

Spatial Data in R: Overview and Examples

Israeli Geographical Association Conference

Michael Dorman

Geography and Environmental Development

dorman@post.bgu.ac.il

2017-12-18



Ben-Gurion University of the Negev

Contents

- ▶ The aim of this tutorial is to provide an **introduction to spatial data analysis in R**
- ▶ **Part I** is an **overview** of notable packages for spatial analysis in R
- ▶ **Parts II & III** comprise a practical tutorial of the **sf package**
- ▶ Slides and code available on <https://goo.gl/S1Cgv1>
- ▶ You need to install several packages to run code examples -

```
install.packages("sf")
install.packages("mapview")
install.packages("units")
install.packages("dplyr")
```

Part I: Overview

Introduction

- ▶ R is a **programming language** and **environment**, originally developed for **statistical computing** and **graphics**
- ▶ Over time, R has had an increasing number of **contributed packages** for handling and analysing spatial data
- ▶ Today, spatial analysis is a **major functionality in R**
- ▶ As of November 2017, there are **182 packages** for the analysis of spatial data in R

`rgdal`, `sf`, `raster`: Handling spatial data

- ▶ Reading and writing -
 - ▶ **Vector formats** such as **ESRI Shapefile**, **GeoJSON** and **GPX** - package `rgdal` or `sf` (using **OGR**)
 - ▶ **Raster formats** such as **GeoTIFF** - package `raster` (using **GDAL**)
 - ▶ Specialized formats can be read with other packages, for example -
 - ▶ **HDF** files can be read with package `gdalUtils`
 - ▶ **NetCDF** files can be read with package `ncdf4`
- ▶ Reprojection (using **PROJ4**)

rgeos, sf: Geoprocessing Vector Layers

- ▶ Geometric operations on **vector layers** - package rgeos or sf (using GEOS) -
 - ▶ **Numeric operators** - Area, Length, Distance...
 - ▶ **Logical operators** - Contains, Within, Within distance, Crosses, Overlaps, Equals, Intersects, Disjoint, Touches...
 - ▶ **Geometry generating operators** - Centroid, Buffer, Intersection, Union, Difference, Convex-Hull, Simplification...

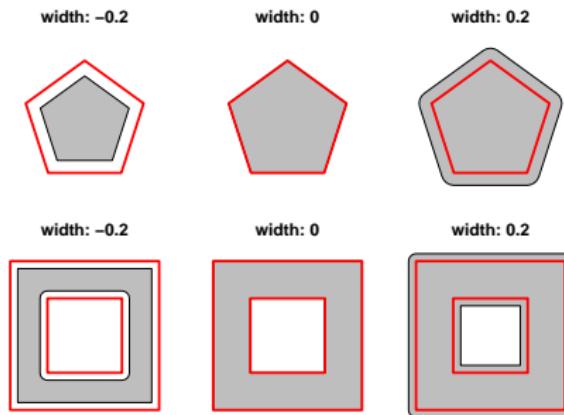


Figure 1: Buffer function

geosphere: Geometric calculations on longitude/latitude

- ▶ Package geosphere implements spherical trigonometry functions for distance and direction-related calculations on geographic coordinates (lon-lat)

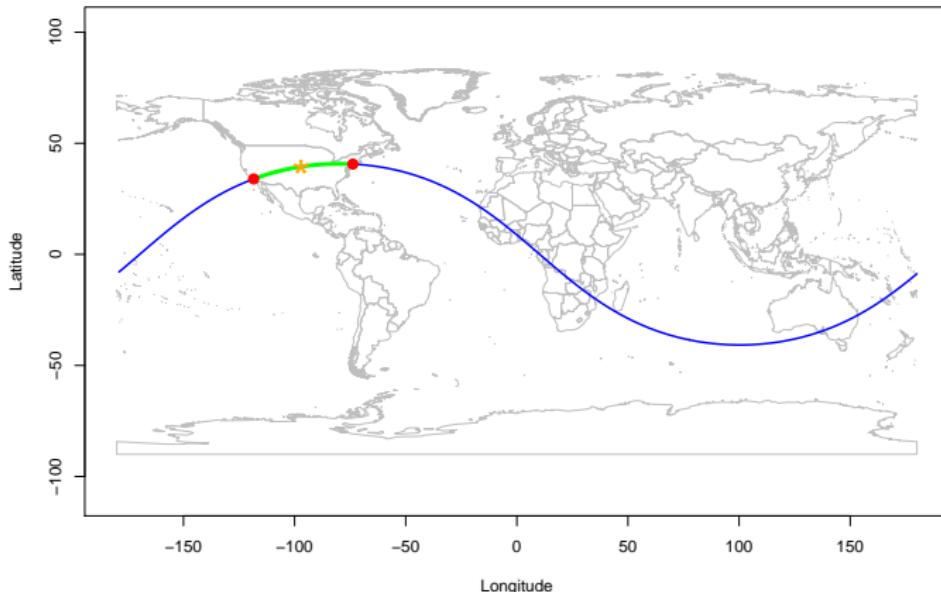


Figure 2: Points on Great Circle

raster: Geoprocessing Rasters

- ▶ Geometric operations on **rasters** can be done with package **raster** -
 - ▶ **Accessing cell values** - As vector, As matrix, Extract to points / lines / polygons, random / regular sampling, Frequency table, Histogram...
 - ▶ **Raster algebra** - Arithmetic (+, -, ...), Math (sqrt, log10, ...), logical (!, ==, >, ...), summary (mean, max, ...), Mask, Overlay...
 - ▶ **Changing resolution and extent** - Crop, Mosaic, (Dis)aggregation, Reprojection, Resampling, Shift, Rotation...
 - ▶ **Focal operators** - Distance, Direction, Focal Filter, Slope, Aspect, Flow direction...
 - ▶ **Transformations** - Vector layers <-> Raster...

raster: Geoprocessing Rasters

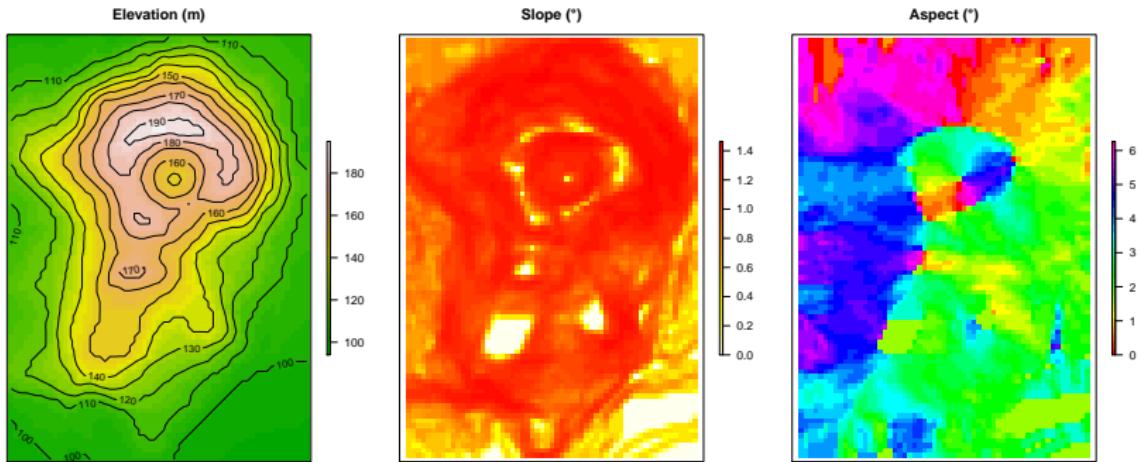


Figure 3: Topographic aspect and slope calculation

gstat: Geostatistical Modelling

- ▶ Univariate and multivariate geostatistics -
 - ▶ Variogram modelling
 - ▶ Ordinary and universal point or block (co)kriging
 - ▶ Cross-validation

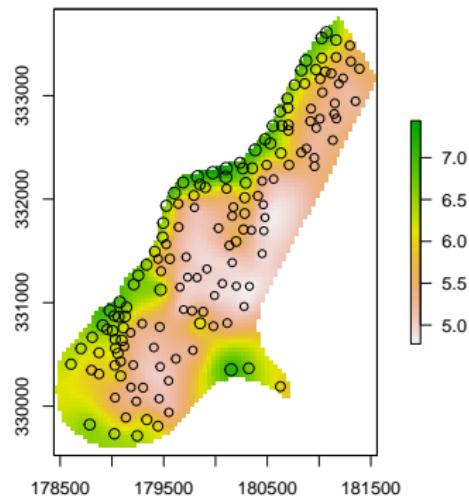


Figure 4: Predicted Zinc concentration, using Ordinary Kriging

spdep: Spatial dependence modelling

- ▶ Modelling with spatial weights -
 - ▶ Building neighbour lists and spatial weights
 - ▶ Tests for spatial autocorrelation for areal data (e.g. Moran's I)
 - ▶ Spatial regression models (e.g. SAR, CAR)

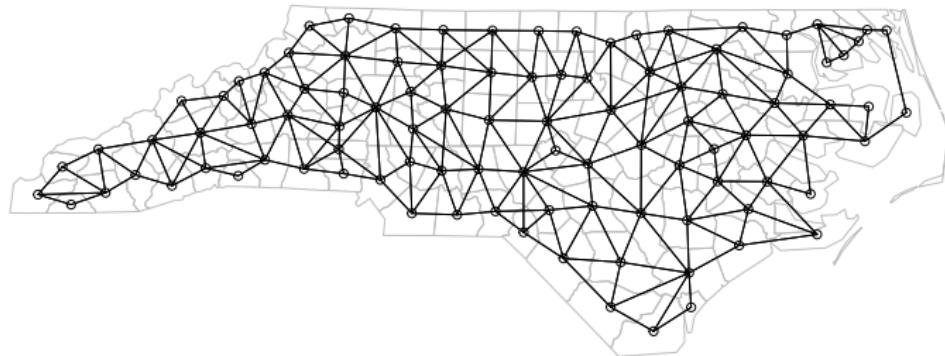


Figure 5: Neighbours list based on regions with contiguous boundaries

spatstat: Spatial point pattern analysis

- ▶ Techniques for statistical analysis of spatial point patterns, such as -
 - ▶ Kernel density estimation
 - ▶ Detection of clustering using Ripley's K-function
 - ▶ Spatial logistic regression

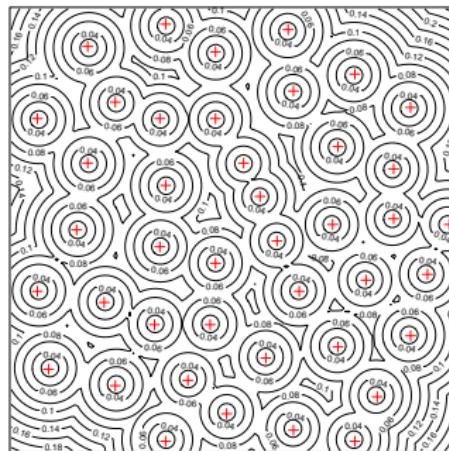


Figure 6: Distance map for the Biological Cells Point Pattern dataset

osmdata: Access to OpenStreetMap data

- ▶ Accessing OpenStreetMap (OSM) data using the [Overpass API](#)

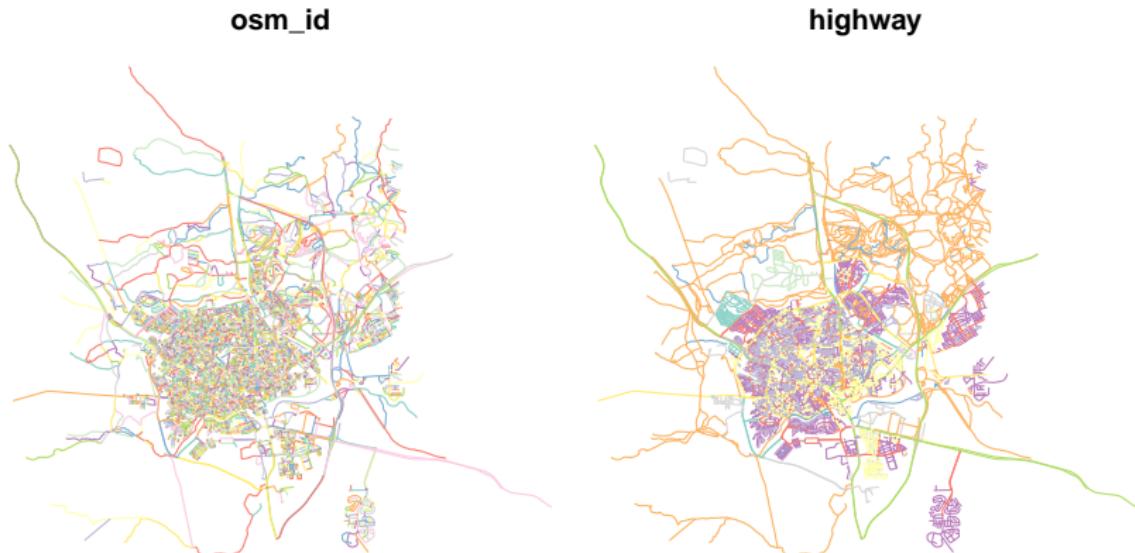


Figure 7: Beer-Sheva road network

leaflet, mapview: Web mapping

- Packages `leaflet` and `mapview` provide methods to produce **interactive maps** using the `Leaflet` JavaScript library

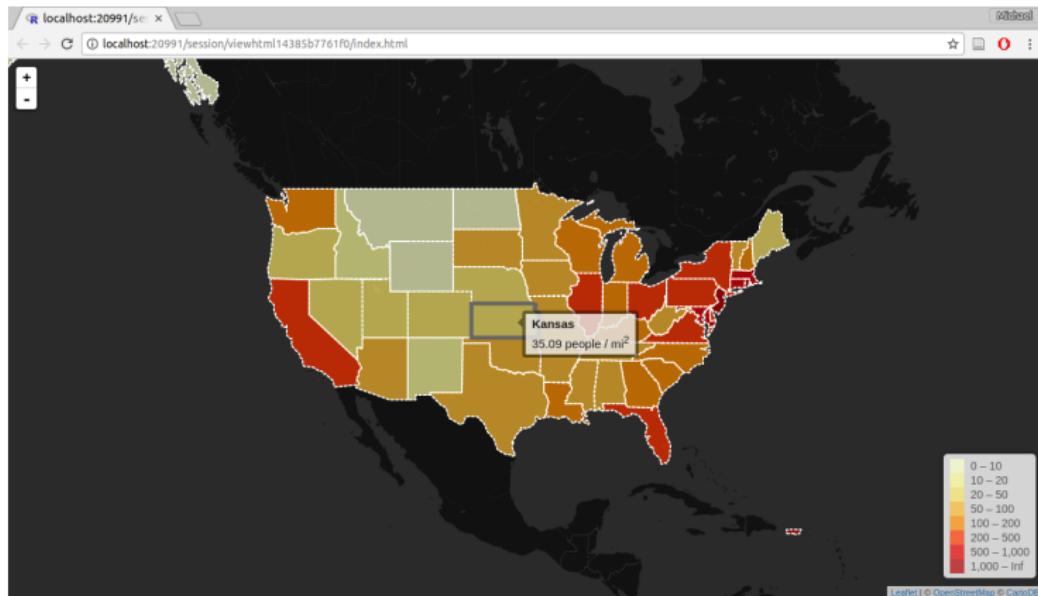


Figure 8: Intractive 'Leaflet' map

Part II: Intro to sf

Vector layers in R: package sf

- ▶ sf is a relative new (2016-) R package for **handling vector layers in R**
- ▶ In the long-term, sf will replace rgdal (2003-), sp (2005-), and rgeos (2011-)
- ▶ The main innovation in sf is a complete implementation of the **Simple Features** standard
- ▶ Since 2003, Simple Features been widely implemented in **spatial databases** (such as **PostGIS**), commercial GIS (e.g., **ESRI ArcGIS**) and forms the vector data basis for libraries such as GDAL
- ▶ When working with spatial databases, Simple Features are commonly specified as **Well Known Text (WKT)**
- ▶ A subset of simple features forms the **GeoJSON** standard

Vector layers in R: package sf

- ▶ The sf class extends the data.frame class to include a **geometry** column
- ▶ This is similar to the way that **spatial databases** are structured

## Simple feature collection with 100 features and 6 fields	
## geometry type: MULTIPOLYGON	
## dimension: XY	
## bbox:	xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## epsg (SRID):	4267
## proj4string:	+proj=longlat +datum=NAD27 +no_defs
## precision:	double (default; no precision model)
## First 3 features:	
## BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79	geom
## 1 1091 1 10 1364 0 19	MULTIPOLYGON(((-81.47275543 ...))
## 2 487 0 10 542 3 12	MULTIPOLYGON(((-81.23989105 ...))
## 3 3188 5 208 3616 6 260	MULTIPOLYGON(((-80.45634460 ...))

Simple feature

Simple feature geometry list-column (sf)

Simple feature geometry (sf)

Figure 9: Structure of an ‘sf’ object¹

¹<https://cran.r-project.org/web/packages/sf/vignettes/sf1.html>

Vector layers in R: package sf

- The **sf** class is actually a hierarchical structure composed of three classes -
 - sf** - Vector **layer** object, a table (`data.frame`) with one or more attribute columns and one geometry column
 - sfc** - Geometric part of the vector layer, the **geometry column**
 - sfg** - **Geometry** of an individual simple feature

```
## Simple feature collection with 100 features and 6 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## epsg (SRID): 4267
## proj4string: +proj=longlat +datum=NAD27 +no_defs
## precision: double (default; no precision model)
## First 3 features:
##   BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79      geom
## 1 1091    1     10 1364    0    19 MULTIPOLYGON((-81.47275543...
## 2 487     0     10  542    3    12 MULTIPOLYGON((-81.23989105...
## 3 3188    5     208 3616    6    260 MULTIPOLYGON((-80.45634460...
```

Simple feature

Simple feature geometry list-column (sfc)

geom

Creating layers from scratch

- ▶ As mentioned above, the main classes in the sf package are -
 - ▶ **sfg** - geometry
 - ▶ **sfc** - geometry column
 - ▶ **sf** - layer
- ▶ Let's create an object for each of these classes to learn more about them
- ▶ First load the sf package -

```
library(sf)
## Linking to GEOS 3.5.1, GDAL 2.2.1, proj.4 4.9.2, lwgeom
```

Geometry (**sfg**)

- ▶ Objects of class **sfg (geometry)** can be created using the appropriate function for each geometry type -
 - ▶ `st_point`
 - ▶ `st_multipoint`
 - ▶ `st_linestring`
 - ▶ `st_multilinestring`
 - ▶ `st_polygon`
 - ▶ `st_multipolygon`
 - ▶ `st_geometrycollection`
- ▶ From coordinates passed as -
 - ▶ **numeric vectors**
 - ▶ **matrix objects**
 - ▶ **list objects**

Geometry (`sfg`)

- ▶ For example, we can create an object named `pnt1` representing a `POINT` geometry with `st_point` -

```
pnt1 = st_point(c(34.812831, 31.260284))
```

- ▶ Printing the object in the console gives the **WKT** representation -

```
pnt1
## POINT(34.812831 31.260284)
```

Geometry (**sfg**)

- ▶ Note the class definition of an **sfg** (**geometry**) object is actually composed of three parts -
 - ▶ XY - The dimensions type (XY, XYZ, XYM or XYZM)
 - ▶ POINT - The geometry type (POINT, MULTIPOLYGON, etc.)
 - ▶ **sfg** - The general class (**sfg** = Simple Feature Geometry)
- ▶ For example, we created has geometry POINT and dimensions XY -

```
class(pnt1)
## [1] "XY"     "POINT"  "sfg"
```

Geometry column (**sfc**)

- ▶ Let's create a second object named `pnt2`, representing a different point -

```
pnt2 = st_point(c(34.798443, 31.243288))
```

- ▶ Geometry objects (`sfg`) can be *collected* into an **sfc** (**geometry column**) objects
- ▶ This is done with function `st_sfc`

Geometry column (**sfc**)

- ▶ **Geometry column** objects contain a **Coordinate Reference System (CRS)** definition, specified with the `crs`
- ▶ The CRS can be **specified** in one of two ways -
 - ▶ A definition in the **PROJ.4** format ("`+proj=longlat +datum=WGS84`")
 - ▶ An **EPSG** code (4326)
- ▶ Let's combine the two POINT geometries `pnt1` and `pnt2` into an **sfc (geometry column)** object `pnt` -

```
geom = st_sfc(pnt1, pnt2, crs = 4326)
```

Geometry column (`sfc`)

- ▶ Here is a summary of the resulting geometry column -

```
geom
## Geometry set for 2 features
## geometry type:  POINT
## dimension:      XY
## bbox:            xmin: 34.79844 ymin: 31.24329 xmax: 34.812831 ymax: 31.260284
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## POINT(34.812831 31.260284)
## POINT(34.798443 31.243288)
```

Layer (`sf`)

- ▶ A `sfc` (**geometry column**) can be combined with non-spatial columns (*attributes*) resulting in an `sf` (**layer**) object
- ▶ In our case the two points in the `sfc` (**geometry column**) `pnt` represent the location of the two railway station in Beer-Sheva
- ▶ Let's create a `data.frame` with a `name` column specifying station name
- ▶ Note that the order of attributes must match the order of the geometries

Layer (**sf**)

- ▶ Creating the attribute table -

```
dat = data.frame(  
  name = c("Beer-Sheva North", "Beer-Sheva Center")  
)
```

```
dat  
##           name  
## 1 Beer-Sheva North  
## 2 Beer-Sheva Center
```

Layer (sf)

- ▶ And combining the **attribute table** with the **geometry column** -

```
pnt = st_sf(dat, geom)
```

```
pnt
## Simple feature collection with 2 features and 1 field
## geometry type: POINT
## dimension: XY
## bbox: xmin: 34.79844 ymin: 31.24329 xmax: 34.812831 ymax: 31.260284
## epsg (SRID): 4326
## proj4string: +proj=longlat +datum=WGS84 +no_defs
##           name          geom
## 1 Beer-Sheva North POINT(34.812831 31.260284)
## 2 Beer-Sheva Center POINT(34.798443 31.243288)
```

Extracting layer components

- ▶ In the last few slides we -
 - ▶ Started from raw **coordinates**
 - ▶ Converted them to **geometry** objects (sfg)
 - ▶ Combined the geometries to a **geometry column** (sfc)
 - ▶ Added attributes to the geometry column to get a **layer** (sf)
- ▶ Sometimes we are interested in the opposite process
- ▶ We may need to extract the simpler components (**geometry**, **attributes**, **coordinates**) from an existing layer

Extracting layer components

- ▶ The **sfc (geometry column)** component can be extracted from an **sf (layer)** object using function **st_geometry** -

```
st_geometry(pnt)
## Geometry set for 2 features
## geometry type:  POINT
## dimension:      XY
## bbox:            xmin: 34.79844 ymin: 31.24329 xmax: 34.812831 ymax: 31.260284
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## POINT(34.812831 31.260284)
## POINT(34.798443 31.243288)
```

Extracting layer components

- ▶ The non-spatial columns of an `sf` (**layer**), i.e. the **attribute table**, can be extracted from an `sf` object using function `st_set_geometry` and `NULL` -

```
st_set_geometry(pnt, NULL)
##                 name
## 1 Beer-Sheva North
## 2 Beer-Sheva Center
```

Extracting layer components

- ▶ The **coordinates** (matrix object) of **sf**, **sfc** or **sfg** objects can be obtained with function **st_coordinates** -

```
st_coordinates(pnt)
##           X           Y
## 1 34.81283 31.26028
## 2 34.79844 31.24329
```

Interactive mapping with mapview

- ▶ Function `mapview` is useful for inspecting spatial data -

```
library(mapview)  
mapview(pnt)
```

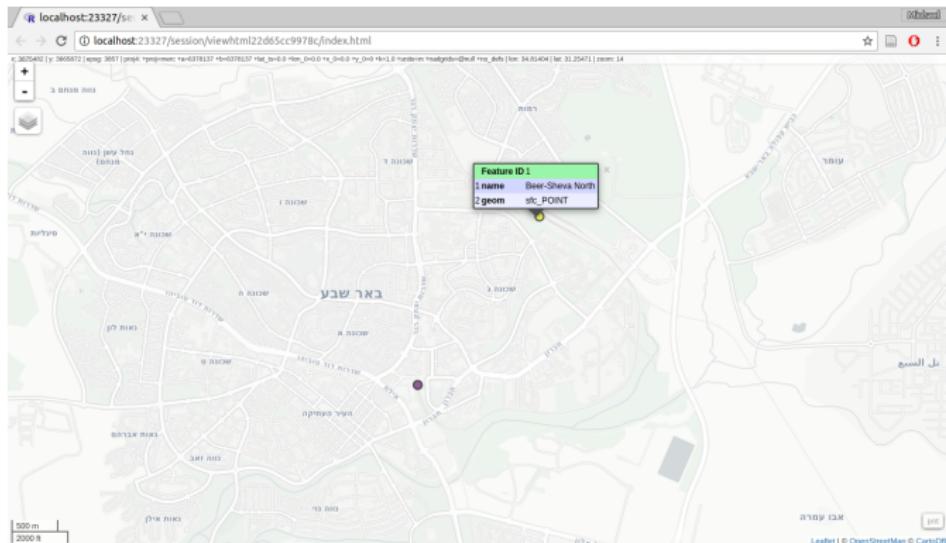


Figure 10: Intractive map of pnt layer

Part III: Advanced sf

Reading layers into R

- ▶ Here are the driver names of some commonly used **vector layer formats** that can be read into R -
 - ▶ ESRI Shapefile
 - ▶ GeoJSON
 - ▶ GPX
 - ▶ KML
- ▶ Note that it is also possible to read / write to **spatial databases** such as -
 - ▶ PostgreSQL/PostGIS
 - ▶ SQLite/Spatialite
- ▶ For complete list of **available drivers** -

```
View(st_drivers(what = "vector"))
```

Reading layers into R

- ▶ In the following examples, we will use two vector layers -
 - ▶ **US state borders**
 - ▶ **Storm tracks**
- ▶ We will import both from **Shapefiles**
 - ▶ Download a ZIP archive with both layers from [here](#)
 - ▶ Extract the file contents into a new directory
 - ▶ Use `setwd` to change the **Working Directory**

```
setwd("C:/Tutorials/sf")
```

Reading layers into R

- ▶ Next we use `st_read` function to **read the layer**
- ▶ In case the Shapefile is located in the Working Directory we just need to specify the **name of the shp file**
- ▶ We can also specify `stringsAsFactors = FALSE` to **avoid conversion** of character to factor

```
states = st_read(  
  dsn = "cb_2015_us_state_5m.shp",  
  stringsAsFactors = FALSE  
)  
  
tracks = st_read(  
  dsn = "hurmjrl020.shp",  
  stringsAsFactors = FALSE  
)
```

Basic plotting

- ▶ states is a **polygonal** layer containing **US states borders**
- ▶ It has **49 features** and **1 attribute** -
 - ▶ state = State name

```
dim(states)
## [1] 49  2
```

```
plot(states, key.pos = NULL)
```

Basic plotting

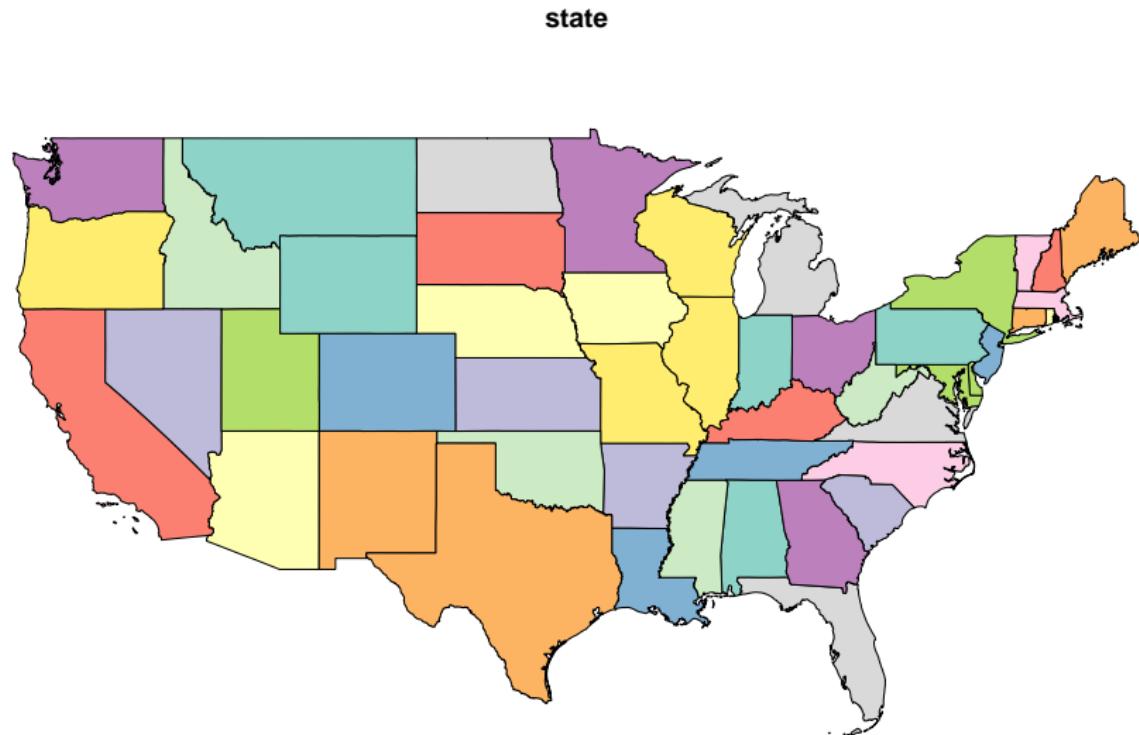


Figure 11: The states' layer

Basic plotting

- ▶ tracks is a **line** layer with **storm trajectories**
- ▶ Each feature represents a storm **segment**
- ▶ It has **4056 features** and **7 attributes** -
 - ▶ btid = Track ID
 - ▶ year = Year
 - ▶ month = Month
 - ▶ day = Day
 - ▶ name = Storm name
 - ▶ wind_mph = Wind speed (miles / hour)
 - ▶ category = Storm category (e.g. H5 = Category 5 Hurricane)

```
dim(tracks)
## [1] 4056     8
```

```
plot(tracks)
```

Basic plotting

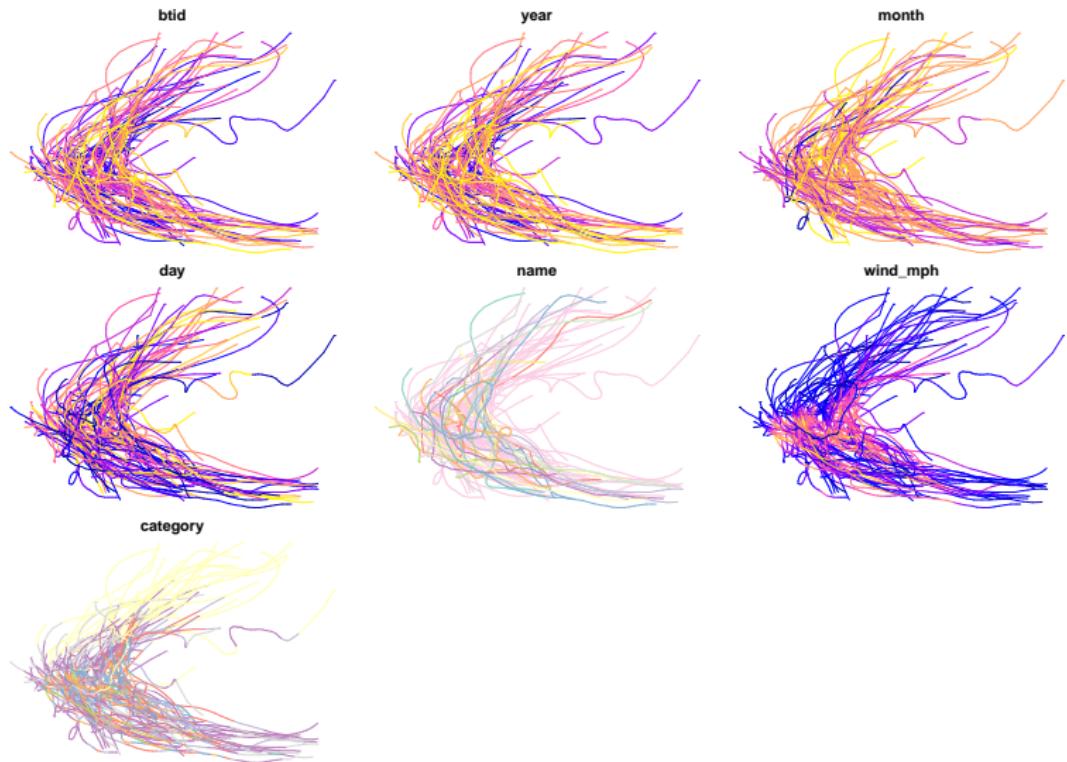


Figure 12: The tracks layer

Basic plotting

- ▶ When we are plotting an sfc **geometry column**, the plot only displays the geometric shape
- ▶ We can use **basic graphical parameters** to control the appearance, such as -
 - ▶ col - Fill color
 - ▶ border - Outline color
 - ▶ pch - Point shape
 - ▶ cex - Point size
- ▶ For example, to draw **states outline in grey** -

```
plot(st_geometry(states), border = "grey")
```

Basic plotting

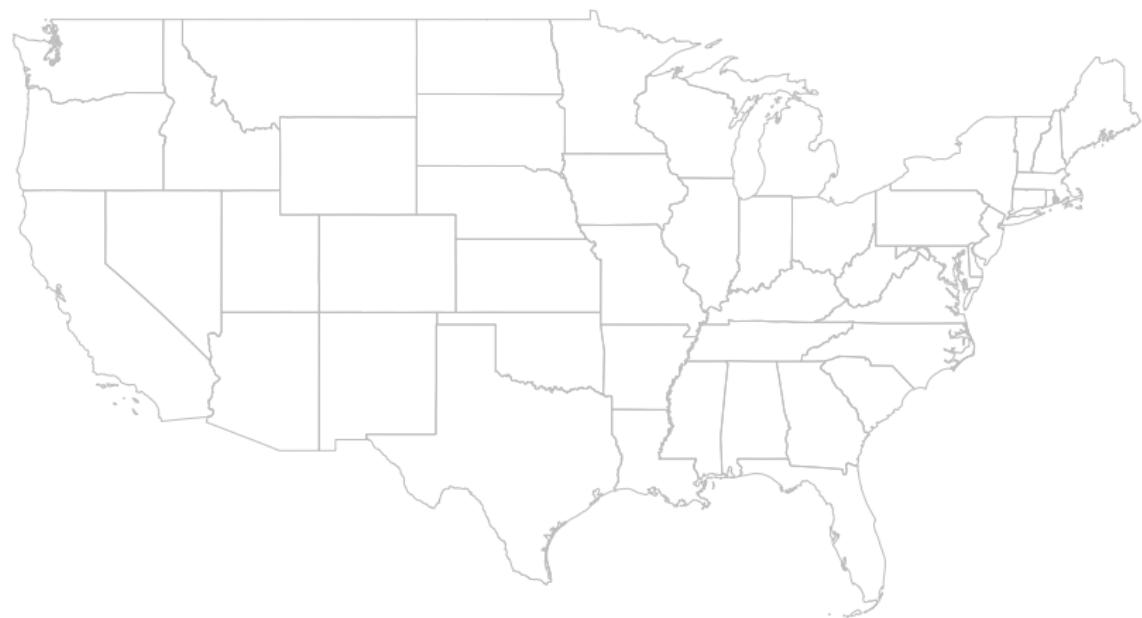


Figure 13: Basic plot of `sfc` object

Basic plotting

- ▶ Additional vector layers can be **drawn** in an **existing graphical window** with add=TRUE
- ▶ For example, the following expressions draw **both** tracks and states
- ▶ Note that the second expression uses add=TRUE

```
plot(st_geometry(states), border = "grey")
plot(st_geometry(tracks), col = "red", add = TRUE)
```

Basic plotting



Figure 14: using `add=TRUE` in plot

Interactive map with mapview

```
mapview(tracks, zcol = "wind_mph", legend = TRUE)
```

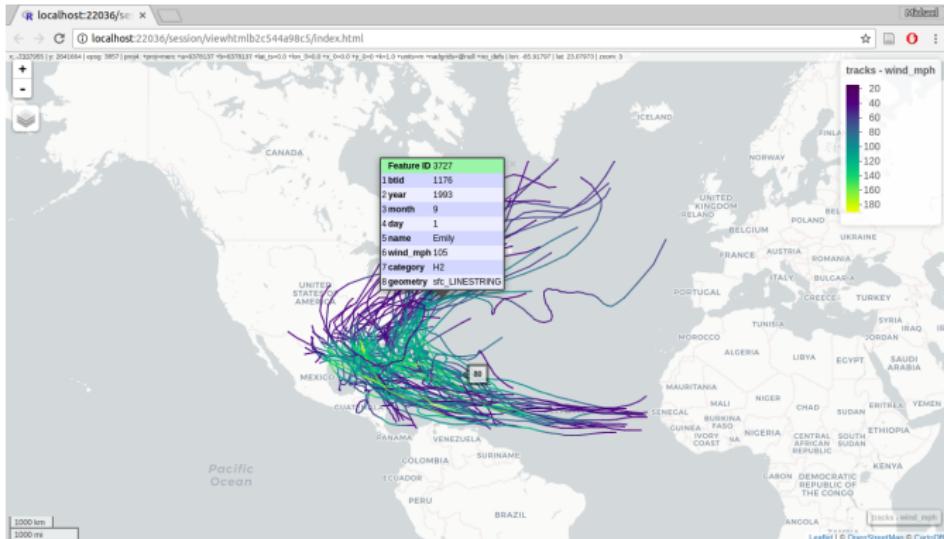


Figure 15: Intractive map of tracks layer

Subsetting

- ▶ **Subsetting** (filtering) of features in an `sf` vector layer is done in exactly the same way as filtering rows in a `data.frame`
- ▶ For example, the following expression filters the `tracks` layer to keep only those storms **which took place after 1949**

```
tracks = tracks[tracks$year > 1949, ]
```

Reprojection

- ▶ **Reprojection** is an important part of spatial analysis workflow since as we often need to -
 - ▶ Transform several layers into the same projection
 - ▶ Switch between un-projected and projected data
- ▶ A vector layer can be reprojected with `st_transform`
- ▶ `st_transform` has **two parameters** -
 - ▶ `x` - The **layer** to be reprojected
 - ▶ `crs` - The **target CRS**
- ▶ The CRS can be **specified** in one of two ways -
 - ▶ A definition in the **PROJ.4** format ("`+proj=longlat +datum=WGS84`")
 - ▶ An **EPSG** code (4326)

Reprojection

- ▶ In the following code section we are reprojecting both the states and tracks layers
- ▶ The **target CRS** is the *US National Atlas Equal Area* projection (EPSG=2163)

```
states = st_transform(states, 2163)
tracks = st_transform(tracks, 2163)
```

Reprojection

state

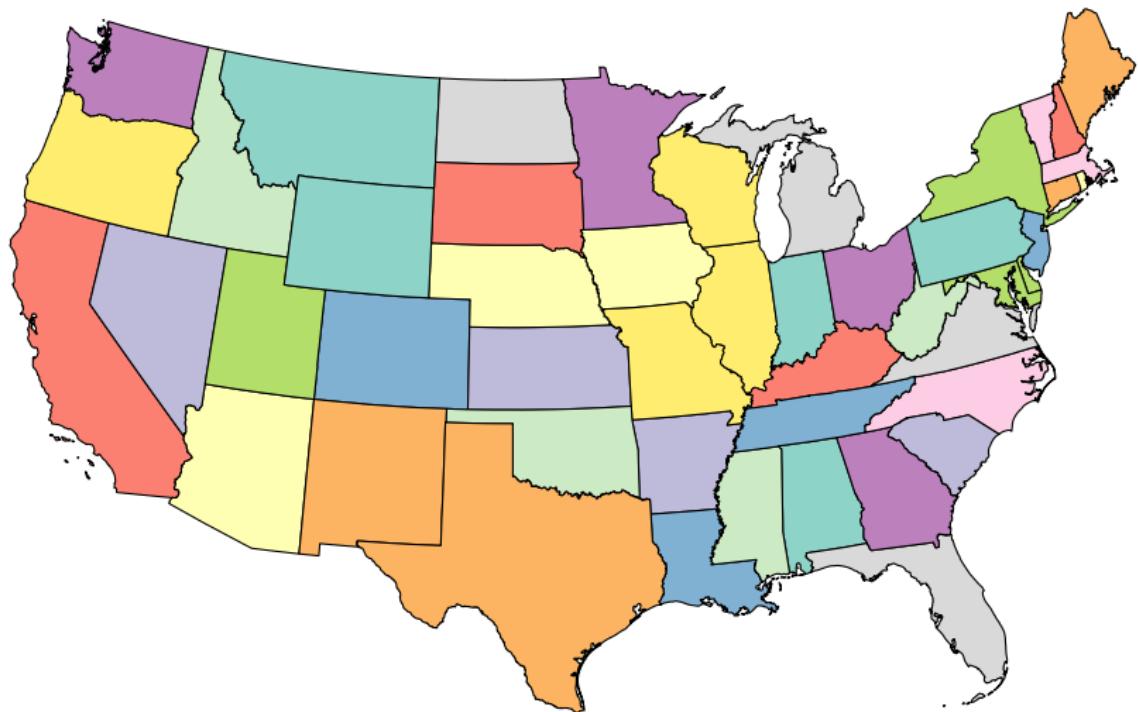


Figure 16: Reprojected states layer

Geometric calculations

Geometric operations on vector layers can conceptually be divided into **three groups** according to their output -

- ▶ **Numeric** values: Functions that summarize geometrical properties of -
 - ▶ A **single layer** (e.g. area, length)
 - ▶ A **pair of layers** (e.g. distance)
- ▶ **Logical** values: Functions that evaluate whether a certain condition holds true, regarding -
 - ▶ A **single layer** (e.g. geometry is valid)
 - ▶ A **pair of layers** (e.g. feature A intersects feature B)
- ▶ **Spatial** layers: Functions that create a new layer based on -
 - ▶ A **single layer** (e.g. centroids)
 - ▶ A **pair of layers** (e.g. intersection area)

Numeric

- ▶ There are several functions to calculate **numeric geometric properties** of vector layers in package sf -
 - ▶ `st_length`
 - ▶ `st_area`
 - ▶ `st_distance`
 - ▶ `st_bbox`
 - ▶ `st_dimension`

Numeric

- ▶ For example, we can calculate the area of each feature in the states layer (i.e. each state) using `st_area` -

```
area = st_area(states)
area[1:3]
## Units: m^2
## [1] 134050489381 295336534486 137732515283
```

- ▶ The result is an object of class `units` -

```
class(area)
## [1] "units"
```

Numeric

- ▶ We can convert measurements to different units with `set_units` from package `units` -

```
library(units)

states$area_km2 = set_units(area, km^2)
states$area_km2[1:3]
## Units: km^2
## [1] 134050.5 295336.5 137732.5
```

Numeric

area_km2

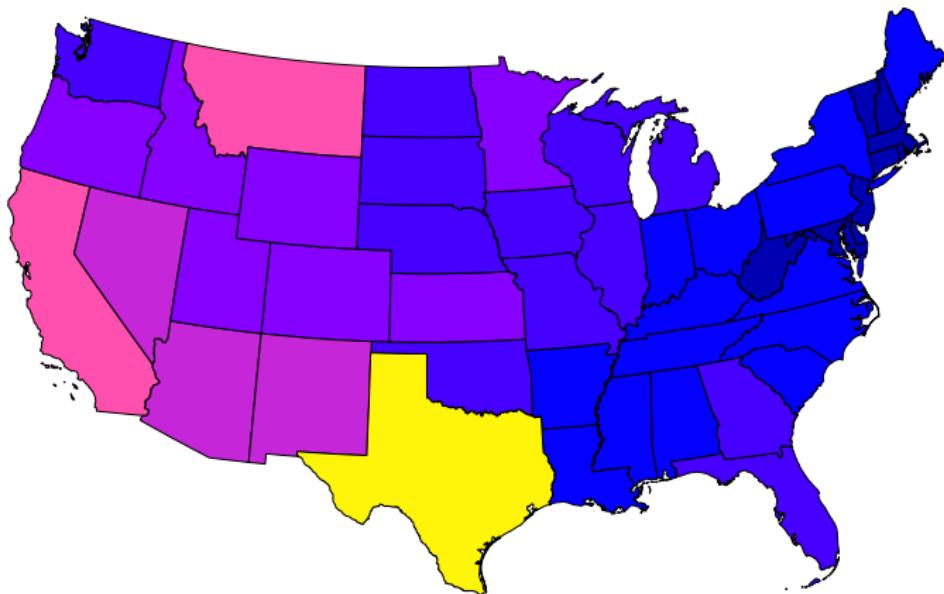


Figure 17: Calculated area_km2 attribute

Logical

- ▶ Given two layers, x and y, the following **logical geometric functions** check whether each feature in x maintains the specified **relation** with each feature in y -
 - ▶ st_intersects
 - ▶ st_disjoint
 - ▶ st_touches
 - ▶ st_crosses
 - ▶ st_within
 - ▶ st_contains
 - ▶ st_overlaps
 - ▶ st_covers
 - ▶ st_covered_by
 - ▶ st_equals
 - ▶ st_equals_exact

Logical

- ▶ When specifying `sparse=FALSE` the functions return a **logical matrix**
- ▶ Each **element** i, j in the matrix is TRUE when $f(x[i], y[j])$ is TRUE
- ▶ For example, this creates a matrix of intersection relations between states -

```
int = st_intersects(states, states, sparse = FALSE)
```

```
int[1:4, 1:4]
##      [,1]  [,2]  [,3]  [,4]
## [1,] TRUE FALSE FALSE FALSE
## [2,] FALSE TRUE FALSE  TRUE
## [3,] FALSE FALSE  TRUE FALSE
## [4,] FALSE  TRUE FALSE  TRUE
```

Logical

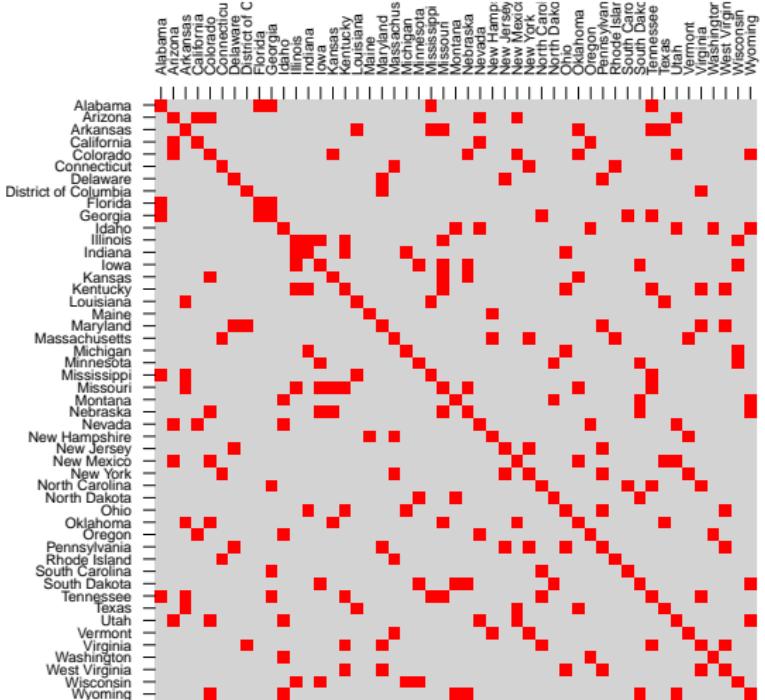


Figure 18: Intersection relations between states features

Spatial

- ▶ sf provides common **geometry-generating** functions applicable to **individual** geometries, such as -
 - ▶ st_centroid
 - ▶ st_buffer
 - ▶ st_sample
 - ▶ st_convex_hull
 - ▶ st_voronoi

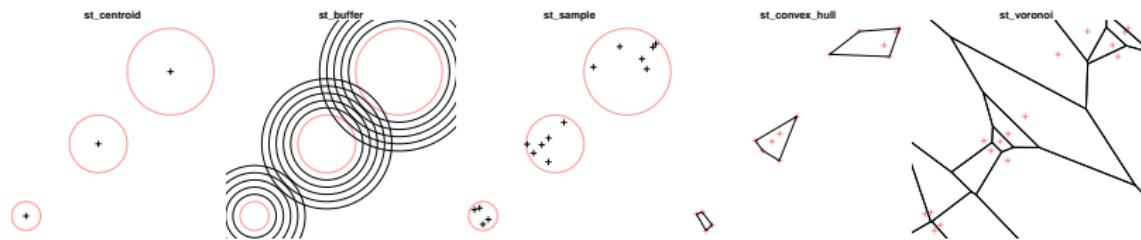


Figure 19: Geometry-generating operations on individual layers

Spatial

- ▶ For example, the following expression uses `st_centroid` to create a layer of **state centroids** -

```
states_ctr = st_centroid(states)
```

- ▶ They can be **plotted** as follows -

```
plot(st_geometry(states))
plot(st_geometry(states_ctr), col = "red", add = TRUE)
```

Spatial

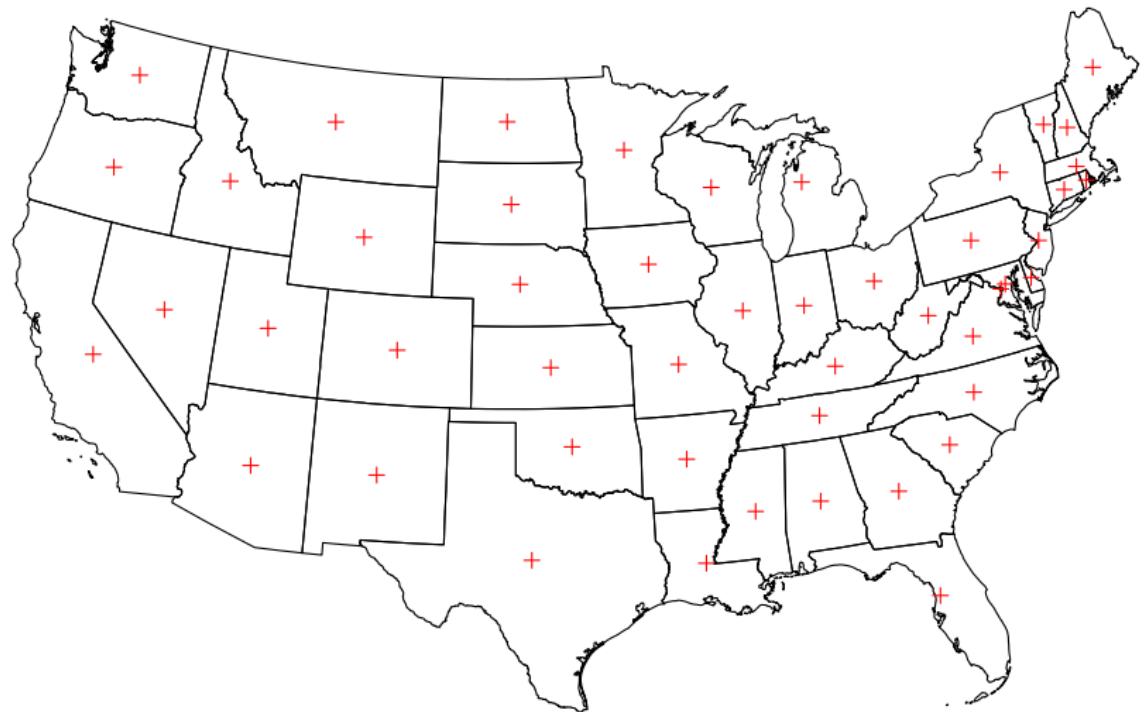


Figure 20: State centroids

Spatial

- ▶ Other **geometry-generating** functions work on **pairs** of input geometries -
 - ▶ `st_intersection`
 - ▶ `st_difference`
 - ▶ `st_sym_difference`
 - ▶ `st_union`

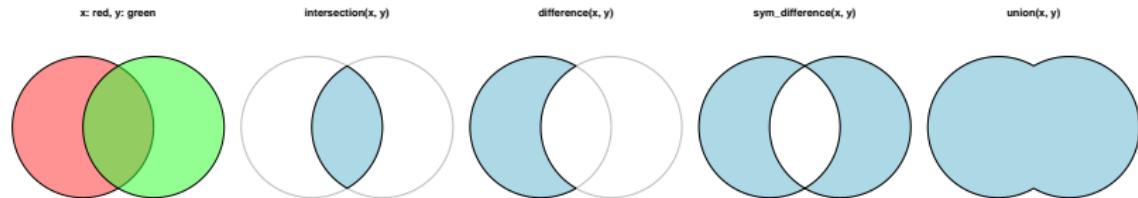


Figure 21: Geometry-generating operations on pairs of layers

Spatial

- ▶ For example, to calculate **total tracks length per state** we can use `st_intersection` to ‘split’ the tracks layer by State

-

```
tracks_int = st_intersection(tracks, states)
```

Spatial

- ▶ The result is a new line layer split by state borders and including a state attribute -

```
plot(  
  st_geometry(states),  
  border = "grey"  
)  
  
plot(  
  tracks_int[, "state"],  
  add = TRUE,  
  lwd = 3,  
  key.pos = NULL  
)
```

Spatial



Figure 22: Intersection result

Spatial

- ▶ The resulting layer has mixed LINESTRING and MULTILINESTRING geometries (Why?)

```
class(tracks_int$geometry)
## [1] "sfc_GEOMETRY" "sfc"
```

- ▶ To calculate line length we need to convert it to MULTILINESTRING -

```
tracks_int = st_cast(tracks_int, "MULTILINESTRING")
```

- ▶ Verifying the conversion succeeded -

```
class(tracks_int$geometry)
## [1] "sfc_MULTILINESTRING" "sfc"
```

Spatial

- ▶ Let's add a **storm track length** attribute called `length` -

```
tracks_int$length = st_length(tracks_int)
tracks_int$length = set_units(tracks_int$length, km)
```

Join layer with table

- ▶ Next we aggregate attribute table of tracks_int by state, to find the sum of length values -

```
library(dplyr)

track_lengths =
  tracks_int %>%
  st_set_geometry(NULL) %>%
  group_by(state) %>%
  summarize(length = sum(length)) %>%
  as.data.frame
```

Join layer with table

- ▶ The result is a `data.frame` with **total length** of storm tracks **per state** -

```
head(track_lengths)
##           state      length
## 1      Alabama 2272.1309 km
## 2      Arkansas 1000.0448 km
## 3 Connecticut  290.7856 km
## 4    Delaware   58.2246 km
## 5     Florida 2932.1240 km
## 6    Georgia 1505.7847 km
```

Join layer with table

- ▶ Next, we can **join** the aggregated table back to the states layer -

```
states = merge(  
  states,  
  track_lengths,  
  by = "state",  
  all.x = TRUE  
)
```

Join layer with table

- ▶ Here is the states layer summary after the join -

```
head(states)
## Simple feature collection with 6 features and 3 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:            xmin: -2031905 ymin: -1562374 xmax: 2295
## epsg (SRID):   2163
## proj4string:   +proj=laea +lat_0=45 +lon_0=-100 +x_0=0
##                 state       area_km2      length
## 1    Alabama  134050.49 km^2 2272.1309 km MULTIPOLYGON(0
## 2    Arizona  295336.53 km^2        NA km MULTIPOLYGON(0
## 3   Arkansas  137732.52 km^2 1000.0448 km MULTIPOLYGON(0
## 4 California 409701.73 km^2        NA km MULTIPOLYGON(0
## 5 Colorado  269373.19 km^2        NA km MULTIPOLYGON(0
## 6 Connecticut 12912.34 km^2  290.7856 km MULTIPOLYGON(0
```

Join layer with table

- ▶ Finally, we **divide** total track length by state area
- ▶ This gives us track **density** per state -

```
states$track_density =  
  states$length / states$area_km2
```

- ▶ Plotting the layer shows the new `track_density` attribute -

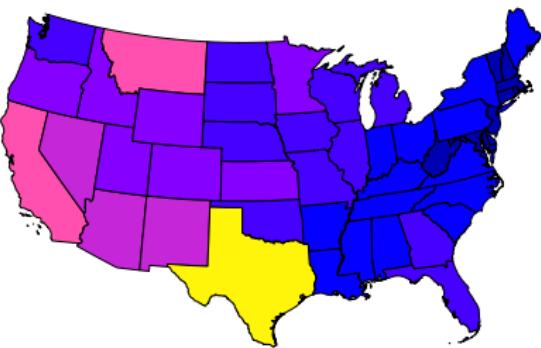
```
plot(states)
```

Join layer with table

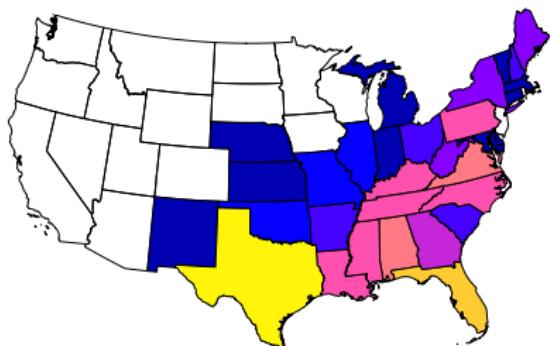
state



area_km2



length



track_density

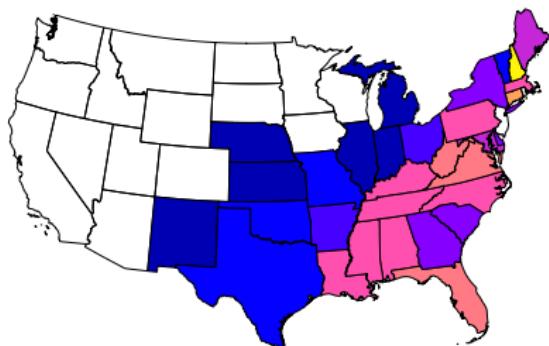


Figure 23: States layer with track density attribute

Thank you for listening!

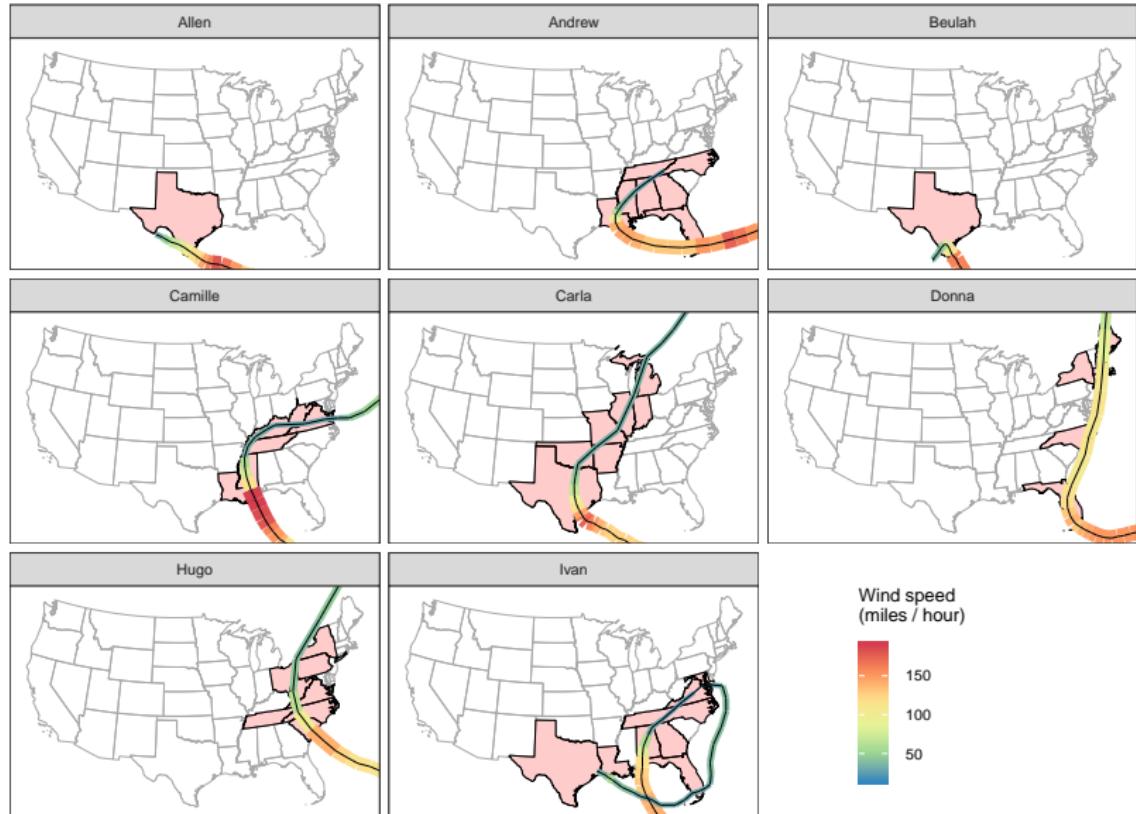


Figure 24: Level 5 hurricanes, wind speed and affected states