Notes on Ch2: Geographic data in R

Norman LIm

2025-06-28

Introduction

Fundamental geographic data models: - vector: represents the world using points, lines, and polygons
- raster: divides the surface up into cells of constant size. Example: background images used in web maps

Which one to use? It depends: - for social science, vector data tends to dominate because human settlements tend to have discrete borders
- for environmental sciences, raster dominates because of reliance on sensor data

Vector data

-based on points located within a coordinate reference system (CRS)

important low-level libraries in the sf package include:
- GDAL
- PROJ
- GEOS: a planar (aka flat or projected) geometry engine
- S2: a spherical (unprojected) geometry engine

Simple features

- simple features is an open standard developed by the Open Geospatial Consortium (OGC)
- there are 18 geometry types supported by the simple features specification
- only 7 are used in most geographic research
- these 7 are fully supported in R via the **sf** package

Simple features fully supported in R via **sf**:
- multilinestring
- multipoint
- multipolygon
- geometrycollection
- polygon - point - linestring

load libraries:

```
library(sf)          # classes & functions for vector data
```

```
## Linking to GEOS 3.13.1, GDAL 3.11.0, PROJ 9.6.0; sf_use_s2() is TRUE
```

```r
library(terra)       # classes & functions for raster data
```

```
## terra 1.8.54
```

```r
library(spData)      # for geographic data
library(spDataLarge) # for larger geographic data
```

In sf,
- simple feature objects in R are stored in a data frame
- geographic data are usually contained in `geom` or `geometry` column

```r
class(world)
```

```
## [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

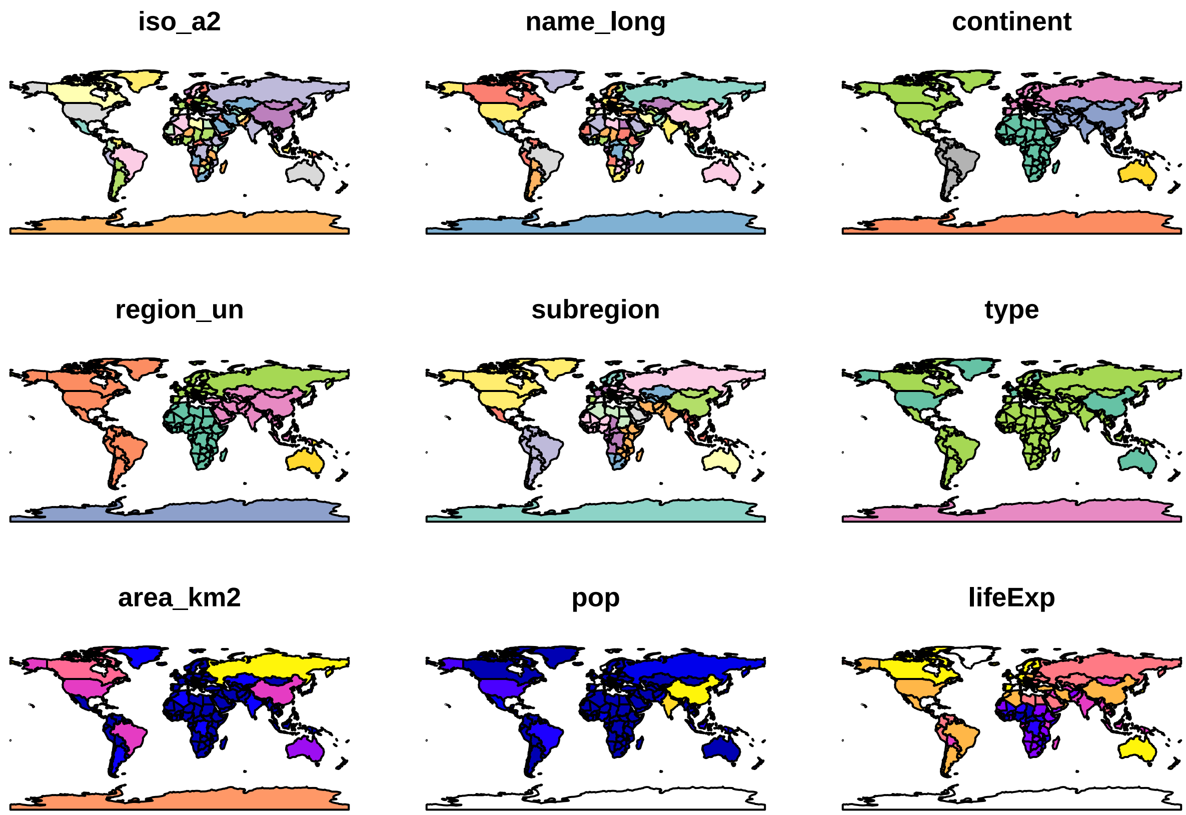
```r
names(world)
```

```
##  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
##  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap" "geom"
```

- in the `world` data, the `geom` column contains a list of all the coordinates of the country polygons
- `sf` objects can be plotted quickly using the function `plot()` or `plot.sf()`

```r
plot(world)
```

```
## Warning: plotting the first 9 out of 10 attributes; use max.plot = 10 to plot
## all
```



Here is where R shines — making statistical calculations — on “geospatial data”

```
summary(world["lifeExp"])
```

```
##     lifeExp                     geom
##  Min.   :50.62   MULTIPOLYGON :177
##  1st Qu.:64.96   epsg:4326    :   0
##  Median :72.87   +proj=long...:   0
##  Mean   :70.85
##  3rd Qu.:76.78
##  Max.   :83.59
##  NA's   :10
```

Note: if we use the `$` to access the column `lifeExp`, we won’t see the geom:

```
summary(world$lifeExp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   50.62   64.96   72.87   70.85   76.78   83.59      10
```

Subsetting `sf` objects

```
world_mini = world[1:2, 1:3]
world_mini
```

```
## Simple feature collection with 2 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -18.28799 xmax: 180 ymax: -0.95
## Geodetic CRS:  WGS 84
##   iso_a2 name_long continent                           geom
## 1     FJ      Fiji   Oceania MULTIPOLYGON (((-180 -16.55...
## 2     TZ  Tanzania    Africa MULTIPOLYGON (((33.90371 -0...
```

Note: again, we use more pythonic syntax here.

Trying the more r-esque sytax:

```r
world_mini2 <- world[1:2, 1:3]
world_mini2
```

```
## Simple feature collection with 2 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -18.28799 xmax: 180 ymax: -0.95
## Geodetic CRS:  WGS 84
##   iso_a2 name_long continent                           geom
## 1     FJ      Fiji   Oceania MULTIPOLYGON (((-180 -16.55...
## 2     TZ  Tanzania    Africa MULTIPOLYGON (((33.90371 -0...
```

- the authors of the book prefer using the `equals assignment` notation for faster typing and compatibility with other languages such as Python and Javascript.

Why simple features?

- faster reading and writing of data
- enhanced plotting performance
- `sf` objects can be treated as data frames in most operations
- consistent and intuitive
- works with the pipe operator

Importing geographic vector data

```r
world_dfr = st_read(system.file("shapes/world.gpkg", package = "spData"))
```

```
## Reading layer ‘world’ from data source
##   ‘/home/cavemancoder/R/x86_64-pc-linux-gnu-library/4.5/spData/shapes/world.gpkg’
##   using driver ‘GPKG’
## Simple feature collection with 177 features and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -89.9 xmax: 180 ymax: 83.64513
## Geodetic CRS:  WGS 84
```

```r
world_tbl = read_sf(system.file("shapes/world.gpkg", package = "spData"))

class(world_dfr)
```

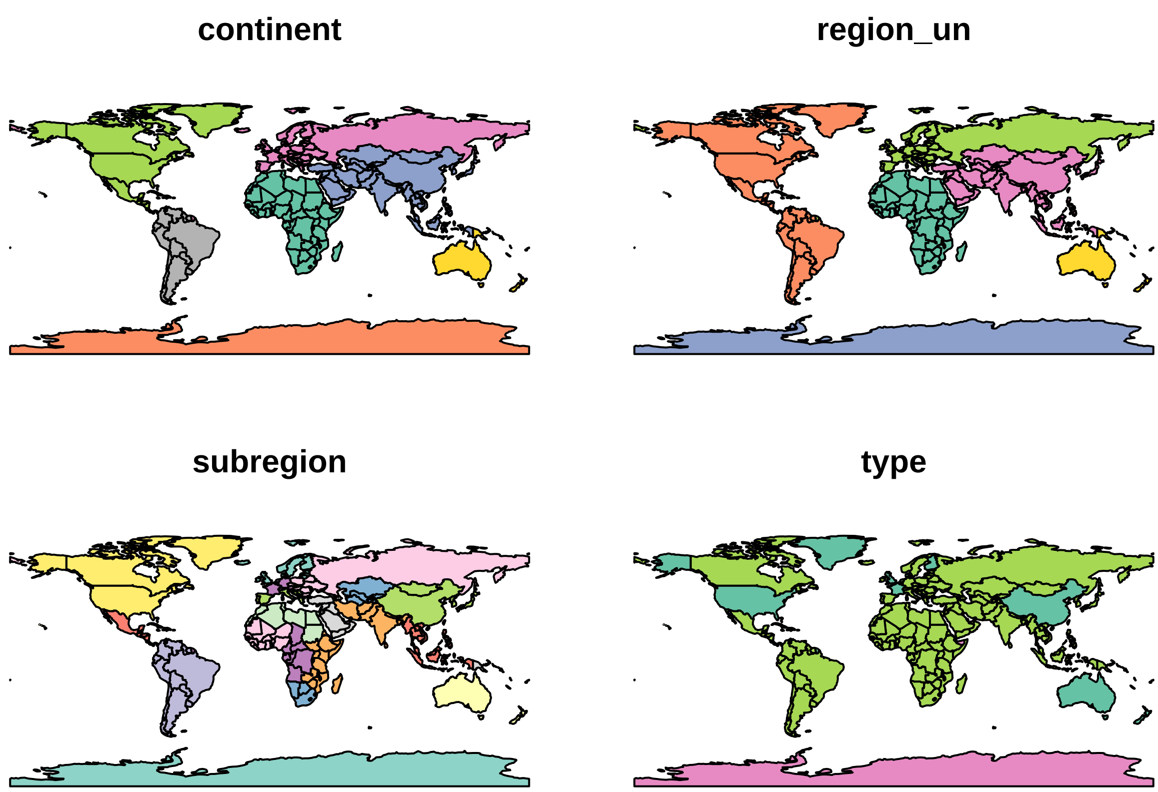
```
## [1] "sf"         "data.frame"
```

```r
class(world_tbl)
```

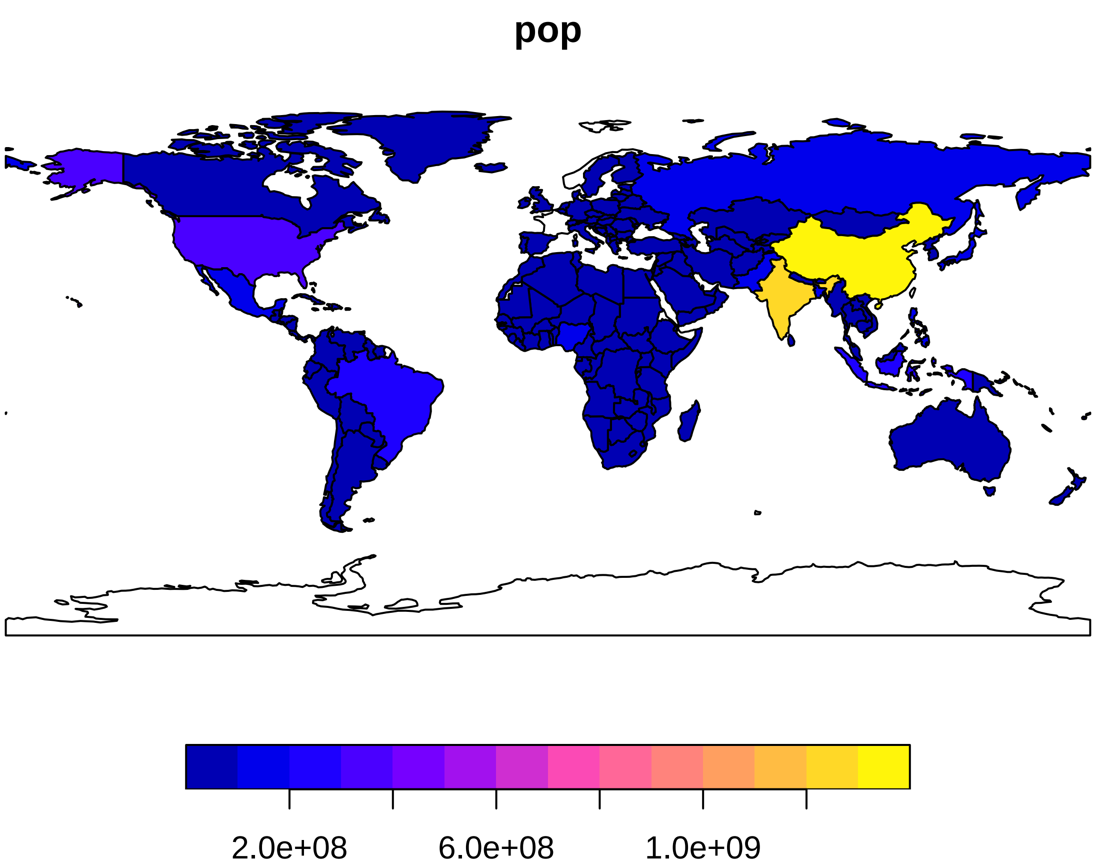
```
## [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

Basic maps

```r
plot(world[3:6])
```




```r
plot(world["pop"])
```



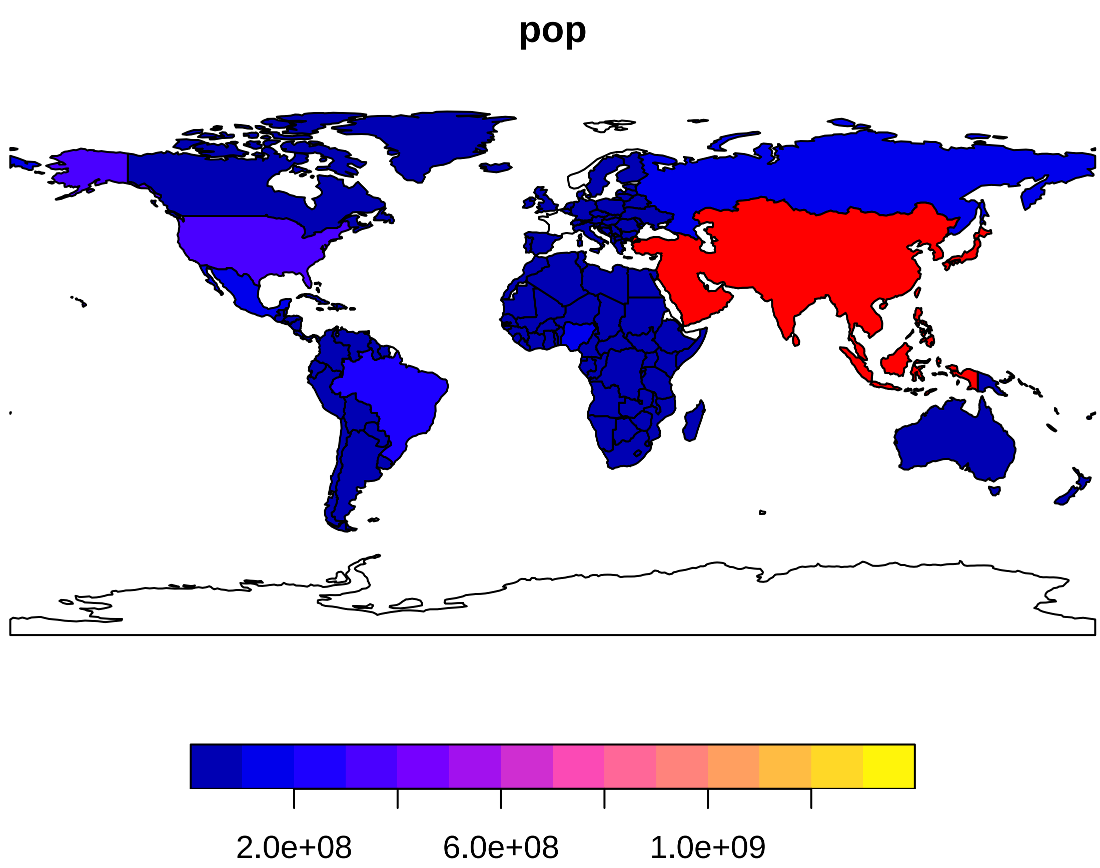
- plots can be added as layers to existing images by setting `add = TRUE`

for example:

```
world_asia = world[world$continent == "Asia",]
asia = st_union(world_asia)
```

Plotting Asia on top of the world map as layer:

```
plot(world["pop"], reset = FALSE)
plot(asia, add = TRUE, col = "red")
```

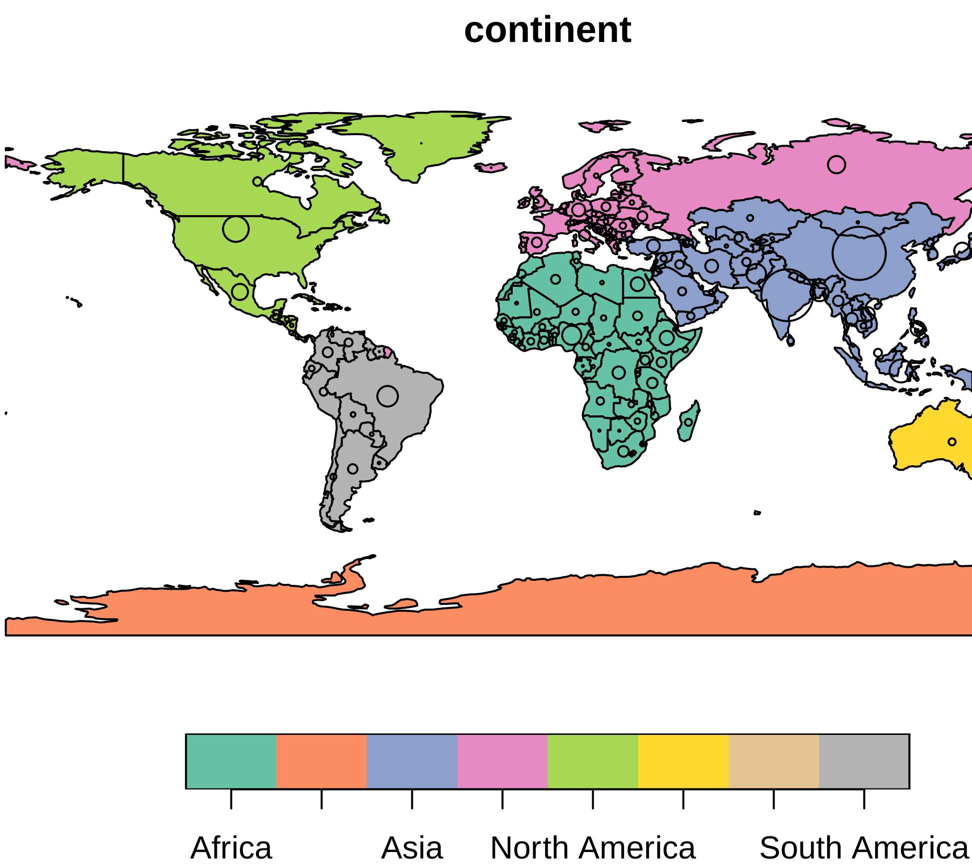


Overlaying circles on plot of map:

```
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
world_cents = st_centroid(world, of_largest = TRUE)
```

```
## Warning: st_centroid assumes attributes are constant over geometries
```

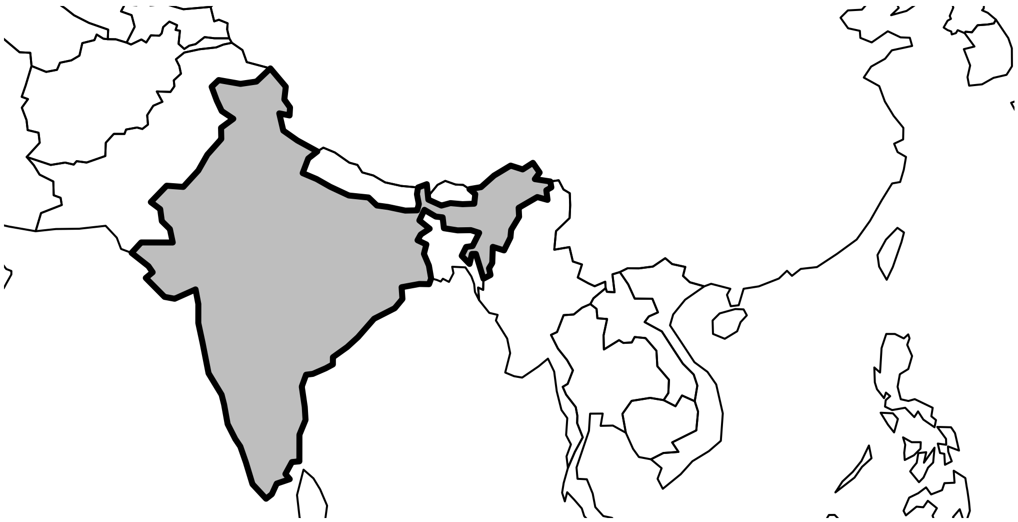
```
plot(st_geometry(world_cents), add = TRUE, cex = cex)
```



- in the code, the function `st_centroid()` was used to "convert" polygons to points. The aesthetics are controlled by the `cex` argument.

Plotting with Expanded bounding box

```
india = world[world$name_long == "India", ]
plot(st_geometry(india), expandBB = c(0, 0.2, 0.1, 1), col= "gray", lwd = 3)
plot(st_geometry(world_asia), add = TRUE)
```



Geometry types

Standard encoding for simple feature geometries:
- well-known binary (WKB): usually in hexadecimal
- well-known text (WKT): human-readable text

Commonly used geometry types in `sf`: - point: coordinates in two-, three-, or four-dimensional space. Example: `POINT (5 2)`
- linestring: a sequence of points with a straight line connecting the points. Example: `LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)`
- polygon: a sequence of points that form a closed, non-intersecting ring. Example: `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`
- Multipoint: Example: `MULTIPOINT (5 2, 1 3, 3 4, 3 2)`
- Multilinestring: Example: `MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))`
- Multipolygon: Example: `MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))`
- Geometry collection: any combination of the geometries above. Example: `GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))`

Combining geometries and non-geographic attributes

```
lnd_point = st_point(c(0.1, 51.1))
lnd_geom = st_sfc(lnd_point, crs = "ESPG:4326")
lnd_attrib = data.frame(
  name = "London",
  temperature = 25,
```

```r
  date = as.Date("2023-06-21")
)
lnd_sf = st_sf(lnd_attrib, geometry = lnd_geom)

lnd_sf
```

```
## Simple feature collection with 1 feature and 3 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 0.1 ymin: 51.1 xmax: 0.1 ymax: 51.1
## CRS:           NA
##     name temperature       date          geometry
## 1 London          25 2023-06-21 POINT (0.1 51.1)
```

- if we need to "make"draw" geometries from scatch, we add an `st_` prefix to the geometry type listed above

```r
st_point(c(5, 2))
```

```
## POINT (5 2)
```

```r
st_point(c(5, 2, 3))
```

```
## POINT Z (5 2 3)
```

```r
st_point(c(5, 2, 1), dim = "XYM")
```

```
## POINT M (5 2 1)
```

```r
st_point(c(5, 2, 3, 1))
```

```
## POINT ZM (5 2 3 1)
```

- XY is 2d, XYZ is 3d, XYZM is 3d with additional variable (like for improving measurement accuracy)

```r
# multipoint
multipoint_matrix = rbind(c(5, 2), c(1, 3), c(3, 4), c(3, 2))
st_multipoint(multipoint_matrix)
```

```
## MULTIPOINT ((5 2), (1 3), (3 4), (3 2))
```

```r
# linestring
linestring_matrix = rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2))
st_linestring(linestring_matrix)
```

```
## LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

Multi-sf examples:

```r
## POLYGON
polygon_list = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
st_polygon(polygon_list)
```

```
## POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
```

```r
## POLYGON with a hole
polygon_border = rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))
polygon_hole = rbind(c(2, 4), c(3, 4), c(3, 3), c(2, 3), c(2, 4))
polygon_with_hole_list = list(polygon_border, polygon_hole)
st_polygon(polygon_with_hole_list)
```

```
## POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))
```

```r
## MULTILINESTRING
multilinestring_list = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                            rbind(c(1, 2), c(2, 4)))
st_multilinestring(multilinestring_list)
```

```
## MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))
```

```r
## MULTIPOLYGON
multipolygon_list = list(list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))),
                         list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2))))
st_multipolygon(multipolygon_list)
```

```
## MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5)), ((0 2, 1 2, 1 3, 0 3, 0 2)))
```

```r
## GEOMETRYCOLLECTION
geometrycollection_list = list(st_multipoint(multipoint_matrix),
                               st_linestring(linestring_matrix))
st_geometrycollection(geometrycollection_list)
```

```
## GEOMETRYCOLLECTION (MULTIPOINT ((5 2), (1 3), (3 4), (3 2)), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))
```

Simple feature columns (sfc)

- a list of `sfg` objects, where an `sfg` object contains only a single simple feature geometry.
- to combine simple features in a objet with two features, we can use the `st_sfc()`
- this looks like a function analogous to the concatenate function in r-base (`c()`):

```r
point1 = st_point(c(5, 2))
point2 = st_point(c(1, 3))

points_sfc = st_sfc(point1, point2)
points_sfc
```

```
## Geometry set for 2 features
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 2 xmax: 5 ymax: 3
## CRS:           NA

## POINT (5 2)

## POINT (1 3)
```

- when `sfc` objects contain objects of the same geometry type, they can be converted into an `sfc` of polygon type
- the geometry type of an object can be verified using `st_geometry_type()`:

```
polygon_list1 = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
polygon1 = st_polygon(polygon_list1)

polygon_list2 = list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2)))
polygon2 = st_polygon(polygon_list2)

polygon_sfc = st_sfc(polygon1, polygon2)
st_geometry_type(polygon_sfc)
```

```
## [1] POLYGON POLYGON
## 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

- multilinesgring examples:

```
# sfc MULTILINESTRING
multilinestring_list1 = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                            rbind(c(1, 2), c(2, 4)))
multilinestring1 = st_multilinestring((multilinestring_list1))
multilinestring_list2 = list(rbind(c(2, 9), c(7, 9), c(5, 6), c(4, 7), c(2, 7)),
                            rbind(c(1, 7), c(3, 8)))
multilinestring2 = st_multilinestring((multilinestring_list2))
multilinestring_sfc = st_sfc(multilinestring1, multilinestring2)
st_geometry_type(multilinestring_sfc)
```

```
## [1] MULTILINESTRING MULTILINESTRING
## 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

- it is also possible to create an `sfc` object from `sfg` objects with different geometry types:

```
point_multilinestring_sfc = st_sfc(point1, multilinestring1)
st_geometry_type(point_multilinestring_sfc)
```

```
## [1] POINT           MULTILINESTRING
## 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

- `sfc` objects can store additional information on the coordinate reference system (CRS).

- the crs information can be verified with `st_crs()`:

```r
st_crs(points_sfc)
```

```
## Coordinate Reference System: NA
```

- all geometries in `sfc` objects must have the same CRS.
- the functions `st_sfc()` or `st_sf()` can be used to specify the CRS.

```r
# Set the crs with an identifier referring to an "EPSG" CRS code
points_sfc_wgs = st_sfc(point1, point2, crs = "EPSG:4326")
st_crs(points_sfc_wgs)
```

```
## Coordinate Reference System:
##   User input: EPSG:4326
##   wkt:
## GEOGCRS["WGS 84",
##     ENSEMBLE["World Geodetic System 1984 ensemble",
##         MEMBER["World Geodetic System 1984 (Transit)"],
##         MEMBER["World Geodetic System 1984 (G730)"],
##         MEMBER["World Geodetic System 1984 (G873)"],
##         MEMBER["World Geodetic System 1984 (G1150)"],
##         MEMBER["World Geodetic System 1984 (G1674)"],
##         MEMBER["World Geodetic System 1984 (G1762)"],
##         MEMBER["World Geodetic System 1984 (G2139)"],
##         MEMBER["World Geodetic System 1984 (G2296)"],
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]],
##         ENSEMBLEACCURACY[2.0]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##         SCOPE["Horizontal component of 3D system."],
##         AREA["World."],
##         BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```

The sfheaders package

- speeds up the construction, conversion, and manipulation of `sf` objects.

Example: a vector converted to `sfg_POINT`

```r
v = c(1, 1)
v_sfg_sfh = sfheaders::sfg_point(obj = v)
v_sfg_sfh
```

```
## POINT (1 1)
```

Example: creation of `sfg` objects from matrices and data frames:

```r
# from matrix
m = matrix(1:8, ncol = 2)
sfheaders::sfg_linestring(obj = m)
```

```
## LINESTRING (1 5, 2 6, 3 7, 4 8)
```

```r
# from data frame
df = data.frame(x = 1:4, y = 4:1)
sfheaders::sfg_polygon(obj = df)
```

```
## POLYGON ((1 4, 2 3, 3 2, 4 1, 1 4))
```

Other examples:

```r
sfheaders::sfc_point(obj = v)
```

```
## Geometry set for 1 feature
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 1 xmax: 1 ymax: 1
## CRS:           NA

## POINT (1 1)
```

```r
sfheaders::sfc_linestring(obj = m)
```

```
## Geometry set for 1 feature
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 5 xmax: 4 ymax: 8
## CRS:           NA

## LINESTRING (1 5, 2 6, 3 7, 4 8)
```

```r
sfheaders::sfc_polygon(obj = df)
```

```
## Geometry set for 1 feature
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 1 xmax: 4 ymax: 4
## CRS:           NA

## POLYGON ((1 4, 2 3, 3 2, 4 1, 1 4))
```

```r
sfheaders::sf_point(obj = v)
```

```
## Simple feature collection with 1 feature and 0 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 1 xmax: 1 ymax: 1
## CRS:           NA
##      geometry
## 1 POINT (1 1)
```

```r
sfheaders::sf_linestring(obj = m)
```

```
## Simple feature collection with 1 feature and 1 field
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 5 xmax: 4 ymax: 8
## CRS:           NA
##   id                       geometry
## 1  1 LINESTRING (1 5, 2 6, 3 7, ...
```

```r
sfheaders::sf_polygon(obj = df)
```

```
## Simple feature collection with 1 feature and 1 field
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 1 ymin: 1 xmax: 4 ymax: 4
## CRS:           NA
##   id                       geometry
## 1  1 POLYGON ((1 4, 2 3, 3 2, 4 ...
```

Example: defining the CRS on an object created by using `sfheaders`:

```r
df_sf = sfheaders::sf_polygon(obj = df)
st_crs(df_sf) = "EPSG:4326"
```

Spherical geometry operations with S2

- the S2 geometry engine is turned on by default

To verify if S2 engine is set to on or off:

```r
sf_use_s2()
```

```
## [1] TRUE
```

What if the S2 engine is turned off?

```r
india_buffer_with_s2 = st_buffer(india, 1) # 1 meter
sf_use_s2(FALSE)
```

```
## Spherical geometry (s2) switched off
```

```r
india_buffer_without_s2 = st_buffer(india, 1) # 1 degree
```

```
## Warning in st_buffer.sfc(st_geometry(x), dist, nQuadSegs, endCapStyle =
## endCapStyle, : st_buffer does not correctly buffer longitude/latitude data
```

```
## dist is assumed to be in decimal degrees (arc_degrees).
```

Leaving the S2 engine turned on, unless explicitly stated:

```r
sf_use_s2(TRUE)
```

```
## Spherical geometry (s2) switched on
```

Raster data

- represents the world with continuous grid of cells aka pixels
- usually consists of a raster header and a matrix that represent the pixels
- the raster header defines the CRS, origin, and extent
- the origin is usually the lower left corner
- in `terra`, the origin is the upper left corner

Resolution formula:

$$\text{resolution} = \frac{\text{xmax} - \text{xmin}}{\text{ncol}}, \frac{\text{ymax} - \text{ymin}}{\text{nrow}}$$

We can easily access and modify each single cell by: - using the ID of a cell
- explicitly specifying the rows and columns

R packages for working with raster data

- **terra**
- **stars**

Converting **terra** objects to **stars**: - `st_as_stars()`

Converting **stars** objects to **terra**: - `rast()`

Introduction to terra

- terra provides the possibility to divide the raster into smaller chunks and process them iteratively instead of loading the whole raster file into RAM

Example: basic terra usage

Creation of a `SpatRaster` object

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
my_rast = rast(raster_filepath)
class(my_rast)
```

```
## [1] "SpatRaster"
## attr(,"package")
## [1] "terra"
```

Inspection of the raster header:

```
my_rast
```

```
## class       : SpatRaster
## size        : 457, 465, 1  (nrow, ncol, nlyr)
## resolution  : 0.0008333333, 0.0008333333  (x, y)
## extent      : -113.2396, -112.8521, 37.13208, 37.51292  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source      : srtm.tif
## name        : srtm
## min value   : 1024
## max value   : 2892
```

Using dedicated functions to see a report on each raster component:

```
dim(my_rast)
```

```
## [1] 457 465   1
```

```
ncell(my_rast)
```

```
## [1] 212505
```

```
res(my_rast)
```

```
## [1] 0.0008333333 0.0008333333
```

```
ext(my_rast)
```

```
## SpatExtent : -113.239583212784, -112.85208321281, 37.1320834298579, 37.5129167631658 (xmin, xmax, ym
```

```r
crs(my_rast)
```

```
## [1] "GEOGCRS[\"WGS 84\",\n    ENSEMBLE[\"World Geodetic System 1984 ensemble\",\n        MEMBER[\"Wo
```

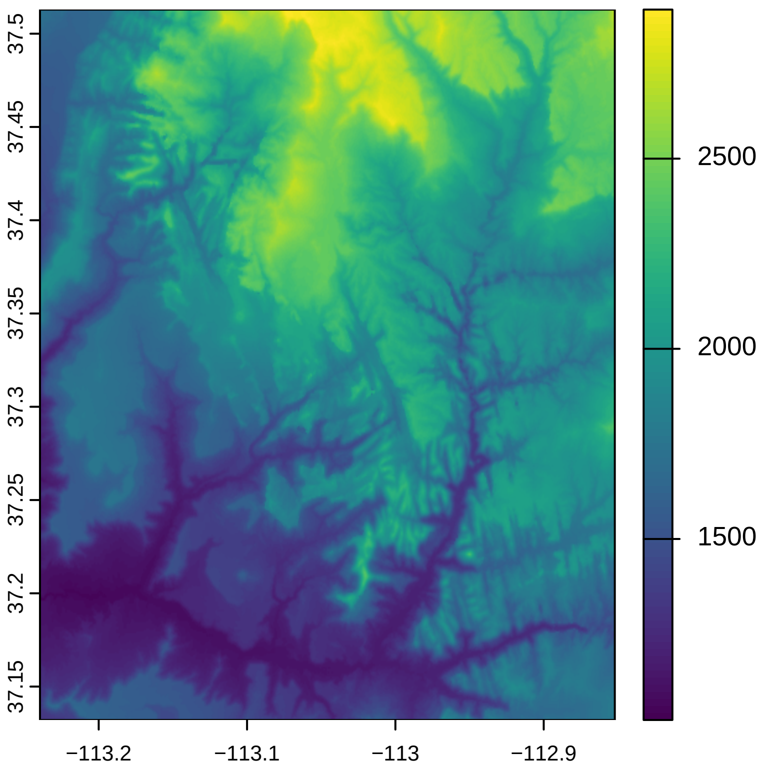
```r
inMemory(my_rast)
```

```
## [1] FALSE
```

Basic map-making

Plotting the raster file

```r
plot(my_rast)
```



Raster classes

- The `SpatRaster` class represents rasters object of **terra**

```r
single_raster_file = system.file("raster/srtm.tif", package = "spDataLarge")
single_rast = rast(raster_filepath)
```

Creating new rasters from scratch with `rast()`

```r
new_raster = rast(
  nrows = 6, ncols = 6,
  xmin = -1.5, xmax = 1.5,
  ymin = -1.5, ymax = 1.5,
  vals = 1:36
)
```

- based on the resolution formula above, the example has a resolution of 0.5 degrees.
- the unit is degrees since the default CRS of raster objects is WGS84.
- the CRS can be specified using the `crs` argument

The `SpatRaster` class also handles multiple layers:

```r
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast = rast(multi_raster_file)
multi_rast
```

```
## class       : SpatRaster
## size        : 1428, 1128, 4  (nrow, ncol, nlyr)
## resolution  : 30, 30  (x, y)
## extent      : 301905, 335745, 4111245, 4154085  (xmin, xmax, ymin, ymax)
## coord. ref. : WGS 84 / UTM zone 12N (EPSG:32612)
## source      : landsat.tif
## names       : landsat_1, landsat_2, landsat_3, landsat_4
## min values  :      7550,      6404,      5678,      5252
## max values  :     19071,     22051,     25780,     31961
```

Displaying the number of layers with `nlyr()`:

```r
nlyr(multi_rast)
```

```
## [1] 4
```

- in multi-layer raster objects, layers can be “subset” using `[[]]` or `$`
- `terra::subset()` can also be used to select layers

```r
multi_rast3 = subset(multi_rast, 3)
multi_rast4 = subset(multi_rast, 4)
```

- just like in r-base, combining can be accomplished using the comcatenate operator:

```r
multi_rast34 = c(multi_rast3, multi_rast4)
```

Geographic coordinate reference systems

- longitude: East-West angular distance from the Prime Meridian plane
- latitude: North-South angular distance from the equatorial plane
- Earth is assumed to be spherical (less accurate) or ellipsoidal (more accurate)
- equatorial radius is about 11.5 km longer than the polar radius
- for the ellipsoidal model, what ellipsoid to use is defined by the *datum*
- geocentric datum (WGS84) has its center located in the Earth's center of gravity
- local datum (e.g. NAD83) the ellipsoidal surface is shifted to align with the surface at a particular location
- with local datum, local variations in the earth's surface are accounted for

Projected coordinate reference systems

- the three-dimensional surface of the Earth is "projected" on a flat surface
- the transformation introduces some deformations
- some parts get distorted
- main projection types include conic, cylindrical, and planar

Units

- most CRSs use meters, but some use feet

```
luxembourg = world[world$name_long == "Luxembourg",]
st_area(luxembourg)
```

```
## 2408817306 [m^2]
```

- be careful with the units when doing calculations
- if we simply divide the area with the conversion factor 1000000 to convert the area to sq. km:

```
st_area(luxembourg) / 1000000
```

```
## 2408.817 [m^2]
```

the result is still in square meters. This is not correct.

- the better way is to use the **units** package to convert units:

```r
units::set_units(st_area(luxembourg), km^2)
```

```
## 2408.817 [km^2]
```

- the `res()` command will not show the units of the raster data
- only **sf** supports units
- we have to know the units used in the projection of choice

Exercises

E1. Use `summary()` on the geometry column of the `world` data object that is included in the **spData** package. What does the output tell us about:

```
Its geometry type?
The number of countries?
Its coordinate reference system (CRS)?
```

Solution:

```r
summary(world$geom)
```

```
##  MULTIPOLYGON     epsg:4326 +proj=long...
##           177             0             0
```

Ans: Multipolygon, 177 countries, ESPG:4326

E2. Run the code that 'generated' the map of the world in Section 2.2.3 (Basic map-making). Find two similarities and two differences between the image on your computer and that in the book.

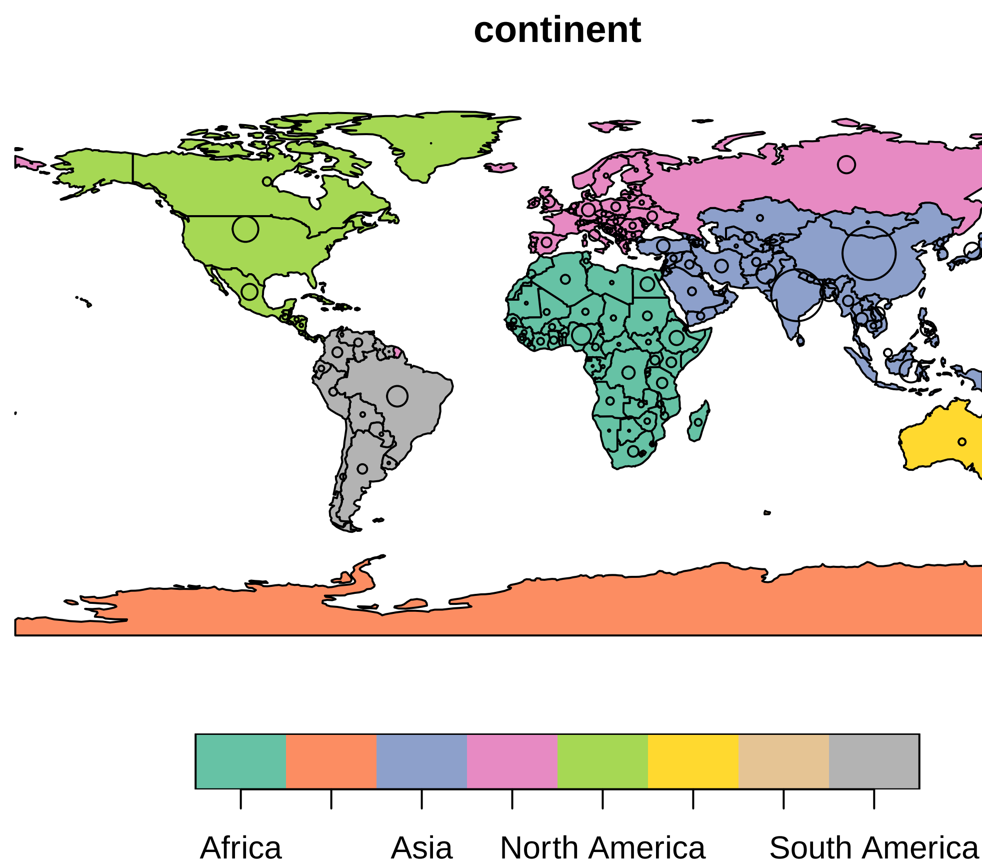
```
What does the `cex` argument do (see `?plot`)?
Why was `cex` set to the `sqrt(world$pop) / 10000`?
Bonus: experiment with different ways to visualize the global population.
```

Solution:

```r
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
world_cents = st_centroid(world, of_largest = TRUE)
```

```
## Warning: st_centroid assumes attributes are constant over geometries
```

```r
plot(st_geometry(world_cents), add = TRUE, cex = cex)
```



Ans: Two similarities: 1. map color.
2. circle marks.

Two differences: 1. legend 2. title

`cex` controls the symbol size. `cex` is set to the square root of population in order to make the area of the circle proportional to the population – doubling the population will correspond to a circle with twice the area. If we don't use sqrt, then we will effectively double the diameter instead.

E3. Use `plot()` to create maps of Nigeria in context (see Section 2.2.3).

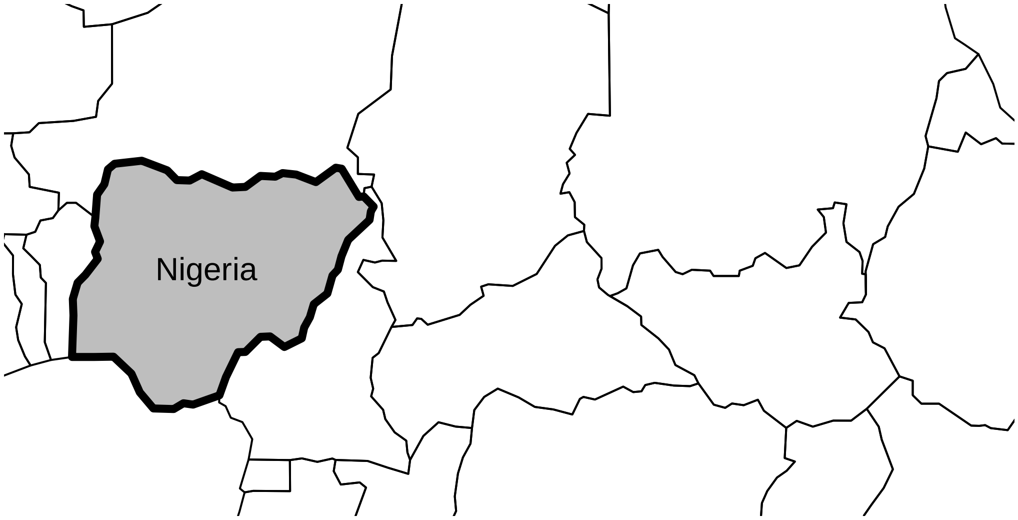
```
Adjust the `lwd`, `col` and `expandBB` arguments of plot().
Challenge: read the documentation of `text()` and annotate the map.
```

Solution:

```r
world_africa = world[world$continent == "Africa", ]
plot(st_geometry(world_africa))
```

```
nigeria = world[world$name_long == "Nigeria", ]
plot(
  st_geometry(nigeria),
  expandBB = c(0, 0.1, 0.2, 2),
  col = "gray",
  lwd = 4
)

plot(st_geometry(world_africa), add = TRUE)
text(7.985654, 9.544975, "Nigeria")
```



E4. Create an empty `SpatRaster` object called `my_raster` with 10 columns and 10 rows. Assign random values between 0 and 10 to the new raster and plot it.

Solution:

```
my_raster = rast(
  nrows = 10, ncols = 10,
  vals = sample(0:10, 100, replace = TRUE)
)

# x11()
# plot(my_raster,
#      main = "My Raster"
# )
my_raster
```

```
## class       : SpatRaster
## size        : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 36, 18  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (CRS84) (OGC:CRS84)
## source(s)   : memory
## name        : lyr.1
## min value   :     0
## max value   :    10
```

E5. Read-in the `raster/nlcd.tif` file from the **spDataLarge** package. What kind of information can you get about the properties of this file?

Solution:

```
raster_file = system.file("raster/nlcd.tif", package = "spDataLarge")
exer5_rast = rast(raster_file)
exer5_rast
```

```
## class       : SpatRaster
## size        : 1359, 1073, 1  (nrow, ncol, nlyr)
## resolution  : 31.5303, 31.52466  (x, y)
## extent      : 301903.3, 335735.4, 4111244, 4154086  (xmin, xmax, ymin, ymax)
## coord. ref. : NAD83 / UTM zone 12N (EPSG:26912)
## source      : nlcd.tif
## color table : 1
## categories  : levels
## name        :   levels
## min value   :    Water
## max value   : Wetlands
```

Ans: Class is SpatRaster. Size is 1359 x 1073 (cols, rows). Number of layers: 1. Resolution (31.5, 31.5). xmin = 301903.3, xmax = 335735.4, ymin = 4111244, ymax = 4154086, CRS = ESPG:26912, datum = NAD83 and others.

E6. Check the CRS of the `raster/nlcd.tif` file from the **spDataLarge** package. What kind of information you can learn from it?

Soln: Datum is NAD83 or the North American Datum of 1983. Coordinate Reference System is ESPG:26912. This is a projected coordinate system common in North America.