Introduction to Vector Data Processing in R

Earth Observation Workshop, Basel

Michael Dorman

Geography and Environmental Development

dorman@post.bgu.ac.il

2018-02-26 - 2018-03-01

Ben-Gurion University of the Negev

Contents

- This tutorial is a practical introduction to working with **vector layers** in R using the **`sf` package**
- **Slides and code** available on GitHub
- **Data** available on Dropbox

Contents

Requirements

- Several packages **need to be installed** to run code examples -

```
install.packages("sf")
install.packages("mapview")
install.packages("units")
install.packages("dplyr")
devtools::install_github("tidyverse/ggplot2")
```

- Set to **working directory** with the data -
  - `cb_2015_us_state_5m` - **US states (Shapefile)**
  - `hurmjrl020` - **Storm tracks (Shapefile)**

Part I: 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`

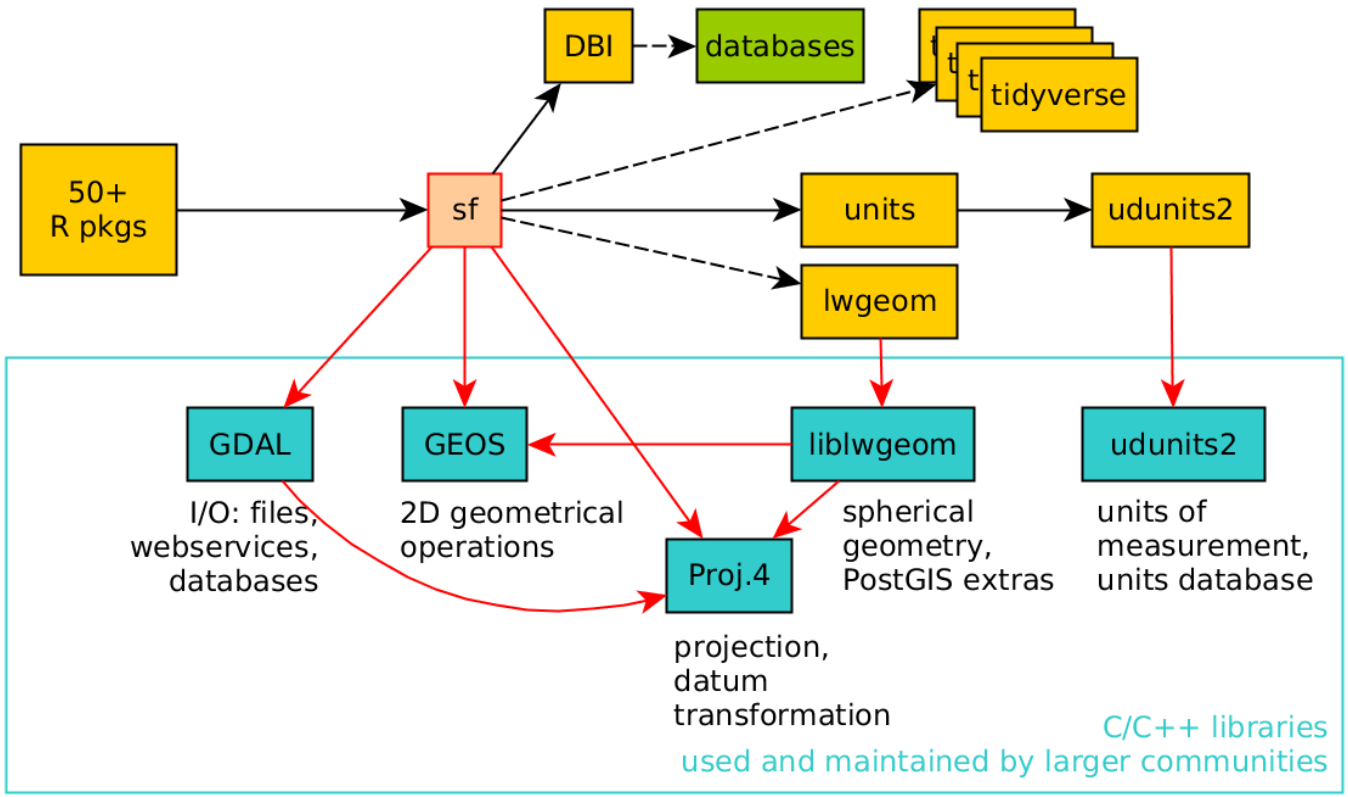

Figure 1: `sf` package dependencies[1]

[1] https://github.com/edzer/rstudio_conf

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...
```

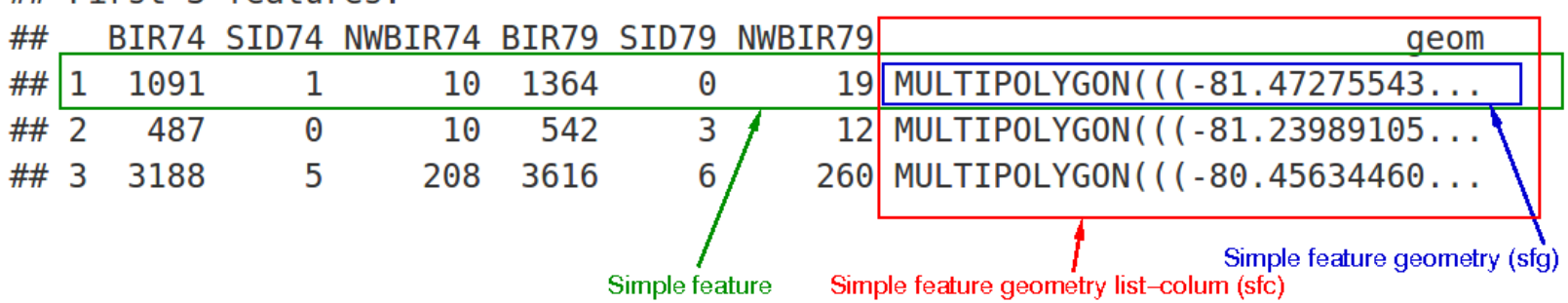

Figure 2: Structure of an `sf` object[2]

[2] 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–colum (sfc)

Simple feature geometry (sfg)

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 2.2.2, proj.4 4.9.2
```

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** - `POINT`
  - `matrix` objects - `MULTIPOINT` or `LINESTRING`
  - `list` objects - All other geometries

Geometry (sfg)

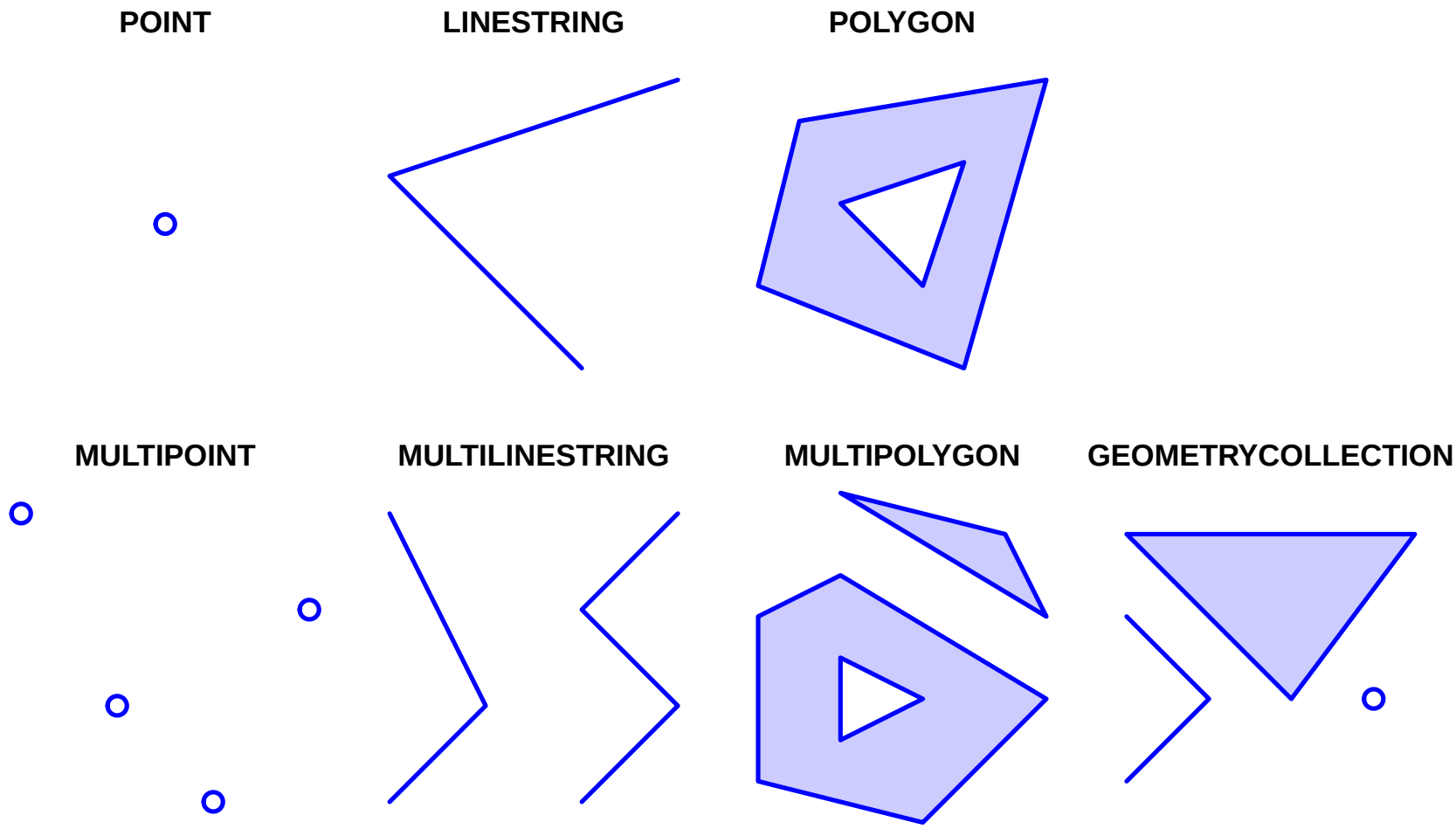

Figure 3: Simple feature geometry (sfg) types in package sf

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 (34.81283 31.26028)
```

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, the `pnt1` object 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)** object
- This is done with function `st_sfc`

Geometry column (`sfc`)

- **Geometry column** objects contain a **Coordinate Reference System (CRS)** definition, specified with `crs`
- Two types of arguments are accepted -
  - A **PROJ.4** definition (`"+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: 34.81283 ymax: 31.26028
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## POINT (34.81283 31.26028)
## POINT (34.79844 31.24329)
```

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 stations 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: 34.81283 ymax: 31.26028
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
##                name                        geom
## 1  Beer-Sheva North POINT (34.81283 31.26028)
## 2 Beer-Sheva Center POINT (34.79844 31.24329)
```

Extracting layer components

- In the last few slides we -
  - Started from raw **coordinates**
  - Convered 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: 34.81283 ymax: 31.26028
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## POINT (34.81283 31.26028)
## POINT (34.79844 31.24329)
```

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)
```

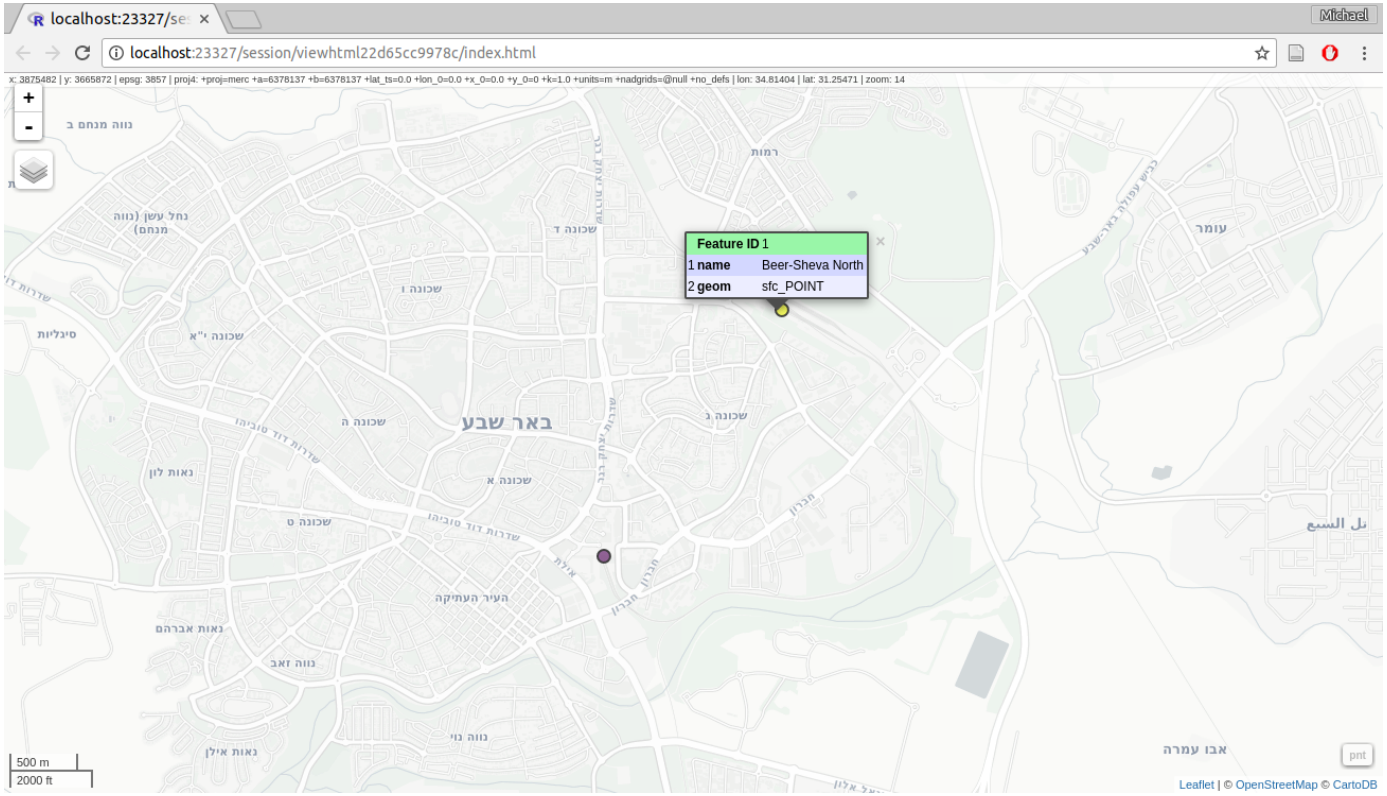

Figure 4: Intractive map of `pnt` layer

Part II: Advanced `sf`

Creating point layer from table

- A common way of creating a point layer is to transform a **table** which has **X and Y coordinate columns**
- Function `st_as_sf` can convert a table (`data.frame`) into a point layer (`sf`)
- In `st_as_sf` we specify -
  - `x` - The **`data.frame`** to be converted
  - `coords` - **Columns names** with the **coordinates (X, Y)**
  - `crs` - The **CRS** (`NA` if left unspecified)

Creating point layer from table

- For example, R has a built-in `data.frame` named **`quakes`**
- `quakes` gives the locations of of **1000 earthquakes near Fiji**
- See **`?quakes`** for more information

```
head(quakes)
##      lat   long depth mag stations
## 1 -20.42 181.62   562 4.8       41
## 2 -20.62 181.03   650 4.2       15
## 3 -26.00 184.10    42 5.4       43
## 4 -17.97 181.66   626 4.1       19
## 5 -20.42 181.96   649 4.0       11
## 6 -19.68 184.31   195 4.0       12
```

Creating point layer from table

- The `quakes` table has two columns **`long`** and **`lat`** specifing the **longitude** and **latitude** of each earthquake
- We can **convert** the table into a point layer with `st_as_sf`

```
quakes_pnt = st_as_sf(
  x = quakes,
  coords = c("long", "lat"),
  crs = 4326
)
```

- Note that the order of `coords` corresponds to X-Y!

Creating point layer from table

```
quakes_pnt
```

```
## Simple feature collection with 1000 features and 3 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 165.67 ymin: -38.59 xmax: 188.13 ymax: -10.72
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## First 10 features:
##    depth mag stations                geometry
## 1    562 4.8       41 POINT (181.62 -20.42)
## 2    650 4.2       15 POINT (181.03 -20.62)
## 3     42 5.4       43     POINT (184.1 -26)
## 4    626 4.1       19 POINT (181.66 -17.97)
## 5    649 4.0       11 POINT (181.96 -20.42)
## 6    195 4.0       12 POINT (184.31 -19.68)
## 7     82 4.8       43   POINT (166.1 -11.7)
## 8    194 4.4       15 POINT (181.93 -28.11)
## 9    211 4.7       35 POINT (181.74 -28.74)
## 10   622 4.3       19 POINT (179.59 -17.47)
```

Creating point layer from table

- When **plotting** an `sf` object we get multiple small maps, one for each attribute
- This can be useful to quickly examine the **types of spatial variation** in our data

```
plot(quakes_pnt)
```

Creating point layer from table

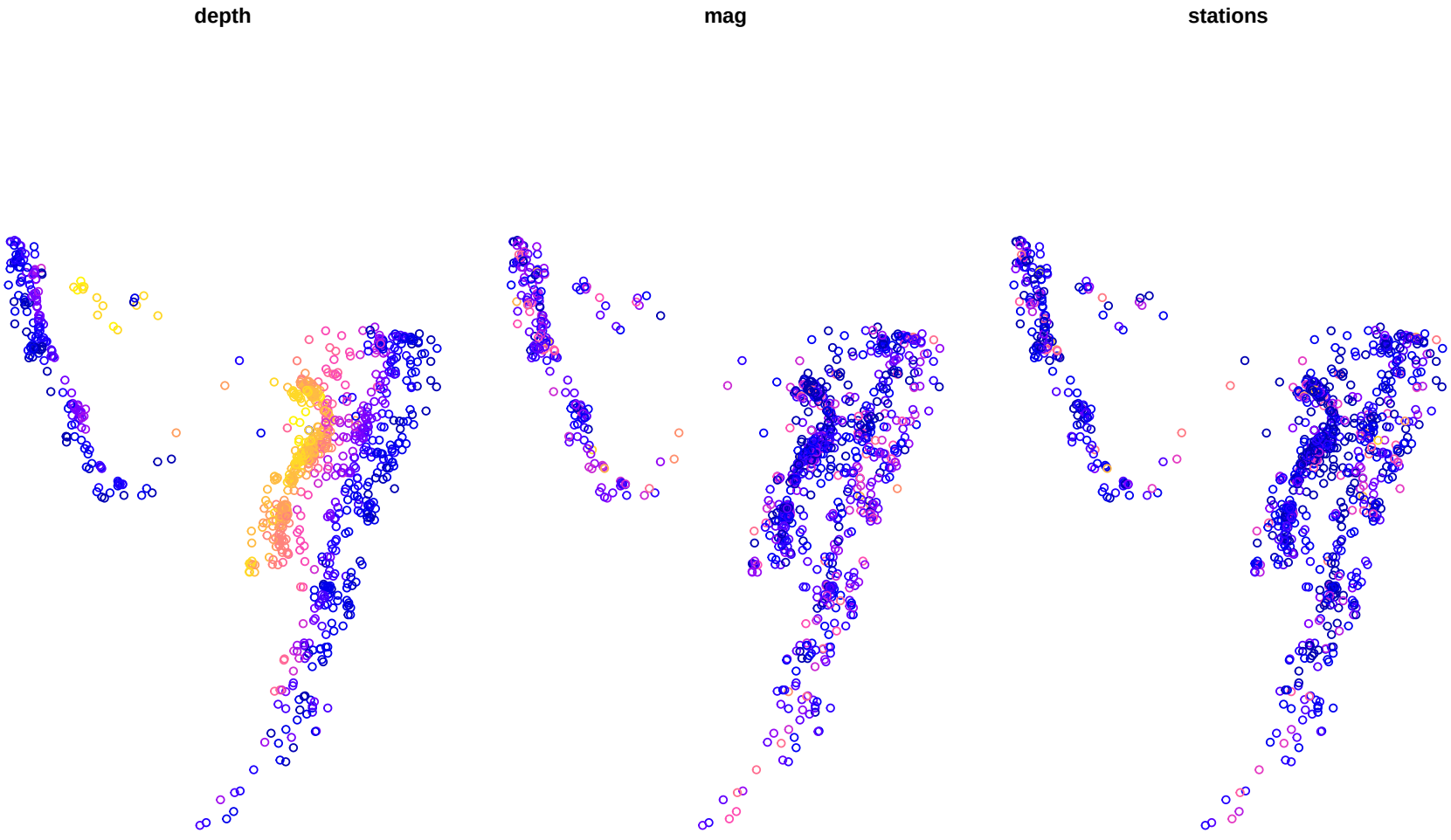

Figure 5: Point layer based on the `quakes` table

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)
```

Basic plotting

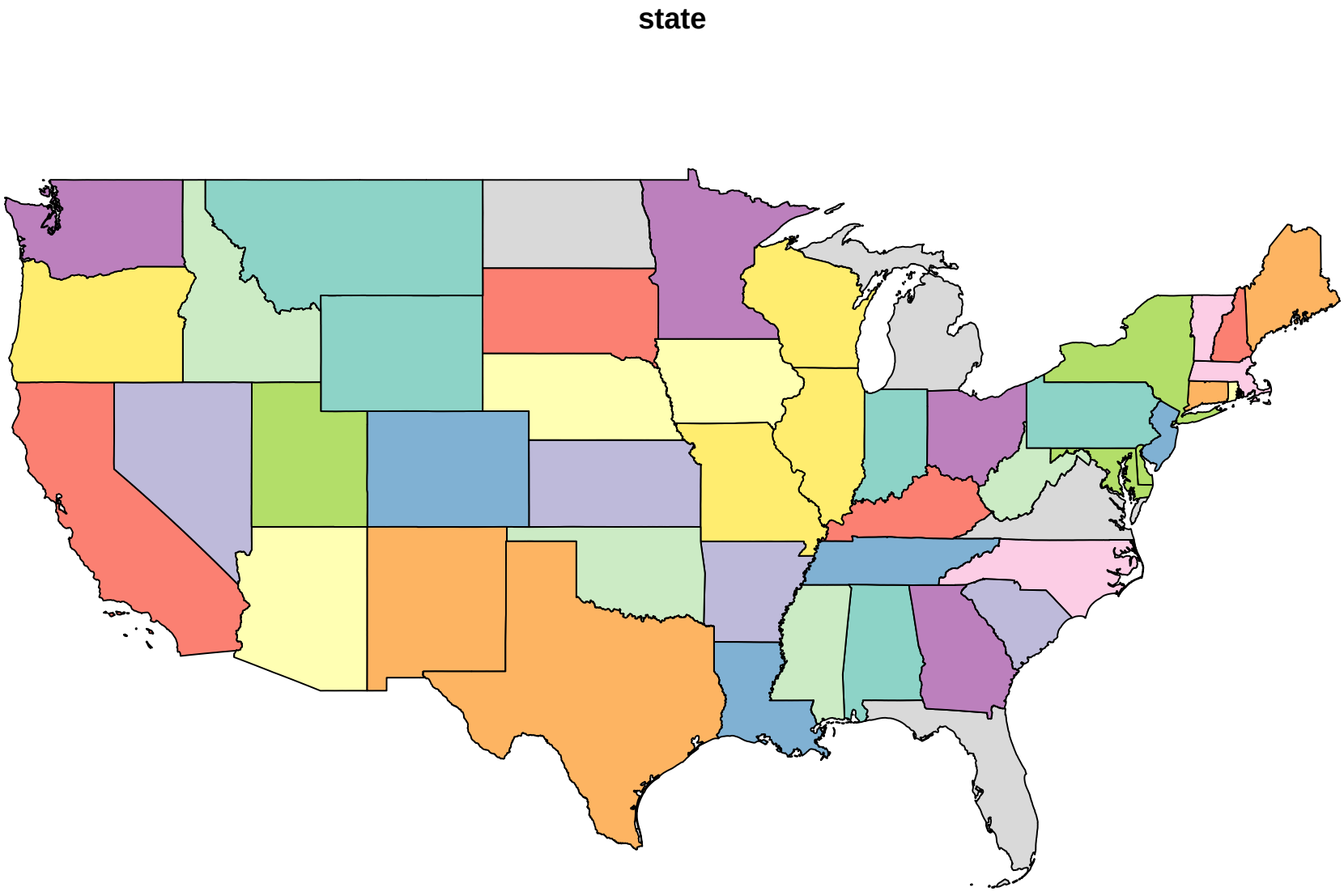

Figure 6: The `states` layer

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

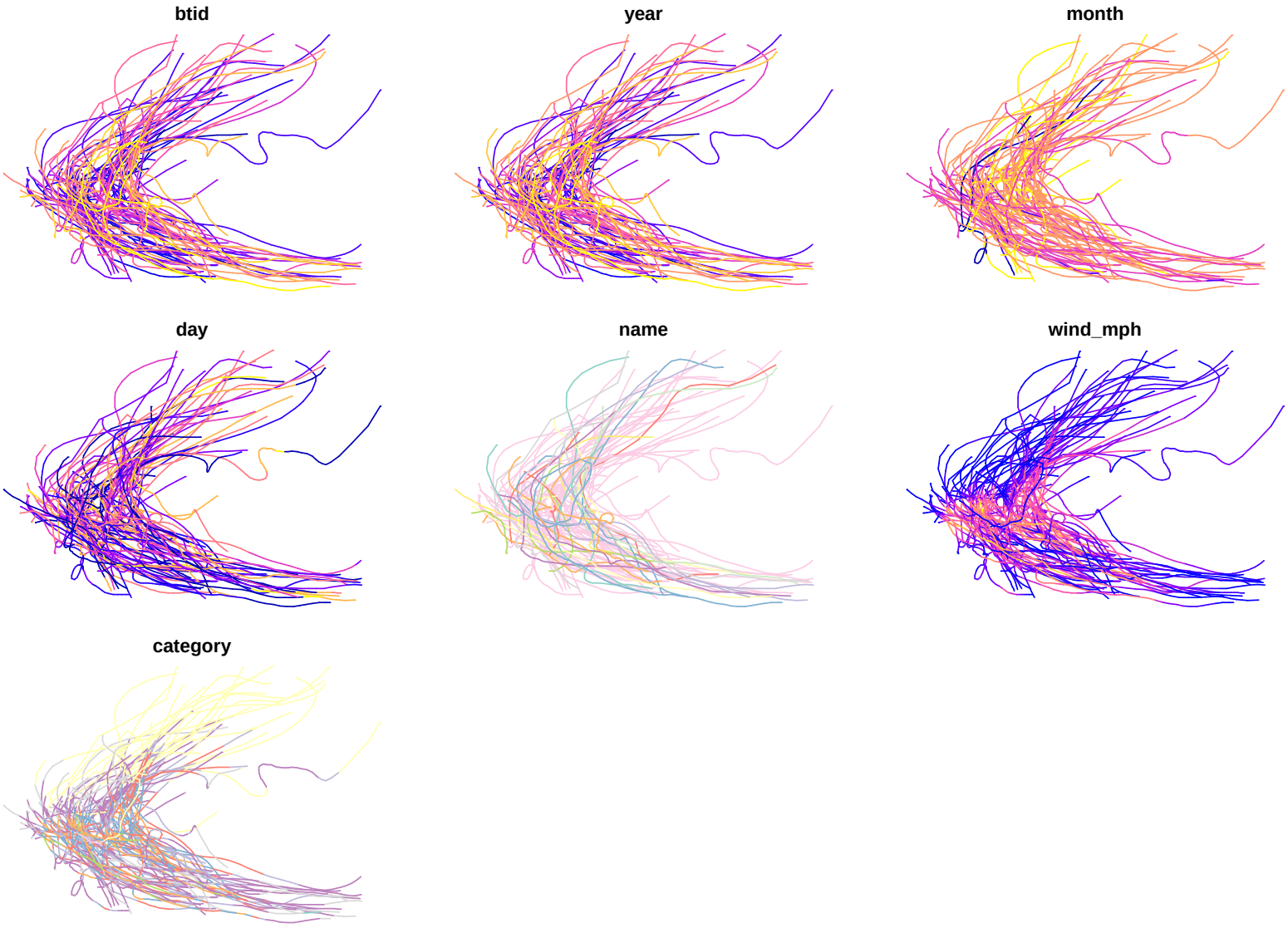

Figure 7: The `tracks` layer

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

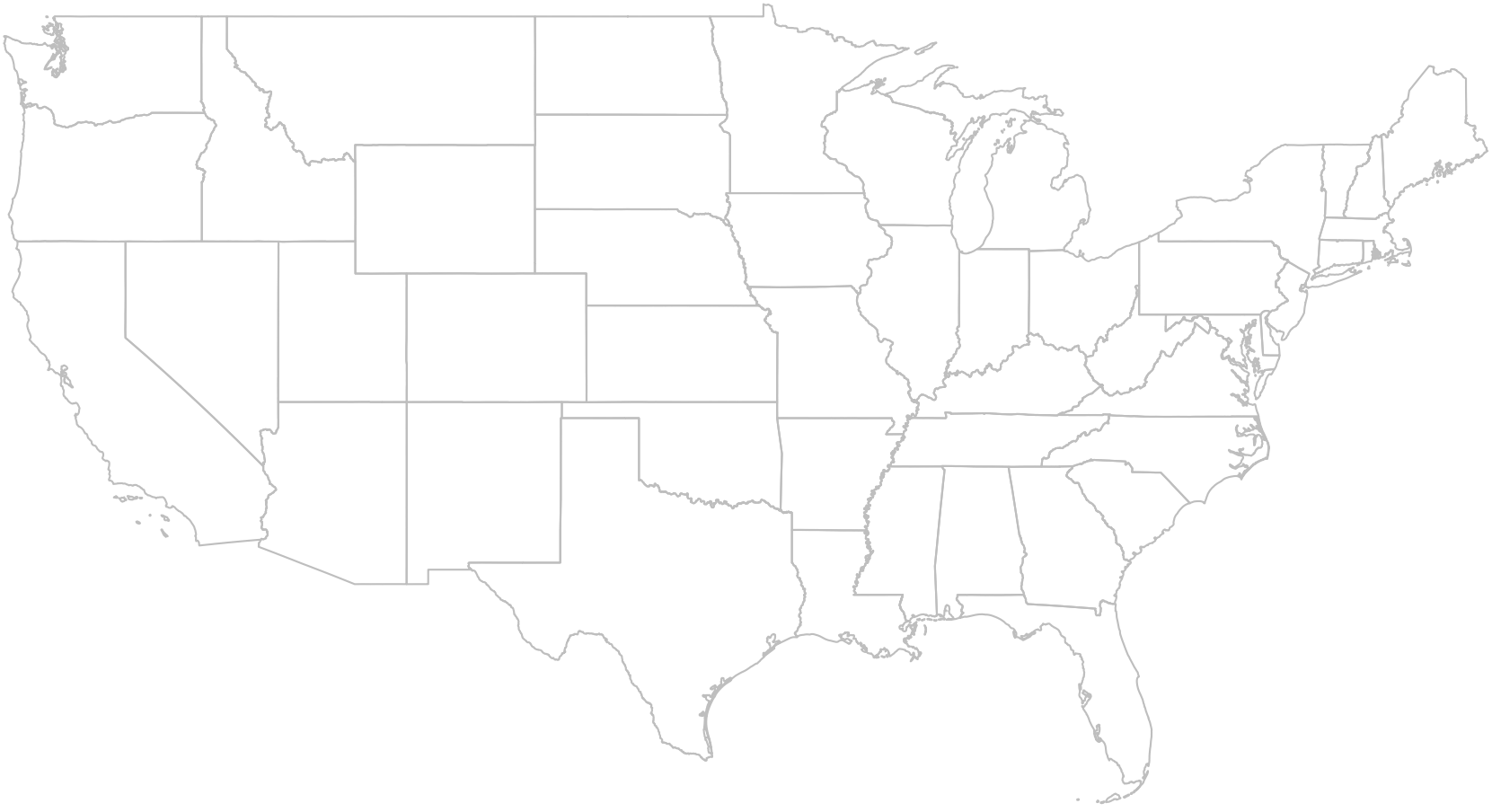

Figure 8: Basic plot of `sfc` object

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

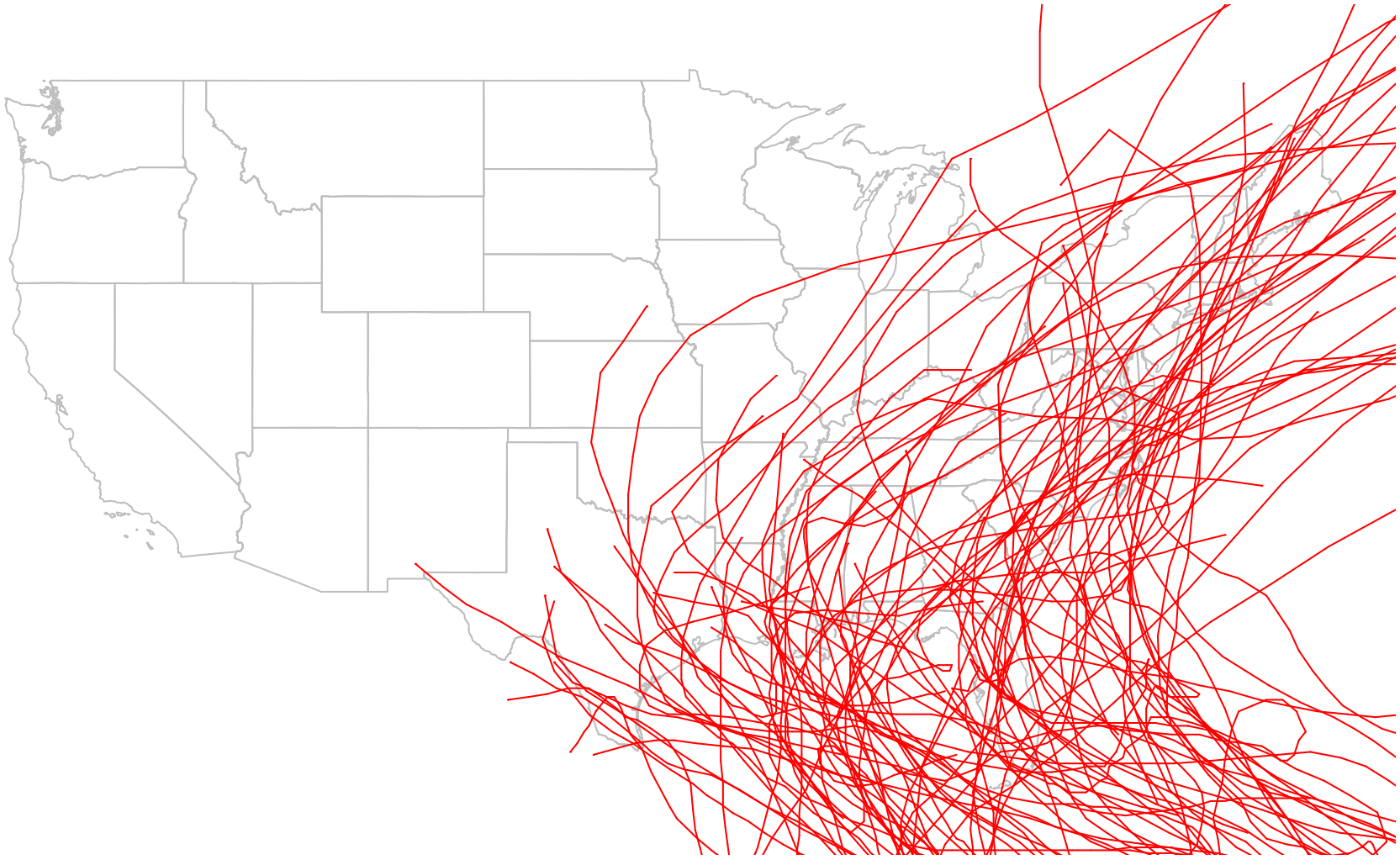

Figure 9: Using `add=TRUE` in `plot`

Interactive map with `mapview`

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

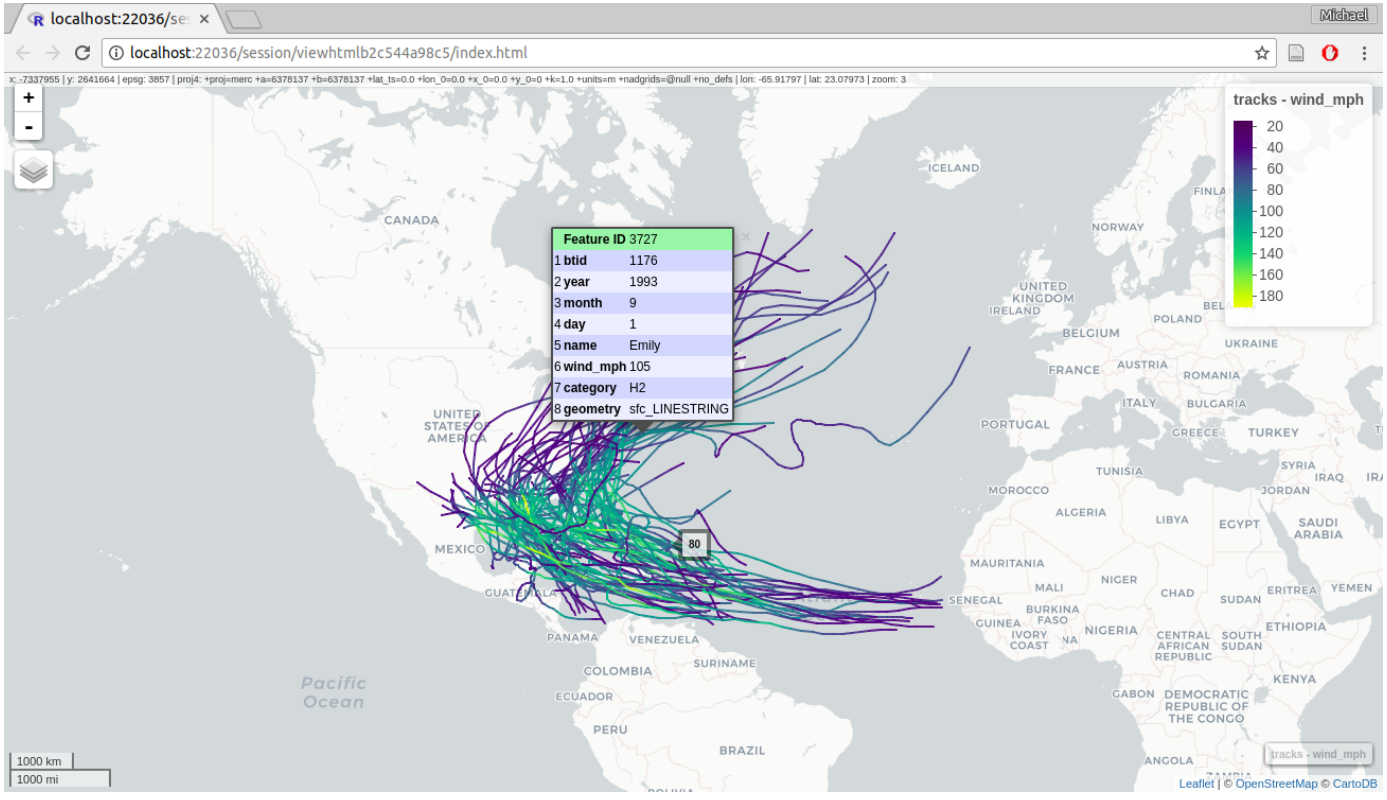

Figure 10: Intractive map of `tracks` layer

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 **PROJ.4** definition (`"+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

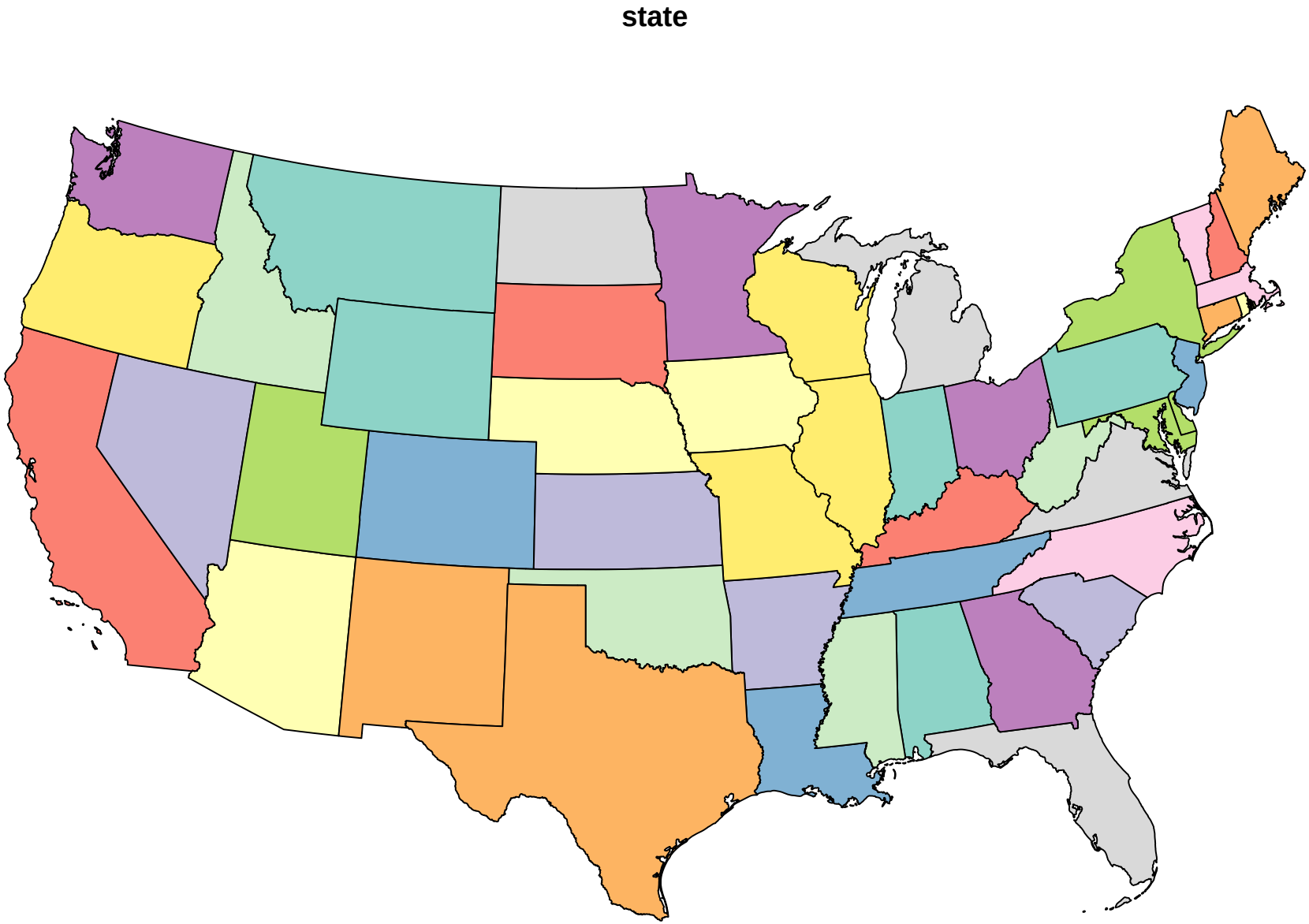

Figure 11: Reprojected `states` layer

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, ]
```

- As another example, we can create a `tracks_h5` layer with all segments that belong to storms which **reached the `"H5"` category**

```
strong = unique(tracks$btid[tracks$category == "H5"])
tracks_h5 = tracks[tracks$btid %in% strong, ]
```

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` -

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

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

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

Numeric

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

```
library(units)

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

- Inspecting result -

```
plot(states[, "area"])
```

Numeric

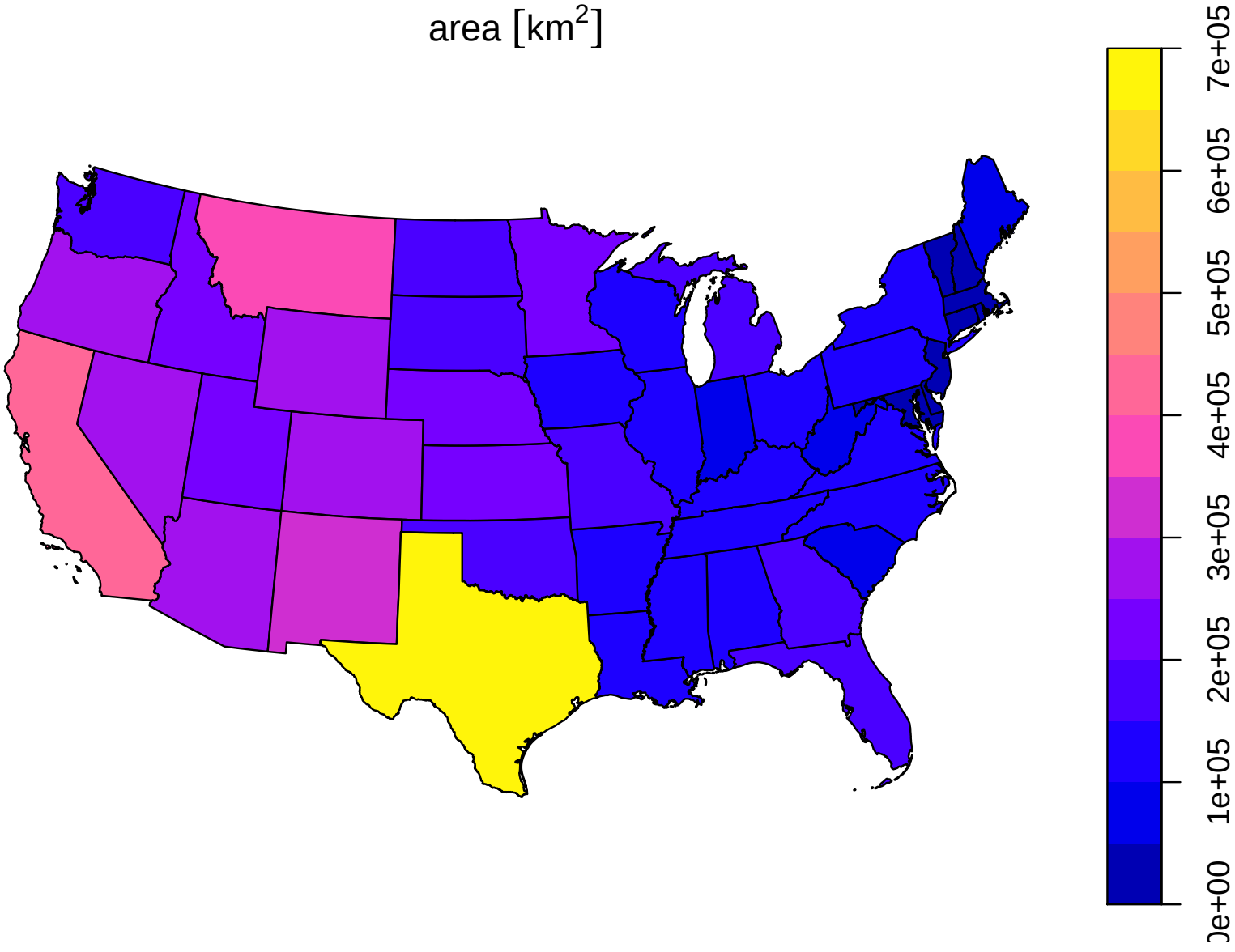

Figure 12: Calculated `area` attribute

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 13: 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`

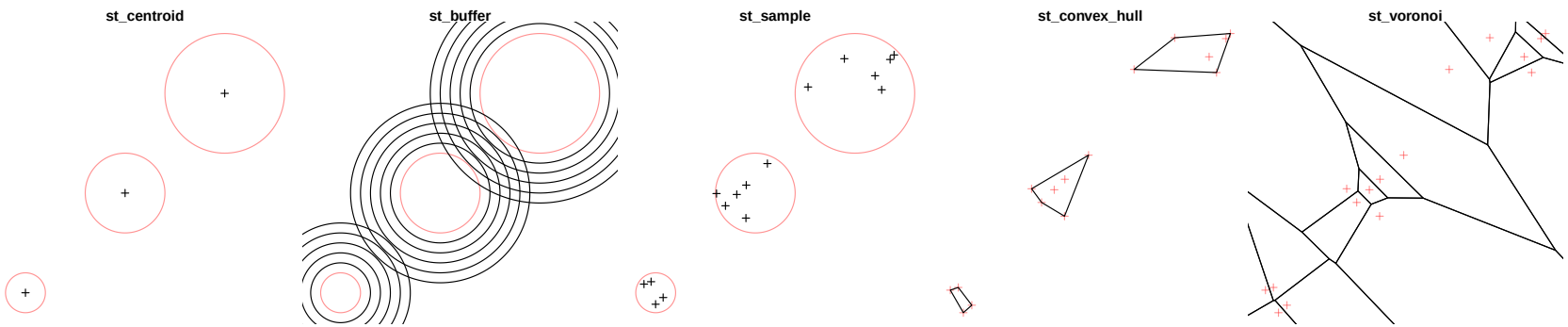

Figure 14: Geometry-generating operations on individual layers

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),
  border = "grey"
)
plot(
  st_geometry(states_ctr),
  col = "red", pch = 3,
  add = TRUE
)
```

Spatial

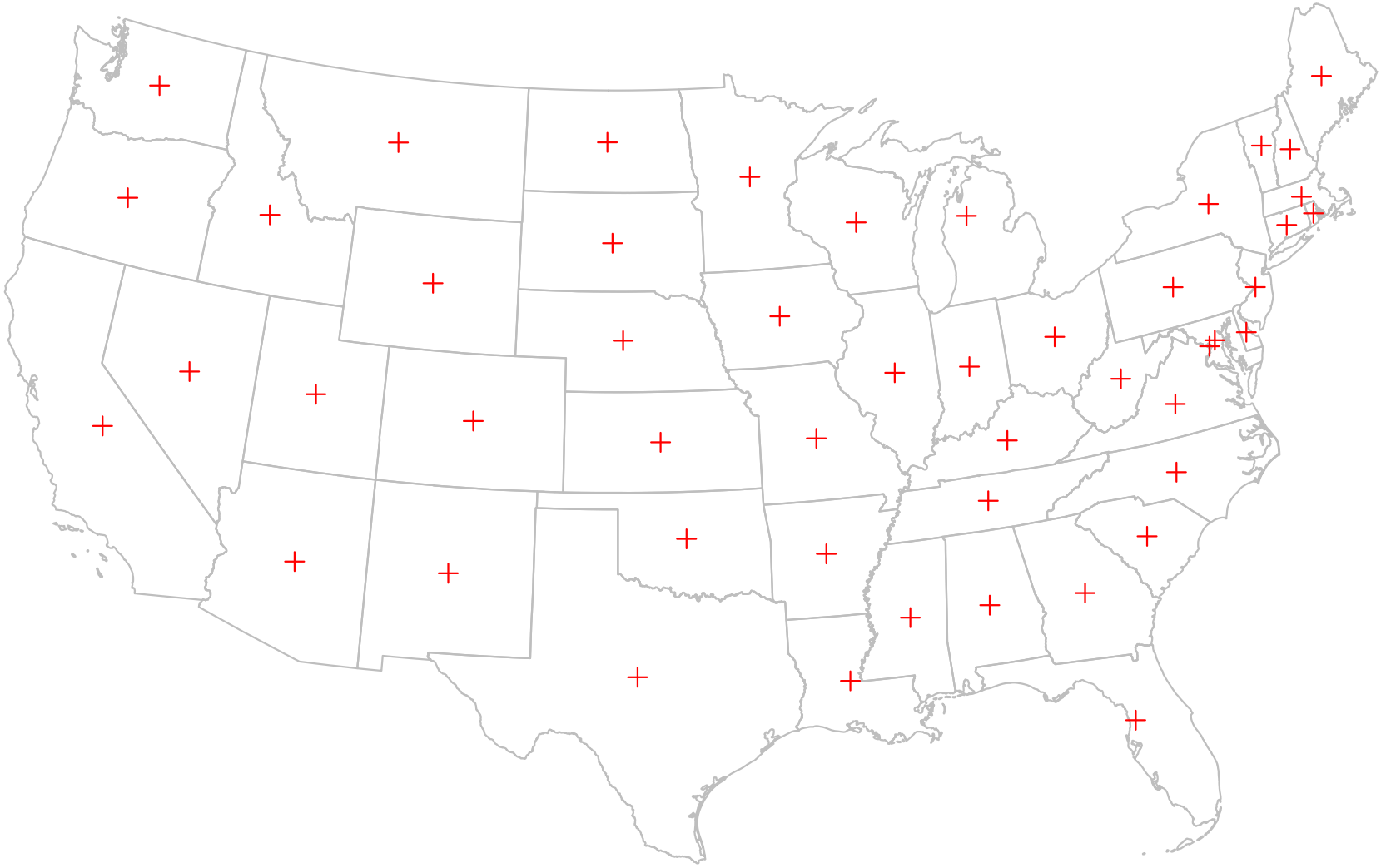

Figure 15: State centroids

Spatial

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

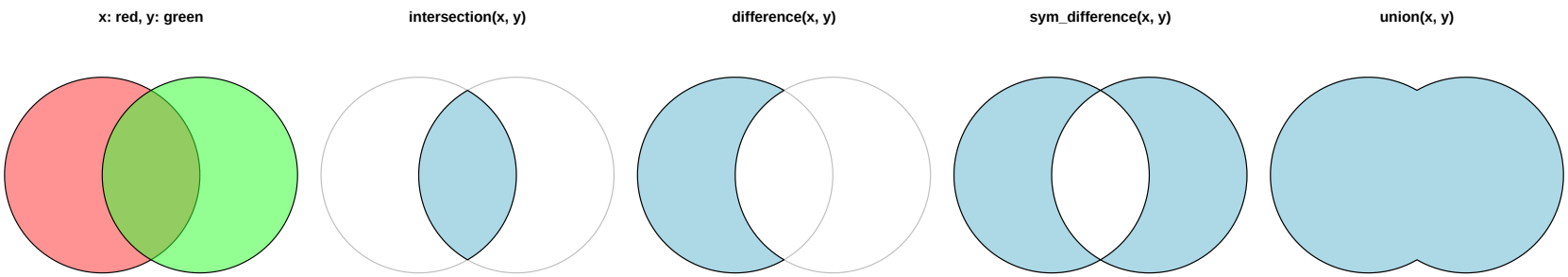

Figure 16: Geometry-generating operations on pairs of layers

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"],
  lwd = 3,
  add = TRUE
)
```

Spatial

Figure 17: Intersection result

Spatial

- The resulting layer has mixed `LINESTERING` 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 how the `states` layer looks like 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: 2295610 ymax: 67481.2
## epsg (SRID):    2163
## proj4string:    +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0 +a=6370997 +b
##         state           area       length                       geometry
## 1     Alabama 134050.49 km^2 2272.1309 km MULTIPOLYGON (((1150023 -15...
## 2     Arizona 295336.53 km^2        NA km MULTIPOLYGON (((-1386136 -1...
## 3    Arkansas 137732.52 km^2 1000.0448 km MULTIPOLYGON (((482001 -928...
## 4  California 409701.73 km^2        NA km MULTIPOLYGON (((-1717278 -1...
## 5    Colorado 269373.19 km^2        NA km MULTIPOLYGON (((-786661.9 -...
## 6 Connecticut  12912.34 km^2  290.7856 km MULTIPOLYGON (((2156197 -83...
```

Join layer with table

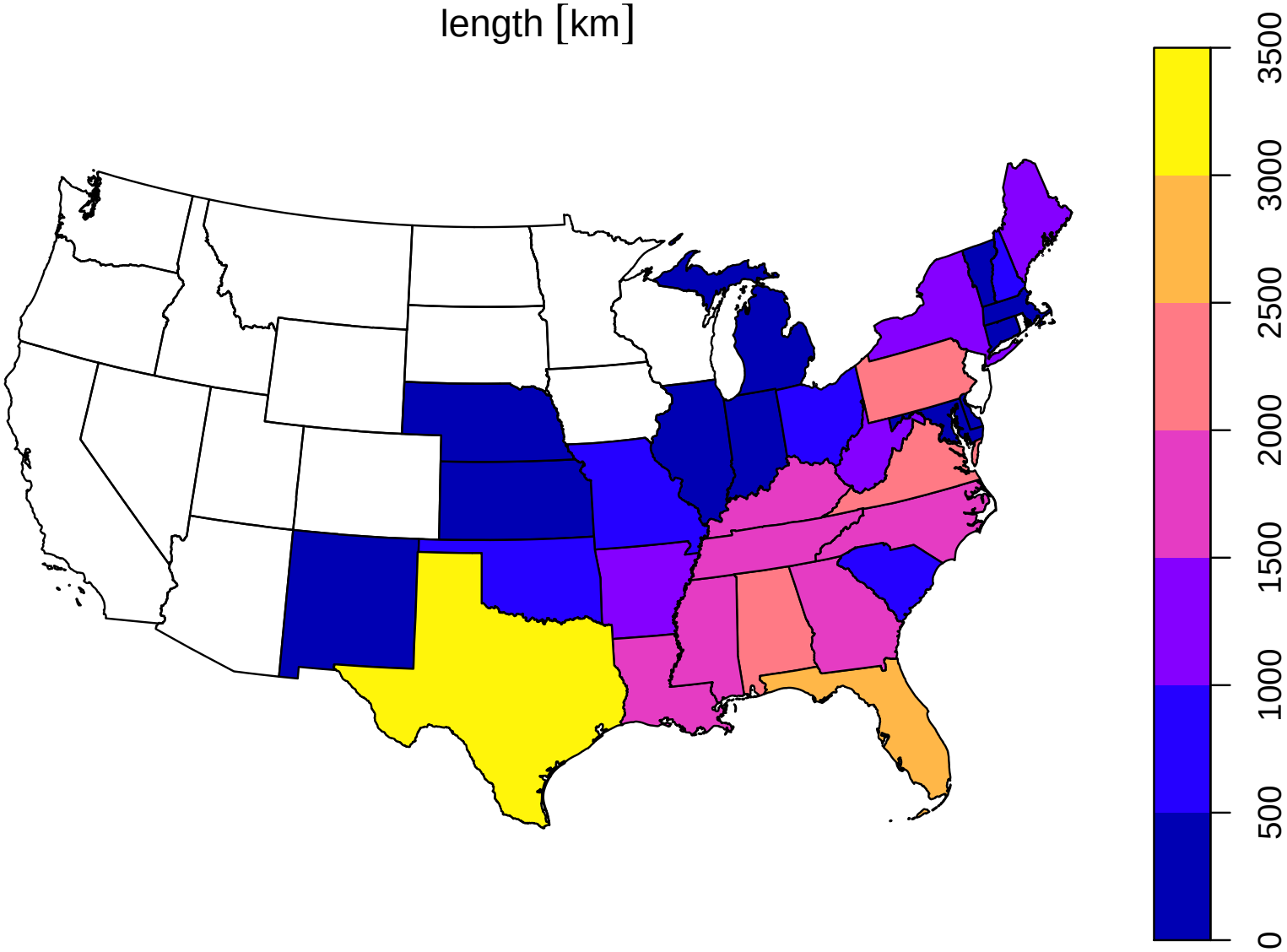

Figure 18: Total track length per state

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
```

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

```
plot(states)
```

Join layer with table

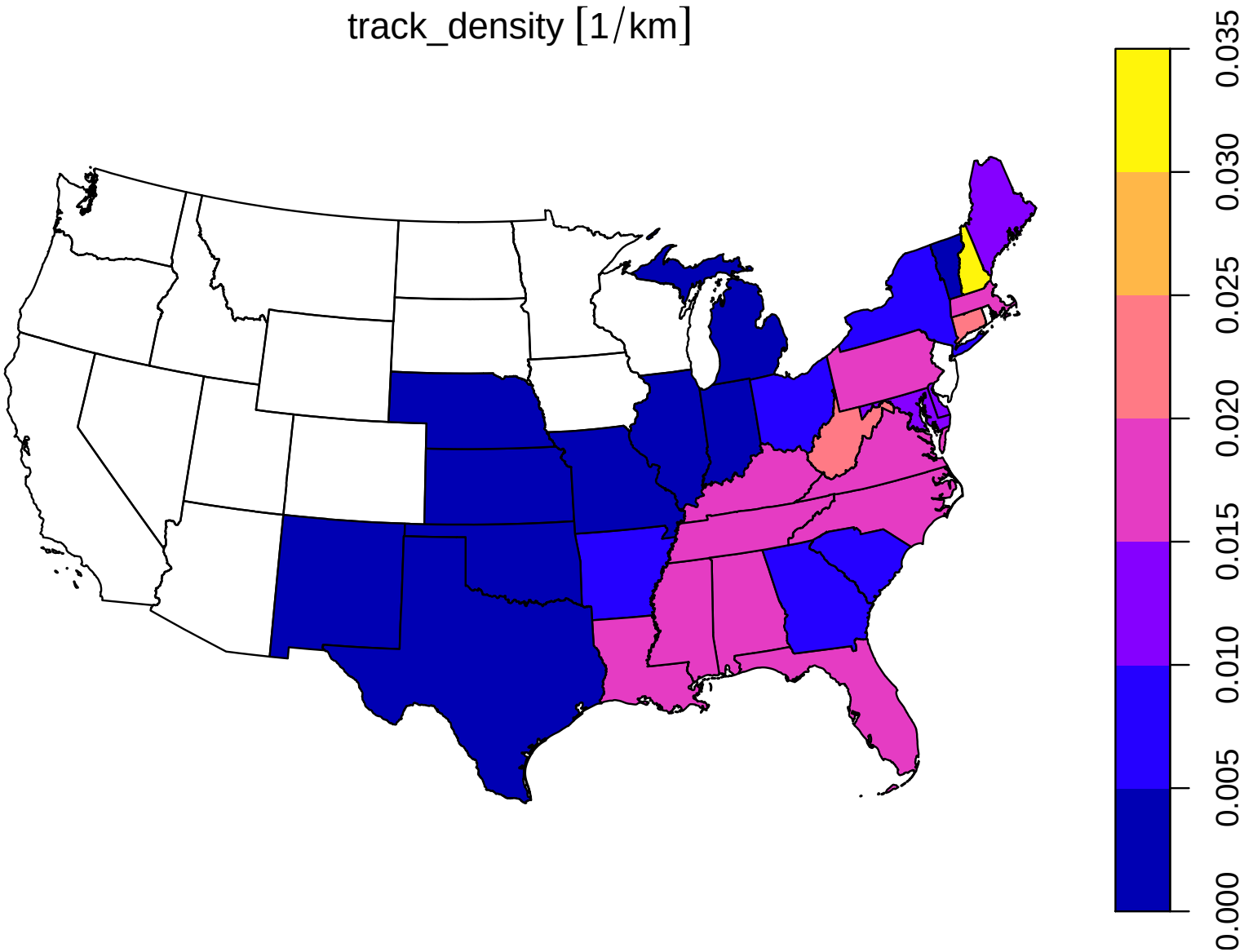

Figure 19: Track density per state

Join layer with table

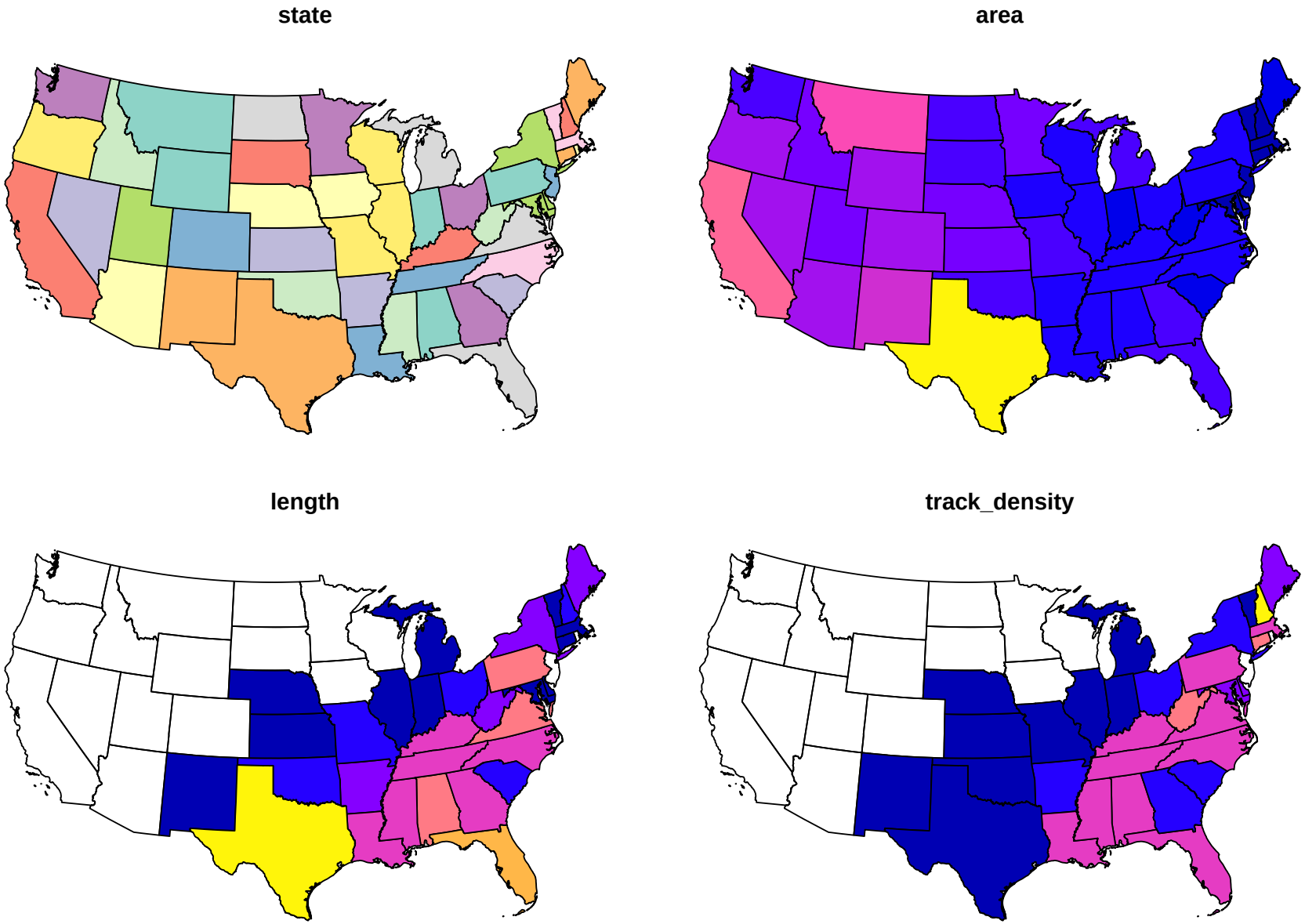

Figure 20: States layer with calculated attributes

Spatial Join

- One of the most useful operators in spatial analysis is the **spatial join**
- Function `st_join` accepts -
  - Two layers `x` and `y`
  - Geometric type `join` (default `st_intersects`)
  - Join type `left` (Left `TRUE`, Inner `FALSE`)
- For example, the following expression joins the `state` attribute of the state each track segment intersects -

```
tracks_states = st_join(
  states[, "state"],
  tracks[, "name"]
)
```

- The new layer has all `states` geometries, duplicated when necessary, with all respective `tracks` attributes for each state/track segment intersection

Spatial Join

```
tracks_states
```

```
## Simple feature collection with 397 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -2031905 ymin: -2116976 xmax: 2516374 ymax: 732352.2
## epsg (SRID):    2163
## proj4string:    +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0 +a=6370997 +b
## First 10 features:
##       state     name                           geometry
## 1   Alabama     Easy MULTIPOLYGON (((1150023 -15...
## 1.1 Alabama     Easy MULTIPOLYGON (((1150023 -15...
## 1.2 Alabama     Easy MULTIPOLYGON (((1150023 -15...
## 1.3 Alabama     Easy MULTIPOLYGON (((1150023 -15...
## 1.4 Alabama     Easy MULTIPOLYGON (((1150023 -15...
## 1.5 Alabama    Hilda MULTIPOLYGON (((1150023 -15...
## 1.6 Alabama    Hilda MULTIPOLYGON (((1150023 -15...
## 1.7 Alabama   Eloise MULTIPOLYGON (((1150023 -15...
## 1.8 Alabama   Eloise MULTIPOLYGON (((1150023 -15...
## 1.9 Alabama Frederic MULTIPOLYGON (((1150023 -15...
```

Spatial Join

- When using `left=FALSE` only geometries that have a match are returned

```
left = st_join(
  states[, "state"],
  tracks[, "name"]
)
inner = st_join(
  states[, "state"],
  tracks[, "name"],
  left = FALSE
)
```

Spatial Join

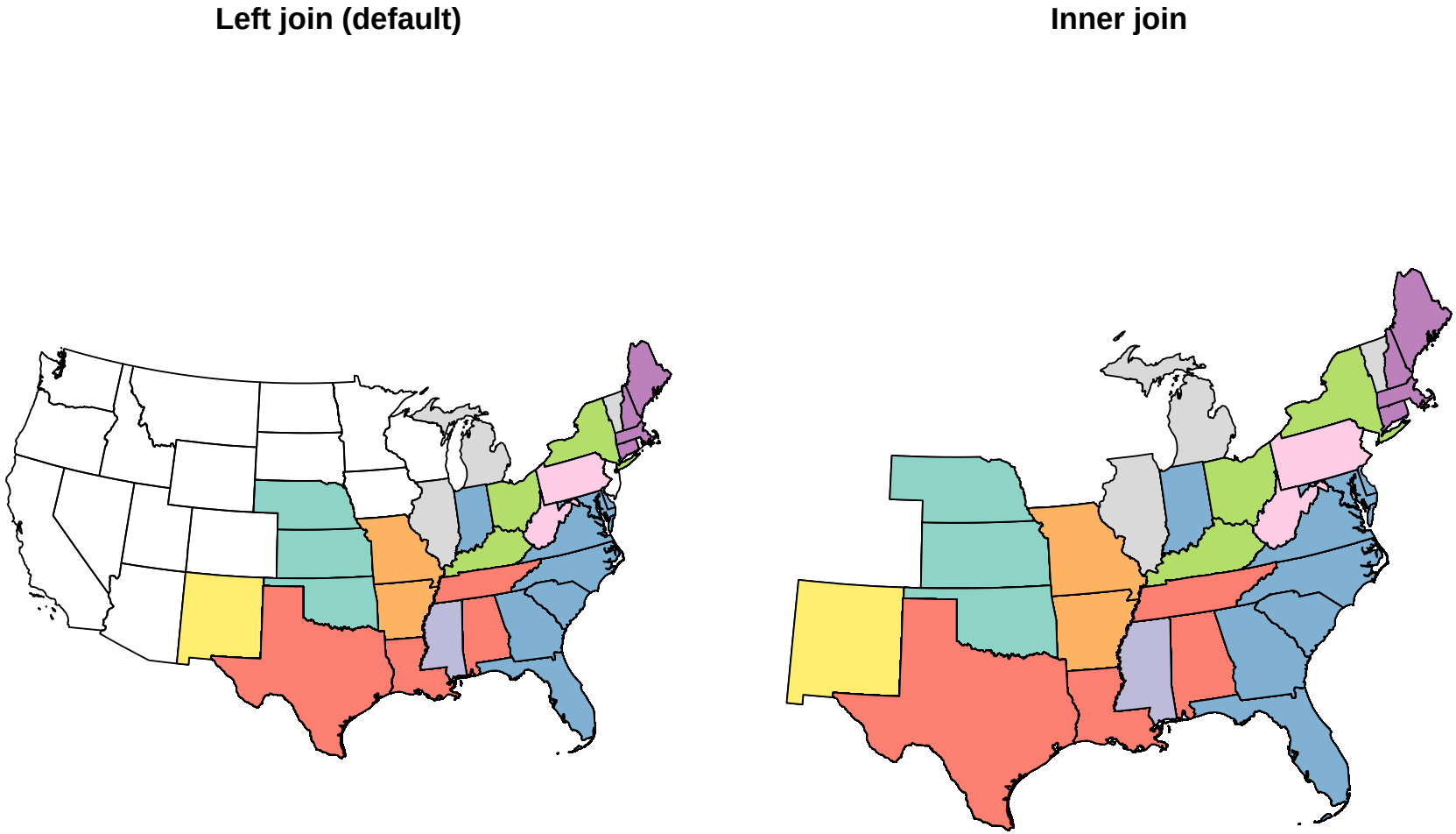

Figure 21: Left vs. Inner spatial join result

Spatial Join

- In the next example we visualize the states affected by each "H5" storm track
- First, we union the segments per storm

```
tracks_h5_agg = aggregate(
  tracks_h5[, "name"],
  data.frame(id = tracks_h5$btid),
  unique
)
```

Spatial Join

```
tracks_h5_agg
```

```
## Simple feature collection with 8 features and 2 fields
## Attribute-geometry relationship: 0 constant, 1 aggregate, 1 identity
## geometry type:  MULTILINESTRING
## dimension:      XY
## bbox:           xmin: -285061.9 ymin: -3250141 xmax: 7729104 ymax: 2534017
## epsg (SRID):    2163
## proj4string:    +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0 +a=6370997 +b
##     id    name                          geometry
## 1  864   Donna MULTILINESTRING ((2389447 -...
## 2  869   Carla MULTILINESTRING ((2570017 -...
## 3  922  Beulah MULTILINESTRING ((2349914 -...
## 4  939 Camille MULTILINESTRING ((1920845 -...
## 5 1050   Allen MULTILINESTRING ((2807307 -...
## 6 1139    Hugo MULTILINESTRING ((3876948 -...
## 7 1166  Andrew MULTILINESTRING ((1703214 -...
## 8 1319    Ivan MULTILINESTRING ((3484779 -...
```

Spatial Join

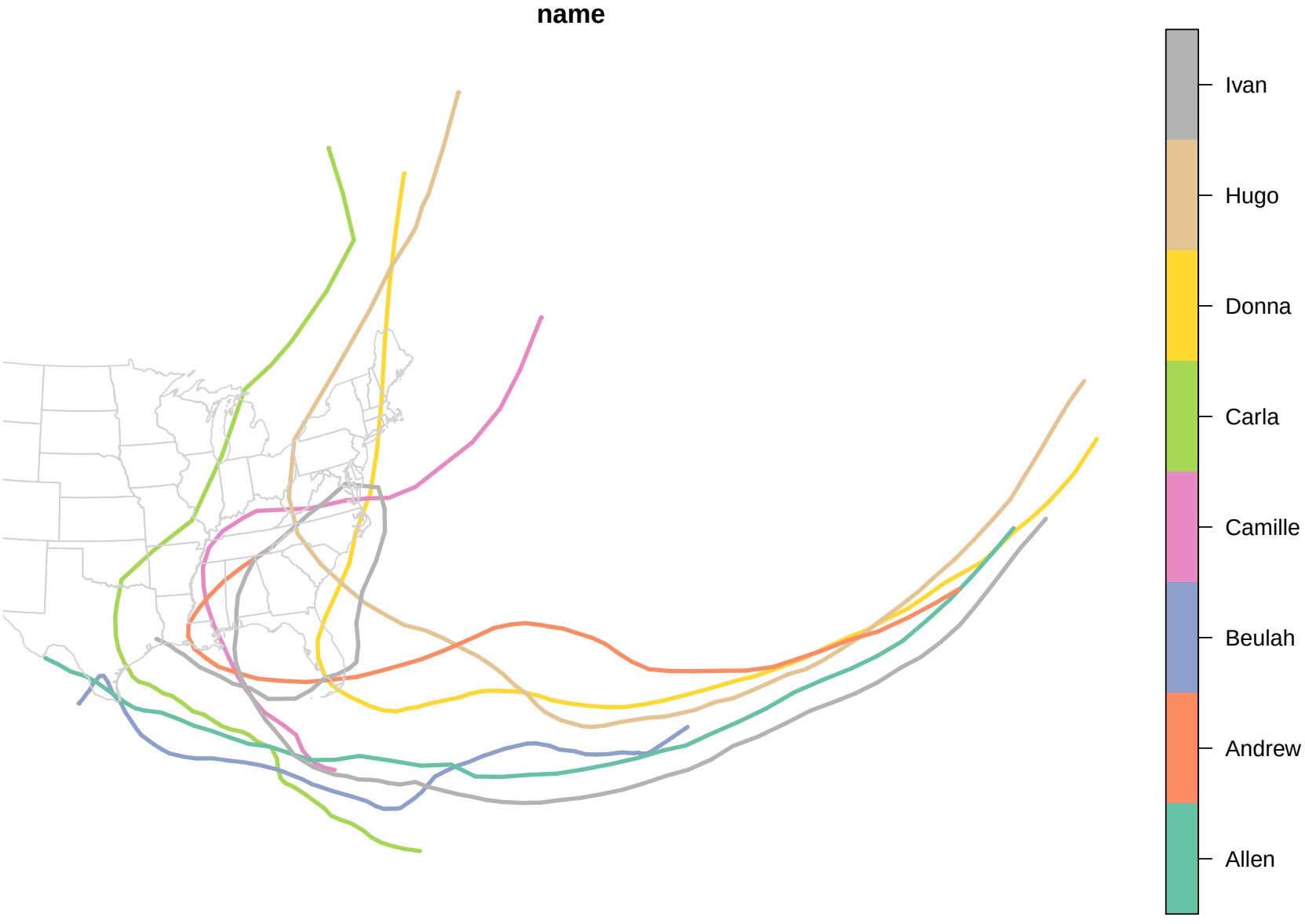

Figure 22: "H5" storm tracks

Spatial Join

- Second, we join the aggregated tracks to the `states` layer
- As a result, the `states` features are duplicated per `tracks_h5_agg` feature they intersect

```
states_tracks_h5 = st_join(
  states,
  tracks_h5_agg,
  left = FALSE
)
```

Spatial Join

```
head(states_tracks_h5)
```

```
## Simple feature collection with 6 features and 6 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 482001 ymin: -2086803 xmax: 2295610 ymax: 67481.2
## epsg (SRID):    2163
## proj4string:    +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0 +a=6370997 +b
##           state           area       length    track_density   id   name
## 1       Alabama 134050.49 km^2 2272.1309 km 0.016949814 1/km 1166 Andrew
## 1.1     Alabama 134050.49 km^2 2272.1309 km 0.016949814 1/km 1319   Ivan
## 3      Arkansas 137732.52 km^2 1000.0448 km 0.007260775 1/km  869  Carla
## 6   Connecticut  12912.34 km^2  290.7856 km 0.022519971 1/km  864  Donna
## 9       Florida 151838.27 km^2 2932.1240 km 0.019310836 1/km  864  Donna
## 9.1     Florida 151838.27 km^2 2932.1240 km 0.019310836 1/km 1166 Andrew
##                           geometry
## 1   MULTIPOLYGON (((1150023 -15...
## 1.1 MULTIPOLYGON (((1150023 -15...
## 3   MULTIPOLYGON (((482001 -928...
## 6   MULTIPOLYGON (((2156197 -83...
## 9   MULTIPOLYGON (((1953691 -20...
## 9.1 MULTIPOLYGON (((1953691 -20...
```

Mapping with ggplot2

- The development version of ggplot2 supports plotting sf layers using function **geom_sf**

```
devtools::install_github("tidyverse/ggplot2")
```

- For example -

```
library(ggplot2)

ggplot() +
  geom_sf(
    data = states
  ) +
  geom_sf(
    data = tracks_h5_agg,
    aes(colour = name)
  )
```

Mapping with ggplot2

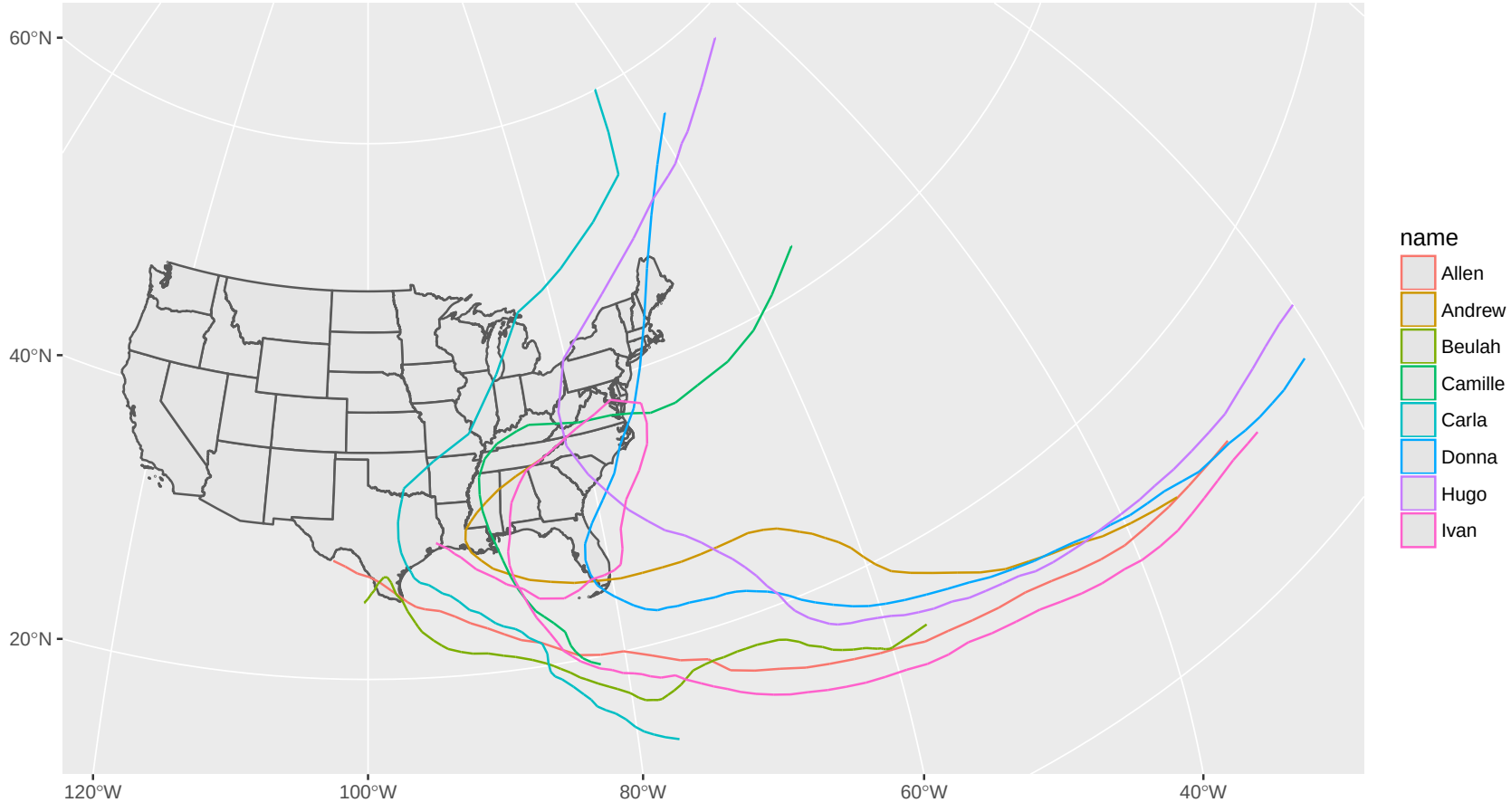

Figure 23: Mapping `sf` layer with `ggplot2`

Mapping with ggplot2

```
ggplot() +
  geom_sf(
    data = states,
    fill = NA, colour = "black", size = 0.1
    ) +
  geom_sf(
    data = states_tracks_h5,
    fill = "red", colour = NA, size = 0.3, alpha = 0.2
    ) +
  facet_wrap(~ name) +
  theme_bw()
```

Mapping with ggplot2

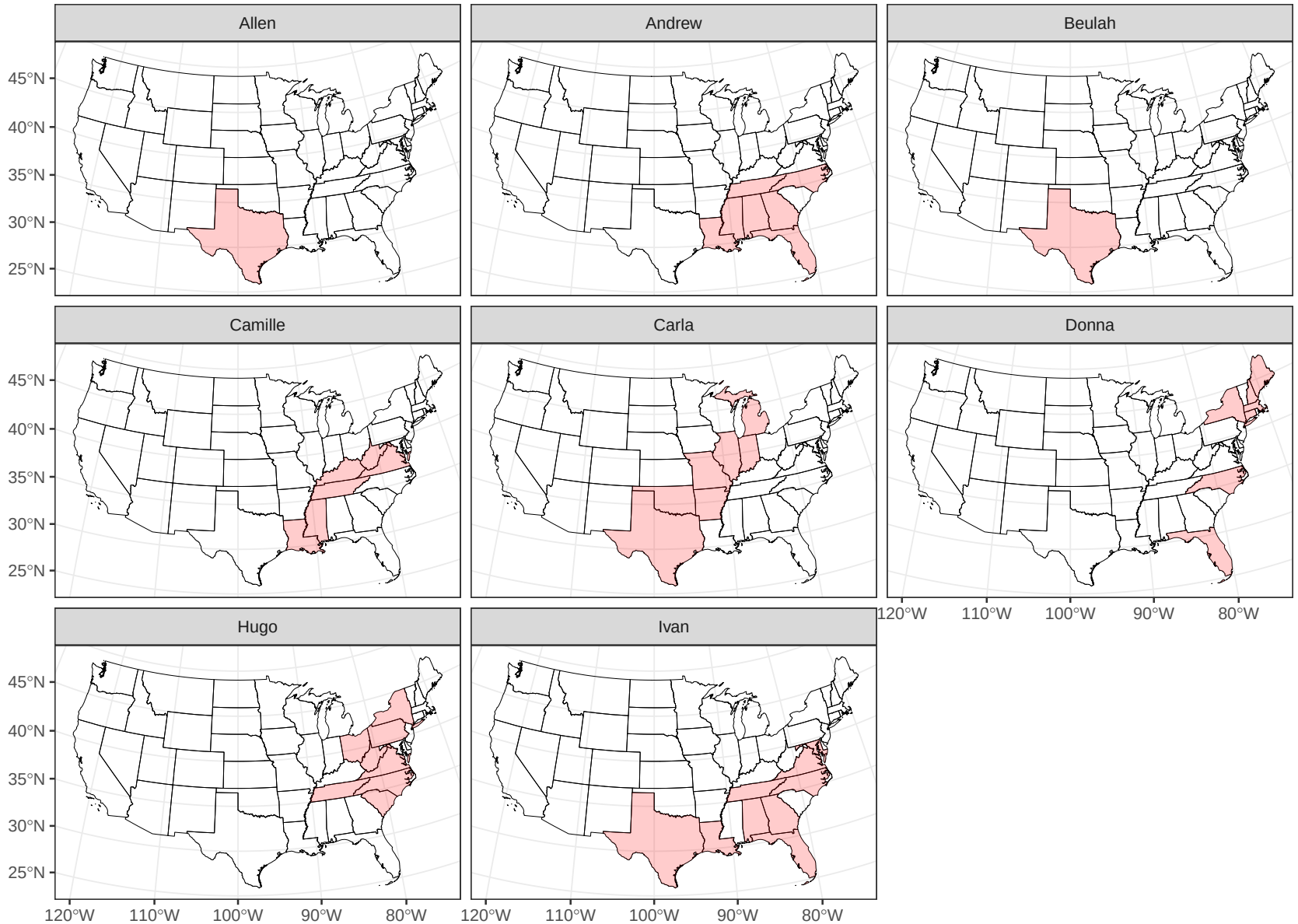

Figure 24: Using ggplot2 facets

Mapping with ggplot2

```
ggplot() +
  geom_sf(
    data = states,
    fill = NA, colour = "darkgrey", size = 0.1
    ) +
  geom_sf(
    data = states_tracks_h5,
    fill = "red", colour = "black", size = 0.3, alpha = 0.2
    ) +
  geom_sf(
    data = tracks_h5,
    aes(size = wind_mph, colour = wind_mph)
    ) +
  geom_sf(
    data = tracks_h5,
    colour = "black", size = 0.2
    ) +
```

Mapping with ggplot2

```
  coord_sf(
    xlim = st_bbox(states)[c(1,3)],
    ylim = st_bbox(states)[c(2,4)]
    ) +
  scale_size_continuous(
    range = c(0.1, 5), guide = FALSE
    ) +
  scale_colour_distiller(
    "Wind speed\n(miles / hour)", palette = "Spectral"
    ) +
  facet_wrap(~ name, ncol = 3) +
  theme_bw() +
  theme(
    panel.grid.major = element_line(colour = "transparent"),
    axis.text = element_blank(),
    axis.ticks = element_blank(),
    legend.position = c(0.8, 0.17)
    )
```

Mapping with ggplot2

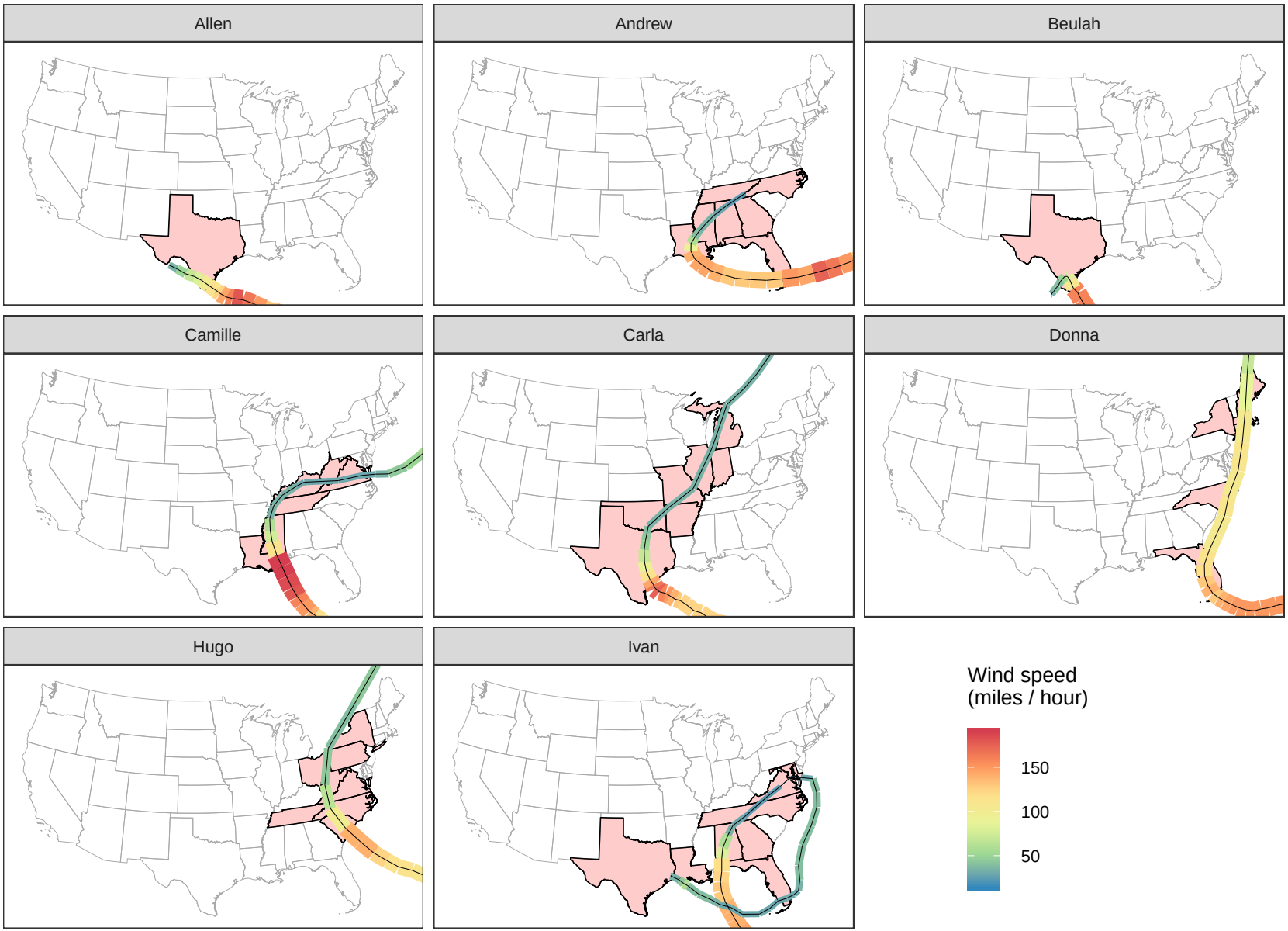

Figure 25: Level 5 hurricanes, wind speed and affected states

More information

- Official `sf` tutorials
- `sf` tutorial from useR!2017 conference
- `sf` tutorial from rstudio::conf 2018 conference
- The r-spatial blog
- Geocomputation with R (2018) book

Thank you for listening!