2 Geographic data in R

Prerequisites

This is the first practical chapter of the book, and therefore it comes with some software requirements. We assume that you have an up-to-date version of R installed and that you are comfortable using software with a command-line interface such as the integrated development environment (IDE) RStudio.[12]

After you've checked you R installation and brushed-up on your R skills where appropriate, the next step is to install and load the packages used in this chapter. Packages are installed with `install.packages("package_name")` . We will use the two packages that provide functions for handling spatial data, loaded with `library(package_name)` as follows:

```
library(sf)          # classes and functions for vector data
library(raster)      # classes and functions for raster data
```

The chapter also relies on two data packages: **spData** and **spDataLarge**. Importantly, the **spDataLarge** package needs be installed with the following command: `install.packages("spDataLarge", repos = "https://nowosad.github.io/drat/", type = "source")` .

```
library(spData)        # load geographic data
library(spDataLarge)   # load larger geographic data
```

> On Mac and Linux a few requirements must be met to install **sf**. These are described in the package's README at github.com/r-spatial/sf.

This chapter will provide brief explanations of the fundamental geographic data models: vector and raster. We will introduce the theory behind each data model and the disciplines in which they predominate, before demonstrating their implementation in R.

The *vector data model* represents the world using points, lines and polygons. These have discrete, well-defined borders, meaning that vector datasets usually have a high level of precision (but not necessarily accuracy as we will see in 2.4). The *raster data model* divides the surface up into cells of constant size. Raster datasets are the basis of background images used in web-mapping and have been a vital source of geographic data since the origins of aerial photography and satellite-based remote sensing devices. Rasters aggregate spatially specific features to a given resolution, meaning that they are consistent over space and scalable (many worldwide raster datasets are available).

Which to use? The answer likely depends on your domain of application:

- Vector data tends to dominate the social sciences because human settlements tend to have discrete borders.
- Raster often dominates in environmental sciences because of the reliance on remote sensing data.

There is much overlap in some fields and raster and vector datasets can be used side-by-side: ecologists and demographers, for example, commonly use both vector and raster data. Whether your work involves more use of vector or raster datasets, it is worth understanding the underlying data model before using them, as discussed in subsequent chapters. This book uses **sf** and **raster** packages to work with vector data and raster datasets respectively.

2.1 Vector data

> Take care when using the word 'vector' as it can have two meanings in this book: geographic vector data and the `vector` class (note the `monospace` font) in R. The former is a data model, the latter is an R class just like `data.frame` and `matrix`. Still, there is a link between the two: the spatial coordinates which are at the heart of the geographic vector data model can be represented in R using `vector` objects.

The geographic vector model is based on points located within a coordinate reference system (CRS). Points can represent self-standing features (e.g. the location of a bus stop) or they can be linked together to form more complex geometries such as lines and polygons. Most point geometries contain only two dimensions (3 dimensional CRSs contain an additional z value, typically representing height above sea level).

In this system London, for example, can be represented by the coordinates `c(-0.1, 51.5)`. This means that its location is -0.1 degrees east and 55.5 degrees north of the origin. The origin in this case is at 0 degrees longitude (the Prime Meridian) and 0 degree latitude (the Equator) in a geographic ('lon/lat') CRS (Figure 2.1, left panel). The same point could also be approximated in a projected CRS with 'Easting/Northing' values of `c(530000, 180000)` in the British National Grid (BNG), meaning that London is located 530 km *East* and 180 km *North* of the $origin$ of the CRS. This can be verified visually: slightly more than 5 'boxes' — square areas bounded by the grey grid lines 100 km in width — separate the point representing London from the origin (Figure 2.1, right panel).

The location of BNG's origin, in the sea beyond South West Peninsular, ensures that most locations in the UK have positive Easting and Northing values.[13] There is more to CRSs, as described in sections 2.3 and 5.2 but, for the purposes of this section, it is sufficient to know that coordinates consist of two numbers representing distance from an origin, usually in x then y dimensions.

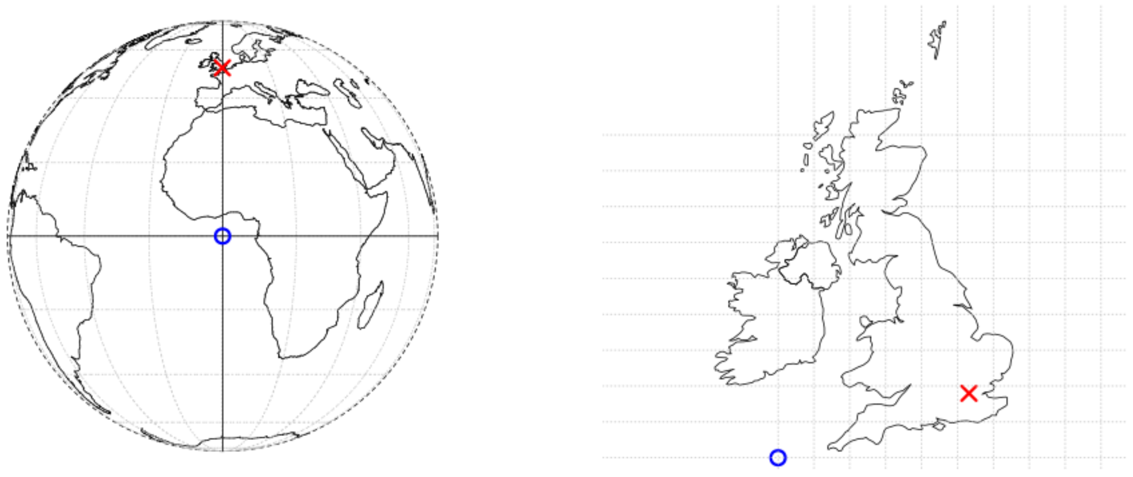


Figure 2.1: Illustration of vector (point) data in which location of London (the red X) is represented with reference to an origin (the blue circle). The left plot represents a geographic CRS with an origin at 0° longitude and latitude. The right plot represents a projected CRS with an origin located in the sea west of the South West Peninsula.

2.1.1 An introduction to simple features

Simple features is an open standard developed and endorsed by the Open Geospatial Consortium (OGC) to represent a wide range of geographic information. It is a hierarchical data model that simplifies geographic data by condensing a complex range of geographic forms into a single geometry class. Only 7 out of 17 possible types of simple feature are currently used in the vast majority of GIS operations (Figure 2.2). The R package **sf** (Pebesma 2018) fully supports all of these (including plotting methods etc.).[14]

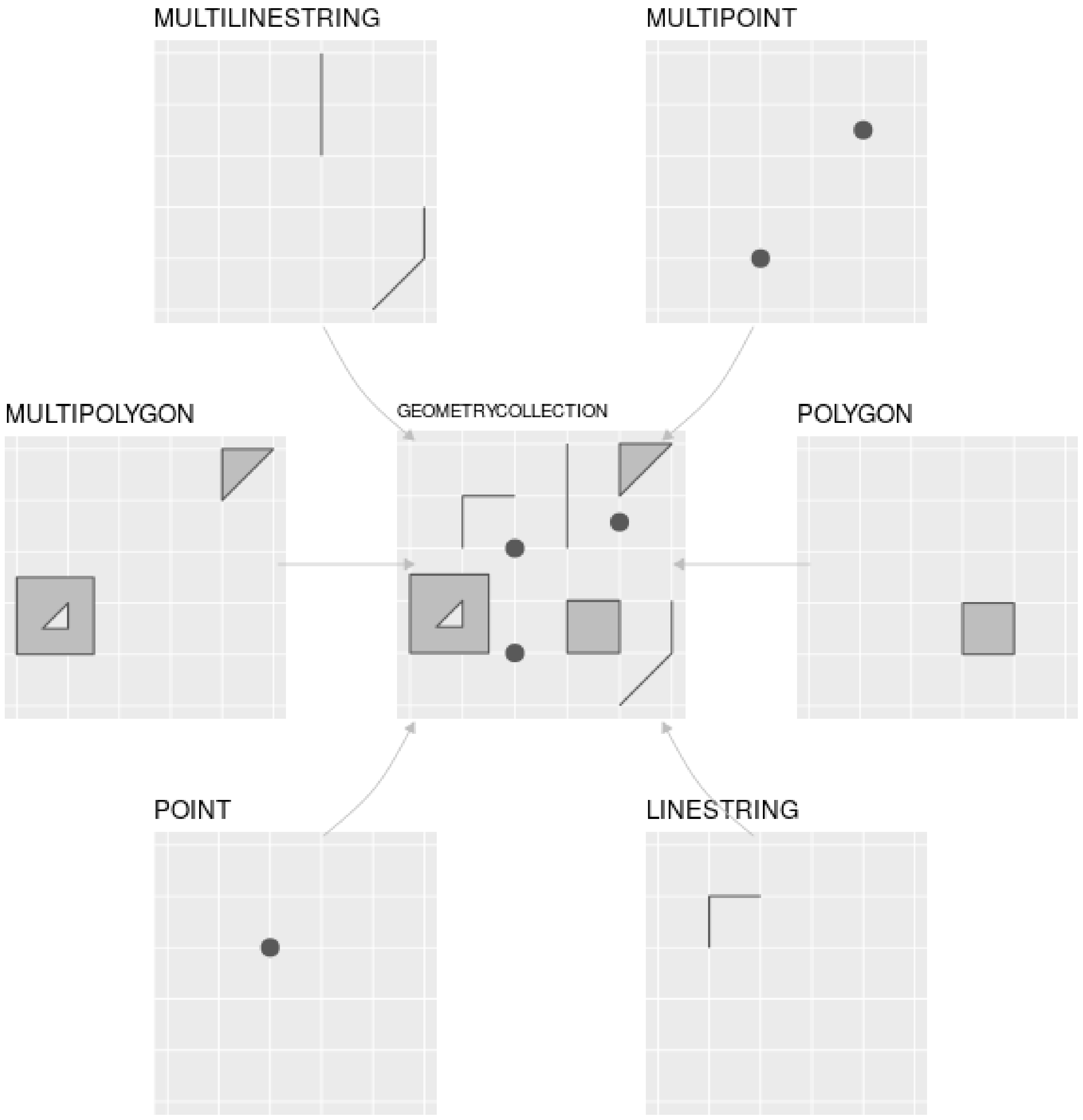


Figure 2.2: The subset of the Simple Features class hierarchy supported by sf.

sf can represent all common vector geometry types (raster data classes are not supported by **sf**): points, lines, polygons and their respective 'multi' versions (which group together features of the same type into a single feature). **sf** also supports geometry collections, which can contain multiple geometry types in a single object. Given the breadth of geographic data forms, it may come as a surprise that a class system to support all of them is provided in a single package, which can be installed from CRAN:[15] **sf** incorporates the functionality of the three main packages of the **sp** paradigm (**sp** (Pebesma and Bivand 2018) for the class system, **rgdal** (Bivand, Keitt, and Rowlingson 2018) for reading and writing data, **rgeos** (Bivand and Rundel 2017) for spatial operations undertaken by GEOS) in a single, cohesive whole. This is well-documented in **sf**'s vignettes.

As the first vignette explains, simple feature objects in R are stored in a data frame, with geographic data occupying a special column, a 'list-column'. This column is usually named 'geom' or 'geometry'. We will use the `world` dataset provided by the **spData**, loaded at the beginning of this chapter (see nowosad.github.io/spData for a list datasets loaded by the package). `world` is a spatial object containing spatial and attribute columns, the names of which are returned by the function `names()` (the last column contains the geographic information):

```
names(world)
#>  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion"
#>  [6] "type"      "area_km2"  "pop"       "lifeExp"   "gdpPercap"
#> [11] "geom"
```

It is the contents of this modest-looking `geom` column that gives `sf` objects their spatial powers, a 'list-column' that contains all the coordinates. The **sf** package provides a `plot()` method for visualizing geographic data: the follow command creates Figure 2.3.

```
plot(world)
```

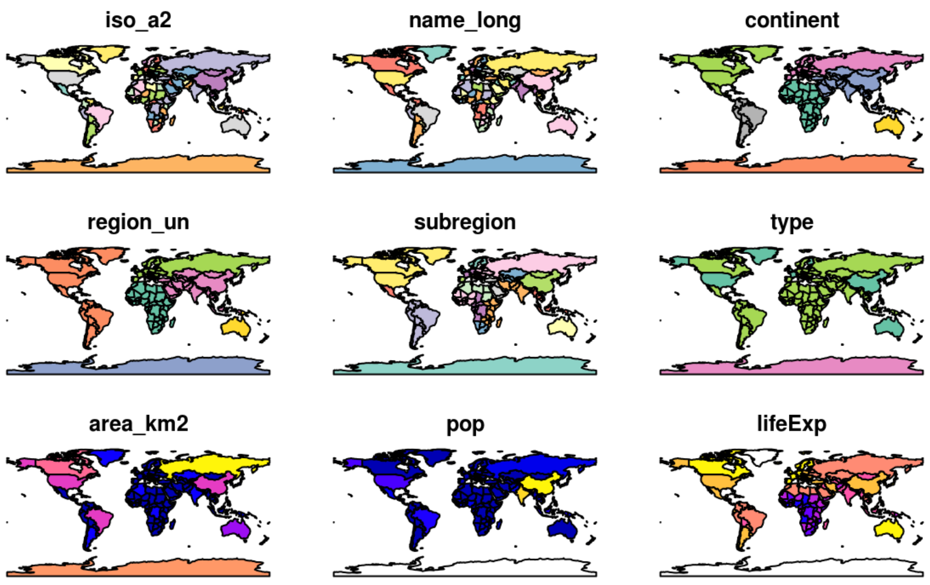


Figure 2.3: A spatial plot of the world using the sf package, with a facet for each attribute.

Note that instead of creating a single map, as most GIS programs would, the `plot()` command has created multiple maps, one for each variable in the `world` datasets. This behavior can be useful for exploring the spatial distribution of different variables and is discussed further in 2.1.3 below.

Being able to treat spatial objects as regular data frames with spatial powers has many advantages, especially if you are already used to working with data frames. The commonly used `summary()` function, for example, provides a useful overview of the variables within the `world` object.

```
summary(world["lifeExp"])
#>     lifeExp                geom    
#>  Min.   :48.9   MULTIPOLYGON :177  
#>  1st Qu.:64.3   epsg:4326    :  0  
#>  Median :72.8   +proj=long...:  0  
#>  Mean   :70.6                      
#>  3rd Qu.:77.1                      
#>  Max.   :83.6                      
#>  NA's   :9
```

Although we have only selected one variable for the `summary` command, it also outputs a report on the geometry. This demonstrates the 'sticky' behavior of the geometry columns of **sf** objects, meaning the geometry is kept unless the user deliberately removes them, as we'll see in section 3.2. The result provides a quick summary of both the non-spatial and spatial data contained in `world` : the mean average life expectancy is 71 years (ranging from less than 50 to more than 80 years with a median of 73 years) across all countries.

> The word `MULTIPOLYGON` in the summary output above refers to the geometry type of features (countries) in the `world` object. This representation is necessary for countries with islands such as Indonesia and Greece. Other geometry types are described in section 2.1.5.

It is worth taking a deeper look at the basic behavior and contents of this simple feature object, which can usefully be thought of as a '**S**patial data**F**rame).

`sf` objects are easy to subset. The code below shows its first two rows and three columns. The output shows two major differences compared with a regular `data.frame` : the inclusion of additional geographic data (`geometry type` , `dimension` , `bbox` and CRS information - `epsg (SRID)` , `proj4string`), and the presence of final `geometry` column:

```
world[1:2, 1:3]
#> Simple feature collection with 2 features and 3 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 11.6 ymin: -17.9 xmax: 75.2 ymax: 38.5
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
#>   iso_a2   name_long continent                           geom
#> 1     AF Afghanistan      Asia MULTIPOLYGON (((61.2 35.7, ...
#> 2     AO      Angola    Africa MULTIPOLYGON (((16.3 -5.88,...
```

All this may seem rather complex, especially for a class system that is supposed to be simple. However, there are good reasons for organizing things this way and using **sf**.

2.1.2 Why simple features?

Simple features is a widely supported data model that underlies data structures in many GIS applications including QGIS and PostGIS. A major advantage of this is that using the data model ensures your work is cross-transferable to other set-ups, for example importing from and exporting to spatial databases.

A more specific question from an R perspective is “why use the **sf** package when **sp** is already tried and tested”? There are many reasons (linked to the advantages of the simple features model) including:

- Fast reading and writing of data
- Enhanced plotting performance
- **sf** objects can be treated as data frames in most operations
- **sf** functions can be combined using `%>%` operator and works well with the tidyverse collection of R packages
- **sf** function names are relatively consistent and intuitive (all begin with `st_`)

Due to such advantages some spatial packages (including **tmap**, **mapview** and **tidycensus**) have added support for **sf**. However, it will take many years for most packages to transition and some will never switch. Fortunately these can still be used in a workflow based on `sf` objects, by converting them to the `Spatial` class used in **sp**:

```r
library(sp)
world_sp = as(world, Class = "Spatial")
# sp functions ...
```

`Spatial` objects can be converted back to `sf` in the same way or with `st_as_sf()`:

```r
world_sf = st_as_sf(world_sp, "sf")
```

2.1.3 Basic map making

You can quickly create basic maps in **sf** with the base `plot()` function. By default, **sf** creates a multi-panel plot (like **sp**’s `spplot()`), one sub-plot for each variable (see left-hand image in Figure 2.4).

```r
plot(world[3:4])
plot(world["pop"])
```

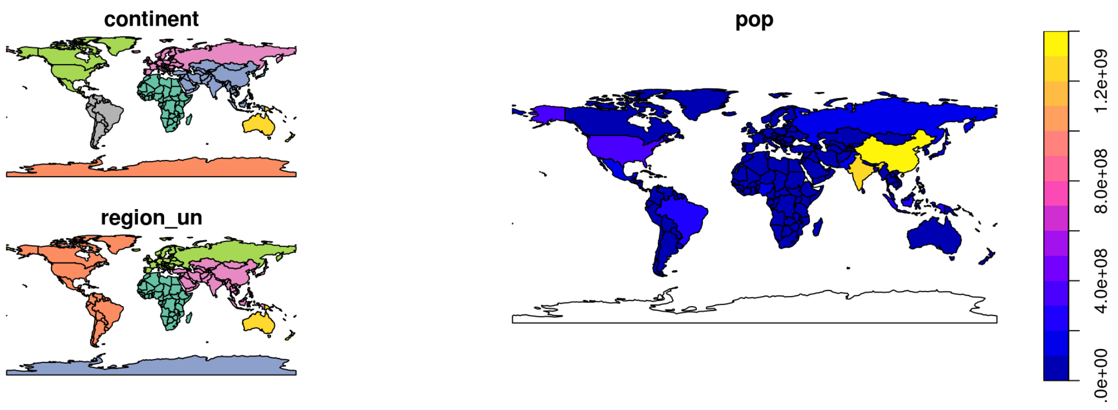


Figure 2.4: Plotting with sf, with multiple variables (left) and a single variable (right).

As with **sp**, you can add further layers to your maps using the `add = TRUE` -argument of the `plot()` function .[16] To illustrate this, and prepare for content covered in chapters 3 and 4 on attribute and spatial data operations, we will subset and combine countries in the `world` object, which creates a single object representing Asia:

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

We can now plot the Asian continent over a map of the world. Note that this only works when the initial plot has only one facet and when `reset` (which resets plot settings) is set to `FALSE` :

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

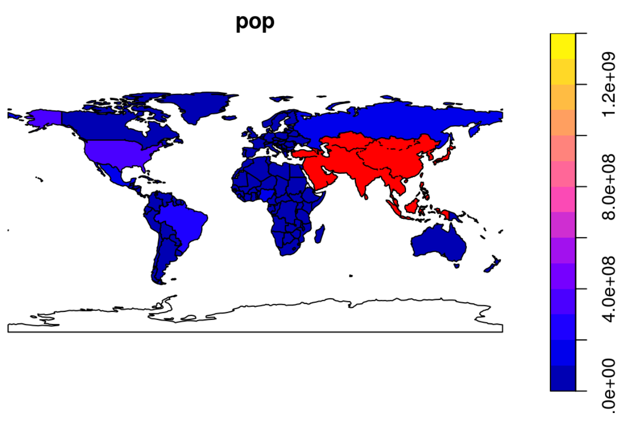


Figure 2.5: A plot of Asia added as a layer on top of countries worldwide.

This can be very useful for quickly checking the geographic correspondence between two or more layers: the `plot()` function is fast to execute and requires few lines of code, but does not create interactive maps with a wide range of options. For more advanced map making we recommend using a dedicated visualization package such as **tmap**, **ggplot2**, **mapview**, or **leaflet**.

2.1.4 Base plot arguments

sf simplifies spatial data objects compared with **sp** and provides a near-direct interface to GDAL and GEOS C++ functions. In theory this should make **sf** faster than **sp**/**rgdal**/**rgeos**. This section introduces **sf** classes in preparation for subsequent chapters which deal with vector data (in particular Chapters 4 and 5).

As a final exercise, we will see one way of how to do a spatial overlay in **sf**. First, we convert the countries of the world into centroids, and then subset those in Asia. Finally, the `summary` command tells us how many centroids (countries) are part of Asia (43) and how many are not (134).

```
world_centroids = st_centroid(world)
sel_asia = st_intersects(world_centroids, asia, sparse = FALSE)
#> although coordinates are longitude/latitude, st_intersects assumes that they are planar
```

```
summary(sel_asia)
#>       V1
#>   Mode :logical
#>   FALSE:134
#>   TRUE :43
```

Note: `st_intersects()` uses GEOS in the background for the spatial overlay operation (see also Chapter 4).

Since **sf**'s `plot()` function builds on base plotting methods, you may also use its many optional arguments (see `?graphics::plot` and `?par`). This provides many options, but may not be the most concise way to generate maps for publication (see Chapter 9). A simple illustration of the flexibility of `plot()` is provided in Figure 2.6, which adds circles representing population size to a map of the world. A basic version of the map can be created with the following commands (see exercises in section 2.5 and the script `02-contplot` in the GitHub repo for further details):

```
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
plot(st_geometry(world_centroids), add = TRUE, cex = cex)
```

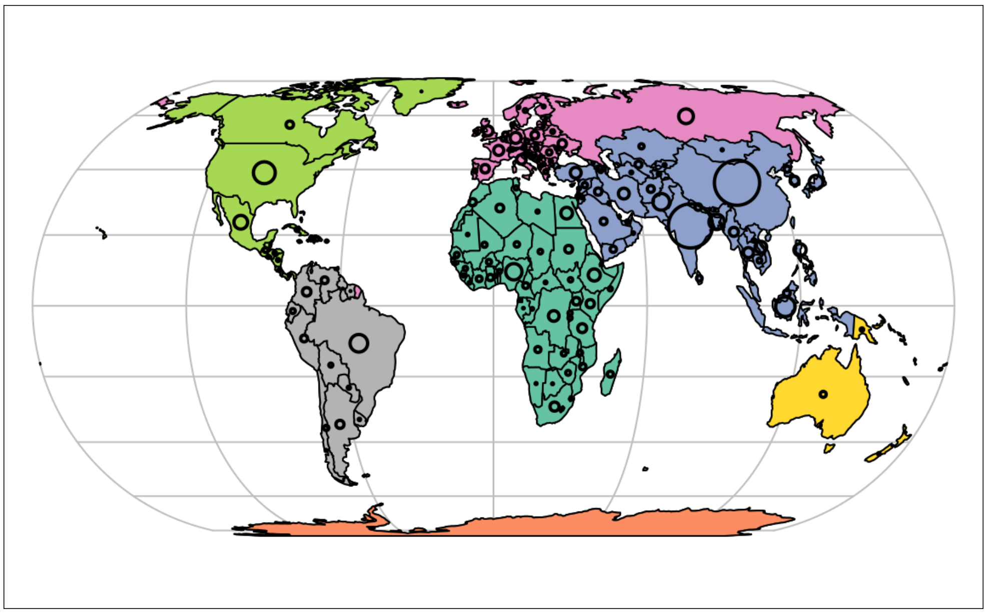


Figure 2.6: Country continents (represented by fill color) and 2015 populations (represented by points, with point area proportional to population) worldwide.

2.1.5 Simple feature classes

To understand new data formats in depth, it often helps to build them from the ground up. This section walks you through vector spatial classes step-by-step, from the elementary simple feature geometry to simple feature objects of class `sf` representing complex spatial data. Before describing each geometry type that the **sf** package supports, it is worth taking a step back to understand the building blocks of `sf` objects. As stated in section 2.1.1, simple features are simply data frames with at least one special column that makes it spatial. These spatial columns are often called `geom` or `geometry` and can be like non-spatial columns: `world$geom` refers to the spatial element of the `world` object described above. These geometry columns are 'list columns' of class `sfc`. In turn, `sfc` objects are composed of one or more objects of class `sfg`: simple feature geometries.

To understand how the spatial components of simple features work, it is vital to understand simple feature geometries. For this reason we cover each currently supported `sfg` type in the next subsections before moving on to describe how these can be combined to form `sfc` and eventually full `sf` objects.

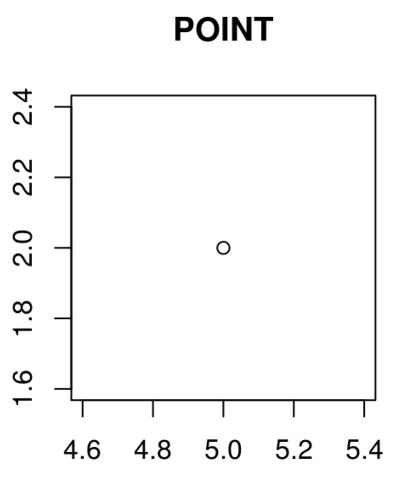
2.1.5.1 Simple feature geometry types

Geometries are the basic building blocks of simple features. Simple features in R can take on one of the 17 geometry types supported by the **sf** package. In this chapter we will focus on the seven most commonly used types: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`. Find the whole list of possible feature types in the PostGIS manual.

Generally, well-known binary (WKB) or well-known text (WKT) are the standard encoding for simple feature geometries. WKB representations are usually hexadecimal strings easily readable for computers. This is why GIS and spatial databases use WKB to transfer and store geometry objects. WKT, on the other hand, is a human-readable text markup description of simple features. Both formats are exchangeable, and if we present one, we will naturally choose the WKT representation.

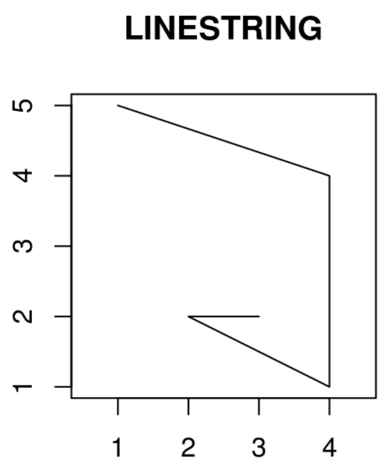
The basis for each geometry type is the point. A point is simply a coordinate in 2D, 3D or 4D space (see `vignette("sf1")` for more information) such as:

- `POINT (5 2)`



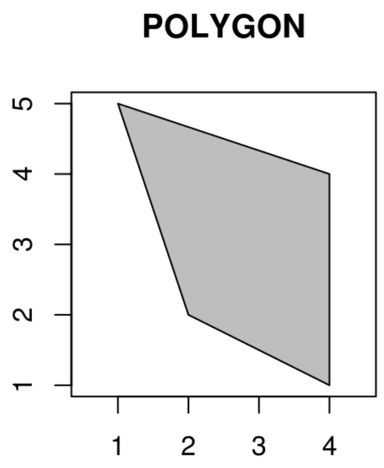
A linestring is a sequence of points with a straight line connecting the points, for example:

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

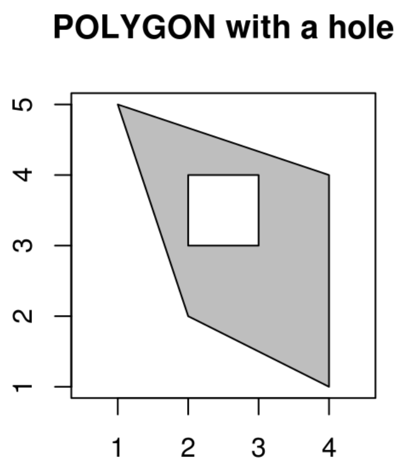


A polygon is a sequence of points that form a closed, non-intersecting ring. Closed means that the first and the last point of a polygon have the same coordinates. By definition, a polygon has one exterior boundary (outer ring) and can have zero or more interior boundaries (inner rings), also known as holes.

- Polygon without a hole - `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`

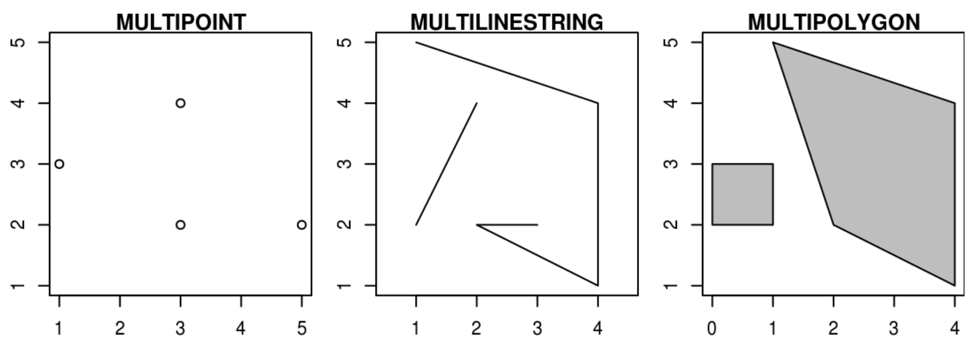


- Polygon with one hole - `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))`



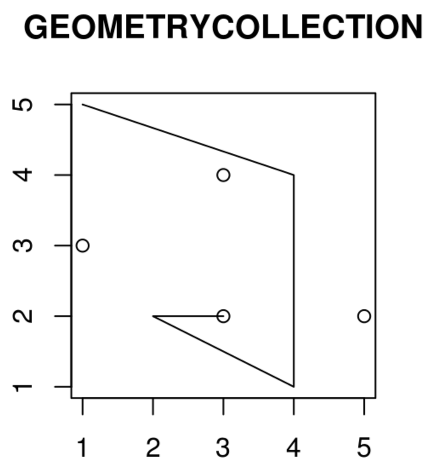
So far we have created geometries with only one geometric entity per feature. However, **sf** also allows multiple geometries to exist within a single feature (hence the term 'geometry collection') using "multi" version of each geometry type:

- Multipoint - `MULTIPOINT (5 2, 1 3, 3 4, 3 2)`
- Multistring - `MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))`
- Multipolygon - `MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))`



Finally, a geometry collection might contain any combination of geometry types:

- Geometry collection - `GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)))`



2.1.5.2 Simple feature geometry (sfg) objects

The `sfg` class represents the different simple feature geometry types: point, linestring, polygon (and their ‘multi’ equivalents, such as multipoints) or geometry collection.

Usually you are spared the tedious task of creating geometries on your own since you can simply import an already existing spatial file. However, there are a set of functions to create simple feature geometry objects (`sfg`) from scratch if needed. The names of these functions are simple and consistent, as they all start with the `st_` prefix and end with the name of the geometry type in lowercase letters:

- A point - `st_point()`
- A linestring - `st_linestring()`
- A polygon - `st_polygon()`
- A multipoint - `st_multipoint()`
- A multilinestring - `st_multilinestring()`
- A multipolygon - `st_multipolygon()`
- A geometry collection - `st_geometrycollection()`

`sfg` objects can be created from three native data types:

1. A numeric vector - a single point
2. A matrix - a set of points, where each row contains a point - a multipoint or linestring
3. A list - any other set, e.g. a multilinestring or geometry collection

To create point objects, we use the `st_point()` function in conjunction with a numeric vector:

```r
# note that we use a numeric vector for points
st_point(c(5, 2)) # XY point
#> POINT (5 2)
st_point(c(5, 2, 3)) # XYZ point
#> POINT Z (5 2 3)
st_point(c(5, 2, 1), dim = "XYM") # XYM point
#> POINT M (5 2 1)
st_point(c(5, 2, 3, 1)) # XYZM point
#> POINT ZM (5 2 3 1)
```

XY, XYZ and XYZM types of points are automatically created based on the length of a numeric vector. Only the XYM type needs to be specified using a `dim` argument.

By contrast, use matrices in the case of multipoint (`st_multipoint()`) and linestring (`st_linestring()`) objects:

```r
# the rbind function simplifies the creation of matrices
## 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)

## 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)
```

Finally, use lists for the creation of multilinestrings, (multi-)polygons and geometry collections:

```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))


## 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))
```

```
## 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))
```

```
## 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)))
```

```
## GEOMETRYCOLLECTION
gemetrycollection_list = list(st_multipoint(multipoint_matrix),
                              st_linestring(linestring_matrix))
st_geometrycollection(gemetrycollection_list)
#> GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3
```

2.1.5.3 Simple feature geometry column

One `sfg` object contains only a single simple feature geometry. A simple feature geometry column (`sfc`) is a list of `sfg` objects, which is additionally able to contain information about the coordinate reference system in use. For instance, to combine two simple features into one object with two features, we can use the `st_sfc()` function. This is important since `sfc` represents the geometry column in **sf** data frames:

```
# sfc POINT
point1 = st_point(c(5, 2))
point2 = st_point(c(1, 3))
st_sfc(point1, point2)
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    NA
#> proj4string:    NA
#> POINT (5 2)
#> POINT (1 3)
```

In most cases, an `sfc` object contains objects of the same geometry type. Therefore, when we convert `sfg` objects of type polygon into a simple feature geometry column, we would also end up with an `sfc` object of type polygon. Equally, a geometry column of multilinestrings would result in an `sfc` object of type multilinestring:

```
# sfc POLYGON
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)
st_sfc(polygon1, polygon2)
#> Geometry set for 2 features
#> geometry type:  POLYGON
#> dimension:      XY
#> bbox:           xmin: 0 ymin: 1 xmax: 4 ymax: 5
#> epsg (SRID):    NA
#> proj4string:    NA
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
#> POLYGON ((0 2, 1 2, 1 3, 0 3, 0 2))


# 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))
st_sfc(multilinestring1, multilinestring2)
#> Geometry set for 2 features
#> geometry type:  MULTILINESTRING
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 1 xmax: 7 ymax: 9
#> epsg (SRID):    NA
#> proj4string:    NA
#> MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 ...
#> MULTILINESTRING ((2 9, 7 9, 5 6, 4 7, 2 7), (1 ...
```

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

```
# sfc GEOMETRY
st_sfc(point1, multilinestring1)
#> Geometry set for 2 features
#> geometry type:  GEOMETRY
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 1 xmax: 5 ymax: 5
#> epsg (SRID):    NA
#> proj4string:    NA
#> POINT (5 2)
#> MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 ...
```

As mentioned before, `sfc` objects can additionally store information on the coordinate reference systems (CRS). To specify a certain CRS, we can use the `epsg (SRID)` or `proj4string` attributes of an `sfc` object. The default value of `epsg (SRID)` and `proj4string` is `NA` (*Not Available*):

```
st_sfc(point1, point2)
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    NA
#> proj4string:    NA
#> POINT (5 2)
#> POINT (1 3)
```

All geometries in an `sfc` object must have the same CRS.

We can add coordinate reference system as a `crs` argument of `st_sfc()`. This argument accepts either an integer with the `epsg` code (for example, `4326`) or a `proj4string` character string (for example, `"+proj=longlat +datum=WGS84 +no_defs"`) (see section 2.3).

```
# EPSG definition
st_sfc(point1, point2, crs = 4326)
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
#> POINT (5 2)
#> POINT (1 3)
```

```
# PROJ4STRING definition
st_sfc(point1, point2, crs = "+proj=longlat +datum=WGS84 +no_defs")
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
#> POINT (5 2)
#> POINT (1 3)
```

For example, we can set the UTM Zone 11N projection with `epsg` code `2955` :

```
st_sfc(point1, point2, crs = 2955)
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    2955
#> proj4string:    +proj=utm +zone=11 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_def
#> POINT (5 2)
#> POINT (1 3)
```

As you can see above, the `proj4string` definition was automatically added. Now we can try to set the CRS using `proj4string` :

```
st_sfc(point1, point2, crs = "+proj=utm +zone=11 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +unit
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    NA
#> proj4string:    +proj=utm +zone=11 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_def
#> POINT (5 2)
#> POINT (1 3)
```

However, the `epsg` string of our result remained empty. This is because there is no general method to convert from `proj4string` to `epsg` .

2.1.5.4 Simple feature objects

So far, we have only dealt with the pure geometries. Most of the time, however, these geometries come with a set of attributes describing them. These attributes could represent the name of the geometry, measured values, groups to which the geometry belongs, and many more. For example, we measured a temperature of 25°C on Trafalgar Square in London on June 21st 2017. Hence, we have a specific point in space (the coordinates), the name of the location (Trafalgar Square), a temperature value and the date of the measurement. Other attributes might include a urbanity category (city or village), or a remark if the measurement was made using an automatic station.

The simple feature class, `sf` , is a combination of an attribute table (`data.frame`) and a simple feature geometry column (`sfc`). Simple features are created using the `st_sf()` function:

```
# sfg objects
london_point = st_point(c(0.1, 51.5))
ruan_point = st_point(c(-9, 53))

# sfc object
our_geometry = st_sfc(london_point, ruan_point, crs = 4326)

# data.frame object
our_attributes = data.frame(name = c("London", "Ruan"),
                            temperature = c(25, 13),
                            date = c(as.Date("2017-06-21"), as.Date("2017-06-22")),
                            category = c("city", "village"),
                            automatic = c(FALSE, TRUE))

# sf object
sf_points = st_sf(our_attributes, geometry = our_geometry)
```

The above example illustrates the components of `sf` objects. Firstly, coordinates define the geometry of the simple feature geometry (`sfg`). Secondly, we can combine the geometries in a simple feature geometry column (`sfc`) which also stores the CRS. Subsequently, we store the attribute information on the geometries in a `data.frame` . Finally, the `st_sf()` function combines the attribute table and the `sfc` object in an `sf` object.

```
sf_points
#> Simple feature collection with 2 features and 5 fields
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: -9 ymin: 51.5 xmax: 0.1 ymax: 53
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
#>     name temperature       date category automatic         geometry
#> 1 London          25 2017-06-21     city     FALSE POINT (0.1 51.5)
#> 2   Ruan          13 2017-06-22  village      TRUE    POINT (-9 53)
```

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

The result shows that `sf` objects actually have two classes, `sf` and `data.frame`. Simple features are simply data frames (square tables), but with spatial attributes (usually stored in a special `geom` list-column in the data frame). This duality is central to the concept of simple features: most of the time a `sf` can be treated as and behaves like a `data.frame`. Simple features are, in essence, data frames with a spatial extension.

2.2 Raster data

The geographic raster data model consists of a raster header[17] and a matrix (with rows and columns) representing equally spaced cells (often also called pixels; Figure 2.7:A). The raster header defines the coordinate reference system, the extent and the origin. The origin (or starting point) is frequently the coordinate of the lower-left corner of the matrix (the **raster** package, however, uses the upper left corner, by default (Figure 2.7:B)). The header defines the extent via the number of columns, the number of rows and the cell size resolution. Hence, starting from the origin, we can easily access and modify each single cell by either using the ID of a cell (Figure 2.7:B) or by explicitly specifying the rows and columns. This matrix representation avoids storing explicitly the coordinates for the four corner points (in fact it only stores one coordinate, namely the origin) of each cell corner as would be the case for rectangular vector polygons. This and map algebra makes raster processing much more efficient and faster than vector data processing. However, in contrast to vector data, a raster cell can only hold a single value. The value might be numeric or categorical (Figure 2.7:C).

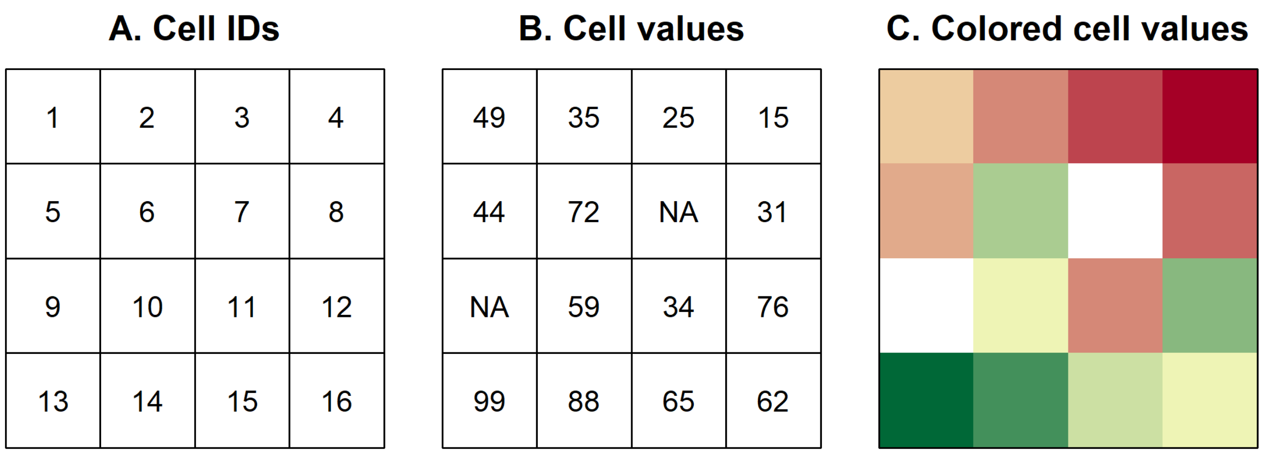


Figure 2.7: Raster data: A - cell IDs; B - cell values; C - a colored raster map.

Raster maps usually represent continuous phenomena such as elevation, temperature, population density or spectral data (Figure 2.8). Of course, we can represent discrete features such as soil or land-cover classes also with the help of a raster data model (Figure 2.8). Consequently, the discrete borders of these features become blurred, and depending on the spatial task a vector representation might be more suitable.

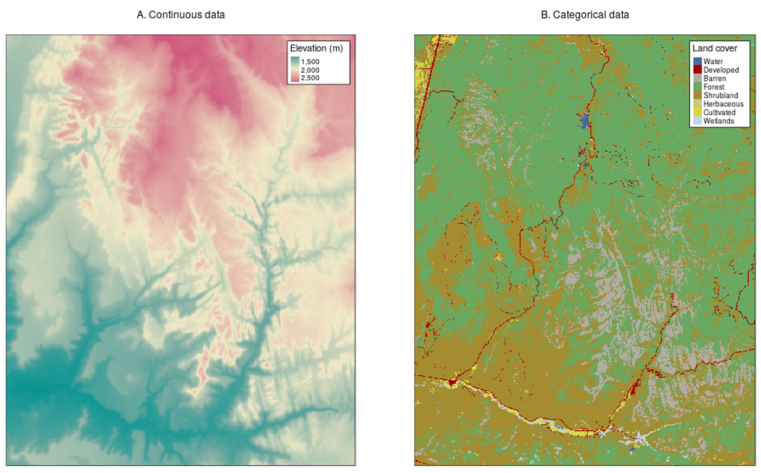


Figure 2.8: Examples of continuous (left) and categorical (right) raster.

2.2.1 An introduction to raster

The **raster** package supports raster objects in R. It provides an extensive set of functions to create, read, export, manipulate and process raster datasets. Aside from general raster data manipulation, **raster** provides many low level functions that can form the basis to develop more advanced raster functionality. **raster** also lets you work on large raster datasets that are too large to fit into the main memory. In this case, **raster** provides the possibility to divide the raster into smaller chunks (rows or blocks), and processes these iteratively instead of loading the whole raster file into RAM (for more information, please refer to `vignette("functions", package = "raster")`.

For the illustration of **raster** concepts, we will use datasets from the **spDataLarge** (note these packages were loaded at the beginning of the chapter). It consists of a few raster and one vector datasets covering an area of the Zion National Park (Utah, USA). For example, `srtm.tif` is a digital elevation model of this area (for more details - see its documentation `?srtm`) First of all, we would like to create a `RasterLayer` object named `new_raster`:

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

Typing the name of the raster into the console, will print out the raster header (extent, dimensions, resolution, CRS) and some additional information (class, data source name, summary of the raster values):

```
new_raster
#> class       : RasterLayer
#> dimensions  : 457, 465, 212505  (nrow, ncol, ncell)
#> resolution  : 0.000833, 0.000833  (x, y)
#> extent      : -113, -113, 37.1, 37.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
#> data source : /home/travis/R/Library/spDataLarge/raster/srtm.tif
#> names       : srtm
#> values      : 1024, 2892  (min, max)
```

To access individual header information, you can use following commands: `dim(new_raster)` (dimensions - number of rows, number of columns, number of cells), `res(new_raster)` (spatial resolution), `extent(new_raster)` (spatial extent), and `crs(new_raster)` (coordinate reference system).

Note that in contrast to the **sf** package, **raster** only accepts the `proj4string` representation of the coordinate reference system.

Sometimes it is important to know if all values of a raster are currently in memory or on disk. Find out with the `inMemory()` function:

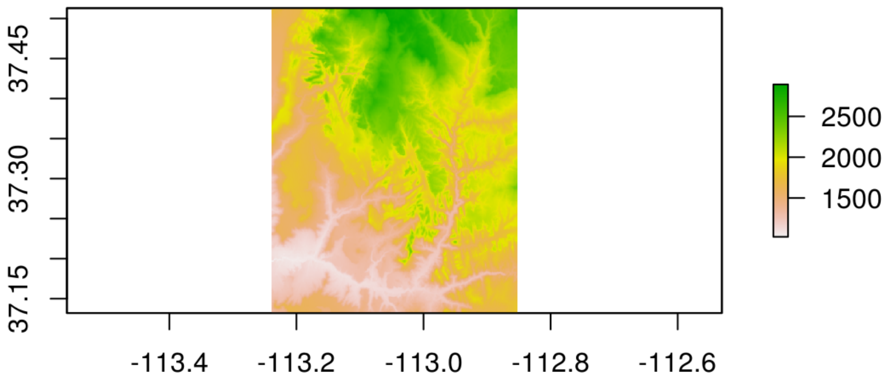
```
inMemory(new_raster)
#> [1] FALSE
```

`help(package = "raster", topic = "raster-package")` returns a full list of all available **raster** functions.

2.2.2 Basic map making

Similar to the **sf** package, **raster** also provides `plot()` methods for its own classes.

```
plot(new_raster)
```



There are several different approaches to plot raster data in R:

- You can use `spplot()` to visualize several (such as spatiotemporal) layers at once. You can also do so with the **rasterVis** package which provides more advanced methods for plotting raster objects.
- Packages such as **tmap**, **mapview** and **leaflet** facilitate especially interactive mapping of both raster and vector objects.

2.2.3 Raster classes

The `RasterLayer` class represents the simplest form of a raster object, and consists of only one layer. The easiest way to create a raster object in R is to read-in a raster file from disk or from a server.

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

The **raster** package supports numerous drivers with the help of **rgdal**. To find out which drivers are available on your system, run `raster::writeFormats()` and `rgdal::gdalDrivers()` .

Rasters can also be created from scratch using the `raster()` function. This is illustrated in the subsequent code chunk, which results in a new `RasterLayer` object. The resulting raster consists of 36 cells (6 columns and 6 rows specified by `nrow` and `ncol`) centered around the Prime Meridian and the Equator (see `xmn` , `xmx` , `ymn` and `ymx` parameters). The CRS is the default of raster objects: WGS84. This means the unit of the resolution is in degrees which we set to 0.5 (`res`). Values (`vals`) are assigned to each cell: 1 to cell 1, 2 to cell 2, and so on. Remember: `raster()` fills cells row-wise (unlike `matrix()`) starting at the upper left corner, meaning the top row contains the values 1 to 6, the second 7 to 12 etc.

```
new_raster2 = raster(nrow = 6, ncol = 6, res = 0.5,
                     xmn = -1.5, xmx = 1.5, ymn = -1.5, ymx = 1.5,
                     vals = 1:36)
```

For still further ways of creating a raster object have a look at the help file - `?raster` .

Aside from `RasterLayer` , there are two additional classes: `RasterBrick` and `RasterStack` . Both can handle multiple layers, but differ regarding the number of supported file formats, type of internal representation and processing speed.

A `RasterBrick` consists of multiple layers, which typically correspond to a single multispectral satellite file or a single multilayer object in memory. The `brick()` function creates a `RasterBrick` object. Usually, you provide it with a filename to a multilayer raster file but might also use another raster object and other spatial objects (see its help page for all supported formats).

```
multilayer_raster_filepath = system.file("raster/landsat.tif", package="spDataLarge")
r_brick = brick(multilayer_raster_filepath)
r_brick
#> class       : RasterBrick
#> dimensions  : 1428, 1128, 1610784, 4  (nrow, ncol, ncell, nlayers)
#> resolution  : 30, 30  (x, y)
#> extent      : 301905, 335745, 4111245, 4154085  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=utm +zone=12 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,
#> data source : /home/travis/R/Library/spDataLarge/raster/landsat.tif
#> names       : landsat.1, landsat.2, landsat.3, landsat.4
#> min values  :      7550,      6404,      5678,      5252
#> max values  :     19071,     22051,     25780,     31961
```

The `nlayers` function retrieves the number of layers stored in a `Raster*` object:

```
nlayers(r_brick)
#> [1] 4
```

A `RasterStack` is similar to a `RasterBrick` in the sense that it consists also of multiple layers. However, in contrast to `RasterBrick` , `RasterStack` allows you to connect several raster objects stored in different files or multiply objects in memory. More specifically, a `RasterStack` is a list of `RasterLayer` objects with the same extent and resolution. Hence, one way to create it is with the help of spatial objects already existing in R's global environment. And again, one can simply specify a path to a file stored on disk.

```
raster_on_disk = raster(r_brick, layer = 1)
raster_in_memory = raster(xmn = 301905, xmx = 335745, ymn = 4111245, ymx = 4154085, res =
values(raster_in_memory) = sample(1:ncell(raster_in_memory))
crs(raster_in_memory) = crs(raster_on_disk)
```

```
r_stack = stack(raster_in_memory, raster_on_disk)
r_stack
#> class       : RasterStack
#> dimensions  : 1428, 1128, 1610784, 2  (nrow, ncol, ncell, nlayers)
#> resolution  : 30, 30  (x, y)
#> extent      : 301905, 335745, 4111245, 4154085  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=utm +zone=12 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,
#> names       :   layer, landsat.1
#> min values  :       1,      7550
#> max values  : 1610784,     19071
```

Another difference is that the processing time for `RasterBrick` objects is usually shorter than for `RasterStack` objects.

Decision on which `Raster*` class should be used depends mostly on a character of input data. Processing of a single mulitilayer file or object is the most effective with `RasterBrick` , while `RasterStack` allows calculations based on many files, many `Raster*` objects, or both.

Operations on `RasterBrick` and `RasterStack` objects will typically return a `RasterBrick` .

2.3 Coordinate Reference Systems

Vector and raster spatial data types share concepts intrinsic to spatial data. Perhaps the most fundamental of these is the Coordinate Reference System (CRS), which defines how the spatial elements of the data relate to the surface of the Earth (or other bodies). CRSs are either geographic or projected, as introduced at the beginning of this chapter (see Figure 2.1). This section will will explain each type, laying the foundations for section 5.2 on CRS transformations.

2.3.1 Geographic coordinate systems

Geographic coordinate systems identify any location on the Earth's surface using two values — longitude and latitude. *Longitude* is location in the East-West direction in angular distance from the Prime Meridian plane. *Latitude* is angular distance North or South of the equatorial plane. Distance in geographic CRSs are therefore not measured in meters. This has important consequences, as demonstrated in section 5.2.

The surface of the Earth in geographic coordinate systems is represented by a spherical or ellipsoidal surface. Spherical models assume that the Earth is a perfect sphere of a given radius. Spherical models have the advantage of simplicity but are rarely used because they are inaccurate: the Earth is not a sphere! Ellipsoidal models are defined by two parameters: the equatorial radius and the polar radius. These are suitable because the Earth is compressed: the equatorial radius is around 11.5 km longer than the polar radius (Maling 1992).[18]

Ellipsoids are part of a wider component of CRSs: the *datum*. This contains information on what ellipsoid to use (with the `ellps` parameter in the proj4 CRS library) and the precise relationship between the Cartesian coordinates and location on the Earth's surface. These additional details are stored in the `towgs84` argument of proj4 notation (see proj4.org/parameters.html for details). These allow local variations in Earth's surface, e.g. due to large mountain ranges, to be accounted for in a local CRS. There are two types of datum — local and geocentric. In a *local datum* such as `NAD83` the ellipsoidal surface is shifted to align with the surface at a particular location. In a *geocentric datum* such as `WGS84` the center is the Earth's center of gravity and the accuracy of projections is not optimized for a specific location. Available datum definitions can be seen by executing `st_proj_info(type = "datum")`.

2.3.2 Projected coordinate systems

Projected CRSs are based on Cartesian coordinates on an implicitly flat surface. They have an origin, x and y axes, and a linear unit of measurement such as meters. All projected CRSs are based on a geographic CRS, described in the previous section, and rely on map projections to convert the three-dimensional surface of the Earth into Easting and Northing (x and y) values in a projected CRS.

This transition cannot be done without adding some distortion. Therefore, some properties of the Earth's surface are distorted in this process, such as area, direction, distance, and shape. A projected coordinate system can preserve only one or two of those properties. Projections are often named based on a property they preserve: equal-area preserves area, azimuthal preserve direction, equidistant preserve distance, and conformal preserve local shape.

There are three main groups of projection types - conic, cylindrical, and planar. In a conic projection, the Earth's surface is projected onto a cone along a single line of tangency or two lines of tangency. Distortions are minimized along the tangency lines and rise with the distance from those lines in this projection. Therefore, it is the best suited for maps of mid-latitude areas. A cylindrical projection maps the surface onto a cylinder. This projection could also be created by touching the Earth's surface along a single line of tangency or two lines of tangency. Cylindrical projections are used most often when mapping the entire world. A planar projection projects data onto a flat surface touching the globe at a point or along a line of tangency. It is typically used in mapping polar regions. `st_proj_info(type = "proj")` gives a list of the available projections supported by the PROJ.4 library.

2.3.3 CRSs in R

Two main ways to describe CRS in R are an `epsg` code or a `proj4string` definition. Both of these approaches have advantages and disadvantages. An `epsg` code is usually shorter, and therefore easier to remember. The code also refers to only one, well-defined coordinate reference system. On the other hand, a `proj4string` definition allows you more flexibility when it comes to specifying different parameters such as the projection type, the datum and the ellipsoid.[19] This way you can specify many different projections, and modify existing ones. This also makes the `proj4string` approach more complicated. `epsg` points to exactly one particular CRS.

Spatial R packages support a wide range of CRSs and they use the long-established proj4 library. Other than searching for EPSG codes online, another quick way to find out about available CRSs is via the `rgdal::make_EPSG()` function, which outputs a data frame of available projections. Before going into more detail, it’s worth learning how to view and filter them inside R, as this could save time trawling the internet. The following code will show available CRSs interactively, allowing you to filter ones of interest (try filtering for the OSGB CRSs for example):

```
crs_data = rgdal::make_EPSG()
View(crs_data)
```

In **sf** the CRS of an object can be retrieved using `st_crs()` . For this, we need to read-in a vector dataset:

```
vector_filepath = system.file("vector/zion.gpkg", package="spDataLarge")
new_vector = st_read(vector_filepath)
```

Our new object, `new_vector` , is a polygon representing the borders of Zion National Park (`?zion`).

```
st_crs(new_vector) # get CRS
#> Coordinate Reference System:
#>   No EPSG code
#>   proj4string: "+proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_def
```

In cases when a coordinate reference system (CRS) is missing or the wrong CRS is set, the `st_set_crs()` function can be used:

```
new_vector = st_set_crs(new_vector, 26912) # set CRS
```

The warning message informs us that the `st_set_crs()` function does not transform data from one CRS to another.

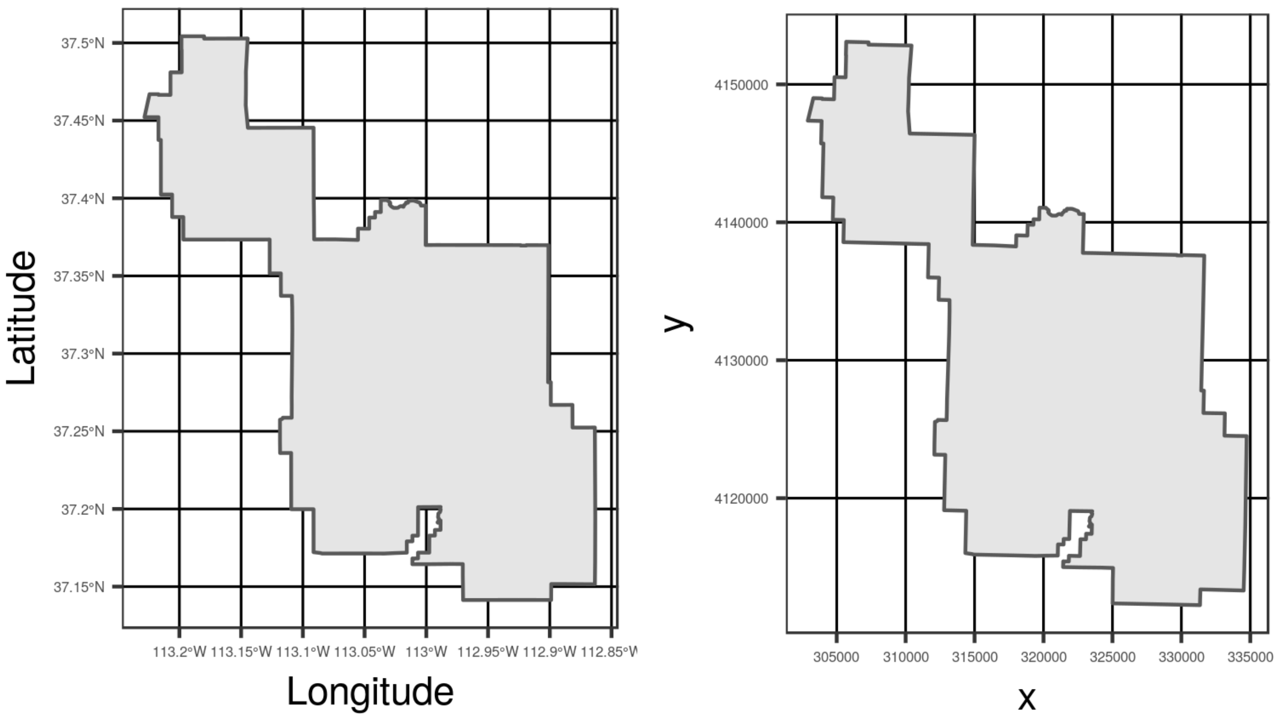


Figure 2.9: Examples of geographic (WGS 84; left) and projected (NAD83 / UTM zone 12N; right) and coordinate systems for a vector data type.

The `projection()` function can be used to access CRS information from a `Raster*` object:

```
projection(new_raster) # get CRS
#> [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

The same function, `projection()`, is used to set a CRS for raster objects. The main difference, compared to vector data, is that raster objects only accept `proj4` definitions:

```
projection(new_raster) = "+proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m
```

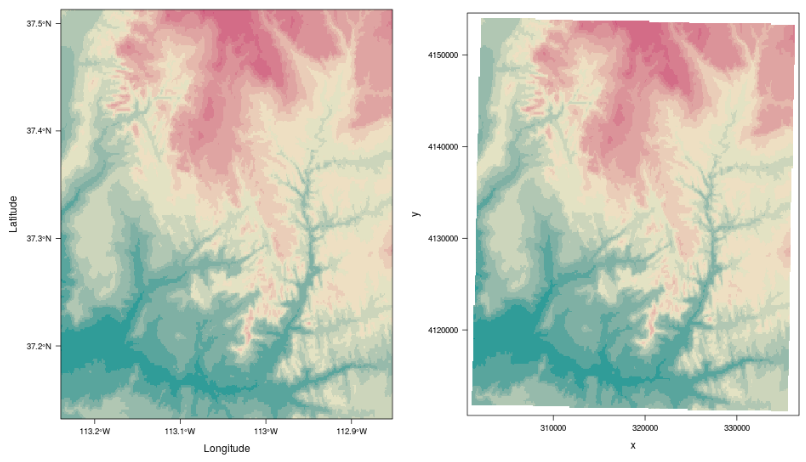


Figure 2.10: Examples of geographic (WGS 84; left) and projected (NAD83 / UTM zone 12N; right) and coordinate systems for a raster data type

We will expand on CRSs and how to project from one CRS to another in much more detail in chapter 5.

2.4 Units

An important feature of CRSs is that they contain information about spatial units. Clearly it is vital to know whether a house's measurements are in feet or meters, and the same applies to maps. It is good cartographic practice to add a *scale bar* onto maps to demonstrate the relationship between distances on the page or screen and distances on the ground. Likewise, it is important to formally specify the units in which the geometry data or pixels are measured to provide context, and ensure that subsequent calculations are done in context.

A novel feature of geometry data in `sf` objects is that they have *native support* for units. This means that distance, area and other geometric calculations in **sf** return values that come with a `units` attribute, defined by the **units** package (Pebesma, Mailund, and Hiebert 2016). This is advantageous because it prevents confusion caused by the fact that different CRSs use different units (most use meters, some use feet). Furthermore, it also provides information on dimensionality, as illustrated by the following calculation which reports the area of Nigeria:

```
nigeria = world[world$name_long == "Nigeria", ]
```

```
st_area(nigeria)
#> 9.05e+11 m^2
```

The result is in units of square meters (m^2), showing a) that the result represents two-dimensional space and b) and that Nigeria is a large country! This information, stored as an attribute (which interested readers can discover with `attributes(st_area(nigeria))`) is advantageous for many reasons, for example it could feed into subsequent calculations such as population density. Reporting units prevents confusion. To take the Nigeria example, if the units remained unspecified, one could incorrectly assume that the units were in km^2. To translate the huge number into a more digestible size, it is tempting to divide the results by a million (the number of square meters in a square kilometer):

```
st_area(nigeria) / 1e6
#> 905062 m^2
```

However, the result is incorrectly given again as square meters. The solution is to set the correct units with the **units** package:

```
units::set_units(st_area(nigeria), km^2)
#> 905062 km^2
```

Units are of equal importance in the case of raster data. However, so far **sf** is the only spatial package that supports units, meaning that people working on raster data should approach changes in the units of analysis (for example, converting pixel widths from imperial to decimal units) with care. The `new_raster` object (see above) uses a UTM projection with meters as units. Consequently, its resolution is also given in meters but you have to know it, since the `res()` function simply returns a numeric vector.

```r
res(new_raster)
#> [1] 0.000833 0.000833
```

If we used the WGS84 projection, the units would change.

```r
repr = projectRaster(new_raster, crs = "+init=epsg:4326")
res(repr)
#> [1] 7.47e-09 7.52e-09
```

Again, the `res()` command gives back a numeric vector without any unit, forcing us to know that the unit of the WGS84 projection is decimal degrees.

2.5 Exercises

1. What does the summary of the `geometry` column tell us about the `world` dataset, in terms of:
    - The geometry type?
    - How many countries there are?
    - The coordinate reference system (CRS)?
2. Using **sf**'s `plot()` command, create a map of Nigeria in context, building on the code that creates and plots Asia above (see Figure 2.5 for an example of what this could look like).
    - Hint: this used the `lwd` , `main` and `col` arguments of `plot()` .
    - Bonus: make the country boundaries a dotted grey line.
    - Hint: `border` is an additional argument of `plot()` for **sf** objects.
3. What does the `cex` argument do in the `plot()` function that generates Figure 2.6?
    - Why was `cex` set to the `sqrt(world$pop) / 10000` instead of just the population directly?
    - Bonus: what equivalent arguments to `cex` exist in the dedicated visualization package **tmap**?
4. Re-run the code that 'generated' Figure 2.6 at the end of 2.1.4 and find 3 similarities and 3 differences between the plot produced on your computer and that in the book.
    - What is similar?
    - What has changed?
    - Bonus: play around with and research base plotting arguments to make your version of Figure 2.6 more attractive. Which arguments were most useful?
    - Advanced: try to reproduce the map presented in Figure 2.1.4. Copy-and-pasting is prohibited!
5. Read the `raster/nlcd2011.tif` file from the **spDataLarge** package. What kind of information can you get about the properties of this file?
6. Create an empty `RasterLayer` object called `my_raster` with 10 columns and 10 rows and resolution of 10 units. Assign random values between 0 and 10 to the new raster and plot it.

References

Pebesma, Edzer. 2018. *Sf: Simple Features for R*. https://CRAN.R-project.org/package=sf.

Pebesma, Edzer, and Roger Bivand. 2018. *Sp: Classes and Methods for Spatial Data*. https://CRAN.R-project.org/package=sp.

Bivand, Roger, Tim Keitt, and Barry Rowlingson. 2018. *Rgdal: Bindings for the 'Geospatial' Data Abstraction Library*. https://CRAN.R-project.org/package=rgdal.

Bivand, Roger, and Colin Rundel. 2017. *Rgeos: Interface to Geometry Engine - Open Source ('GEOS')*. https://CRAN.R-project.org/package=rgeos.

Maling, D. H. 1992. *Coordinate Systems and Map Projections*. 2nd ed. Oxford ; New York: Pergamon Press.

Pebesma, Edzer, Thomas Mailund, and James Hiebert. 2016. "Measurement Units in R." *The R Journal* 8 (2): 486–94. https://journal.r-project.org/archive/2016-2/pebesma-mailund-hiebert.pdf.

12. If you need to install R, we recommend reading section 2.3 and section 2.5 of the freely available book *Efficient R Programming* Gillespie and Lovelace (2016). Packages can be kept up-to-date with `update.packages()` .

    If you are not a regular R user, we recommend that you familiarize yourself with the language before proceeding with this chapter. You can do so using resources such as Gillespie and Lovelace (2016), Grolemund and Wickham (2016) as well as online interactive guides such as DataCamp.

    We recommend organising your work as you learn, for example with the help of an RStudio 'project' called `geocomp-learning` . Creating new script for each chapter or section of interest will help consolidate and extend the skills learned. The code you type to help learn the content of this chapter could be placed in a file called `chapter-02.R` , for example. Everyone learns in a different way; structure your work so it makes sense to you; and avoid copy-pasting to get used to typing code.↩

13. The origin we are referring to, depicted in blue in Figure 2.1, is in fact the 'false' origin. The 'true' origin, the location at which distortions are at a minimum, is located at 2° W and 49° N. This was selected by the Ordnance Survey to be roughly in the center of the British landmass longitudinally.↩

14. The full OGC standard includes rather exotic geometry types including 'surface' and 'curve' geometry types, which currently have limited application in real world applications. All 68 types can be represented with the **sf** package, although (at the time of writing) all methods, such as plotting, are only supported for the 7 types described in this chapter.↩

15. The development version, which may contain new features, can be installed with `devtools::install_github("r-spatial/sf")`. ↩

16. In fact, when you `plot()` an **sf** object, R is calling `sf:::plot.sf()` behind the scenes. `plot()` is a generic method that behaves differently depending on the class of object being plotted.↩

17. Depending on the file format the header is part of the actual image data file, e.g., GeoTiff, or stored in an extra header or world file, e.g., ASCII grid formats↩

18. The degree of compression is often referred to as *flattening*, defined in terms of the equitorial radius (a) and polar radius (b) as follows: $f = (a - b)/a$. The terms *ellipticity* and *compression* can also be used (Maling 1992). Because f is a rather small value, digital ellipsoid models use the 'inverse

flattening' ($rf = 1/f$) to define the Earth's compression. Values of a and rf used in a variety of ellipsoidal models can be seen be executing `st_proj_info(type = "ellps")` .↩

19. Complete list of the `proj4string` parameters can be found at http://proj4.org/parameters.html#parameter-list.↩