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

Today's goals

- Understand why location matters in social science research
- Learn the main types of spatial data: vector and raster

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

Today's goals

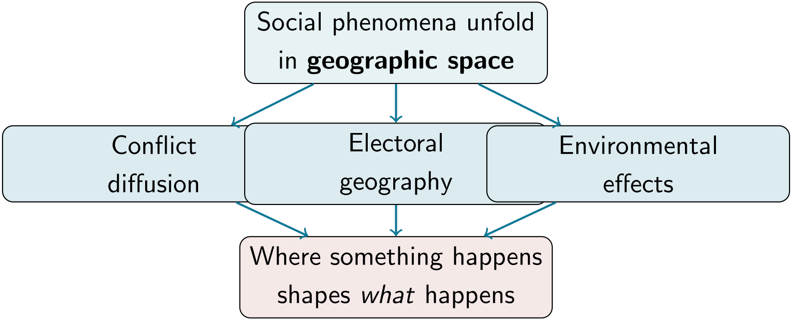
- Understand why location matters in social science research
- Learn the main types of spatial data: vector and raster
- Grasp what a coordinate reference system (CRS) is and why it matters
- Work with spatial data in R using the `sf` package

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`

Roadmap

Location matters



Spatial questions in political science

- **Conflict**: Do armed events cluster? Does violence diffuse across borders?

Spatial questions in political science

- **Conflict**: Do armed events cluster? Does violence diffuse across borders?
  - → Data: ACLED event locations (lat/lon coordinates)

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?

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

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?

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

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?

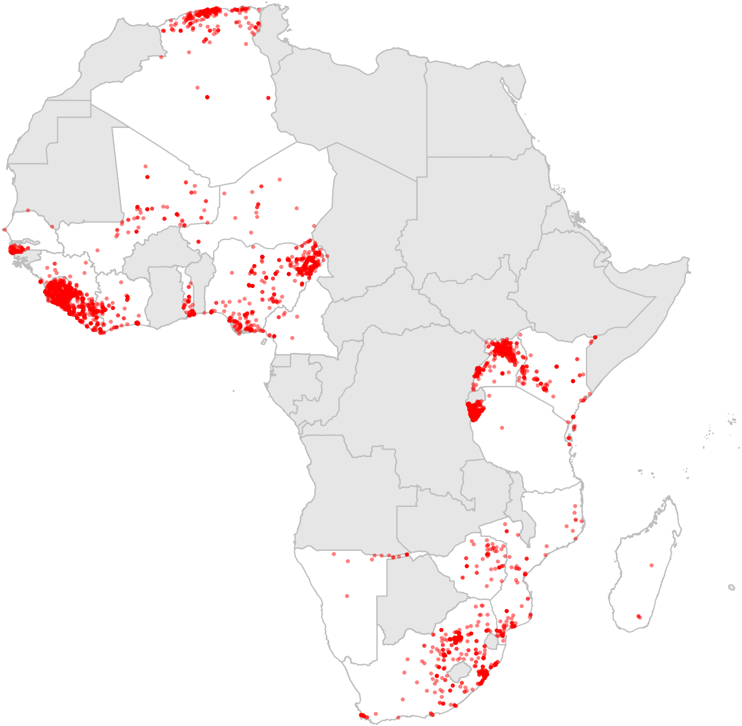
Spatial questions in political science

- **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

Think about your own research topic.

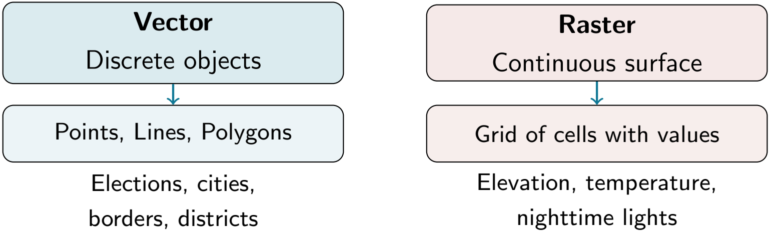
What is its **spatial dimension**?

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



Roadmap

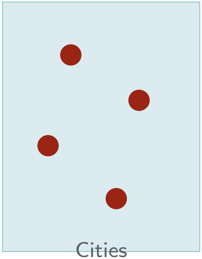
Two families of spatial data



- Today we'll focus more on **vector** data and how to work with it (more common in social science)

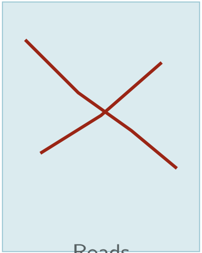
Vector data: three geometry types

Points



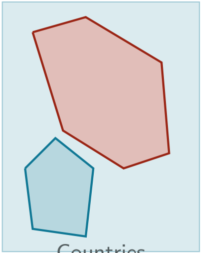
Cities
Events
Survey respondents

Lines

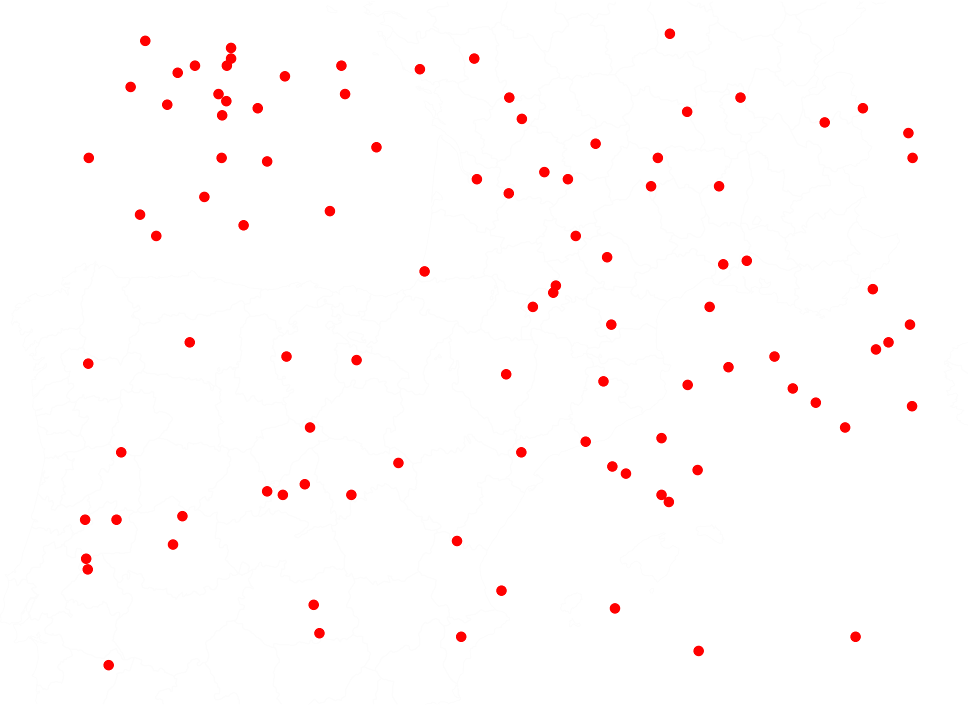


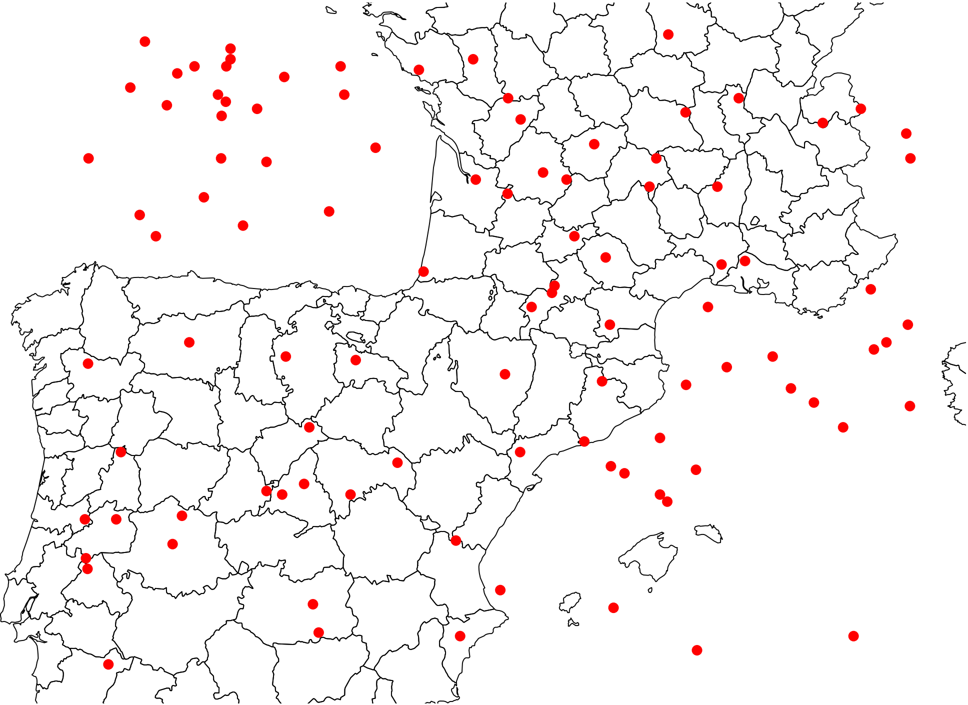
Roads
Rivers
Administrative borders

Polygons



Countries
Municipalities
Electoral districts





How does it look internally: 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

How does it look internally: 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

How does it look internally: 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, . . .))

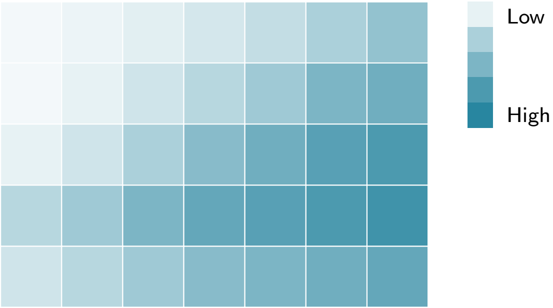
- A spatial object is a **data frame** with a `geometry` column
- All regular data operations work as usual
- Geometry encodes the shape: coordinates of vertices for polygons

How does it look internally: 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, . . .))

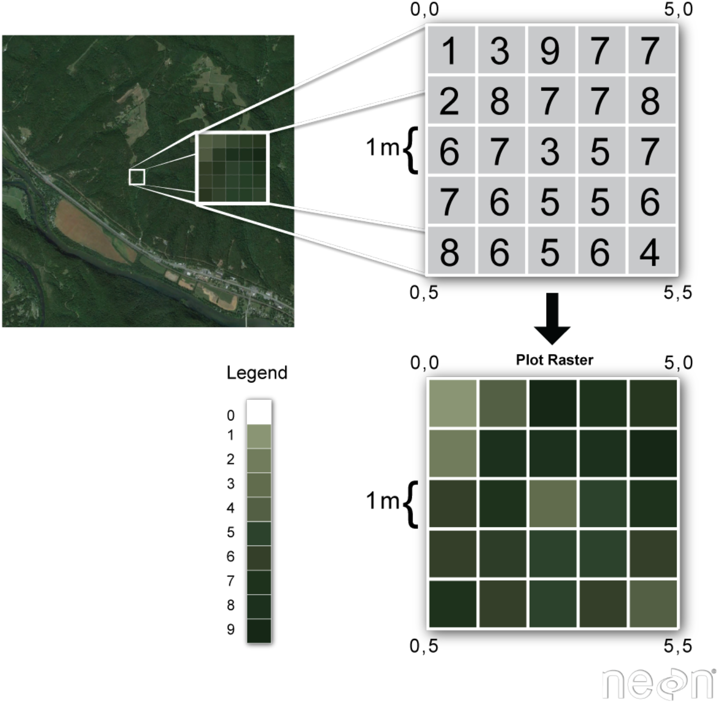
- A spatial object is a **data frame** with a `geometry` column
- All regular data operations work as usual
- Geometry encodes the shape: coordinates of vertices for polygons
- One row = one spatial feature (a city, a district, a country)

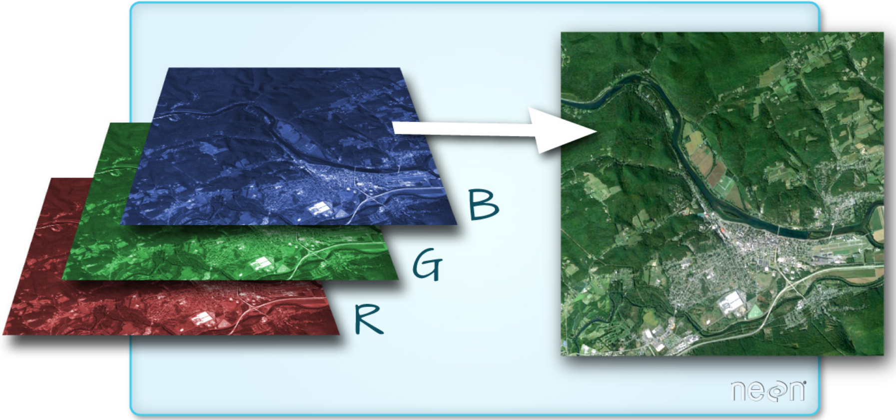
Raster data: a brief preview

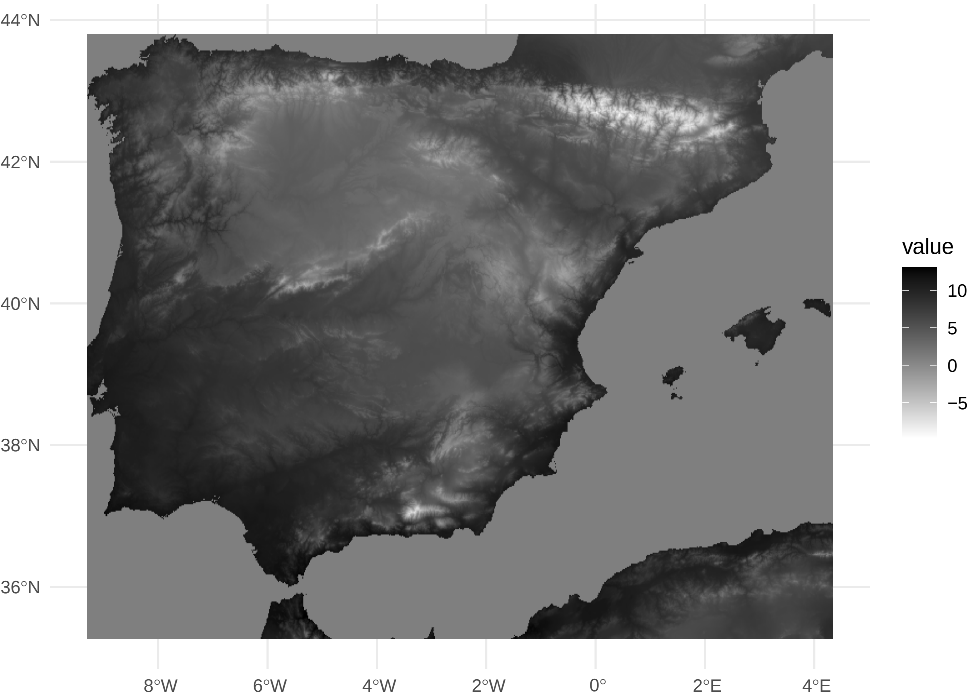


Each cell stores one value

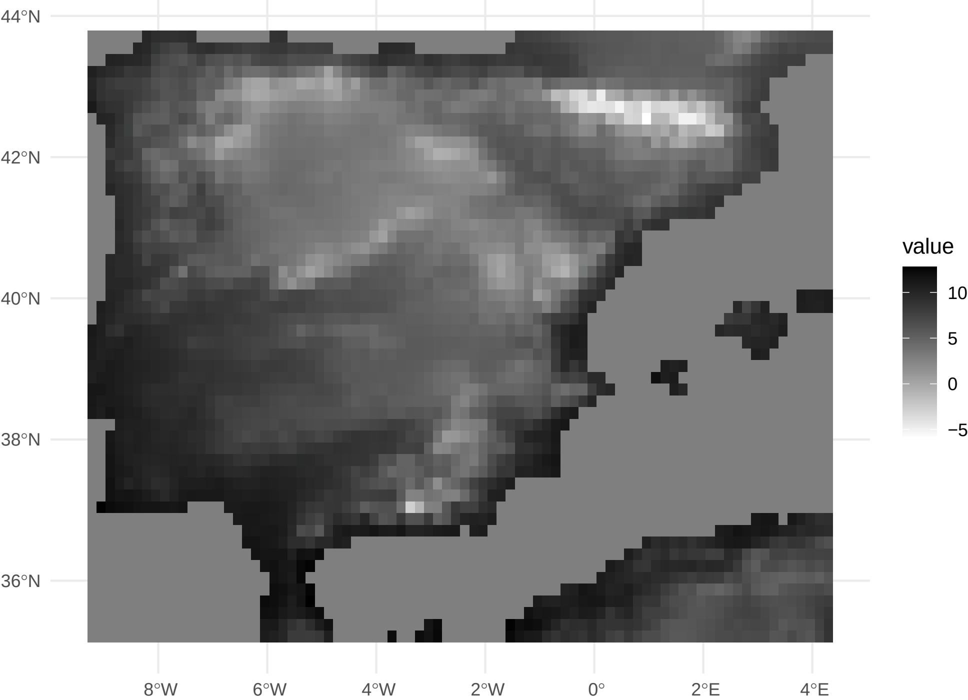
- Grid of equal-size cells, each storing a value (elevation, precipitation, . . .), which are **georeferenced**
- **Continuous surface**: the entire area is covered
- In R: `terra` package (`rast()` objects)


0,0
5,0
1 3 9 7 7
2 8 7 7 8
1m
6 7 3 5 7
7 6 5 5 6
8 6 5 6 4
0,5
5,5
Plot Raster
0,0
5,0
1m
0,5
5,5
Legend
0
1
2
3
4
5
6
7
8
9

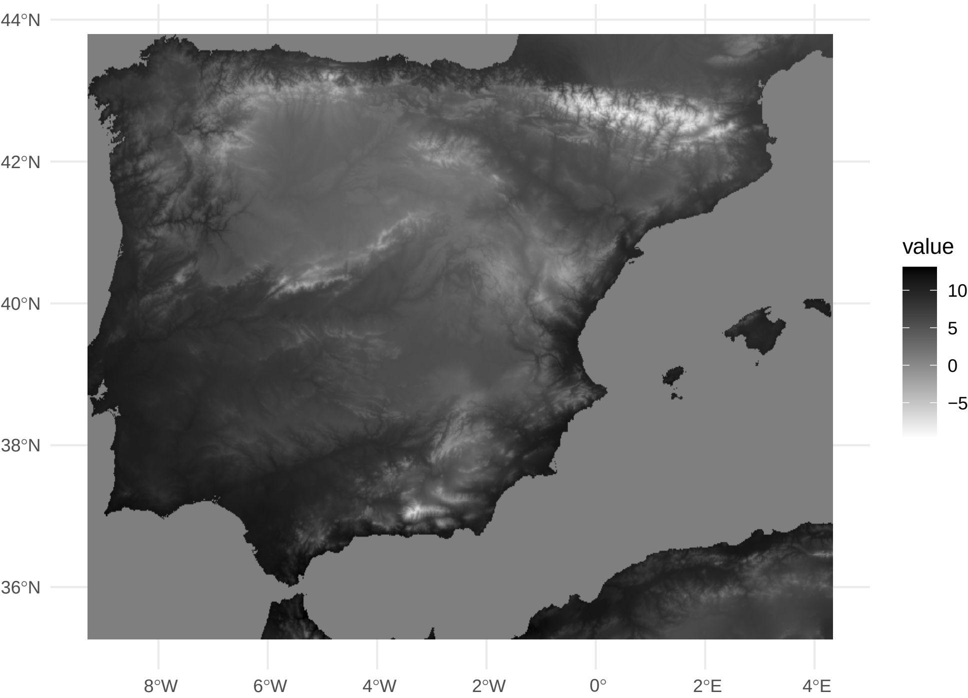

B
G
R
neon



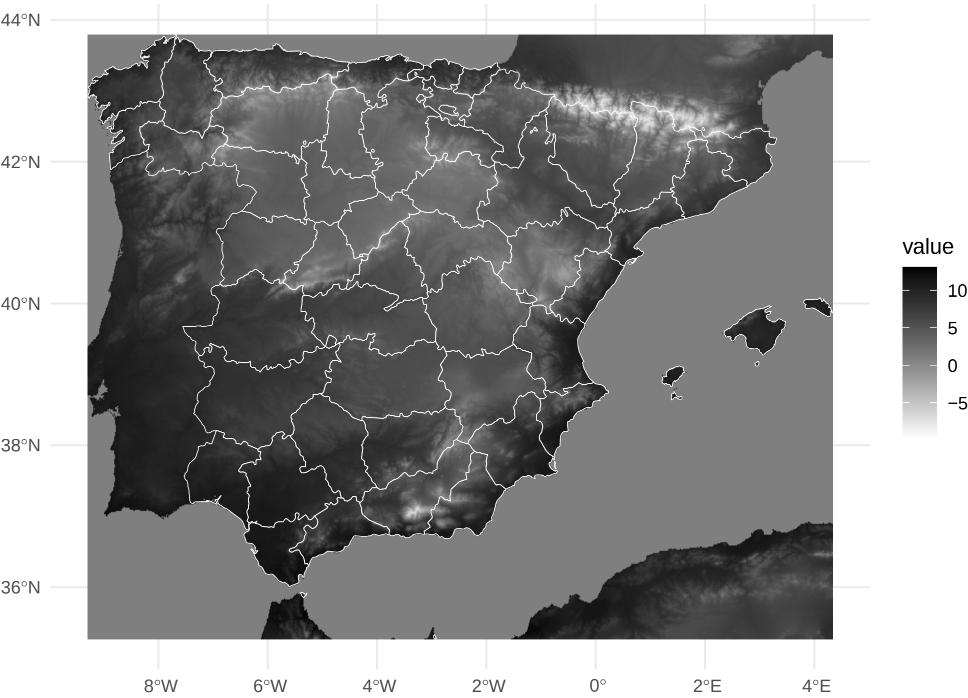
44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5

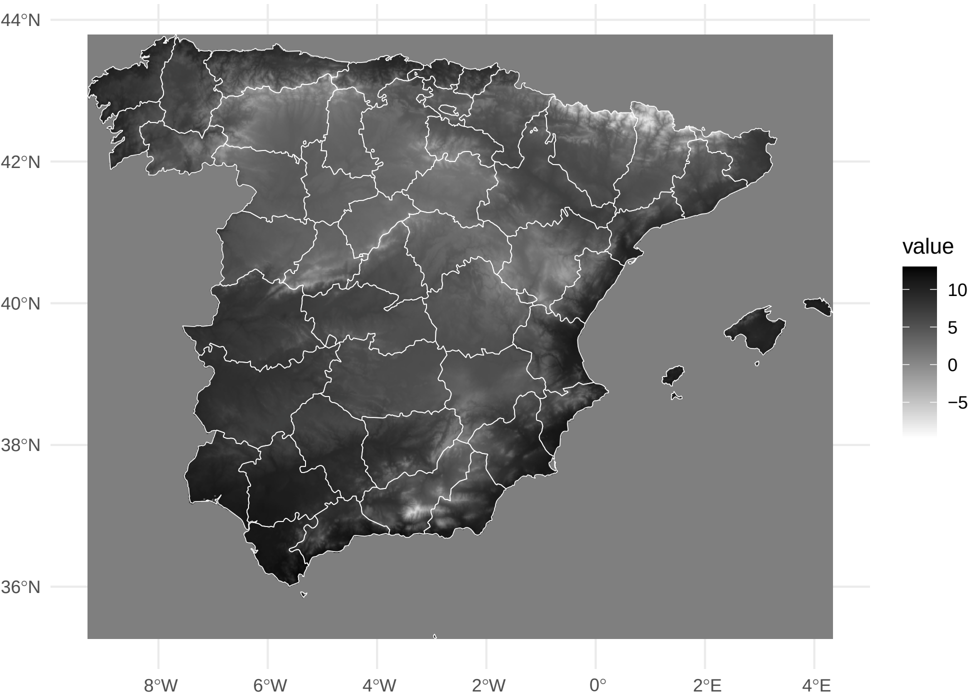


44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5

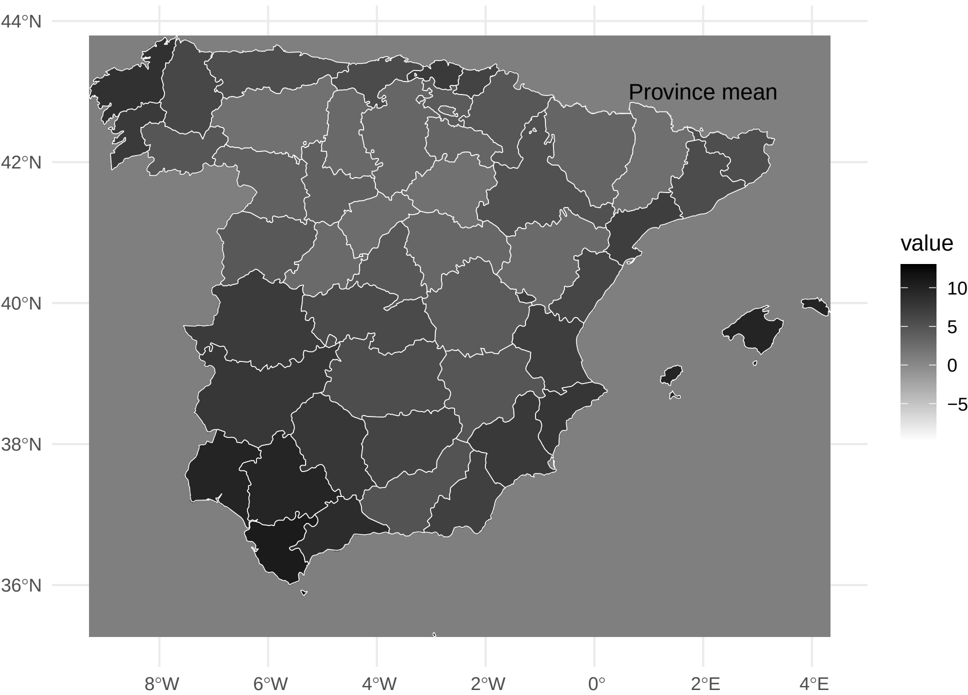


44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5

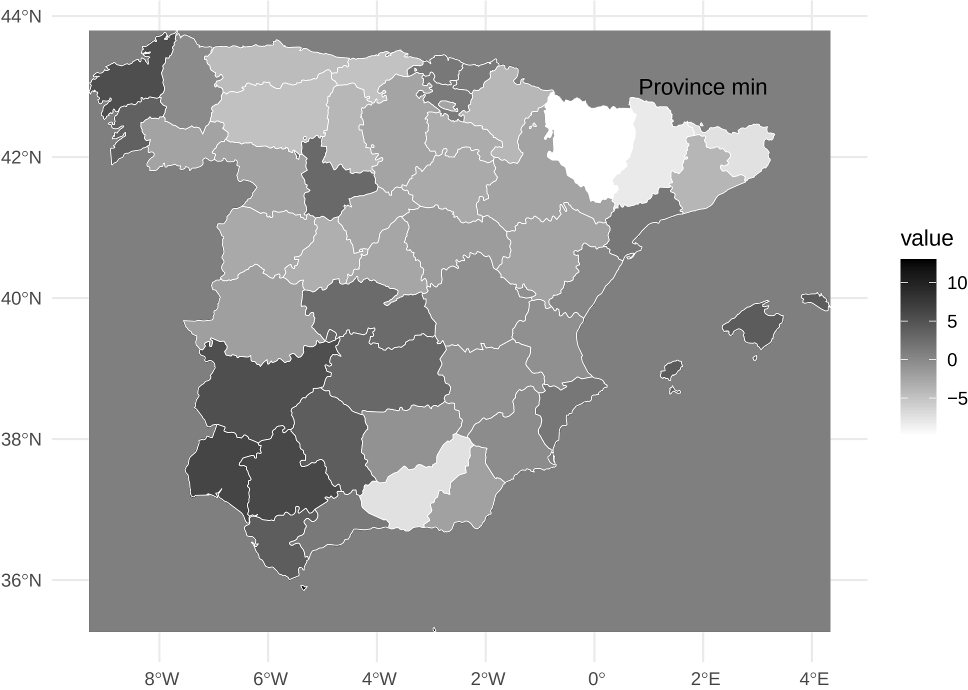

44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5



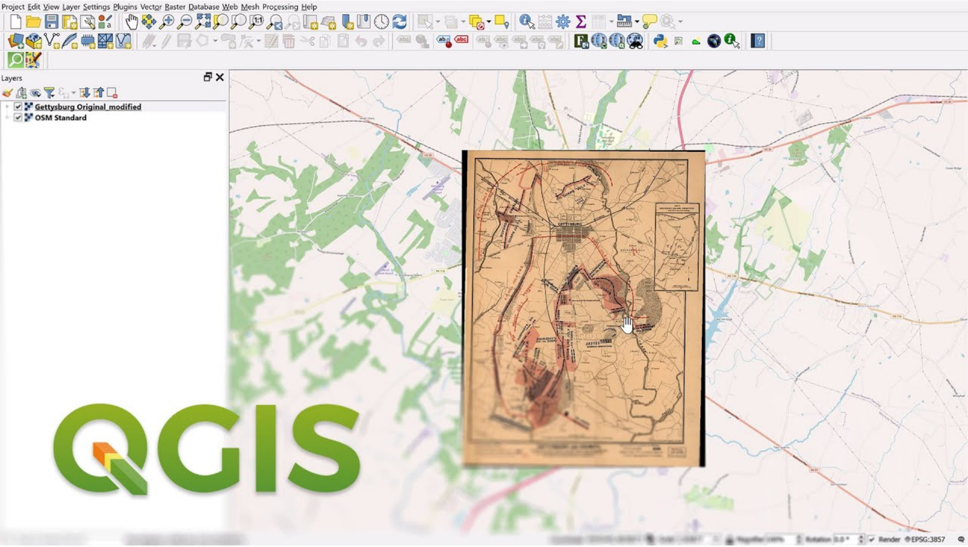
44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5



Province mean
44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E
value
10
5
0
−5



Province min
value
10
5
0
−5
44°N
42°N
40°N
38°N
36°N
8°W
6°W
4°W
2°W
0°
2°E
4°E


Project Edit View Layer Settings Plugins Vector Raster Database Web Mesh Processing Help
Layers
Gettysburg Original_modified
OSM Standard
GETTYSBURG
Render
EPSG:3857
QGIS

Hunziker (2014): https://doi.org/10.3929/ethz-a-010433796

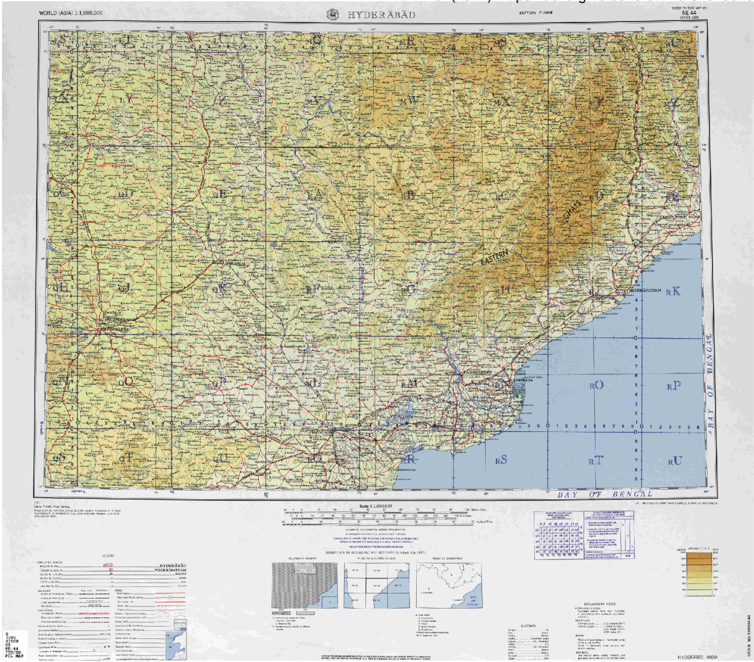


Figure 6.1: AMS 1301 map NE-44, edition 7, covering Hyderabad in 1954.

(a) Original map scan.

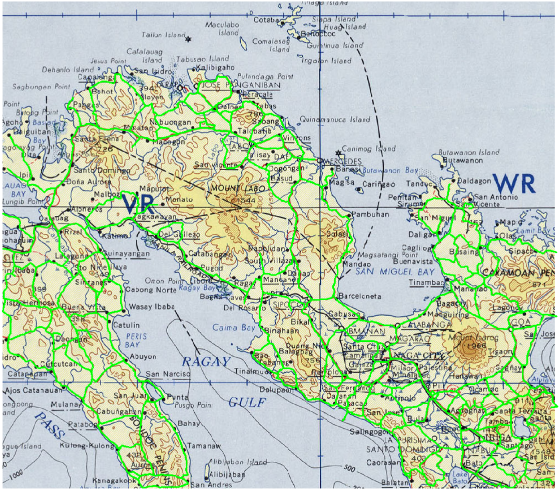
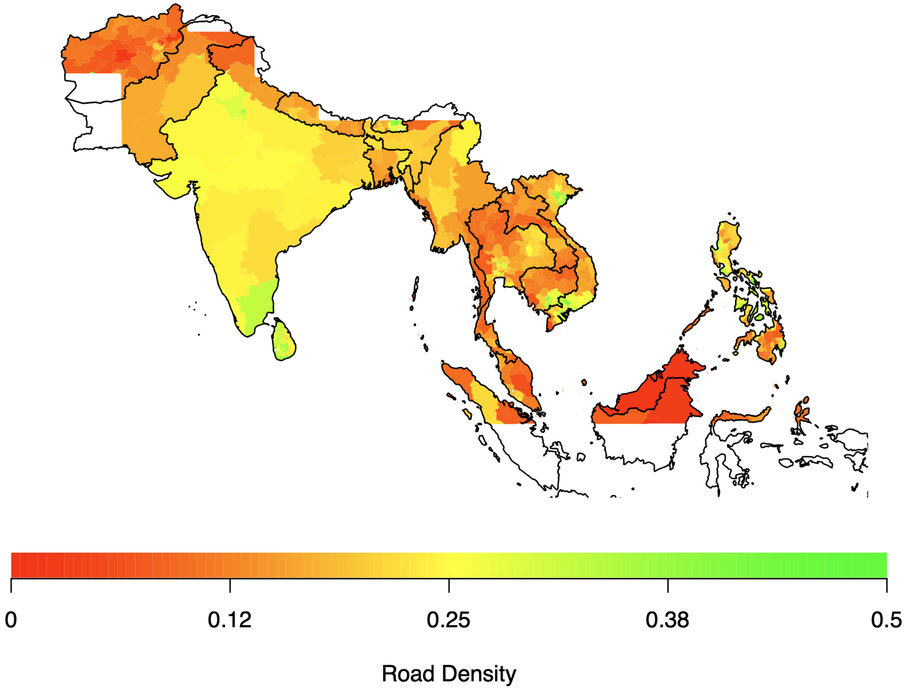


Figure 6.7: Final digitized road network (in green) superimposed on the original scanned map image.


0
0.12
0.25
0.38
0.5
Road Density

Roadmap

How do we locate things on Earth?

- The Earth is a (roughly) spherical three-dimensional object

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

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:

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

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 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

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:

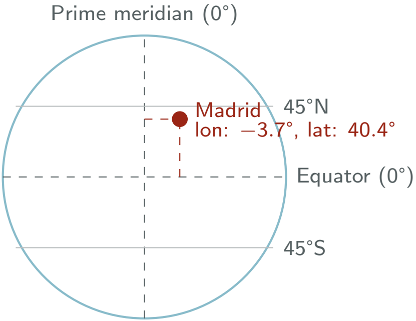
How do we locate things on Earth?

- 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

How do we locate things on Earth?

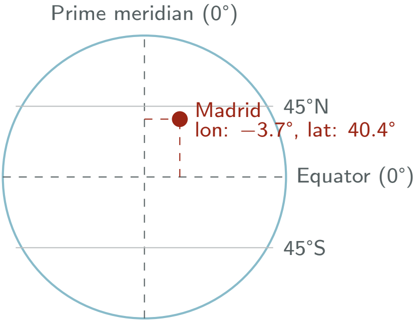
- The Earth is a (roughly) spherical three-dimensional object
- We need a system to assign coordinates to locations on its surface
- A **Coordinate Reference System (CRS)** defines:
  - → The **origin** and **axes** of the coordinate system
  - → The **shape** of the Earth used (datum / ellipsoid)
  - → 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

Geographic CRS: longitude and latitude



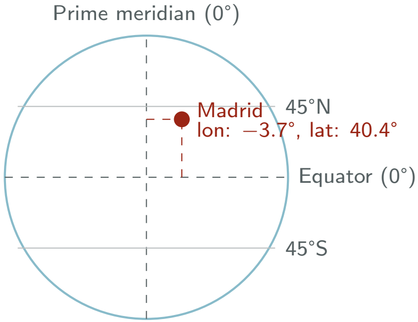
- **Longitude**: degrees East/West from the Prime Meridian (-180 to $+180$)

Geographic CRS: longitude and latitude



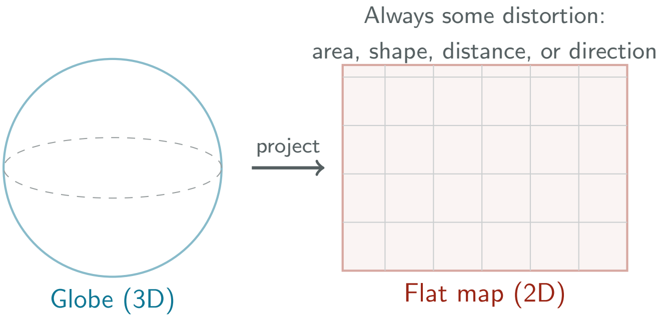
- **Longitude**: degrees East/West from the Prime Meridian (-180 to $+180$)
- **Latitude**: degrees North/South from the Equator (-90 to $+90$)

Geographic CRS: longitude and latitude



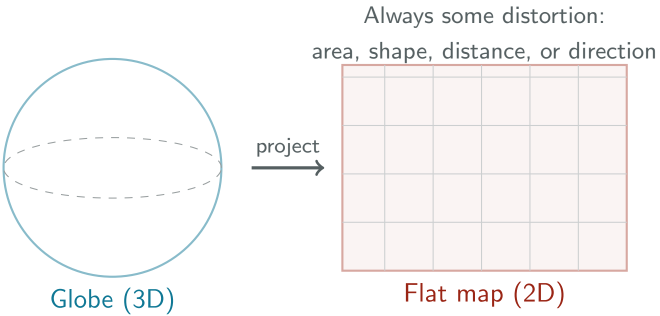
- **Longitude**: degrees East/West from the Prime Meridian (−180 to +180)
- **Latitude**: degrees North/South from the Equator (−90 to +90)
- **WGS84** (EPSG:4326): the universal standard — used by GPS, Google Maps

Projected CRS: flattening the Earth



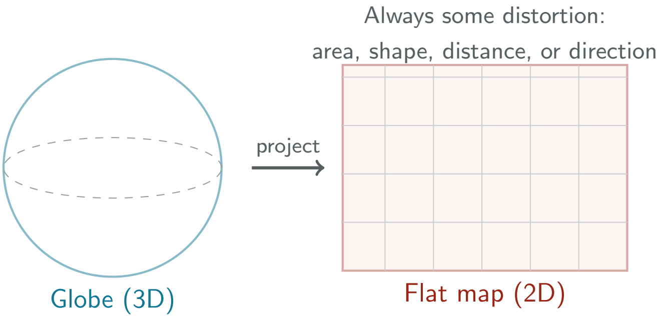
- Converts degrees to meters (or feet) on a flat surface

Projected CRS: flattening the Earth



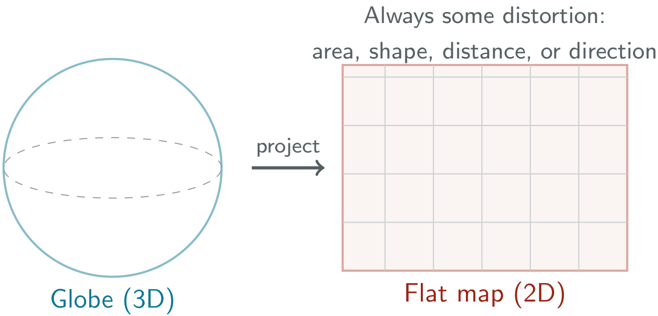
- Converts degrees to meters (or feet) on a flat surface
- **Mercator** (EPSG:3857): shapes preserved, areas hugely distorted at poles

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

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**

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()`

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**

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)`

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)

`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

`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

`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

`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:

`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)

`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

`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

`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

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

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)

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

Reading spatial data

From a shapefile or GeoPackage

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

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

Reading spatial data

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)
```

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
```

Attribute operations: dplyr works as usual

```
# 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))

# Geometry is ``sticky'' --- always retained
# To get a plain data frame: st_drop_geometry()
world_df = st_drop_geometry(world)
```

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

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

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

Geometric operations

- `st_area(polygons)` — area of each polygon (in m^2 if projected)

Geometric operations

- `st_area(polygons)` — area of each polygon (in m^2 if projected)
- `st_distance(pts, pts)` — pairwise distance matrix between features

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

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

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?

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

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)

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

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: `join = st_within`, not `st_within()`

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: `join = st_within`, not `st_within()`
- `st_nearest_feature`: attach nearest polygon (even if no overlap)

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: `join = st_within`, not `st_within()`
- `st_nearest_feature`: attach nearest polygon (even if no overlap)
- Result: `events` data frame + municipality attributes appended

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: `join = st_within`, not `st_within()`
- `st_nearest_feature`: attach nearest polygon (even if no overlap)
- Result: events data frame + municipality attributes appended
- Use case: "which district does each conflict event belong to?"

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
munis = left_join(munis, event_counts, by = "muni_code")
```

Roadmap

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)

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_*()`

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

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:

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
```

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))
```

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

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

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

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

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

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

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)) +
  theme_void()
```

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  → Sequential: `scale_fill_viridis_c()` for one-direction variables

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  - → Sequential: `scale_fill_viridis_c()` for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` for +/- variables

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  - → Sequential: `scale_fill_viridis_c()` for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` for +/- variables
- **Borders**, subtle: `color = "white", linewidth = 0.1`

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  - → Sequential: `scale_fill_viridis_c()` for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` for +/- variables
- **Borders**, subtle: `color = "white", linewidth = 0.1`
- **Theme**: `theme_void()` for clean maps; add `theme(legend.position = "bottom")`

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  - → Sequential: `scale_fill_viridis_c()` for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` for +/- variables
- **Borders**, subtle: `color = "white", linewidth = 0.1`
- **Theme**: `theme_void()` for clean maps; add `theme(legend.position = "bottom")`
- **Missing data**: always set `na.value = "grey80"` in scale

Map aesthetics: quick tips

- **Color scales for continuous data**, some options:
  - → Sequential: `scale_fill_viridis_c()` for one-direction variables
  - → Diverging: `scale_fill_distiller(type = "div")` for +/- variables
- **Borders**, subtle: `color = "white", linewidth = 0.1`
- **Theme**: `theme_void()` for clean maps; add `theme(legend.position = "bottom")`
- **Missing data**: always set `na.value = "grey80"` in scale
- **Projection**, some are useful: e.g. use `coord_sf(crs = 3035)` for Europe without reprojecting data

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

Key takeaways

- **Spatial data** = regular data + geometry column
  - → Vector: points, lines, polygons — discrete objects with attributes

Key takeaways

- **Spatial data** = regular data + geometry column
  - $\rightarrow$ Vector: points, lines, polygons — discrete objects with attributes
- **CRS**: always check, always match before joining

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

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

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()`

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`

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`

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()`

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

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)

Questions?