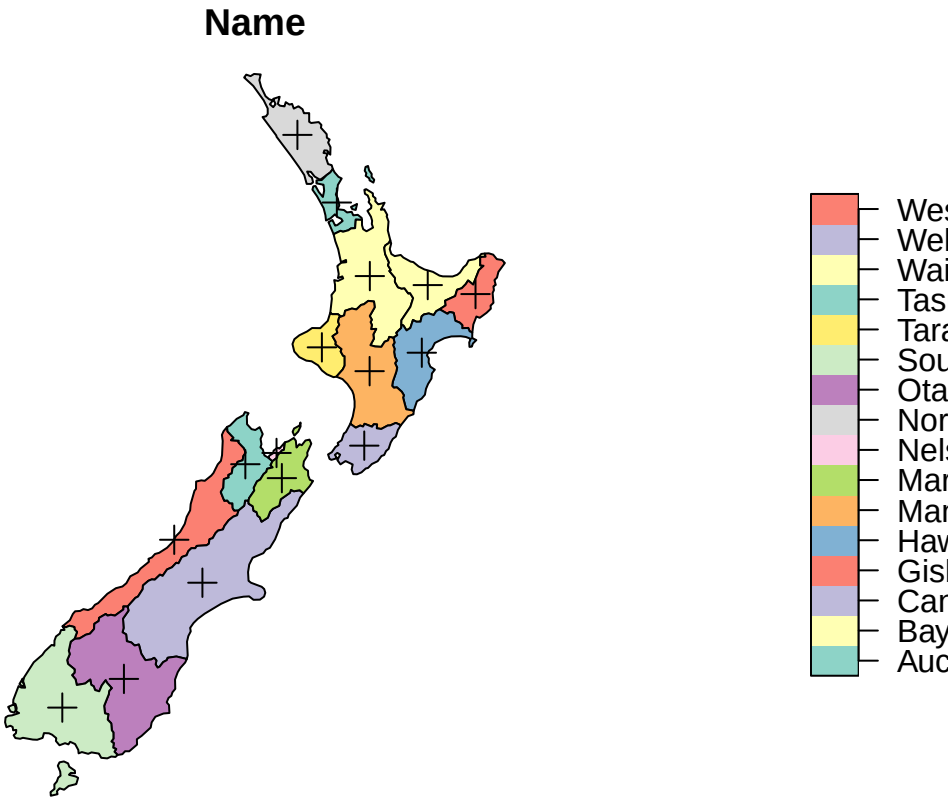


We have already seen how to compute the centroid.

```
all_centroid <- st_centroid(nz)
```

```
## Warning in st_centroid.sf(nz): st_centroid assumes attributes are constant over
## geometries of x
```

```
# Plot
plot(nz[1], reset = FALSE)
plot(st_geometry(all_centroid), add = TRUE, pch = 3, cex = 1.4)
```

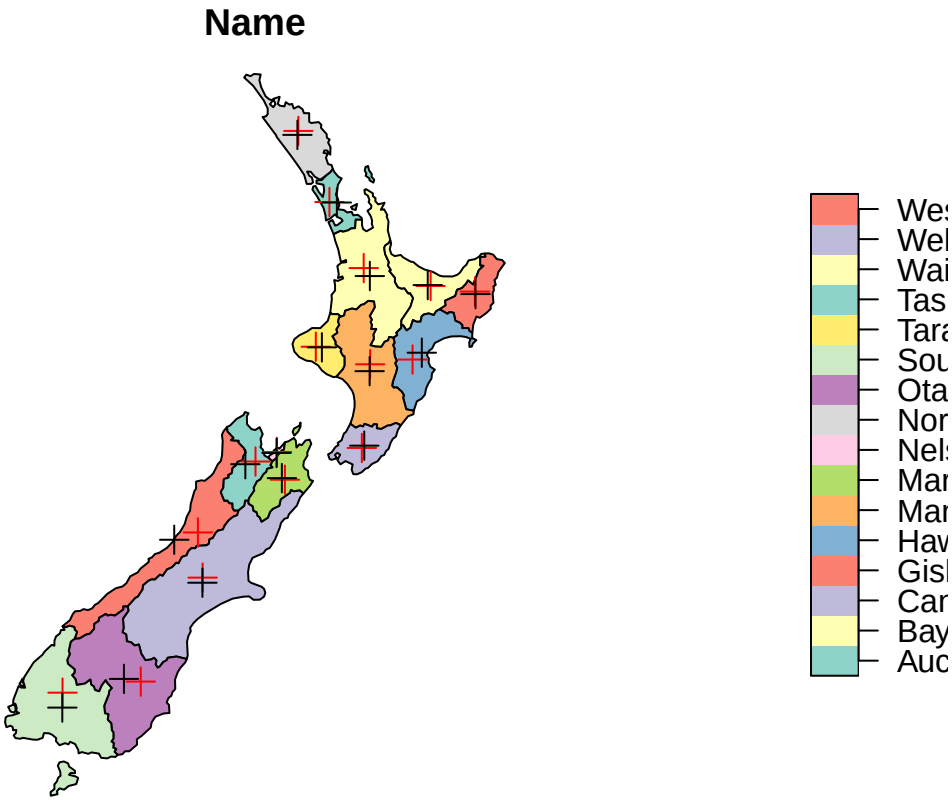


The function `st_point_on_surface()` alters the point so that a point appears on the parent object. This may be more useful than centroids for labels.

```
nz_ptsonsurface <- st_point_on_surface(nz)
```

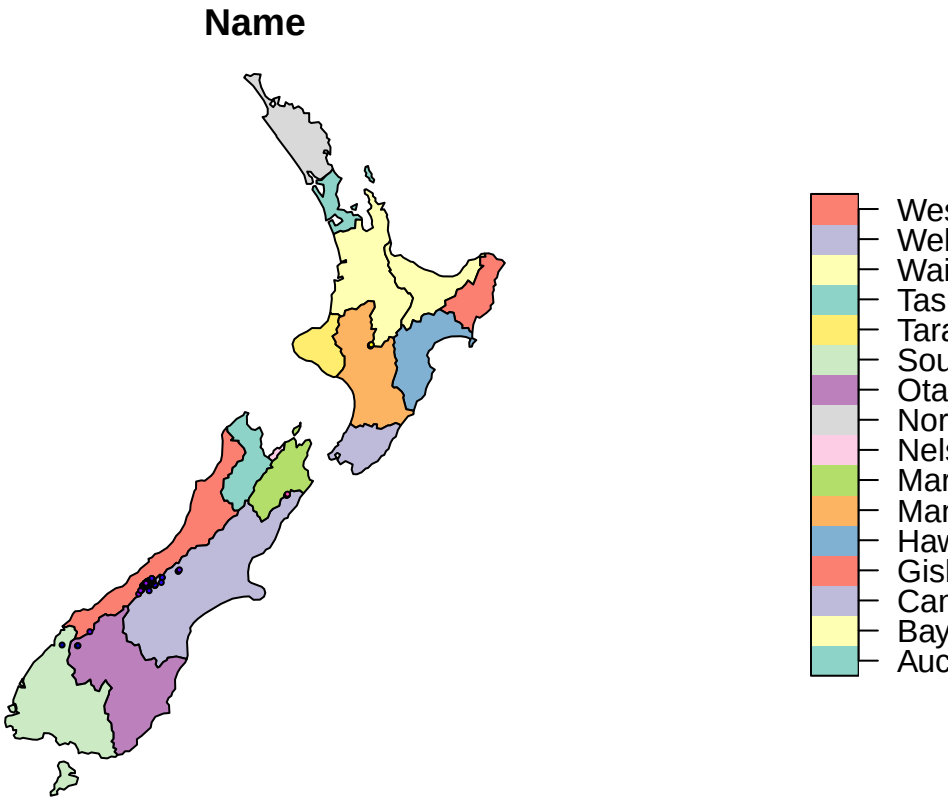
```
## Warning in st_point_on_surface.sf(nz): st_point_on_surface assumes attributes
## are constant over geometries of x
```

```
plot(nz[1], reset = FALSE)
plot(st_geometry(nz_ptsonsurface), add = TRUE, pch = 3, cex = 1.4, col = "red")
plot(st_geometry(all_centroid), add = TRUE, pch = 3, cex = 1.4) # Add centroids for comparison
```

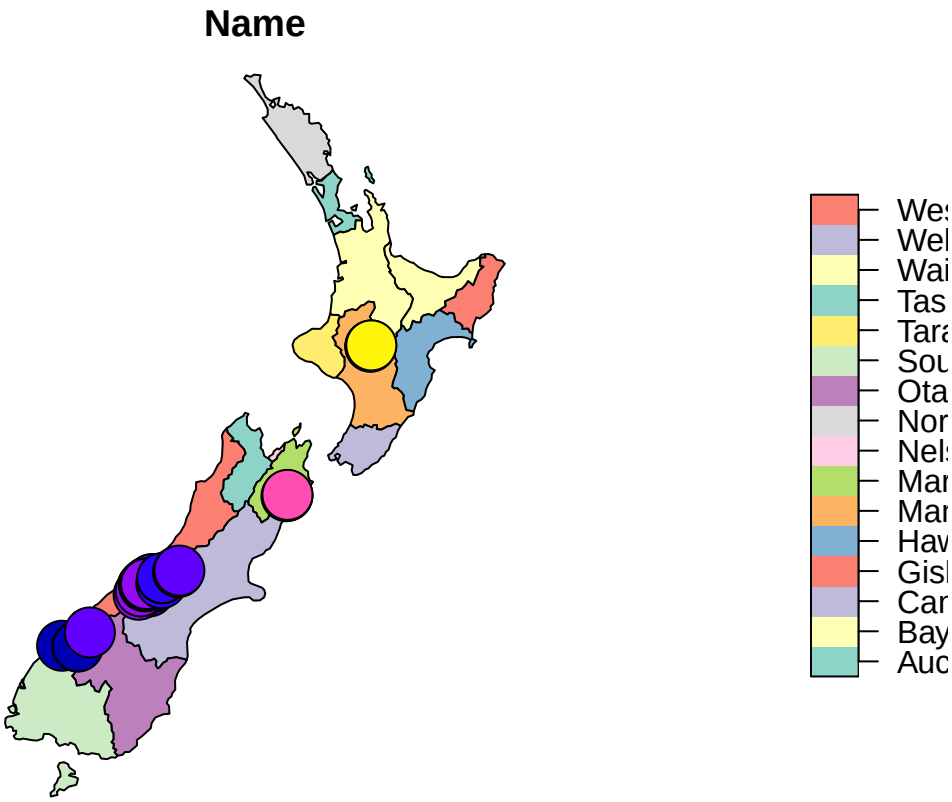


The function `st_buffer()` allows you to compute buffers around geographies. It returns the same sf object, but the geometry column now contains a buffer around the original geometry of the specified distance (in meters).

```
nz_height_buff_5km <- st_buffer(nz_height, dist = 5000)
nz_height_buff_50km <- st_buffer(nz_height, dist = 50000)

plot(nz[1], reset = FALSE)
plot(nz_height_buff_5km[1], add = TRUE)
```

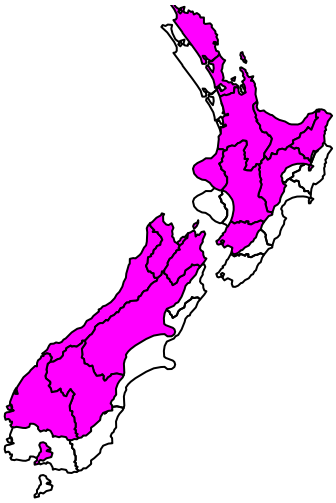



```
plot(nz[1], reset = FALSE)
plot(nz_height_buff_50km[1], add = TRUE)
```

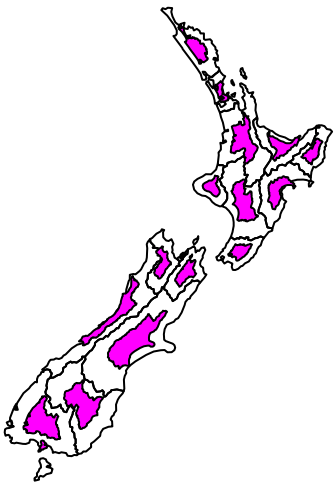


Affine transformations can be done with spatial data. They should be approached with caution as angles and length are not always preserved even though lines are.

```
nz_g <- st_geometry(nz)

# Shift
nz_shift <- nz_g + c(0, 100000)

plot(nz_g, reset = FALSE)
plot(nz_shift, col = "magenta", add = TRUE)
```



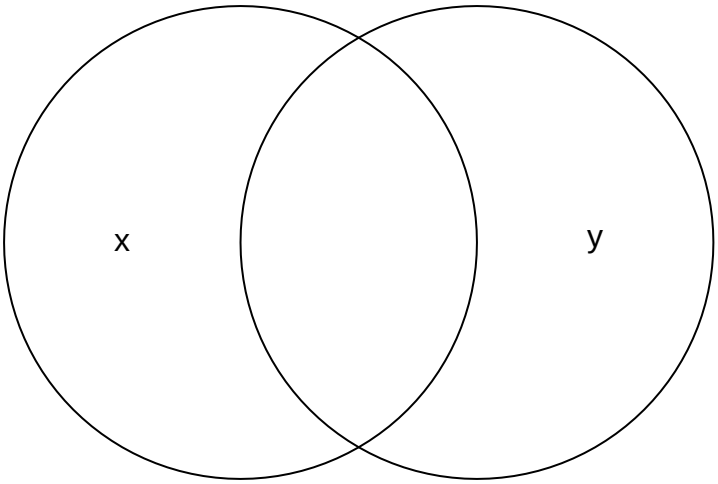
```
# Scaling around the centroid
nz_scale <- (nz_g - st_centroid(nz_g)) * 0.5 + st_centroid(nz_g)

plot(nz_g, reset = FALSE)
plot(nz_scale, col = "magenta", add = TRUE)
```



```
# Set the new geography
nz_changed <- st_set_geometry(nz, nz_shift)
```

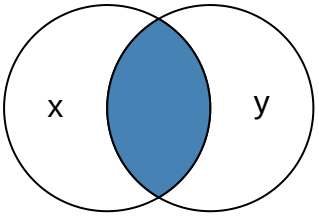
Spatial subsetting with lines or polygons involving changes to the geometry columns is called spatial clipping. Forillustaration, consider two circles created as follows.

```r
# Create two points and buffers
b <- st_sfc(st_point(c(0, 1)), st_point(c(1, 1)))
b <- st_buffer(b, dist = 1)
x <- b[1]
y <- b[2]

# Plot
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
```

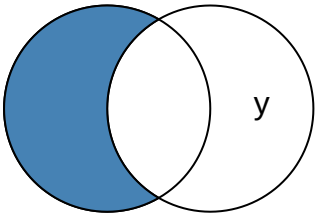


With these circles, we can see the possible spatial clippings.

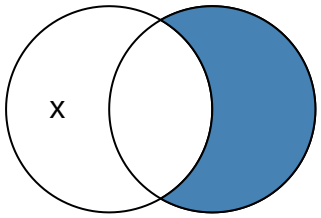
```r
# Intersection
x_int_y <- st_intersection(x, y)
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
plot(x_int_y, add = TRUE, col = "steelblue")
```



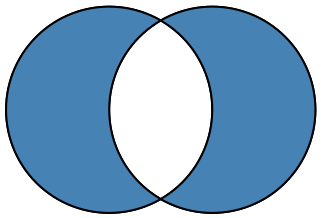

```r
# Difference
x_diff_y <- st_difference(x, y)
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
plot(x_diff_y, add = TRUE, col = "steelblue")
```



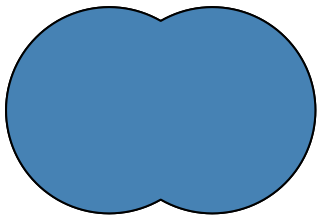
```
y_diff_x <- st_difference(y, x)
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
plot(y_diff_x, add = TRUE, col = "steelblue")
```




```
# Both differences
x_sym_y <- st_sym_difference(x, y)
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
plot(x_sym_y, add = TRUE, col = "steelblue")
```

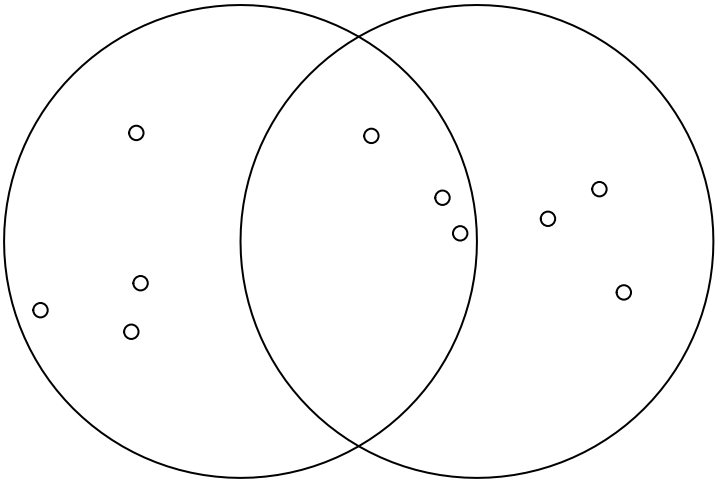


```
# Union
x_uni_y <- st_union(x, y)
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
plot(x_uni_y, add = TRUE, col = "steelblue")
```



The function `st_sample()` randomly selects points within an area.

```
b_sample <- st_sample(b, size = 10)
plot(b)
plot(b_sample, add = TRUE)
```



Merge

Attribute

Suppose you have two datasets with a common identifier variable (e.g., ID, name). It is possible to merge these datasets with this variable following chapter 3, even if there are spatial columns. Here is an example.

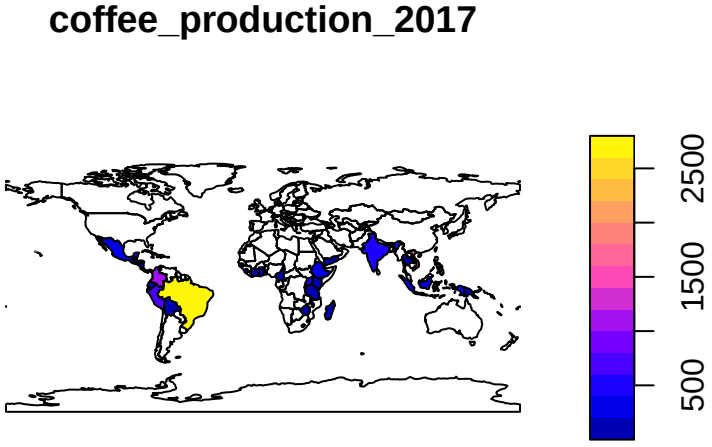
```
# Load coffee_data
data("coffee_data")
names(coffee_data)
```

```
## [1] "name_long"              "coffee_production_2016" "coffee_production_2017"
```

```
class(coffee_data)
```

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

```
# Join the world and coffee data
world_coffee <- full_join(world, coffee_data, by = "name_long")

# Plot coffee production 2017
plot(world_coffee["coffee_production_2017"])
```



Spatial

If you have two (or more) data sets with spatial elements, it is natural to think about merging them based on spatial concepts. Instead of sharing an identifier variable, they may share geographic space. The function `st_join()` allows for many types of spatial overlays. For example, it is possible to join points to multipolygons.

```
names(nz)
```

```
## [1] "Name"          "Island"        "Land_area"     "Population"
## [5] "Median_income" "Sex_ratio"     "geom"
```

```
names(nz_height)
```

```
## [1] "t50_fid"   "elevation" "geometry"
```

```
nz_joined <- st_join(nz, nz_height)
```

The default is a left join. Inspecting the data reveals that regions with more than one point in `nz_height` appear more than once. The corresponding attribute data from `nz_height` are added to `nz`. The geography type is multipolygon, following the first argument's type. If we switch the order, the geography type changes to point.

```
nz_joined_rev <- st_join(nz_height, nz)
```

To do an inner join, set `left = FALSE`. The default operator is `st_intersects()`. To change the operator, change the `join` argument. The sf cheat sheet is a great resource to think about spatial overlays and the possible operators.

There may be contexts in which two spatial datasets are related, but do not actually contain overlapping elements. Augmenting the `st_join()` function allows for a buffer distance to create near matches. Here is an example using the `cycle_hire` and `cycle_hire_osm` datasets from the `spData` package.

```
data("cycle_hire")
data("cycle_hire_osm")
any(st_touches(cycle_hire, cycle_hire_osm, sparse = FALSE))
```

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

```
# Change the CRS to be able to use meters
cycle_hire <- cycle_hire %>%
  st_transform(crs = 27700)

cycle_hire_osm <- cycle_hire_osm %>%
  st_transform(crs = 27700)

# Join within 20 meters
cycle_joined <- st_join(cycle_hire, cycle_hire_osm,
                        join = st_is_within_distance,
                        dist = 20)
```

Practice Exercises 7.1

1. The below code randomly selects ten points distributed across Earth. Create an object named `random_sf` that merges these random points with the `world` dataset. In which countries did your random points land? (Review: How can you make the below code reproducible?)

```
# Coordinate bounds of the world
bb_world <- st_bbox(world)
```

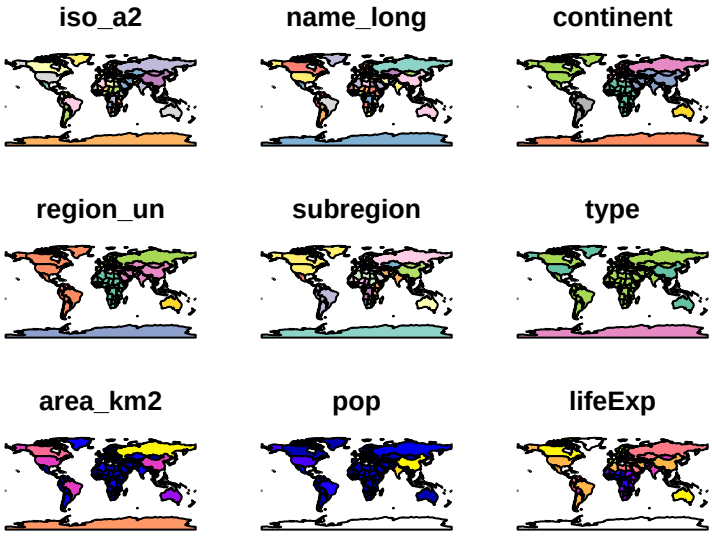
```
random_df <- tibble(
  x = runif(n = 10, min = bb_world[1], max = bb_world[3]),
  y = runif(n = 10, min = bb_world[2], max = bb_world[4])
)

# Set coordinates and CRS
random_points <- random_df %>%
  st_as_sf(coords = c("x", "y")) %>%
  st_set_crs(4326)
```

2. We saw that 70 of the 101 highest points in New Zealand are in Canterbury. How many points in `nz_height()` are within 100 km of Canterbury?
3. Find the geographic centroid of New Zealand. How far is it from the geographic centroid of Canterbury?

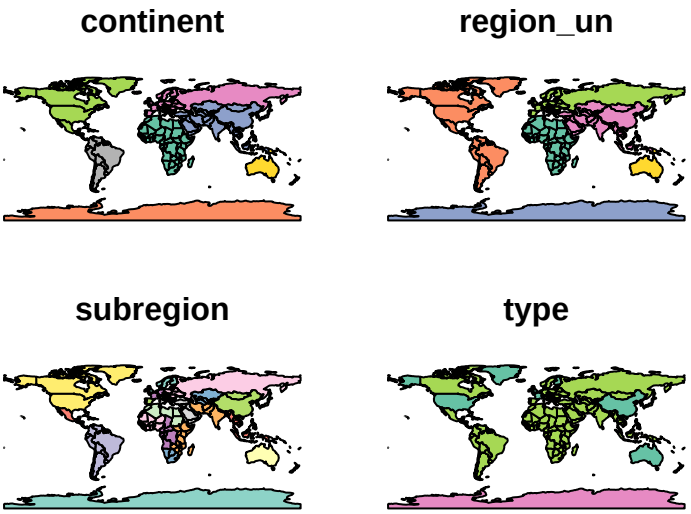
Visualize

The function `plot()` can be used with the `sf` object directly. It relies on the same arguments as for non-geographic data.

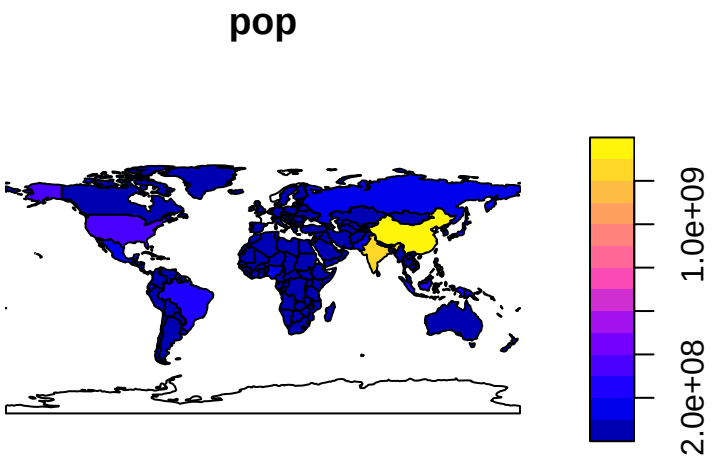
```
plot(world)
```



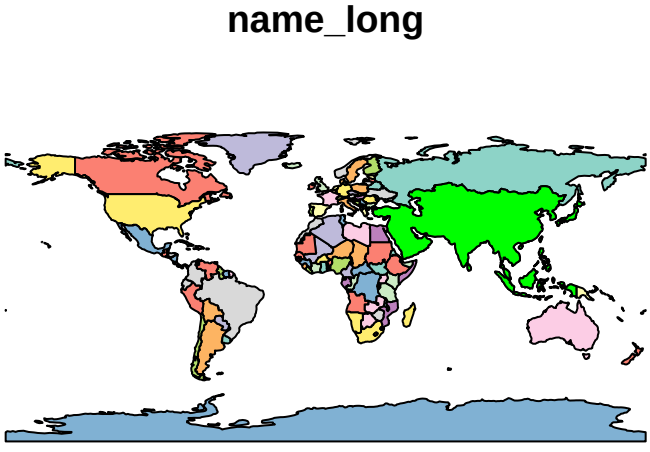

```
plot(world[3:6])
```




```
plot(world["pop"])
```

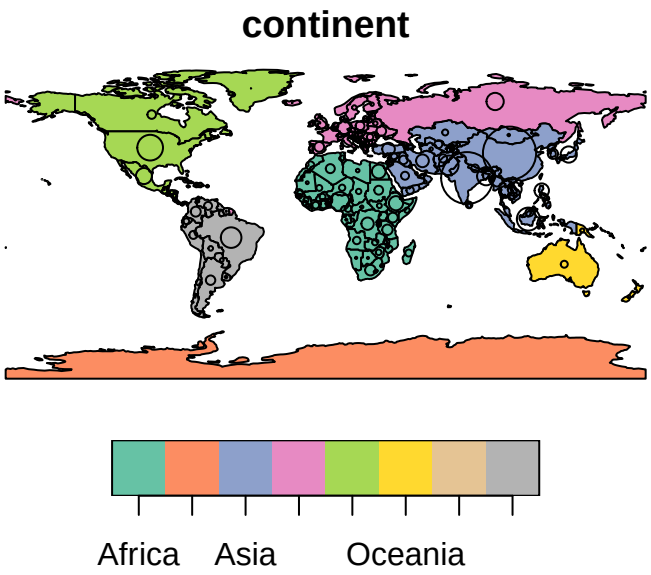



```
# Example adding layers
world_asia <- world[world$continent == "Asia", ] %>% st_union()
plot(world[2], reset = FALSE)
plot(world_asia, add = TRUE, col = "green")
```



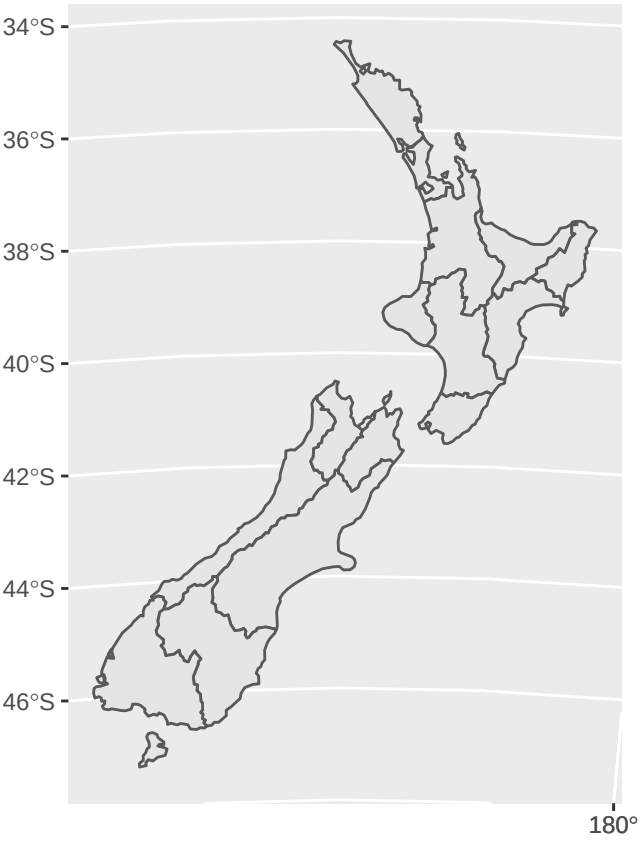

```
# More complex example
plot(world["continent"], reset = FALSE)
```

```
world_centroids <- st_centroid(world, of_largest_polygon = TRUE)
plot(st_geometry(world_centroids), add = TRUE, cex = sqrt(world$pop) / 10000)
```

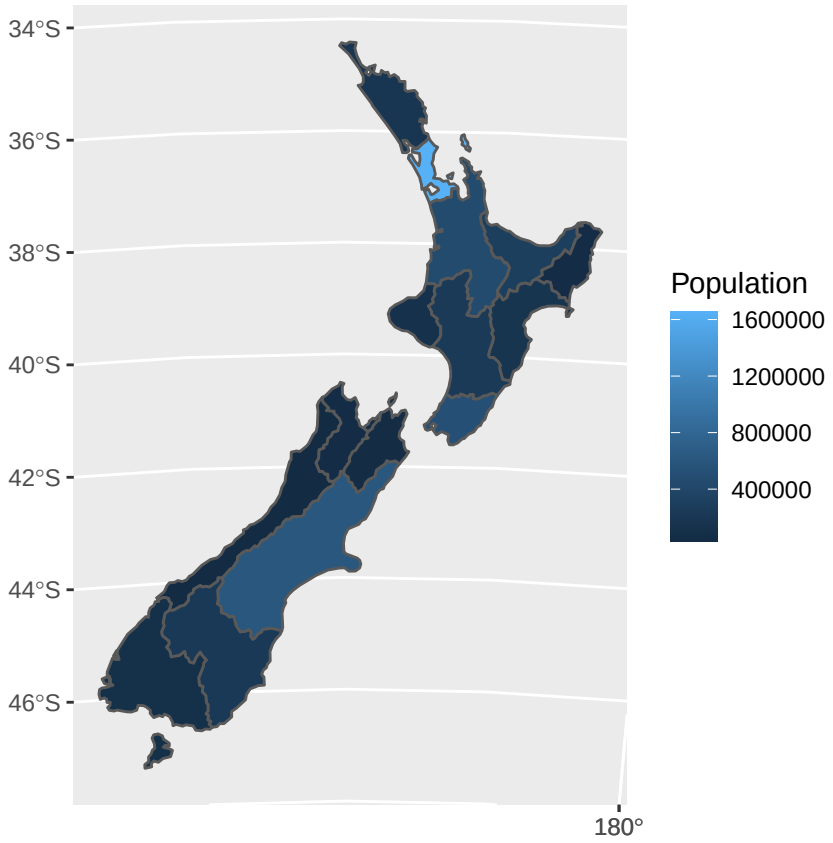


The package `tmap` has the same logic as `ggplot2`, but specialized for maps. There are options for interactive maps as well. The functions of `ggplot2` can also be used with the geom `geom_sf()`.

```
ggplot(nz) +
  geom_sf()
```

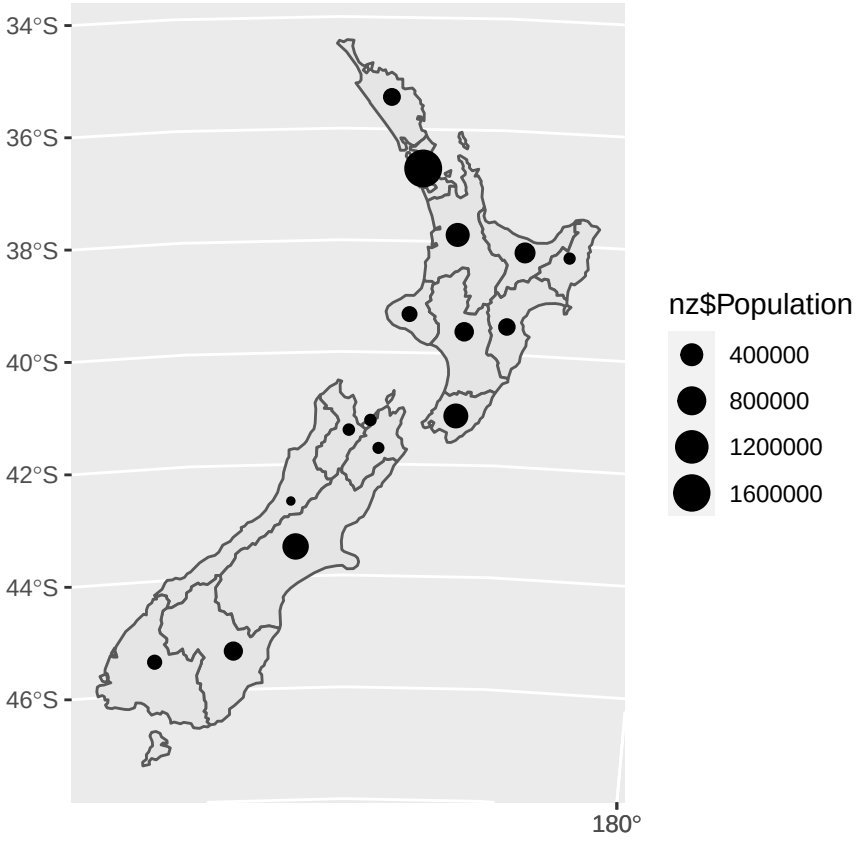


```
ggplot(nz) +
  geom_sf(aes(fill = Population))
```




```
ggplot() +
  geom_sf(data = nz) +
  geom_sf(data = st_geometry(st_point_on_surface(nz)), aes(size = nz$Population))
```

```
## Warning in st_point_on_surface.sf(nz): st_point_on_surface assumes attributes
## are constant over geometries of x
```



Here are some simple examples with `tmap`.

```
tm_shape(nz) +
  tm_borders()
```

```
tm_shape(nz) +
  tm_fill()
```

```
tm_shape(nz) +
  tm_fill() +
  tm_borders()
```

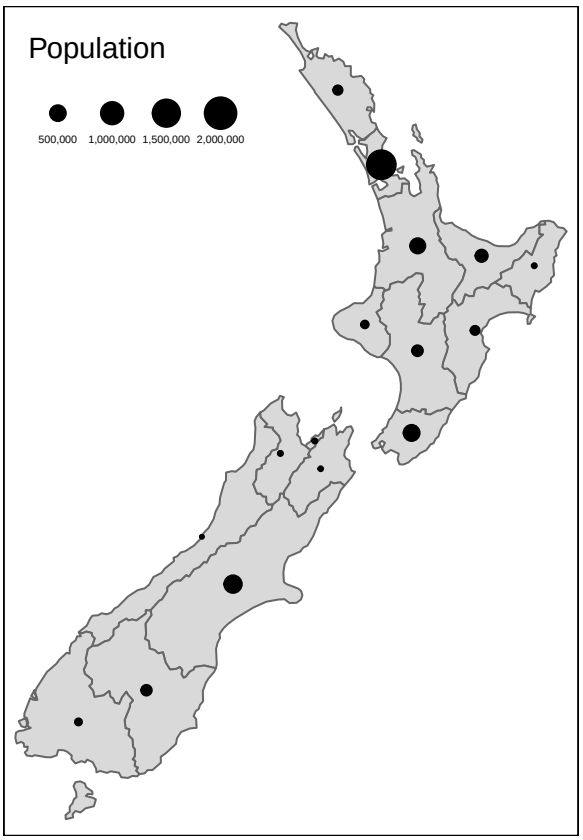
```
tm_shape(nz) +
  tm_polygons()
```

```
# Shortcut for the above
qtm(nz)
```

Here are other examples adding more layers. See chapter 8 of Lovelace, Nowosad, and Muenchow (2021) for more details on the possibilities.

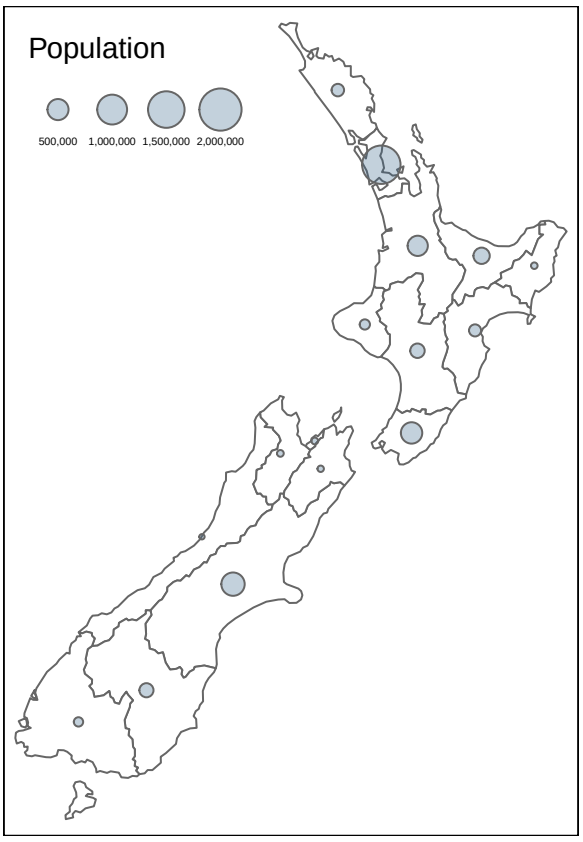
```
tm_shape(nz) +
  tm_polygons() +
  tm_dots(size = "Population")
```

```
## Legend labels were too wide. Therefore, legend.text.size has been set to 0.35. Increase legend.width
```

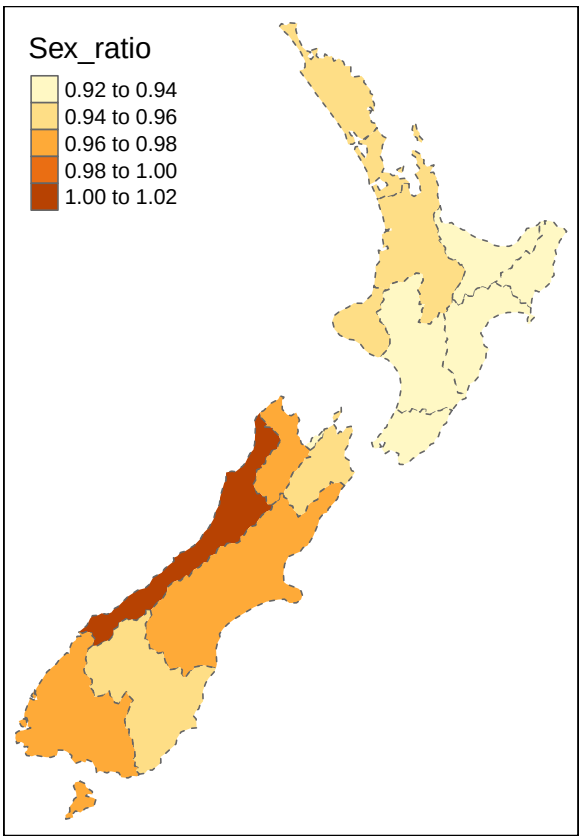



```
tm_shape(nz) +
  tm_polygons(col = "white") +
  tm_bubbles(size = "Population", alpha = 0.3, col = "steelblue4")
```

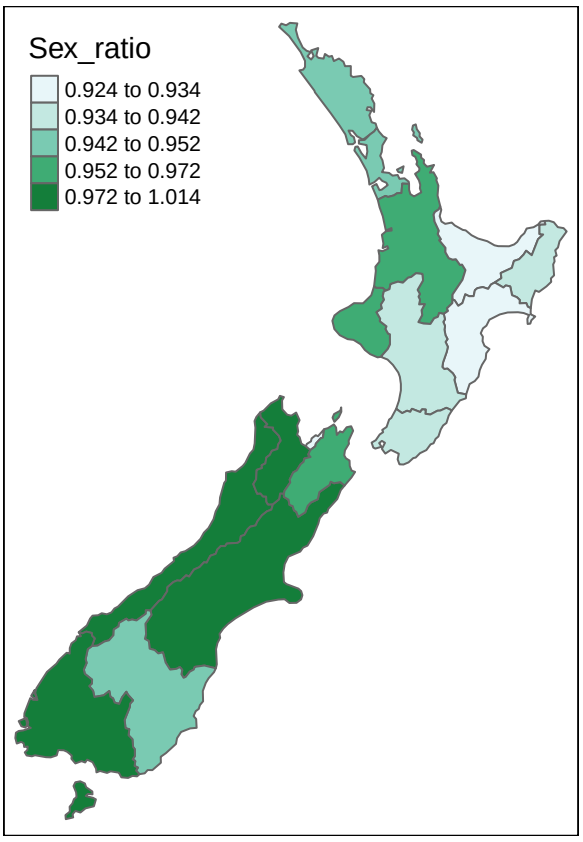
```
## Legend labels were too wide. Therefore, legend.text.size has been set to 0.35. Increase legend.width
```



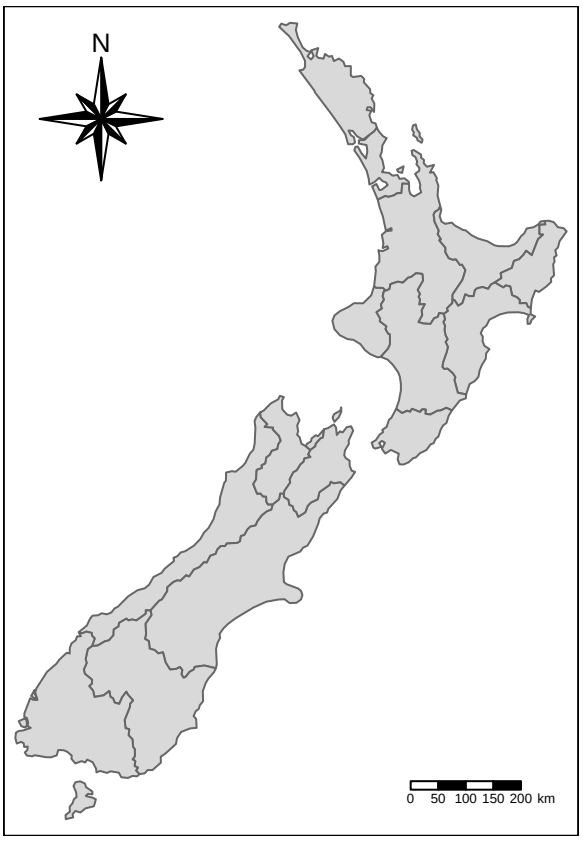

```
tm_shape(nz) +
  tm_borders(lty = 2, lwd = 0.8) +
  tm_fill(col = "Sex_ratio")
```



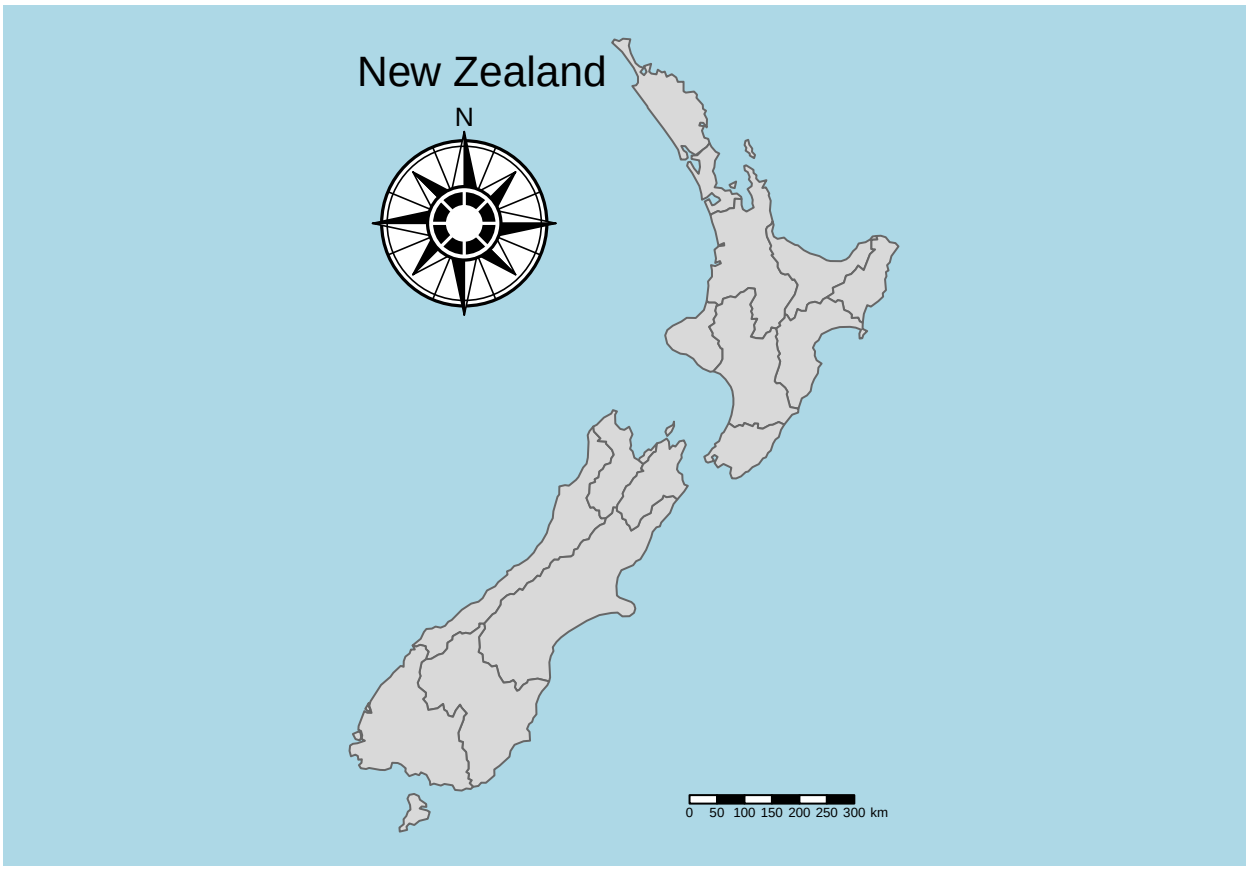

```
# Check out this great tool for map colors
# tmaptools::palette_explorer()
tm_shape(nz) +
  tm_polygons(col = "Sex_ratio", palette = "BuGn", style = "quantile")
```



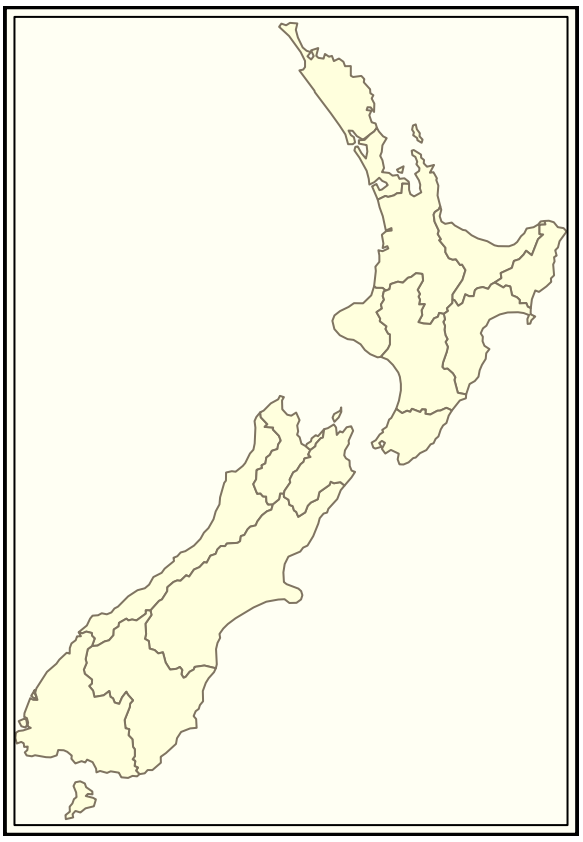

```
tm_shape(nz) +
  tm_polygons() +
  tm_compass(type = "8star", position = c("left", "top")) +
  tm_scale_bar(breaks = c(0, 50, 100, 150, 200))
```




```
tm_shape(nz) +
  tm_polygons() +
  tm_compass(type = "rose", position = c("left", "top")) +
  tm_scale_bar() +
  tm_layout(frame = FALSE, bg.color = "lightblue", title = "New Zealand")
```




```
tm_shape(nz) +
  tm_polygons() +
  tm_style("classic")
```



```
tm_shape(nz) +
  tm_polygons() +
  tm_style("bw") +
  tm_facets(by = "Island")
```

Practice Exercises 7.2

1. Using the `world` dataset, plot one country of your choice. (Hint: You may need to use `st_geometry()`.) Add whatever aesthetic elements you would like.

Raster Data

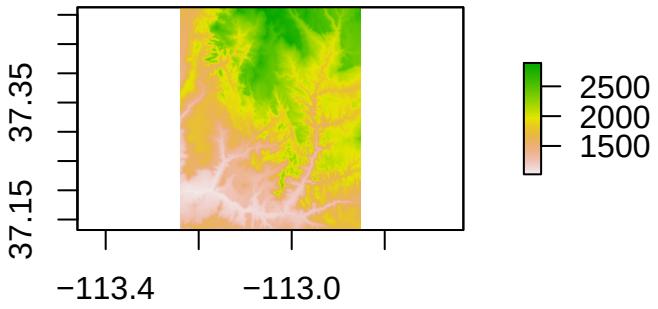
Raster data are comprised of equally sized cells and corresponding data values. Because there can only be one value per cell, raster data can be ill-suited to represent human-invented borders. Because of the matrix representation of the geography (rather than coordinate points), raster data is well-suited to efficiently represent continuous spatial data. The package `raster` allows you to load, analyze, and map raster objects.

```
data("elevation")
elevation
```

```
## class      : RasterLayer
## dimensions : 457, 465, 212505  (nrow, ncol, ncell)
## resolution : 0.0008333333, 0.0008333333  (x, y)
## extent     : -113.2396, -112.8521, 37.13208, 37.51292  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## source     : memory
## names      : srtm
## values     : 1024, 2892  (min, max)
```

Printing the example raster displays the raster header and information. This example raster is from Zion National Park in Utah, U.S.. Just as for vector data, the `plot()` function can be used with raster data.

```
plot(elevation)
```



To access and set the CRS for raster objects, use `projection()`.

```
projection(elevation)
```

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

Accessing and managing the attribute data of rasters is different than the usual approaches. For example, variables cannot be character strings. See chapter 3 of Lovelace, Nowosad, and Muenchow (2021) for information on this. See chapter 4 for spatial operations on raster data and chapter 5 for geometry operations on raster data.

Further Reading

The above information comes from Lovelace, Nowosad, and Muenchow (2021) and Wickham (2016).

References

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2021. *Geocomputation with R*. https://geocompr.robinlovelace.net/.

Wickham, Hadley. 2016. *Ggplot2*. Use R! Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-24277-4.