Transport and Geographic Data Science with R

Practical Exercises

Robin Lovelace and Malcolm Morgan, Institute for Transport Studies, University of Leeds

These practicals aim to test your knowledge of material covered in a short course delivered for the Department for Transport. Code and data supporting the content can be found in the GitHub repository (repo) at github.com/ITSLeeds/TDS. To see the course contents and timings see https://git.io/tds2day which links to the file `2day.md` in the repo.

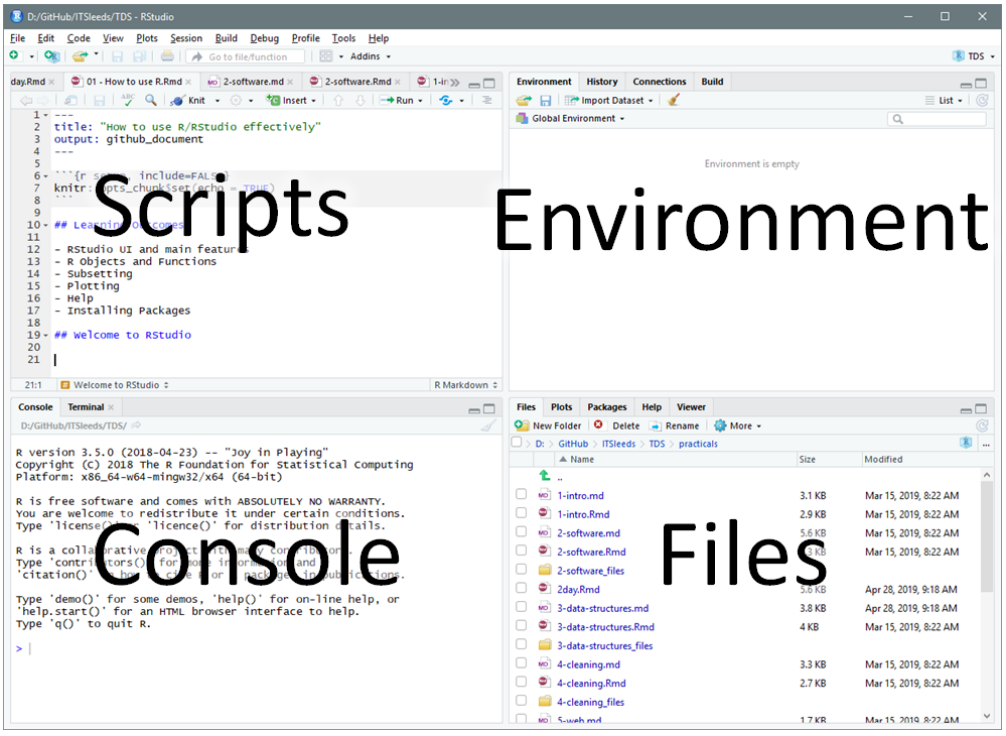
It will use the following CRAN packages:

```
library(osmdata)   # a package for accessing OpenStreetMap data
library(pct)       # download and process data from DfT-funded Propensity to Cycle Tool project
library(sf)        # spatial vector data classes
library(stats19)   # get stats19 data
library(stplanr)   # transport planning tools
library(spData)    # example spatial datasets
library(tidyverse)# packages for 'data science'
library(tmap)      # interactive maps
```

1 How to use R/RStudio effectively

The learning outcomes of this first session are to learn: RStudio UI and main features, R Objects and Functions, Subsetting, Plotting, and Help.

The course assumes that you have already got some basic knowledge of working with R and the editor (IDE) RStudio. If you don't see the prerequisites at git.io/tds2day. The main components of RStudio are shown in the figure below.



1.0.1 Projects

Projects are a way to organise related work together. Each project has its own folder and Rproj file.

Start a new project with:

> File > New Project

You can choose to create a new directory (folder) or associate a project with an existing directory. Make a new project called TDS and save it in a sensible place on your computer. Notice that TDS now appears in the top right of RStudio.

Always do your work within a project

1.0.2 R Scripts

We could simply type all our code into the console, but that would require us to retype all our code every time we wish to run it. So we usually save code in a script file (with the .R extension).

Make a new script:

> Flie > New File > Rscript

Or use the new script button on the toolbar.

Save the script and give it a sensible name like `TDS-lesson-1.R` with:

> File > Save

Or the save button on the toolbar.

1.1 Code

1.1.1 Writing Code

Let's start with some basic R operations

```
x = 1:5
y = c(0,1,3,9,18)
plot(x, y)
```

This code creates two objects, both vectors of length == 5, and then plots them.

1.1.2 Running Code

We have several ways to run code within a script.

1. Place the cursor on a line of code and press `CTRL + Enter` to run that line of code.
2. Highlight a block of code or part of a line of code and press `CTRL + Enter` to run the highlighted code.
3. Press `CTRL + Shift + Enter` to run all the code in a script.
4. Press the Run button on the toolbar to run all the code in a script.
5. Use the function `source()` to run all the code in a script e.g. `source("TDS-lesson-1.R")`

1.1.3 Vewing Objects

Lets create some different types of object:

```
cat = data.frame(name = c("Tiddles", "Chester", "Shadow"),
                 type = c("Tabby", "Persian", "Siamese"),
                  age = c(1, 3, 5),
                 likes_milk = c(TRUE, FALSE,TRUE))
even_numbers = seq(from = 2, to = 4000, by = 2)
```

```r
random_letters = sample(letters, size = 100, replace = TRUE)
small_matrix = matrix(1:24, nrow = 12)
```

We can view the objects in a range of ways:

1. Type the name of the object into the console e.g. `cat`, what happens if we try to view all 2000 even_numbers?
2. Use the `head()` function to view the first few values e.g. `head(even_numbers)`
3. Use the view table button next to matrix or data.frame objects in the environment tab.

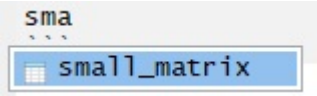
We can also get an overview of an object using a range of functions.

1. `summary()`
2. `class()`
3. `class()`
4. `dim()`
5. `length()`

Exercise try these functions, what results do they give?

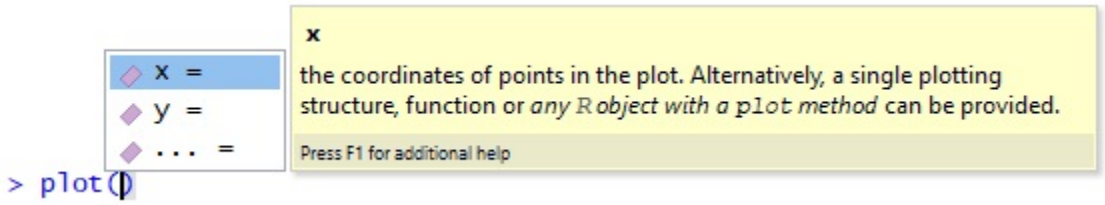
1.1.4 Using Autocomplete

RStudio can help you write code by autocompleting. RStudio will look for similar objects and functions after typing the first three letters of a name.



When there is more than one option you can select from the list using the mouse or arrow keys.

Within a function, you can get a list of arguments by pressing Tab.



Notice the help popup.

1.1.5 Getting help

Every function in R has a help page. You can view the help using `?` for example `?sum`. Many packages also contain vignettes, these are long form help documents containing examples and guides. `vignette()` will show a list of all the vignettes available, or you can show a specific vignette for example `vignette(topic = "sf1", package = "sf")`.

1.1.6 Commenting Code

It is good practice to use comments in your code to explain what your code does. You can comment code using `#`

For example:

```r
# A whole line comment
x = 1:5 # An inline comment
y = x * 2
```

You can comment a whole block of text by selecting it and using CTRL + Shift + C

You can add a comment section using CTRL + Shift + R

1.1.7 Cleaning your environment and removing objects

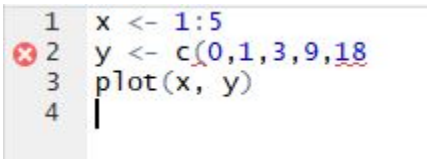
The Environment tab shows all the objects in your environment, this includes Data, Values, and Functions. By default, new objects appear in the Global Environment but you can see other environments with the drop-down menu. For example, each package has its own environment.

Sometimes you wish to remove things from your environment, perhaps because you no longer need them or things are getting cluttered.

You can remove an object with the `rm()` function e.g. `rm(x)` or `rm(x,y)` or you can clear your whole environment with the broom button on the Environment Tab.

1.1.8 Debugging Code

This code example will run, but we can see some of RStudio's debugging features by changing it. See that when the bracket is removed the red X and the underlying highlight the broken code. You may need to save the code you see the debugging prompt.



Always address debugging prompts before running your code

1.1.9 Saving your work

We have already seen that you can save an R script. You can also save R objects in the RDS format.

```
saveRDS(cat,"cat.Rds")
```

We can also read back in our data.

```
cat2 = readRDS("cat.Rds")
identical(cat, cat2)
```

R also supports many other formats. For example CSV files.

```
write.csv(cat, "cat.csv")
cat3 = read.csv("cat.csv")
identical(cat3, cat1)
```

Notice that `cat3` and `cat` are not identical, what has changed? Hint: use `?write.csv`.

1.2 Classes

1.2.1 Subsetting

We can subset any R object to just get part of the object. Subsetting can be done by either providing the positional numbers of the subset or logical vector of the same length. For two dimension object such as matrices and data.frames you can subset by row or column. Subsetting is done using square brackets `[]` after the name of an object.

```
even_numbers[1:5] # Just the first five even_numbers
x[c(TRUE, FALSE,TRUE,FALSE,TRUE)] # The 1st, 3rd, and 5th element in x
cat[c(1,2),] # First and second row of cat
```

```r
cat[,c(1,3)] # First and third column of cat
cat[,c("name","age")] # First and third column of cat by name
```

It is also possible to create logical vector for subsetting by creating a query

```r
x[x == 5] # Only when x == 5 (notice the use of double equals)
even_numbers[even_numbers < 50] # Just the  even_numbers less than 50
even_numbers[even_numbers %% 9 == 0] # Just the  even_numbers that are a multiple of 9
cat[cat$name == "Tiddles",] # THe rows where the name is Tiddles (notice the use of $)
```

1.2.2 Dealing with NAs

R object can have a value of NA. This is how R represents missing data.

```r
z = c(4,5,NA,7)
```

NA values are common in real-world data but can cause trouble, for example

```r
sum(z) # Result is NA
```

Some functions can be told to ignore NA values.

```r
sum(z, na.rm = TRUE) # Result is equal to 4 + 5 + 7
```

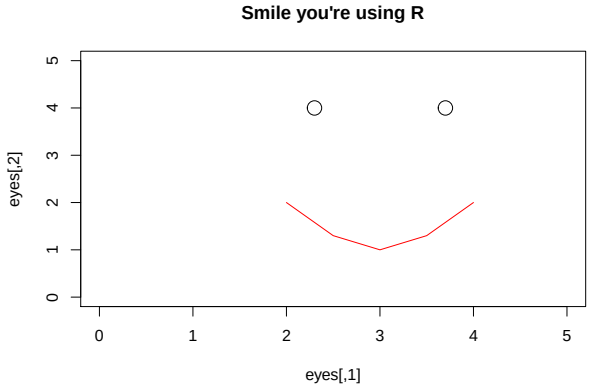
You can find NAs using the `is.na()` function, and then remove them

```r
is.na(z)
z_nona = z[!is.na(z)] #Note the use of the not operator !
sum(z)
```

Be careful of NAs especially in statistical analysis, for example, the average of a value excluding NAs may not be representative of the whole.

1.3 Now you are ready to use R

```r
eyes = c(2.3,4,3.7,4)
eyes = matrix(eyes, ncol = 2, byrow = T)
mouth = c(2,2,2.5,1.3,3,1,3.5,1.3,4,2)
mouth = matrix(mouth, ncol = 2, byrow = T)
plot(eyes, type = "p", main = "Smile you're using R",
     cex = 2, xlim = c(0,5), ylim = c(0,5))
lines(mouth, type = "l", col = "red")
```



2 Packages: ggplot2, the tidyverse and sf

2.1 What are packages?

R has lots of functionality built in, but the real value in R is the community of package developers. Packages add new functions to R. Some packages are so useful they have become almost essential while others are only used for specific purposes.

There are two stages to using a package: installing it and loading it.

Packages that you don't have on your computer can be installed using `install.packages()` Packages come from The Comprehensive R Archive Network there are over 10,000 packages on CRAN. You only need to install a package once.

Note: it is bad practice to install packages with `install.packages()` within a script*

Packages only need to be installed once.

You can use `remotes::install_cran()` or `remotes::install_github()` to only install a package if it is not yet installed and up-to-date

```
install.packages("sf")
remotes::install_cran("sf")
remotes::install_github("r-spatial/sf")
```

Once you have a package on your computer you need to load or 'attach' it to your current environment. Usually, the package will load silently. In some cases the package will provide a message, as illustrated below.

```
library(sf)
```

```
## Linking to GEOS 3.5.1, GDAL 2.1.2, PROJ 4.9.3
```

2.2 sf objects

The `sf` package provides a generic class for spatial vector data: points, lines and polygons. `sf` objects are simply data frames. However, they have a special 'geometry column', typically called 'geom' or 'geometry'. This is illustrated below with reference to `iow`, an `sf` object representing the Isle of Wight, that we will download using the `pct` package (note: the `[1:9]` appended to the function selects only the first 9 columns).

```
iow = pct::get_pct_zones("isle-of-wight")[1:9]
```

```
## Loading required package: sp
```

```
class(iow)
```

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

```
names(iow)
```

```
##  [1] "geo_code"      "geo_name"      "lad11cd"       "lad_name"
##  [5] "all"           "bicycle"       "foot"          "car_driver"
##  [9] "car_passenger" "geometry"
```

```
iow[1:2, c(1, 5, 6, 7, 8)]
```

```
## Simple feature collection with 2 features and 5 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -1.345963 ymin: 50.72766 xmax: -1.282956 ymax: 50.76684
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
##       geo_code  all bicycle foot car_driver                       geometry
```

```
## 3491 E02003581 3506     202  838       1755 MULTIPOLYGON (((-1.300856 5...
## 3492 E02003582 3325     234  458       2110 MULTIPOLYGON (((-1.305353 5...
```

2.3 Attribute operations on sf objects

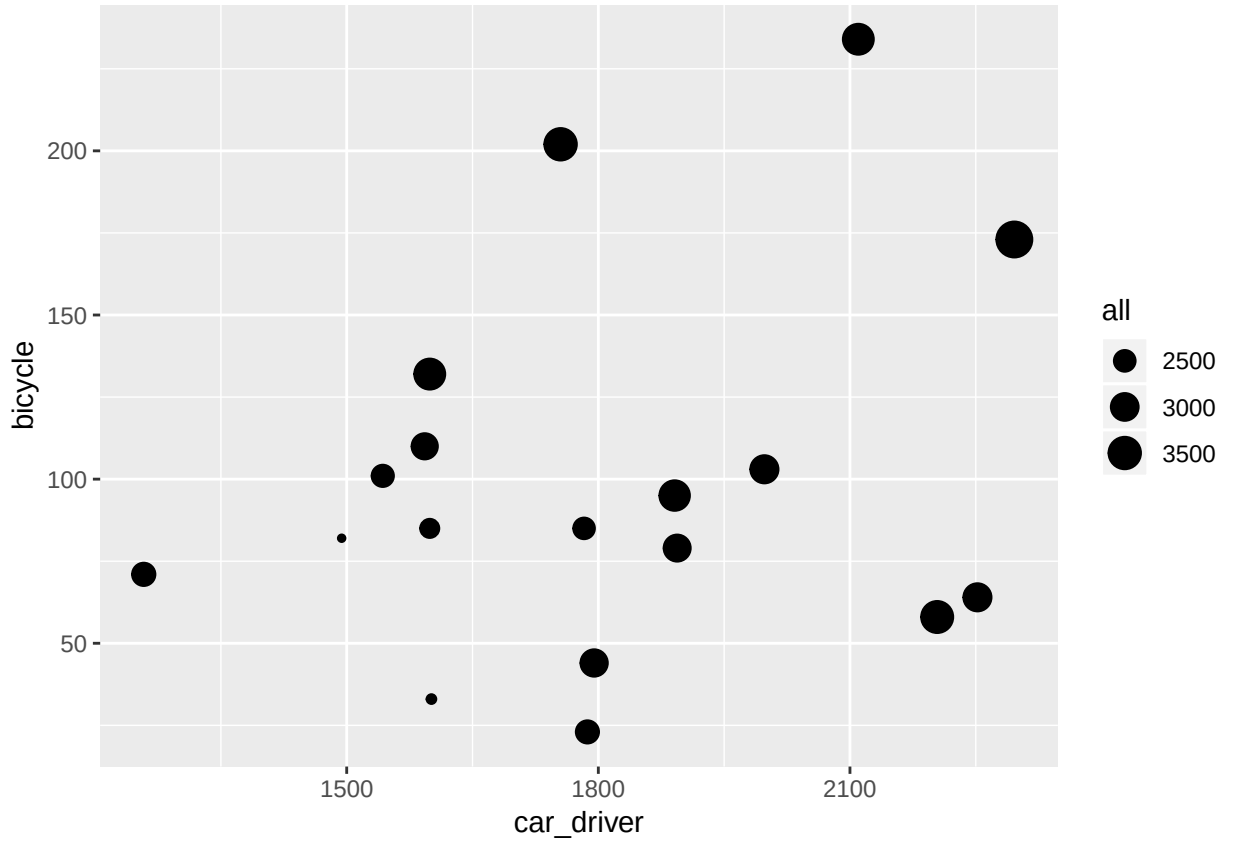
One of the nice things about the `sf` package is that an `sf data.frame` behaves just like a normal `data.frame` for non-spatial operations. Load the example dataset for Isle of Wight.

```
# load example dataset if it doesn't already exist
if(!exists("iow")) {
  iow = pct::get_pct_zones("isle-of-wight")
}
sel = iow$all > 3000  # create a subsetting object
iow_large = iow[sel, ]          # subset areas with a popualtion over 100,000
iow_2 = iow[iow$geo_name == "Isle of Wight 002",] # subset based on 'equality' query
five_in_name = iow[grepl(pattern = "5", x = iow$geo_name), ] # subset based on text string match
iow_first_and_third_column = iow[c(1, 3)]
iow_just_all = iow["all"]
```

2.4 ggplot2, dplyr and pipes

Another useful package is `ggplot2`, a generic plotting package that is part of the 'tidyverse' meta-package, which provides packages for data science that have intuitive function names and work well together. It is flexible, popular, and has dozens of add-on packages which build on it, such as `gganimate`. To plot non-spatial data, it works as follows:

```
library(ggplot2)
ggplot(iow) + geom_point(aes(x = car_driver, y = bicycle, size = all))
```



Note that the + operator adds layers onto one another.

Another useful operator, that is part of `dplyr` is `%>%`. This is the pipe, that puts the output of one command into the first argument of another, as shown below (note the results are the same):

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
class(iow)
```

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

```
iow %>% class()
```

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

Useful `dplyr` functions are demonstrated below.

```
iow %>%
  filter(bicycle > 200) # filter rows
iow %>%
```

```
  select(all) # select just the 'all' column
iow %>%
  group_by(bicycle > 200) %>%
  summarise(mean_walk = mean(foot))
```

2.5 Exercises

1. Check your packages are up-to-date with `update.packages()`
2. Create an RStudio project with an appropriate name for this course (e.g. `tds`)
3. Create a script called `set-up.R`, e.g. with the following command: `file.edit("set-up.R")`
4. Practice subsetting techniques you have learned on the `sf data.frame` object `iow`:
    1. Create an object called `iow_small` which contains only regions with less than 3000 people in the `all` column
    2. Create a selection object called `sel_high_car` which is `TRUE` for regions with above median numbers of people who travel by car and `FALSE` otherwise
    3. How many regions have the number '1' in them? What percentage of the regions in the Isle of Wight is this?
    4. Create an object called `iow_foot` which contains only the foot attribute from `iow`
    5. Bonus: plot the result to show where the high-income regions are
    6. Bonus: use `filter()` from the `dplyr` package to subset small and high car use regions
5. Bonus: What is the population density of each region (use the 'all' column)?
6. Bonus: Which zone has the highest percentage who cycle?
7. Bonus: Find the proportion of people who drive to work (`car_driver`) in areas in which more than 500 people walk to work

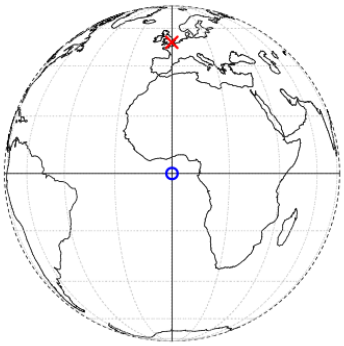
3 Spatial data analysis

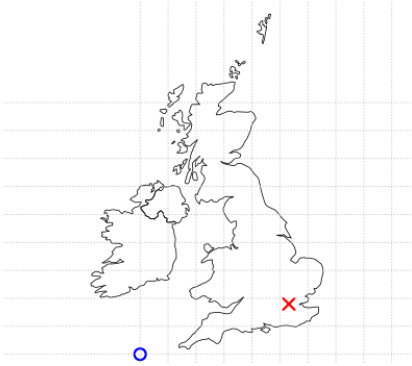
What makes an `sf data.frame` different from a normal `data.frame` is the inclusion of a geometry column and spatial operations.

3.1 Projections and Coordinate Reference Systems

When plotting a map you need X and Y coordinates to specify where objects should appear. While this is simple on a flat surface spatial data must fit onto the curved surface of the earth. You may know that it is impossible to unwrap a sphere into a single flat surface without distorting (stretching, twisting, cutting) the surface in some way. The process of making a flat map from a curved Earth is known as projection, and there are many valid ways to project a map.

Coordinate Reference Systems (CRS) refer to different ways of defining the X and Y coordinates used in different projections. Largely they fall into two categories, **geographic** and **projected**. These are illustrated in the figure below, from Chapter 2 of Geocomputation with R, 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.





- Geographical Coordinate Systems: use latitude and longitude to represent any place on the Earth
- Projected Coordinate Systems: use distances from an origin point to represent a small part of the Earth, e.g. a country. The advantage of a projects CRS is that it is easier to calculate properties such as distance and area as coordinates are in metres.

You can find a catalogue of different CRSs at http://spatialreference.org/

CRSs are often referred to by the EPSG number. The European Petroleum Survey Group publish a database of different coordinate systems. Two useful projections to commit to memory are:

- 4326 - the World Geodetic System 1984 which is a widely used geographical coordinate system, used in GPS datasets and the .geojson file format, for example.
- 27700 - the British National Grid

Every `sf data.frame` has a CRS.

```
st_crs(iow) # 2193 the CRS for Isle of Wight Transverse Mercator
iow_latlng = st_transform(iow, 4326) # Transfrom from one CRS to another
st_crs(iow) # 4326 the CRS for World Geodetic System 1984
iow_latlng = st_transform(iow, 2193) # Transfrom back
```

Warning It is possible to change the CRS without reprojecting the data by:

```
st_crs(iow) = 4326
```

This is risky as you may confuse your data by having the wrong CRS.

For more information see Chapter 6 of Geocompuation with R.

3.2 Spatial operations

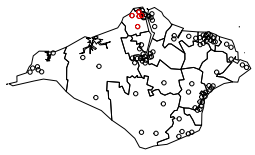
It is possible to subset an `sf data.frame` by location as well as attributes.

Let's load some centroids and find out which are in differe `iow` areas.

```
iow_cents = pct::get_pct_centroids(region = "isle-of-wight", geography = "lsoa") # Load the iow_cents d
iow_cents2 = iow_cents[iow_2,]
```

```
## although coordinates are longitude/latitude, st_intersects assumes that they are planar
## although coordinates are longitude/latitude, st_intersects assumes that they are planar
```

```
plot(iow$geometry)
plot(iow_cents, col = "black", add = TRUE)
plot(iow_cents2, col = "red", add = TRUE)
```



`st_intersects()` is a 'binary predicate' that identifies which features in one `sf` object intersect with another `sf` object in geographic space:

```
st_intersects(iow_cents, iow_large)
```

There are many spatial predicates, including `st_overlaps()` and `st_difference()`. You can see a list of them in the help.

```
?st_intersects
```

You could use a different function by adding the `op` argument

```
iow_cents3 = iow_cents[iow_2, , op = st_disjoint]
```

```
## although coordinates are longitude/latitude, st_intersects assumes that they are planar
## although coordinates are longitude/latitude, st_intersects assumes that they are planar
```

3.3 Aggregation

With a normal data.frame it is possible to group and aggregate variables using the `dplyr` packages.

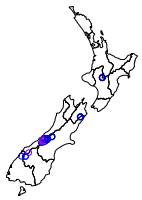
group_by() %>% summarise()

It is also possible to do this for sf `data.frames` by default a `st_union` is performed on the geometries.

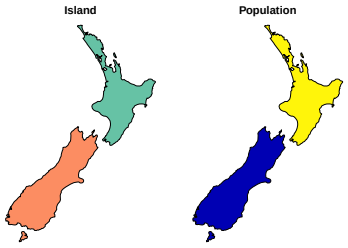
The input data for this part will be point and polygon data representing New Zealand (`nz`), from Chapter 4 of *Geocomputation with R*. The best place to start is to plot the data so the first exercise is:

- Create a plot of the data so it looks like the map of New Zealand below (hint: use the `add = TRUE` argument).

```
library(spData)
plot(nz$geom)
plot(nz_height["elevation"], add = TRUE)
```

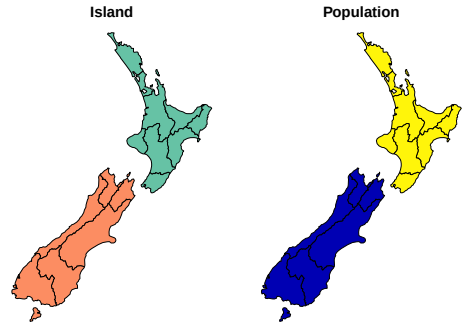


```
nz_islands = nz %>%
  group_by(Island) %>%
  summarise(Population = sum(Population))
plot(nz_islands)
```



Note that the implicit `st_union` has resolved all the internal boundaries of each island. If you wished to keep the boundaries you can use `st_combine`.

```
nz_islands = nz %>%
  group_by(Island) %>%
  summarise(Population = sum(Population), do_union = FALSE)
plot(nz_islands)
```



3.4 Geometric Operations

Geometric operation change or derive from the geometry of our data. The most commonly used functions are:

- `st_simplify` To simplify a complex shape
- `st_centroid` To find the geographical centre of a shape
- `st_buffer` To create a buffer around a shape

For more see Section 5.2 of Geocompuation with R

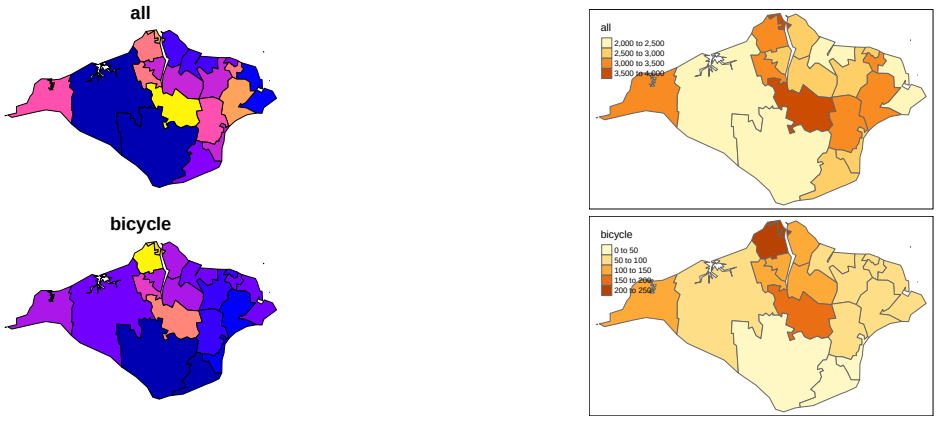
1. Canterbury is the region of New Zealand containing most of the 100 highest points in the country. Create an object called `canterbury` and use this to subset all points within the region. How many of these high points does the Canterbury region contain?
2. Which region has the second highest number of `nz_height` points in, and how many does it have?
    - Bonus: generalizing the question to all regions, how many of New Zealand's 16 regions contain points which belong to the top 100 highest points in the country? Which regions?
    - Bonus: create a table listing these regions in order of the number of points and their name.

4 Visualising spatial datasets

Many packages can be used to create maps in R. The most basic is with the `plot()` function. `ggplot2` is a powerful plotting package that is part of the `tidyverse`, that can create maps with the function `geom_sf()`. However, for publication maps, we recommend using `tmap` for reasons outlined in Chapter 8 of Geocomputation with R. Load the package as follows:

```
library(tmap)
```

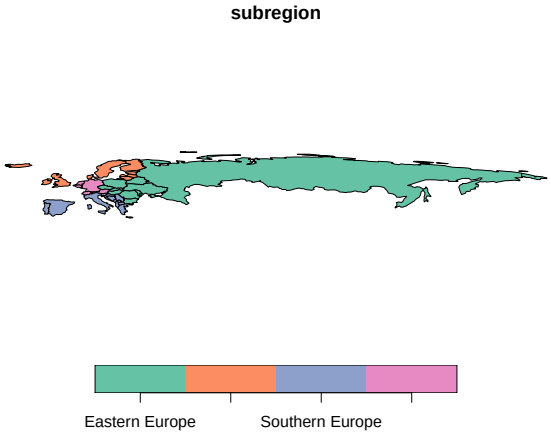
Create the following plots using `plot()` and `tm_shape() + tm_polygons()` functions.



These exercises rely on the object `europe`. Create it using the `world` and `worldbank_df` datasets from the **spData** package as follows (see Section 2.2 of Geocomputation with R for details):

```
europe = world %>%
  filter(continent == "Europe", !is.na(iso_a2)) %>%
  left_join(worldbank_df, by = "iso_a2") %>%
  dplyr::select(name, subregion, gdpPercap, HDI, pop_growth) %>%
  st_transform("+proj=aea +lat_1=20 +lat_2=-23 +lat_0=0 +lon_0=25")
```

1. Create a map showing the geographic distribution of the `gdpPercap` across Europe with base **graphics** (hint: use `plot()`) and **tmap** packages (hint: use `tm_shape(europe) + ...`).
2. Extend the **tmap** created for the previous exercise so the legend has three bins: "High" (above 30000), "Medium" (between 30000 and 20000) and "Low" (below 20000).
   - Bonus: improve the map aesthetics, for example by changing the legend title, class labels and colour palette.
3. Represent `europe`'s subregions on the map.



5 stats19 data analysis - with spatial/temporal analysis

1. Download and plot all crashes reported in Great Britain in 2017 (hint: see the stats19 vignette)
2. Filter crashes that happened in the Isle of Wight
3. Get and plot origin-destination data in `isle-of-wight` with the `pct` package hosted at: https://github.com/ITSLeeds/pct (bonus: look at the source code of `get_od()` and download the origin-destination data with `download.file()`)

5.1 Bonus exercises

Identify a region and zonal units of interest from http://geoportal.statistics.gov.uk/

1. Read them into R as an `sf` object
2. Join-on data from a non-geographic object
3. Add a data access section to your in-progress portfolio
4. Get origin-destination data from Uber

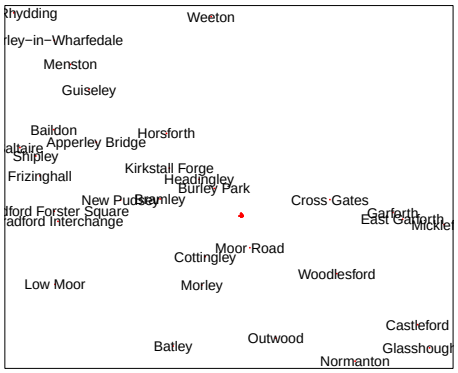
6 OD data with stplanr

1. Create an object representing desire lines in Isle of Wight, e.g, with: `desire_lines_all = pct::get_pct_lines(region = "isle-of-wight")`
2. Get data from Leeds and subset the desire lines with a value of `all` of 200 or above
3. Create a buffer of 500 m for each desire line and calculate the number of crashes that happened within each (using STATS19 data downloaded in the previous exercise)
4. Create a faceted plot showing the temporal distribution of crashes in Isle of Wight (you can choose whether to show these over the months of the year, over days of the week, or over the hours of a day)
5. Do a spatiotemporal subset to identify the crashes that happened within the most commonly travelled desire line between 07:00 and 10:00 during weekdays.

6.1 Accessing crowd-sourced data from OSM

- Type code into the script created in the previous section so that it can reproduce this plot:

```
library(osmdata)
location = opq("leeds") %>%
  add_osm_feature(key = "railway", value = "station") %>%
  osmdata_sf()
station_points = location$osm_points["name"]
tm_shape(location$osm_polygons) +
  tm_polygons(col = "red") +
  tm_shape(station_points) +
  tm_dots(col = "red")  +
  tm_text(text = "name", size = 1)
```



1. Download cycleway data with the tag highway=cycleway for Leeds from https://overpass-turbo.eu/
2. Load the data in R and plot it with `tmap` (bonus: now try to get the same data using the **osmdata** package)

7 Local route network analysis

Routing is the process of finding the "shortest" path from A to B. In this context shortest does not just mean in distance, it may be in time (quickest), or some other characteristic e.g. safest, quietest.

There are many packages that enable you to do routing in R. When choosing a package you should consider several characteristics:

Some packages can do local routing on your own computer. While others allow you to connect to a service.

Local Routing

- Usually requires more effort to set up
- No cost (except for time and hardware)
- Control over data, custom scenarios possible futures etc

Remote Routing

- Easy setup
- May charge or limit the number of routes
- May support more complex options e.g. traffic, public transport
- Usually limited to routing in the present e.g. current road network current transport timetables.

7.1 Routing Features

Not all routing services can do all types of routing, or do them equally well. Most do driving directions but consider if they do:

- Walking / Cycling (if so does it include specialist road types, exclude dangerous roads)
- Take account of hilliness
- Public Transport (if so does it include fares, which types?)
- Are public transport routes based on timetables or real-time service status?
- Take account of steps and disabled access
- Support specialist vehicles (e.g. lorries and low bridges)
- Does it support real-time or historical traffic data?

7.2 Routing packages for R

A non-comprehensive list of routing packages for R

7.2.1 Packages on CRAN

- googleway support for Google Maps and Directions
- mapsapi alternative for google maps
- osrmr Open Source Routing Machine, can connect to remote
- CycleStreets Specialist cycling routing, used by
- dodgr Routing done in R
- igraph General network analysis, not transport specific
- stplanr Limited routing functions based on dodgr and igraph, and some other services.
- gtfsrouter For integrating GTFS public transport timetables

7.2.2 Packages on GitHub

- Open Route Service Connect to ORS website
- TransportAPI An ITS Leeds Package, in development
- OpenTripPlanner An ITS Leeds Package, local or remote OTP routing
- graphhopper

7.3 Getting some Routes

Many services require you to sign up for a free API key, to save some time we will use the TransportAPI: https://developer.transportapi.com/signup

We will install the packages

```
# Install packages from GitHub
remotes::install_github("ITSleeds/transportAPI")
remotes::install_github("ITSleeds/opentripplanner") # For the bonus exercises
```

And load the packages

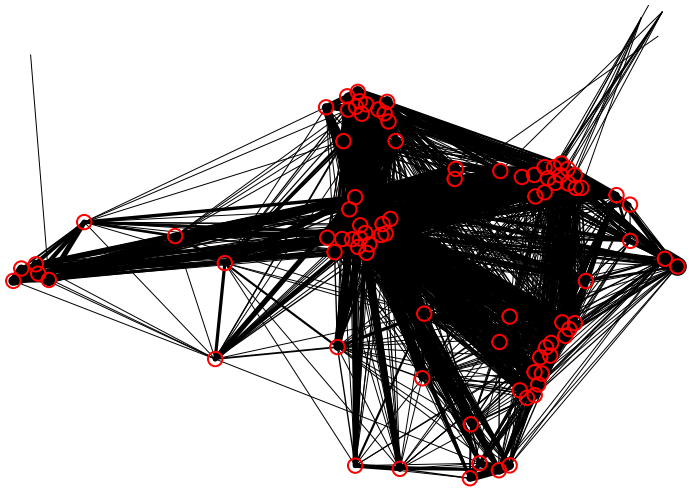
```
# Load packages
library(transportAPI)
library(opentripplanner)
```

TransportAPI you to signup for a free API key.

```
usethis::edit_r_environ()
# TRANSPORTAPI_app_id=your_id_here
# TRANSPORTAPI_app_key=your_key_here
```

Now we will get some data from the PCT. The `get_pct_lines` function returns the desire lines from the PCT.

```
lines = pct::get_pct_lines("isle-of-wight", "commute","lsoa")
lines = lines[,c("id","geo_code1","geo_code2","all",
                 "bicycle","car_driver","train_tube","bus")]
centroids = pct::get_pct_centroids("isle-of-wight", "commute","lsoa")
centroids = centroids[,"geo_code"]
plot(lines$geometry, lwd = lines$all / mean(lines$all))
plot(centroids, col = "red", add = T)
```



Now we will take the top 3 desire lines and route them though Transport API. First, we will subset the top lines and create from and to coordinates

```
lines_top = top_n(lines, 10, all)
from = centroids[match(lines_top$geo_code1, centroids$geo_code),]
to = centroids[match(lines_top$geo_code2, centroids$geo_code),]
```

Then we will use the `tapi_journey_batch` to find multiple routes at once (note: the numbers used are not real, add your own keys).

```
Sys.setenv(TRANSPORTAPI_app_id = "7e8661c5")
Sys.setenv(TRANSPORTAPI_app_key = "ce106381f6e5787f223e720b6055d4f8")
routes_car = transportAPI::tapi_journey_batch(from$geometry,
                                              to$geometry,
                                              fromid = from$geo_code,
                                              toid = to$geo_code,
                                              apitype = "car")
tmap_mode("view")
tm_shape(routes_car) +
  tm_lines()
```

If you cannot get this to work with your own API key, you can download the pre-generated file as follows:

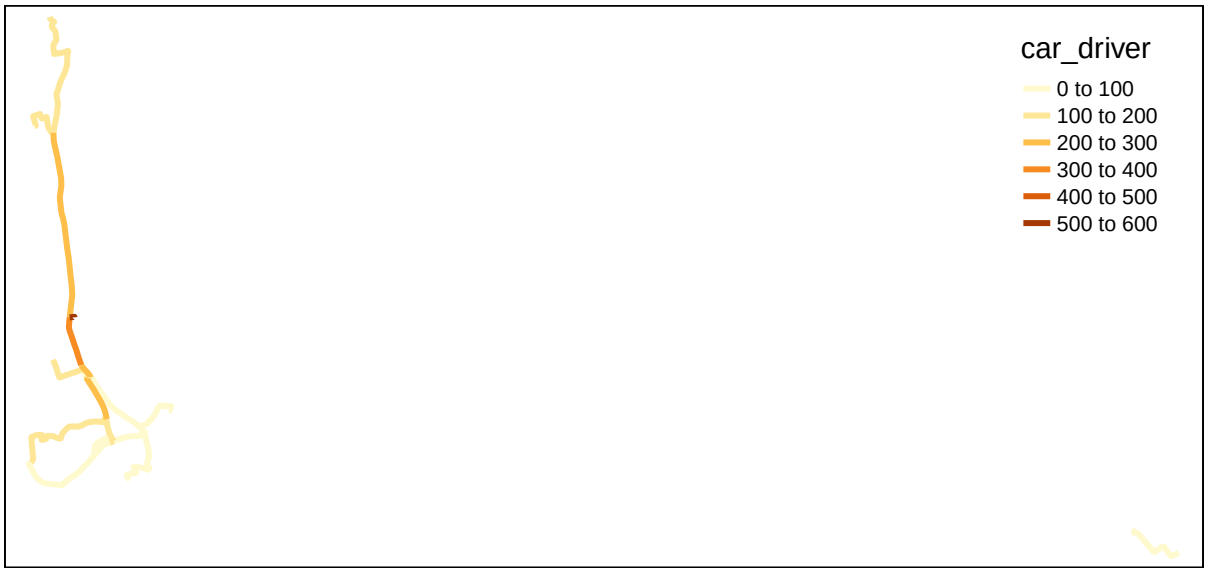
```
u = "https://github.com/ITSLeeds/TDS/releases/download/0.2/routes_car.geojson"
routes_car = sf::read_sf(u)
```

We need to join the number of commuters onto the geometry of the routes.

```r
st_geometry(lines_top) = NULL
routes_car = left_join(routes_car, lines_top, by = c("fromid" = "geo_code1", "toid" = "geo_code2"))
```

Finally, we can combine the routes into a route network

```r
library(stplanr)
rnet = overline2(routes_car, attrib = c("all","bicycle","car_driver","train_tube","bus"))
tm_shape(rnet) +
  tm_lines(col = "car_driver", lwd = 3)
```



7.4 Local Routing With Open Trip Planner

We will repeat the analysis using a local routing tool. This tutorial is based on the package vignette

First, we need some basic data.

```r
# Create Folders for Data
dir.create("OTP")
path_data = file.path("OTP")
path_otp = file.path(path_data, "otp.jar")
dir.create(file.path(path_data,"graphs")) # create a folder structure for the data
dir.create(file.path(path_data,"graphs","default"))

# Download OTP and Data
download.file(
  url = "https://repo1.maven.org/maven2/org/opentripplanner/otp/1.3.0/otp-1.3.0-shaded.jar",
  destfile = path_otp, mode = "wb")
```

```
download.file(
  "https://github.com/ITSLeeds/opentripplanner/releases/download/0.1/isle-of-wight-demo.zip",
  destfile = file.path(path_data,"isle-of-wight-demo.zip") , mode="wb")
unzip(file.path(path_data,"isle-of-wight-demo.zip"), file.path(path_data, "graphs", "default"))
```

Now we set up the OTP

```
log = otp_build_graph(otp = path_otp, dir = path_data)
otp_setup(otp = path_otp, dir = path_data)
otpcon = otp_connect()
```

Next, we find the routes

```
routes_driving = otp_plan(otpcon, fromPlace = from, toPlace = to, mode = "CAR")
```

7.5 Exercises

1. Get routes for the top 5 desire lines for different modes e.g. car, public transport, bike
2. How are public transport routes different from car and bike routes?
3. Plot these routes together on a map, where are there complementary and conflicting routes?

8 Data and methods for assessing cycling potential

1. Identify the top 10 desire lines in Isle of Wight along which at least 100 people travel to work, by:
   - The percentage who walk
   - The percentage who cycle
   - Bonus: Find the top 10 for *all* modes of transport and plot the results
2. Download origin-destination data from 2011 Census using the function `pct::get_od()`.
3. Convert these origin-destination pairs into geographic desire lines between centroids in Leeds (e.g. as generated by the function `pct::get_pct_centroids()`) and plot the result.
4. Find the route along the most travelled desire line in Leeds and plot the result.
5. Get cycle route data for the Isle of Wight and use the function `overline2()` to identify the routes along which most people walk to work.