Spatial Data I: Working with Spatial Data

Francisco Villamil

Applied Quantitative Methods II

MA in Social Sciences, Spring 2026

Today's goals

- Understand why location matters in social science research
- Learn the main types of spatial data: vector and raster
- Grasp what a coordinate reference system (CRS) is and why it matters
- Work with spatial data in R using the `sf` package
- Visualize spatial data with `ggplot2`

This session opens the two-part spatial data block. We are building on students' R and regression skills but introducing an entirely new data structure. The key insight to drive home early: spatial data is just a data frame with an extra geometry column. Once students see that, the `sf` package feels natural because it works with `dplyr` just like any other data frame. Plan roughly 15 minutes per section. The visualization section at the end can expand or contract depending on time.

Roadmap

Why Spatial Data?

Types of Spatial Data

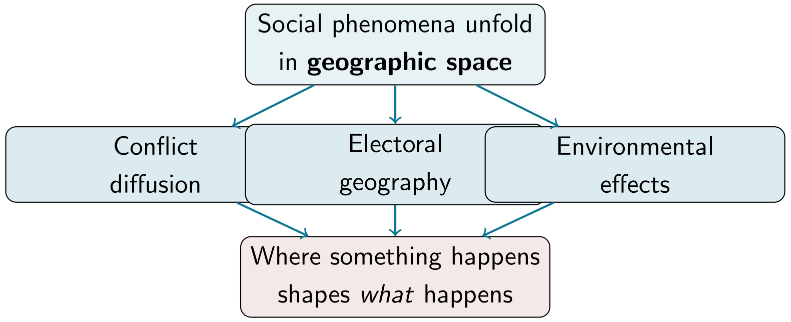
Coordinate Reference Systems

The sf Package

Visualization with ggplot2

Wrap-up

Location matters



Social phenomena unfold in **geographic space**
Conflict diffusion
Electoral geography
Environmental effects
Where something happens shapes *what* happens

Open by motivating the session. Ask students: how many of the research designs we have discussed depend on geography? Conflict events cluster in space; electoral behavior varies by region; pollution affects nearby residents; migration flows follow geographic paths. The point is that ignoring the spatial dimension of data is like ignoring the temporal dimension — you lose structure that can help or, worse, you get wrong answers because of spatial correlation. Spend about 2 minutes here before moving to concrete examples.

Spatial questions in political science

- **Conflict**: Do armed events cluster? Does violence diffuse across borders?
  - → Data: ACLED event locations (lat/lon coordinates)
- **Elections**: How does vote share vary across districts?
  - → Data: municipality-level polygons with electoral results
- **Environment**: Do voters near polluting facilities vote differently?
  - → Data: facility locations + electoral district polygons
- **Inequality**: How does poverty vary within a city?
  - → Data: census tract polygons + socioeconomic attributes

Concrete examples are essential here. ACLED (Armed Conflict Location and Event Data) is a widely used dataset in conflict research that provides exact GPS coordinates for conflict events. Electoral studies increasingly use fine-grained geographic data at the municipality or polling station level. The environmental politics literature asks whether exposure to pollution — measured by distance to facilities — shapes political attitudes. Each of these requires spatial data skills. These examples also preview two key data types: point data (event coordinates) and polygon data (district boundaries). Note to students: many of these datasets are freely available and they may want to use them for their final essays.

Think about your own research topic.

What is its **spatial dimension**?

What data would you need — points, lines, or polygons?

Brief discussion: 1–2 minutes. Ask students to think about their research interests or final essay topics. The goal is to activate prior knowledge and personal investment in the material before the technical content begins. Expected answers: electoral geography (municipality polygons), conflict (event point coordinates), immigration (country or region polygons), environmental effects (facility locations as points, district polygons as receiving units). This also previews the geometry types covered in Section 2.

[IMAGE PLACEHOLDER] This frame will show a map of Africa with armed conflict event locations (ACLED data), demonstrating spatial clustering of violence in the Sahel, Horn of Africa, and DRC. Describe verbally if the image is not yet available: dots representing individual events, clearly concentrated in certain sub-regions. This visual makes the motivation for spatial analysis immediate and concrete.

Roadmap

Why Spatial Data?

Types of Spatial Data

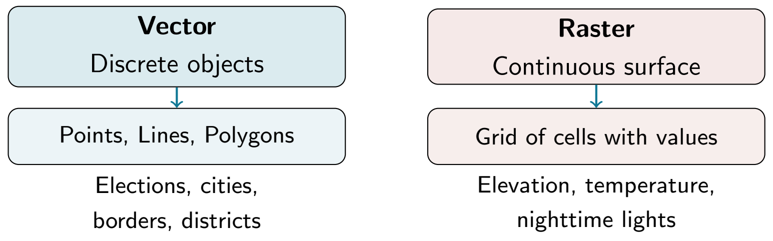
Coordinate Reference Systems

The `sf` Package

Visualization with `ggplot2`

Wrap-up

Two families of spatial data

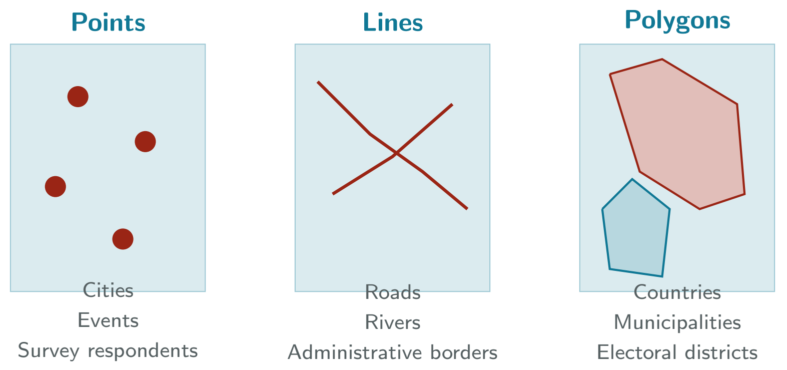


Vector
Discrete objects
Points, Lines, Polygons
Elections, cities, borders, districts
Raster
Continuous surface
Grid of cells with values
Elevation, temperature, nighttime lights

- Today: **vector** data only (by far the most common in social science)
- Raster: briefly in Session 10 (Spatial data II)

Set expectations clearly: we focus on vector data today. Raster data is important (nighttime lights as a proxy for economic activity, precipitation data for agricultural analysis, land cover for environmental studies) but it requires different tools and is covered in Session 10. The key conceptual distinction is discrete vs. continuous: vector data represents identifiable objects (a city, a border, a district), while raster data represents a field that varies continuously across space (temperature measured at every point on the Earth's surface). Most political science and sociology research uses vector data, because we are interested in units (countries, municipalities, people) rather than continuous surfaces.

Vector data: three geometry types



Walk through each geometry type with a political science example. Points: conflict events in ACLED, city locations, household survey respondents. Lines: trade routes, rivers that serve as borders, road networks for accessibility research. Polygons: the workhorse of political science spatial data — country boundaries, municipality boundaries, electoral constituencies, census tracts. Each geometry type can carry an arbitrary number of attribute columns (population, vote share, GDP). The geometry is just one more column in the data frame. Spend about 1 minute on each type and ask students to suggest examples from their own research interests.

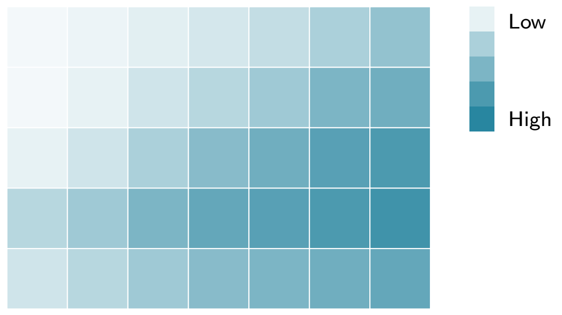
Vector data: geometry + attributes

name	population	vote_share	gdp_pc	geometry
Madrid	3,305,408	0.34	38,200	POLYGON((-3.8 40.7, . . .))
Barcelona	1,620,343	0.21	41,500	POLYGON((2.0 41.2, . . .))
Valencia	814,208	0.29	29,100	POLYGON((-0.4 39.4, . . .))
Sevilla	690,566	0.41	24,800	POLYGON((-6.0 37.3, . . .))

- A spatial object is a **data frame** with a `geometry` column
- All regular data operations work as usual
- Geometry encodes the shape: coordinates of vertices for polygons
- One row = one spatial feature (a city, a district, a country)

This is the key conceptual slide for vector data. The table shows that a spatial data frame is exactly like any other data frame — it just has a `geometry` column appended. Students who already know `dplyr` can immediately apply `filter()`, `mutate()`, `group_by()` to spatial data. The geometry column stores the actual shape: for points, a pair of coordinates; for lines, a sequence of coordinate pairs; for polygons, a closed sequence of coordinate pairs. In the Simple Features standard, these are encoded as Well-Known Text (WKT) strings like `POLYGON((-3.8 40.7, -3.6 40.8, ...))`, which is what appears in the `geometry` column. The `sf` package handles all of this automatically.

Raster data: a brief preview



Each cell stores one value

- Grid of equal-size cells, each storing a value (elevation, precipitation, . . .)
- **Continuous surface**: the entire area is covered
- In R: `terra` package (`rast()` objects)

Keep this slide brief — it is a preview only. The raster grid shows the key difference: every cell in the grid has a value, so the entire geographic surface is represented. Compare this with polygon data, where only the interior of each polygon has attributes. Raster data is used in political science mainly when the variable of interest is inherently continuous: nighttime light intensity (a proxy for economic development), precipitation (affecting agricultural productivity and potentially conflict), land cover type (forest, urban, agricultural). The `terra` package has largely replaced the older `raster` package. We will come back to raster data in Session 10.

Roadmap

Why Spatial Data?

Types of Spatial Data

Coordinate Reference Systems

The `sf` Package

Visualization with `ggplot2`

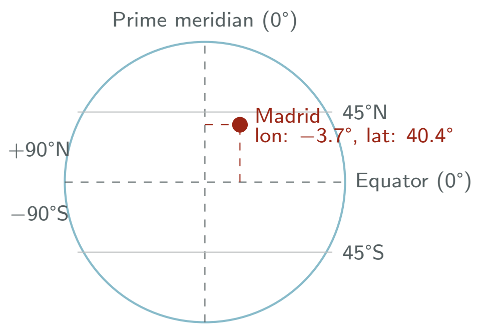
Wrap-up

How do we locate things on Earth?

- The Earth is a (roughly) spherical three-dimensional object
- We need a system to assign coordinates to locations on its surface
- A **Coordinate Reference System (CRS)** defines:
  - → The **origin** and **axes** of the coordinate system
  - → The **shape** of the Earth used (datum / ellipsoid)
  - → How coordinates map to the actual surface
- Two main families:
  - → **Geographic CRS**: longitude and latitude on a sphere
  - → **Projected CRS**: Cartesian x/y on a flat surface

Students often find CRS confusing because they have been using GPS coordinates (latitude, longitude) their whole lives without thinking about what system they refer to. The key message: coordinates are meaningless without knowing the CRS, because the same (x, y) pair means different things in different systems. The CRS defines how numbers map to real-world locations. Start with the intuition: to say where a point is on a globe, you need to agree on (1) where the origin is, (2) how the axes are oriented, and (3) what model of the Earth's shape you use. The two families (geographic vs. projected) are conceptually distinct: one works directly on the 3D sphere, the other flattens it.

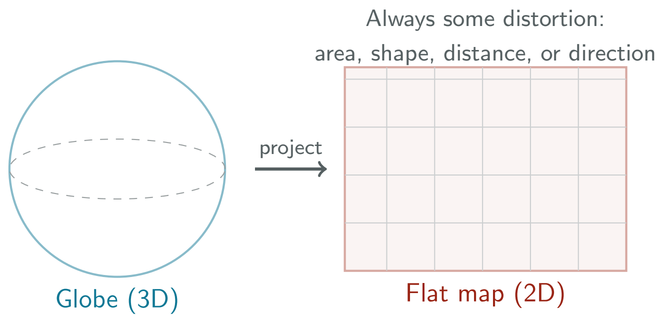
Geographic CRS: longitude and latitude



- **Longitude**: degrees East/West from the Prime Meridian (−180 to +180)
- **Latitude**: degrees North/South from the Equator (−90 to +90)
- **WGS84** (EPSG:4326): the universal standard — used by GPS, Google Maps

WGS84 is the most important CRS to know. It is what GPS devices use, what Google Maps uses, and what most raw datasets come with when coordinates are provided as longitude and latitude columns. When students download a CSV with a `lon` and `lat` column, they should almost always assume WGS84 (EPSG:4326). The EPSG code (European Petroleum Survey Group) is a standard registry of CRS definitions — each CRS has a unique integer code. EPSG:4326 is the code for WGS84. Longitude comes first in most programming contexts (even though we usually say "latitude and longitude" in speech). This trips up many beginners: when converting a CSV to an sf object, the syntax is `coords = c("lon", "lat")`, not `c("lat", "lon")`.

Projected CRS: flattening the Earth



- Converts degrees to meters (or feet) on a flat surface
- **Mercator** (EPSG:3857): shapes preserved, areas hugely distorted at poles
- **UTM zones**: accurate locally, 60 zones worldwide
- Use projected CRS for **distance and area calculations**

The key practical point: you cannot accurately measure distances or areas using WGS84 (geographic) coordinates, because the Earth is not flat. A degree of longitude in Madrid corresponds to about 80 km, but a degree of longitude in Helsinki corresponds to only about 55 km (the Earth is narrower near the poles). If you compute Euclidean distance between two lat/lon points, you get garbage. Always transform to a projected CRS before computing distances or areas. For Europe, ETRS89-LAEA (EPSG:3035) is a common choice; for the US, EPSG:5070 (Albers Equal Area). UTM zones give accurate local measurements but different zones apply to different parts of the world. The Mercator distortion is a good pedagogical example: Greenland appears as large as Africa on a Mercator map, even though Africa is 14 times larger.

EPSG codes: the practical shorthand

EPSG	**Name**	**Use case**
4326	WGS84 (geographic)	GPS, raw CSV coordinates, global data
3857	Web Mercator	Google Maps, web tiles (not for analysis)
3035	ETRS89-LAEA	Europe: equal-area, good for area/distance
32630	UTM Zone 30N	Spain and Portugal (local accuracy)
5070	Albers Equal Area	USA (equal-area for national maps)

- Always check the CRS of your data with `st_crs()`
- All layers in an analysis must use the **same CRS**
- Transform with `st_transform(data, crs = 3035)`

This table is a practical reference students can keep. The most important rule: before any spatial operation involving two datasets, make sure they are in the same CRS. Spatial joins, intersections, and buffer operations all silently fail or give wrong results if CRS do not match. The `sf` package will sometimes warn about mismatching CRS, but not always. Make it a habit to check with `st_crs()` at the start of every analysis. The choice of projected CRS depends on the region and the type of analysis: for equal-area operations (computing region areas, density), use an equal-area projection (EPSG:3035 for Europe); for distance operations, use a conformal or UTM projection. Web Mercator (EPSG:3857) is designed for web map tiles and should almost never be used for analysis.

Roadmap

Why Spatial Data?

Types of Spatial Data

Coordinate Reference Systems

The `sf` Package

Visualization with `ggplot2`

Wrap-up

`sf`: Simple Features for R

- **Simple Features** is an open standard for vector spatial data (ISO 19125)
- The `sf` package is the modern R implementation
  - → Replaced the older `sp` + `rgdal` + `rgeos` ecosystem
  - → Seamless `dplyr` and `ggplot2` integration
- An `sf` object is a **data frame** with:
  - → Regular attribute columns (any type)
  - → A `geometry` column (type `sfc`) storing the shapes
  - → CRS metadata attached to the object
- All geometry types: `POINT`, `LINESTRING`, `POLYGON` and their multi-variants

The historical context is useful: until about 2017, the dominant approach was the `sp` package (for spatial points, lines, polygons) combined with `rgdal` (for reading/writing files) and `rgeos` (for geometric operations). These worked, but the objects were S4 classes that did not play well with `dplyr`. The `sf` package (Pebesma 2018) unified all of this: it reads files, stores geometry, performs operations, and plugs directly into the tidyverse. The `sp` ecosystem is largely deprecated and students should use `sf` for all new work. The "multi-" variants (`MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`) handle features that consist of multiple separate geometries, like a country with offshore islands (e.g., France includes islands in the Caribbean).

What does an `sf` object look like?

Console output: `print(world[1:3,])`

```
Simple feature collection with 3 features and 5 fields
Geometry type:  MULTIPOLYGON
CRS: EPSG 4326 (WGS84)

  name_long continent      pop    geom
1 Afghanistan      Asia 32564342 MULTIPOLYGON(((61.2 35.6...
2      Angola    Africa 24227524 MULTIPOLYGON(((16.3 -5.8...
3     Albania    Europe  2893654 MULTIPOLYGON(((20.6 41.9...
```

- Header: feature count, geometry type, CRS — always shown first
- Rows = features; `geom` column stores the shapes (truncated in display)
- Otherwise: a regular data frame — `dplyr` verbs work immediately

This is the visual anchor for the sf section. Before showing any code, students need to see what an sf object looks like when printed. The header is diagnostic: if the geometry type or CRS looks wrong, you know immediately something went wrong on read. The truncated coordinates (`MULTIPOLYGON(((61.2 35.6...)` are expected: sf stores the full geometry internally but only shows a preview. Most importantly, the output looks like a data frame because it IS a data frame. This is the message to drive home: once you know how to use dplyr, you already know 80% of how to use sf.

Reading spatial data

From a shapefile or GeoPackage

```
library(sf)
world = st_read("data/world.shp")
munis = st_read("data/municipalities.gpkg")
```

From a CSV with coordinate columns

```
df = read.csv("events.csv")
# df has columns:  lon, lat, event_type, casualties
events = st_as_sf(df,
    coords = c("lon", "lat"),
    crs = 4326)
```

- `st_read()` handles: shapefiles (.shp), GeoPackage (.gpkg), GeoJSON, and more

Two reading workflows students will use constantly. The shapefile is the classic format: it is actually four or five separate files (.shp, .dbf, .prj, .shx) that must all be in the same folder. GeoPackage (.gpkg) is the modern replacement: a single file, more efficient, and open standard. GeoJSON is common for web data. When reading from CSV (a very common workflow when working with event data like ACLED), the key is specifying the coordinate column names correctly and the CRS (almost always 4326 for raw GPS data). Students often mix up lon and lat order — stress that it is `c("lon", "lat")`, not `c("lat", "lon")`. After reading, always run `st_crs()` to confirm the CRS was read correctly; a common problem is shapefiles with missing or wrong .prj files.

Inspecting an sf object

```
class(world)
# [1] "sf" "data.frame"

st_crs(world)$epsg
# [1] 4326

st_geometry_type(world, by_geometry = FALSE)
# [1] MULTIPOLYGON

nrow(world) # number of features
ncol(world) # number of columns (incl.  geometry)

head(world)   # shows first rows with geometry
```

Walk through these commands as if at the console. The `class()` call shows that `sf` objects have two classes: `"sf"` and `"data.frame"`. This is by design — they ARE data frames, with additional spatial machinery. The `st_crs()` function returns a full CRS object; accessing `$epsg` pulls out the numeric code. `st_geometry_type()` tells you what kind of geometries you have (useful when you receive a dataset and are not sure). `nrow()` and `ncol()` work just like on a data frame. `head()` shows the first rows, including a truncated representation of the geometry column. Tell students: the geometry column is called `geometry` by default but can have any name; `sf` tracks it internally via `attr(world, "sf_column")`.

Attribute operations: dplyr works as usual

```
library(dplyr)

# Filter to European countries
europe = world %>% filter(continent == "Europe")

# Select columns + compute log population
world = world %>%
  select(name, pop, gdp_pc, geometry) %>%
  mutate(log_pop = log(pop))

# Summarize:  total population by continent
cont = world %>%
  group_by(continent) %>%
  summarize(total_pop = sum(pop, na.rm = TRUE))
```

This slide is where students feel most comfortable: they already know `dplyr`. The key message is that all the familiar verbs — `filter()`, `select()`, `mutate()`, `group_by()`, `summarize()` — work on `sf` objects without any modification. The geometry column is sticky: it is retained automatically unless you explicitly drop it. The `summarize()` example is particularly powerful: when you group_by continent and summarize, `sf` automatically unions the geometries within each continent, producing a new `sf` object with continent-level polygons. This geometric aggregation is free — no extra code needed. Point out that the only difference from a regular data frame is that the geometry column is always there, being quietly updated as needed.

CRS operations

Check the CRS

```
st_crs(world) # full CRS info
st_crs(world)$epsg # just the EPSG code
```

Transform to a different CRS

```
# Reproject to ETRS89-LAEA (Europe equal-area)
europe_proj = st_transform(europe, crs = 3035)
```

- **Always** transform before computing distances or areas
- **Always** ensure all layers share the same CRS before spatial joins
- `st_transform()` reprojects precisely — do not manually change coordinates

Two of the most important `sf` functions. `st_crs()` is diagnostic; `st_transform()` is operative. A common mistake is to try to manually change the coordinate values in the geometry column, which corrupts the data. Always use `st_transform()` to reproject. The function applies the full mathematical transformation between CRS, accounting for the datum shift and projection equations. Another common mistake: joining two datasets where one is EPSG:4326 and the other is EPSG:3035 — `st_join()` will throw an error or give wrong results. Make it a rule: at the start of every spatial analysis, check all CRS with `st_crs()`, and transform everything to a common CRS before proceeding.

Geometric operations

- `st_area(polygons)` — area of each polygon (in m^2 if projected)
- `st_distance(pts, pts)` — pairwise distance matrix between features
- `st_centroid(polygons)`— centroid point of each polygon
- `st_buffer(pts, dist = 5000)`— 5 km buffer around each point
- `st_intersects(x, y)` — which features of x overlap with y?
- `st_union(polygons)` — merge all polygons into one

These are the workhorses of spatial analysis. `st_area()` returns areas in square meters when the CRS is projected, or in square degrees (useless) when geographic. `st_distance()` computes a matrix of pairwise distances — useful for, e.g., distance from each municipality to the national capital, or distance from each event to the nearest army base. `st_centroid()` converts polygons to points (their centroids), useful for computing distances from polygon centroids. `st_buffer()` creates a buffer zone around features — e.g., a 10 km buffer around industrial facilities to define the "exposed" population. `st_intersects()` returns a logical list: for each feature in x, which features of y does it overlap? This is the basis for spatial joins. Always use projected CRS for distance and area operations.

Spatial joins: `st_join()`

Attach polygon attributes to points

```
# events:  sf with POINT geometry (conflict events)
# munis:  sf with POLYGON geometry (municipalities)
events_muni = st_join(events, munis,
     join = st_within)
```

- Default: `st_intersects` (any overlap)
- `st_within`: point must fall *inside* the polygon
  - → Passed as a function object — no () — `join = st_within`, not `st_within()`
- `st_nearest_feature`: attach the nearest polygon (even if no overlap)
- Result: events data frame + municipality attributes appended
- Use case: "which district does each conflict event belong to?"

Spatial joins are one of the most powerful and frequently used operations in political science spatial analysis. The canonical use case: you have point data (conflict events, factories, polling stations) and polygon data (municipalities, districts, countries), and you want to know which polygon each point falls in. After `st_join()`, the result is the point data frame with all the polygon attributes appended as new columns. This allows you to, for example, count the number of conflict events per municipality, or aggregate point-level data to the district level. The `join` argument specifies the spatial predicate: `st_within` is usually the right choice for points-in-polygons (a point is within one and only one polygon). Note: CRS must match before joining — check with `st_crs()` and transform if needed.

Worked example: events per municipality

```
# 1.  Ensure same CRS
events = st_transform(events, crs = 3035)
munis = st_transform(munis, crs = 3035)

# 2.  Spatial join:  assign each event to a
municipality
events_m = st_join(events, munis, join = st_within)

# 3.  Count events per municipality
event_counts = events_m %>%
  st_drop_geometry() %>%
  group_by(muni_code) %>%
  summarize(n_events = n())

# 4.  Merge back to municipality polygons
```

Walk through this worked example step by step. It illustrates the complete workflow: transform CRS to match, spatial join to assign polygon attributes to points, drop geometry to work with a plain data frame, aggregate by grouping variable, then merge back to the polygon object for visualization. The `st_drop_geometry()` call is important: after the spatial join we no longer need the geometry for counting, so we drop it to avoid confusion. The `left_join()` at the end brings the counts back to the municipality polygons so we can make a choropleth map. This full pipeline — from event coordinates to choropleth map — is one of the most common spatial analysis workflows in political science. Students should practice this in the assignment.

Roadmap

Why Spatial Data?

Types of Spatial Data

Coordinate Reference Systems

The `sf` Package

Visualization with `ggplot2`

Wrap-up

`geom_sf()`: maps in ggplot2

Basic map of world countries

```
library(ggplot2)
ggplot(world) +
  geom_sf() +
  theme_void()
```

- `geom_sf()` automatically detects geometry type (points, lines, polygons)
- Works within the full `ggplot2` grammar: `aes()`, `scale_*()`, `theme_*()`
- `theme_void()` removes axes, gridlines, background — ideal for maps
- `coord_sf()` controls projection and extent:

```
    coord_sf(crs = 3035)  # reproject for display
    coord_sf(xlim = c(-10, 40), ylim = c(35, 72))
```

The `geom_sf()` function was added to `ggplot2` in version 3.0 and is now the standard way to plot spatial data in R. The key insight is that it fits seamlessly into the `ggplot2` grammar: you still use `aes()` for aesthetics, `scale_fill_*()` for color scales, `theme_*()` for appearance. The `coord_sf()` function is important for two reasons: (1) you can reproject the display without actually transforming the data — useful for quick visual exploration; (2) you can set bounding box limits to zoom into a region. `theme_void()` is the default map theme: it removes the x and y axis labels (which would show lat/lon degrees, generally not useful on a map) and the background grid. Mention that `theme_minimal()` is also fine if you want to keep some structure.

Choropleth maps

Countries colored by GDP per capita

```
ggplot(world) +
  geom_sf(aes(fill = gdp_pc), color = "white", size =
0.1) +
  scale_fill_viridis_c(
    name = "GDP per capita",
    na.value = "grey80",
    option = "magma") +
  theme_void() +
  labs(title = "GDP per capita (USD)")
```

- Map `fill` to a continuous variable ⇒ choropleth
- `color = "white", size = 0.1`: thin white borders between polygons
- `na.value`: color for missing data

The choropleth is the most common type of thematic map in political science. The key `aes()` mapping is `fill`: which variable determines polygon color. `scale_fill_viridis_c()` is the standard for continuous variables: the viridis palette is perceptually uniform and accessible to color-blind readers. The `option` argument selects the palette variant: `"viridis"` (blue-green-yellow), `"magma"` (black-red-yellow), `"plasma"` (purple-red-yellow). For diverging variables (e.g., party vote share change), `scale_fill_distiller(type = "div")` from RColorBrewer is appropriate. The `na.value` argument is important: countries with missing GDP data should not just disappear from the map, they should show as a neutral grey. The `color` and `size` arguments control polygon borders: thin white lines improve readability on dark-fill maps.

Layering: polygons + points

Country outlines + conflict event locations

```
ggplot() +
  geom_sf(data = world,
    fill = "grey90", color = "white", size = 0.2) +
  geom_sf(data = events,
    aes(color = event_type),
    size = 0.8, alpha = 0.6) +
  scale_color_manual(values = c(...))  +
  coord_sf(xlim = c(-18, 52), ylim = c(-35, 38)) +
  theme_void()
```

- Add multiple `geom_sf()` layers with `data =` argument
- Each layer can use a different `sf` object
- Order matters: later layers drawn on top

Layering is a key strength of ggplot2: you can combine polygon backgrounds with point overlays, add line layers for borders or roads, etc. The important syntax detail: when using multiple datasets, pass `data =` explicitly to each `geom_sf()` call rather than to the top-level `ggplot()`. The `coord_sf(xlim, ylim)` call here zooms to Africa by specifying longitude and latitude bounds. The `alpha = 0.6` makes overlapping points semi-transparent so dense clusters are visible. This exact code pattern — grey country polygons + colored event points — is extremely common in conflict research papers. Students can adapt it by changing the polygon dataset (from world countries to a specific country's municipalities) and the point dataset (to their own events).

A complete example: European unemployment

```
library(sf); library(dplyr); library(ggplot2)

# Read NUTS-2 regions (Eurostat shapefile)
nuts2 = st_read("data/NUTS_RG_20M_2021.shp") %>%
  filter(LEVL_CODE == 2)

# Join unemployment data
nuts2 = left_join(nuts2, unemp_df, by = "NUTS_ID")

# Map
ggplot(nuts2) +
  geom_sf(aes(fill = unemp_rate),
    color = "white", size = 0.1) +
  scale_fill_distiller(palette = "YlOrRd",
    direction = 1, name = "Unemployment (%)") +
  coord_sf(xlim = c(-25, 45), ylim = c(34, 72)) +
```

This worked example ties together everything from the session: reading a shapefile, filtering with dplyr, joining external data, and making a choropleth with a sensible color scale. The Eurostat NUTS shapefile is freely available (search "Eurostat NUTS shapefiles") and contains European NUTS-0, NUTS-1, NUTS-2, and NUTS-3 regions. Filtering to `LEVL_CODE == 2` gives the NUTS-2 level (roughly equivalent to regions or provinces). The `unemp_df` would be a data frame downloaded from Eurostat containing unemployment rates by NUTS_ID code. The `scale_fill_distiller()` function applies RColorBrewer palettes; `"YlOrRd"` (yellow-orange-red) works well for unemployment rates. Allow students to ask questions about the code; this is a realistic example they could adapt for their final essay.

Map aesthetics: quick tips

- **Color scales for continuous data**:
  - → Sequential: `scale_fill_viridis_c()` — for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` — for +/- variables
- **Borders**: `color = "white", linewidth = 0.1` for subtle borders
- **Theme**: `theme_void()` for clean maps; add `theme(legend.position = "bottom")`
- **Missing data**: always set `na.value = "grey80"` in scale
- **Projection**: use `coord_sf(crs = 3035)` for Europe without reprojecting data
- **Saving**: `ggsave("map.pdf", width = 8, height = 6)`

These practical tips save students time. The most common mistakes in student maps: (1) using the default ggplot2 theme which adds lat/lon axis labels and a grey background, (2) not setting a na.value so missing countries turn white and look like ocean, (3) using the wrong color scale (sequential palette for a diverging variable), (4) saving at low resolution. The `coord_sf(crs = 3035)` trick is very useful: it reprojects the display to the LAEA projection (which looks much better for European maps — no distorted Greenland) without actually transforming the data stored in the object. The original data stays in whatever CRS it was read in; the projection only affects how ggplot2 renders it on screen. For publications, save as PDF for vector graphics; for presentations, PNG at 150-200 dpi is sufficient.

Roadmap

Why Spatial Data?

Types of Spatial Data

Coordinate Reference Systems

The `sf` Package

Visualization with `ggplot2`

Wrap-up

Key takeaways

- **Spatial data** = regular data + geometry column
  - → Vector: points, lines, polygons — discrete objects with attributes
- **CRS**: always check, always match before joining
  - → EPSG:4326 (WGS84) for raw data; project before computing distances
- `sf` **package**: the modern R standard
  - → `st_read()`, `st_as_sf()`, `st_transform()`, `st_join()`
  - → Works seamlessly with `dplyr` and `ggplot2`
- **Visualization**: `geom_sf()` inside `ggplot2`
  - → Choropleth: `aes(fill = variable)` + `scale_fill_viridis_c()`
- **Workflow**: read → check CRS → transform → join → visualize

Summarize the arc of the session. Students should leave with a mental model: spatial data is just a data frame with a geometry column, the CRS is the coordinate system that makes the coordinates meaningful, and the `sf` package integrates smoothly into the R workflow they already know. The workflow arrow at the bottom — read, check CRS, transform, join, visualize — is a practical checklist they can follow for any spatial analysis. Next session covers spatial autocorrelation and spatial regression: the analytical core of spatial econometrics, building on the data skills introduced today.

For next session

- Complete Assignment 7 (spatial data in R)
  - → Load a shapefile, inspect CRS, reproject, make a choropleth
  - → Perform a spatial join: assign event points to municipality polygons
- Next session (Spatial data II):
  - → Spatial autocorrelation: Moran's I
  - → Spatial weights matrices
  - → Spatial regression models (SAR, SEM, SLM)

The assignment reinforces the core workflows: reading shapefiles, checking CRS, reprojecting, choropleth mapping, and spatial join. Students should have access to a shapefile (e.g., municipality boundaries for Spain from the Spanish National Statistics Institute, or world country boundaries from Natural Earth) and a CSV with coordinates. Mention that Natural Earth (naturalearthdata.com) provides free country and regional shapefiles at multiple scales and is an excellent resource. Preview Session 10: once we can work with spatial data, the natural next question is whether outcomes are correlated across space (spatial autocorrelation), how to measure it (Moran's I), and how to model it (spatial regression). This builds directly on the panel data and regression content from earlier sessions.

Questions?