WorldPop

Workshop on Advanced Sampling Methodologies

Earth Observation data for sampling

Dr. Gianluca Boo, WorldPop, University of Southampton

2025-09-17

Previously covered

- Introduction to R and RStudio
- Data types and data structures
- Importing data
- Data manipulation and visualization (`tidyverse`)
- Saving and closing sessions

Agenda

- Spatial data in R
- Vector and raster data
- Packages: `sf` and `terra`
- Reading and visualising spatial data
- Spatial analysis and visualisation
- Remote sensing with Google Hearth Engine

Spatial data in R

- Two main types:
  - **Vector**: points, lines, polygons
  - **Raster**: regular grids with values
- R packages:
  - **`sf`**: modern vector handling
  - **`terra`**: raster analysis
- Both integrate well with `ggplot2` for visualisation

Vector data

Raster data

Reading vector data with `sf`

- **Simple Features Standard**: implements the Simple Features standard for spatial vector data, allowing consistent and interoperable representation of geometries like points, lines, and polygons.
- **Data Frame Integration**: stores spatial data in a regular R data frame (or tibble), with a special geometry column, making it easy to manipulate using familiar R tools like dplyr. Efficient
- **Spatial Operations**: provides a wide range of spatial functions (e.g. intersections, buffers, joins) and interfaces with libraries like GEOS, GDAL, and PROJ for robust, fast, and accurate spatial processing.

```r
library(sf)
admin_boundaries <-
  "data/pak_adm_wfp_20220909_shp/pak_admbnda_adm1_wfp_20220909.shp" |>
  st_read(quiet = T)

admin_boundaries |>
  st_geometry() |>
  plot()
```

ℹ️ Try to run the code in your Script Editor.

Vector data attributes in `sf`

- **Simple feature collection**: an `sf` object is a data frame-like structure where each row represents a spatial feature (e.g. a region), and includes attribute fields alongside a geometry column.
- **Geometry and dimensions**: each feature has a specific geometry type (e.g. MULTIPOLYGON), with defined spatial dimensions (e.g. XY), used to represent shapes on a map.
- **Spatial metadata**: the object includes spatial metadata such as a coordinate reference system (CRS), e.g. WGS 84, which defines how the geometries relate to real-world locations.

```r
library(sf); library(tidyverse)
admin_boundaries |>
  select(ADM1_EN) |>
  head(n=1)
```

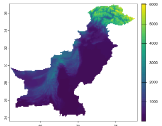
```
Simple feature collection with 1 feature and 1 field
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 73.39804 ymin: 32.76917 xmax: 74.86488 ymax: 35.13466
Geodetic CRS:  WGS 84
       ADM1_EN                       geometry
1 Azad Kashmir MULTIPOLYGON (((74.46635 35...
```

ℹ️ Try to run the code in your Script Editor.

Reading raster data with `terra`

- **Modern Raster and Vector Support**: `terra` provides tools for reading, writing, analyzing, and manipulating raster and vector spatial data, designed as a faster and more efficient successor to the older raster package.
- **High Performance and Scalability**: built for performance, `terra` can handle large spatial datasets efficiently, supports parallel processing, and integrates well with geospatial libraries like GDAL, PROJ, and GEOS.
- **Intuitive Syntax and Consistency**: offers a consistent and user-friendly syntax for spatial operations (e.g., cropping, masking, aggregation, reprojection), making it easy to work with complex geospatial workflows in R.

```r
library(terra)
elevation <-
  "data/pak_elevation_merit103_10km_v1.tif" |>
  rast()
elevation |>
plot()
```



Raster data attributes in `terra`

- **Spatial Structure**: raster data is represented as a `SpatRaster` object with defined dimensions (rows, columns, layers), resolution (cell size), and extent (bounding coordinates of the spatial area).
- **Coordinate Reference System (CRS)**: each raster includes a `CRS` (e.g., WGS 84, EPSG code) that defines how the raster aligns with real-world geographic space.
- **Metadata and Values**: rasters store metadata such as source file, layer names, and value ranges (e.g., min and max), which are critical for interpretation and analysis of the data.

```r
library(terra)
elevation
```

```
class       : SpatRaster
dimensions  : 162, 204, 1  (nrow, ncol, nlyr)
resolution  : 0.08333333, 0.08333333  (x, y)
extent      : 60.8725, 77.8725, 23.59167, 37.09167  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
source      : pak_elevation_merit103_10km_v1.tif
name        : elevation_merit103_100m_v1
min value   :                   2.328116
max value   :                6050.907715
```

ℹ️ Try to run the code in your Script Editor.

`sf::st_crs()` vs `terra::crs()`

- **SCRS Representation:** in `sf`, `st_crs()` retrieves the CRS of a vector object (e.g., admin_boundaries), while in `terra`, `crs()` is used for raster objects (e.g., elevation) to access or set their CRS.
- **SCRS Compatibility**: spatial operations between vector and raster data require both to have the same CRS. If they differ, transformation is needed to align them.
- **SCRS Transformation**: Use `st_transform()` (from `sf`) to reproject vector data to match the CRS of another dataset (e.g., `st_transform(crs(elevation))` ensures admin_boundaries matches the raster CRS).

```
st_crs(admin_boundaries)
crs(elevation)

admin_boundaries <-
  admin_boundaries |>
  st_transform(crs(elevation))
```

ℹ️ Try to run the code in your Script Editor.

Summarising raster values within polygons

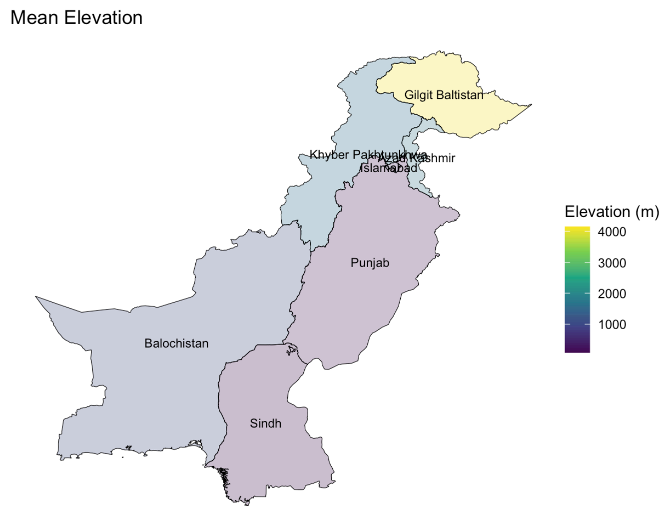
```r
admin_elevation_mean <-
  elevation |>
  terra::extract(admin_boundaries, fun = mean, na.rm = TRUE)
admin_boundaries<-
  admin_boundaries |>
  mutate(elevation_mean=admin_elevation_mean[,2])
admin_boundaries |>
  st_drop_geometry() |> dplyr::select(ADM1_EN, elevation_mean)
```

```
             ADM1_EN elevation_mean
1        Azad Kashmir     1866.30845
2         Balochistan      983.67310
3    Gilgit Baltistan     4151.73991
4           Islamabad      620.50027
5  Khyber Pakhtunkhwa     1727.04567
6              Punjab      226.31326
7               Sindh       78.63546
```

ℹ️ Try to run the code in your Script Editor.

Vector map in ggplot2

```
library(ggplot2)
ggplot() +
  geom_sf(data = admin_boundaries, aes(fill=elevation_mean),  color = "blac
  scale_fill_viridis_c()+
  geom_sf_text(data = admin_boundaries, aes(label = ADM1_EN), size = 3, col
  theme_void()+
  labs(title="Mean Elevation", fill= "Elevation (m)")
```



ℹ️ Try to run the code in your Script Editor.

Raster map in `ggplot2`

```r
library(ggplot2)
elevation_df <- as.data.frame(elevation, xy = TRUE)
colnames(elevation_df) <- c("x", "y", "elevation")
ggplot(elevation_df) +
  geom_raster(aes(x = x, y = y, fill = elevation)) +
  scale_fill_viridis_c() +
  coord_equal() +
  theme_void()+
  labs(title = "Elevation", fill = "Elevation (m)")
```

ℹ️ Try to run the code in your Script Editor.

Google Earth Engine (GEE)

- **Cloud-based platform** for planetary-scale environmental data analysis
- **No need to download massive datasets locally**
- Access to a variety of **pre-processed datasets** including:
  - **Satellite imagery:** Landsat, Sentinel-2, MODIS
  - **Climate data:** ERA5, CHIRPS precipitation
  - **Land cover and vegetation:** Copernicus, GlobCover, NDVI indices
  - **Topography:** SRTM, ASTER DEM
  - **Socioeconomic:** Population density, night-time lights

GEE Code Editor

- Open **GEE Code Editor**
- **Step-by-step:**
  1. Sign in with a Google account
  2. Click **New Script**
  3. Use the **search bar** to find datasets
  4. Load the dataset with `ee.ImageCollection` or `ee.Image`
  5. Use **filters** (e.g., date, location, bands)
  6. Display results with `Map.addLayer()`
  7. Export data with `Export.image.toDrive()` or `Export.table.toDrive()`

GEE Example Script

```javascript
// Example: extract mean NDVI for a region in 2020
var roi = ee.Geometry.Rectangle([34.5, -1.5, 35.5, -0.5]);
var collection = ee.ImageCollection('MODIS/006/MOD13Q1')
                  .filterBounds(roi)
                  .filterDate('2020-01-01', '2020-12-31')
                  .select('NDVI');

var mean_ndvi = collection.mean();
Map.centerObject(roi);
Map.addLayer(mean_ndvi, {min:0, max:9000, palette: ['white','green']}, 'Mea

// Export to Google Drive
Export.image.toDrive({
  image: mean_ndvi,
  description: 'mean_ndvi_2020',
  scale: 250,
  region: roi
});
```

ℹ️ Try to run the code in the GEE Script Editor.

Downloading data from GEE

Use `Export.image.toDrive()` to save a GeoTIFF

Options:

- **Scale**: spatial resolution (meters per pixel)
- **Region**: area of interest (polygon)
- **CRS**: coordinate reference system
- **Format**: GeoTIFF for rasters, CSV/GeoJSON for tables

After export, download the file from Google Drive

ℹ️ Try to download the file from Google Drive.

Reading GEE data

```r
library(terra); library(sf); library(ggplot2)

admin_boundaries <- st_read("data/pak_adm_wfp_20220909_shp/pak_admbnda_adm1

ndvi_raster <- rast("data/mean_ndvi_2020.tif")

admin_boundaries <-
  admin_boundaries |>
  st_transform(crs(ndvi_raster))

ndvi_mean <- terra::extract(ndvi_raster, polygons, fun = mean, na.rm = TRUE
admin_boundaries$mean_ndvi <- ndvi_mean[,2]

ggplot(polygons) +
  geom_sf(aes(fill = mean_ndvi)) +
  scale_fill_viridis_c() +
  ggtitle("Mean NDVI per Polygon from GEE")
```

ℹ️ Try to run the code in your Script Editor.

Take home

- Spatial data: vector (sf) vs raster (terra)
- Always check CRS before analysis
- Use sf for points, lines, polygons
- Use terra for raster analysis
- Combine rasters and vectors for summaries and plots
- Use GEE to access earth observation data

Resources

- Geocomputation with R – comprehensive online book
- `sf` package reference
- `terra` package documentation
- GEE Developer Guide
- Spatial Data Science in R (RStudio Education)
- R Spatial Cheatsheet

Exercise

Please download the R script with exercises from GitHub and try to complete it.