Spatial analysis in R

Christopher Jarvis

Spatial data models

In order to use spatial data we need a way of representing the spatial information called a data model. There are two main data models for representing spatial data: vector and raster.[1]

[1] If you want to learn more about vector and raster data with R, then there is a book called **Geocomputation in R** which can be accessed for free online. This book covers a range of ways to interact and manipulate with spatial data in R.

Vector data consists of points, lines, and polygons. Vector data is often used to represent discrete spatial objects such as household locations, roads, or countries. In this course, we will focus predominantly on using vector data in R. A common format for vector data is the shapefile.

- Points: Houses or disease cases
- Lines: Roads or rivers
- Polygons: Countries or administrative areas

Raster data consists of splitting an area up into grids and giving each grid a value. It is often used for measures such as population density, land cover, or elevation. We will not cover raster data in R[2]. Raster data is really just an image and because of this, there is quite a bit of overlap between spatial statistics and image processing methods. A common form of raster data is satellite imagery and they can be stored as a GeoTIFF file.

[2] There is a very good package called `raster` which can be used for interacting with raster data in R.

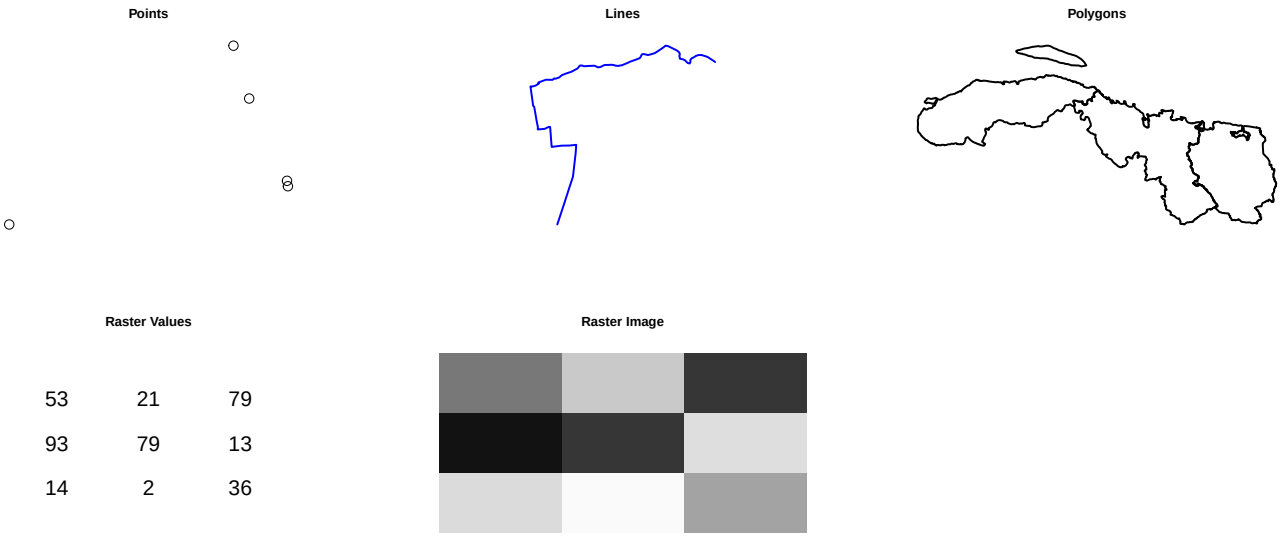

Figure 1: Spatial data types

Coordinate reference systems

Coordinate reference systems (CRS) define how the spatial data relates to the Earth (or other objects). In order to represent space, a system is required so that we can uniquely identify different locations. A CRSs can be categorised by whether it is geographic or projected.

Geographic CRSs can represent any location on the Earth's surface and consist of longitude and latitude. Longitude represents the East and West directions and latitude represents the North and South directions. Longitude and latitude coordinates are represented in degrees, which can make calculating and interpreting distances more difficult as the distance is based on a spherical rather than a flat shape.

Projected CRSs are produced by transforming the 3 dimensional surface of the Earth onto a two dimensional flat surface. Transitioning between 3 and 2 dimensions, inevitably results in a loss of structure and information such as area, distance, or shape. As these CRSs are based on a flat surface, they can be represented in distances such as metres, kms, or miles; this can make calculating and interpreting distances easier. Due to this, reprojecting data to a projected coordinate system is often done before calculating distances.

Each CRS is related to an `EPSG` code or a definition called a `proj4string`. The `EPSG` code and the `proj4string` are just names that allow people and computers to be sure they are using the same coordinates systems.

CRSs can cause confusion when first presented, but in practice the main things to keep in mind are:

- When working on a global scale a geographic CRS is required.
- The most common geographic CRS is WGS 84 which has an `EPSG` code of 4326 (A good code to learn.)
- When calculating distances and working at a country level a projected coordinate system is a good idea.
- The most common projected CRS is Universal Transverse Mercator (UTM), each country will have one or more UTM.
- If you work in only one specific country then learn the UTM code for that country. Otherwise search for the country specific UTM to find out the relevant code.

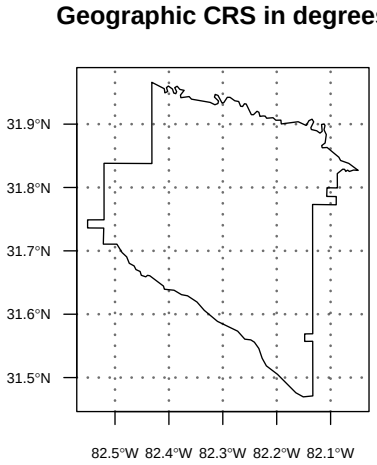

Figure 2: Map of geographic CRS

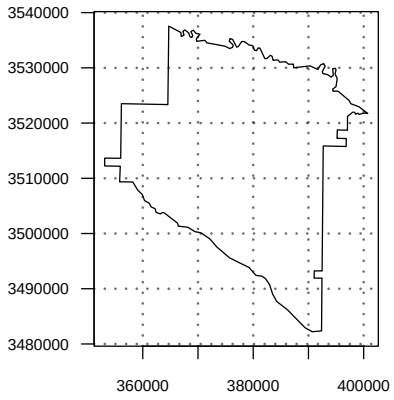

Figure 3: Map of projected CRS

R for spatial analyses

R has become a very capable software for spatial analyses. It allows for a wide range of simple and complex spatial manipulation and analyses. We will first focus on two packages for working with spatial data. In the previous sessions we learned about `ggplot2` for visualisation, `readr` for reading/writing data, and `dplyr` for data manipulation. In this session, we will learn about the package `tmap` for visualising spatial data and the `sf` package for reading/writing and manipulating spatial data.

Packages

The `sf` package simplifies the process for working with and manipulating vector spatial data in R. The package allows for reading/writing and manipulating spatial data.

It stores the data as an `sf` object which is a data frame. This means that the `dplyr` verbs that you learned in the previous session can be used on the spatial data. Therefore, without any extra knowledge you are already able to filter the spatial data using `dplyr`. The `sf` package also provides its own functions for data manipulation.

The `tmap` package provide functions to do thematic mapping in R. It has a similar syntax to `ggplot2`, with the main difference being that you can provide multiple spatial shapes in an easy way. We will be using `tmap` to create some maps.[3]

[3] When loading multiple packages a semi colon can be used to put the function on the same line.

```
library(dplyr);library(ggplot2);library(sf);library(tmap);
```

Reading and writing spatial data

Spatial data can be read into R using the `st_read()` command[4]. This command works in the same way as the `read_csv()` command. The `sf` package can be used to read and write a range of spatial data formats, including shapefile and geojson. The `st_write()` command can be used to write spatial files. Writing files requires the option `delete_layer=TRUE` to tell R to overwrite the existing file.

[4] Most of the functions within the sf package start with the prefix `st_`

Reading/writing shapefiles

```
adm2 <- st_read('Data/haiti/adm2.shp')
st_write(adm2, 'Data/haiti/adm2new.shp', delete_layer=TRUE)
```

Reading and projecting a csv file

Spatial data is often provided as a csv file with columns for coordinates. Turning this type of data into a spatial object requires an extra step to tell R what columns the coordinates are in. This code reads in the csv file and turns it into a spatial file.

```
hc <- read_csv('Data/haiti/haiti-healthsites.csv') # Read data
hc <- st_as_sf(hc, coords = c("x", "y"))  # Turn into spatial data
```

Checking and setting the projection of the data

When you read a spatial file into R it will hopefully have a projection file. The projection of the data can be checked using the `st_crs()` to check the crs of an object and `st_set_crs()` to set the crs when the object does not have a crs.

```
st_crs(hc) # Check the CRS
hc <- hc %>% st_set_crs(4326) # Set the CRS
st_crs(hc) # Recheck
```

If you know the crs of the data but it hasn't been assigned then the `st_set_crs()` command can be used to set the crs. This command should only be used when the data does not have a crs. If the data already has a crs then the `st_set_crs()` command will change the name of the crs but will not actually change the projection. In order to change the projection the `st_transform()` function is needed. If you wanted to project the health centre data to UTM you would use the `EPSG` code 32618 as follows:

```
# Project data
hc_utm <- hc %>%
  st_transform(crs = 32618)
st_crs(hc_utm) # Check the CRS
```

Basic plotting

You can create basic plots of the spatial data using the base R plotting function `plot()`. The default is to plot every variable in the data frame. To avoid this you can use the function `st_geometry()` inside of the `plot()` function.

```
plot(adm2) # plots every variable
plot(st_geometry(hc)) # plots just the shape
```

You can create multi layer plots using the option `add=TRUE`.

```
plot(st_geometry(adm2))
plot(st_geometry(hc), add = T) # add to the existing plot
```

Exercise 1

1. Read in `rivers.shp` using `st_read()`.
2. Check the crs with `st_crs()`. Should you rename to `river_utm`?
3. Explore the object using `head()`, `tail()`, and `str()`, what structure does it have?.
4. Check the crs of `adm2`.
5. Create an object `adm2_utm` where the data is reprojected to `EPSG` code 32618.
6. Do a basic plot of `adm2_utm` spatial data using `plot()` and `st_geometry()`.
7. Add the points from the `hc` object.
8. Add the points from the `hc_utm` object.
9. Add the rivers to the map.

Solution

```
library(dplyr);library(ggplot2);library(sf);library(tmap);

rivers_utm <- st_read('Data/haiti/rivers.shp')
st_crs(rivers_utm)
head(rivers_utm)
tail(rivers_utm)
str(rivers_utm)

st_crs(adm2)
adm2_utm <- st_transform(adm2, crs = 32618)

plot(st_geometry(adm2_utm))
plot(st_geometry(hc), add = T)
plot(st_geometry(hc_utm), add = T)
plot(st_geometry(rivers_utm), add = T, col = "blue")
```

Making maps with ggplots

The `ggplot2` package can also be used with spatial data. The main functions is the `geom_sf` which will plot simple features data.

We can create a basic map just by passing the spatial dataframe into ggplot and adding the geom_sf function.

```
ggplot(adm2_utm) +
  geom_sf()
```

To create a choropleth map you need to use the `fill` argument in the `aes` of the `geom_sf` function.

```
ggplot(adm2_utm) +
  geom_sf(aes(fill = female))
```

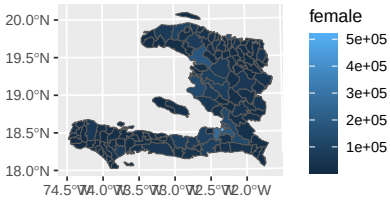

Figure 4: Choropleth map - ggplot2

To create a map with points and lines on, we can add additional `geom_sf` functions but we need to tell `ggplot2` to plot different data. When using `geom_sf` we do not need to explicitly say the x and y co-ordinate to plot as that information is stored in the spatial dataframe.

```
ggplot(adm2_utm) +
  geom_sf() +
  geom_sf(data = hc_utm) +
  geom_sf(data = rivers_utm, color = "blue")
```

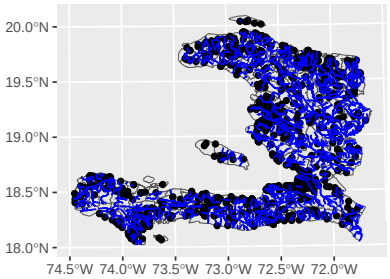

Figure 5: Detailed Haiti map - ggplot2

The maps can be saved in the same way as before using the `ggsave` function from `ggplot2`.

Making maps with tmap

The `tmap` package is used to create thematics map in R. It is similar in style to `ggplot2`.

Syntax

All `tmap` plots begin with the command `tm_shape()`. This tells R which object you want to map. Once you've defined the `tm_shape()` add extra functions to tell R how you want the map to be created. For example, to create a basic map of the admin areas. You tell R that `adm2_utm` is the object you want to map, and add `tm_polygons()` to tell R to create polygons in the map.

```
tm_shape(adm2_utm) + # What you want to map
  tm_polygons() # How you want to map it
```

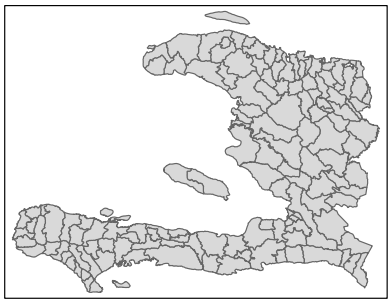

Figure 6: Basic Haiti map - tmap

There are several different commands for different types of maps.

Table 1: tmap drawing commands

Data type	Command	Description
polygon	tm_polygons	Draw polygons
polygon	tm_borders	Draws polygon borders
polygon	tm_fill	Fills the polygons
line	tm_lines	Draws lines
point	tm_bubbles	Draws bubbles
point	tm_squares	Draws squares
point	tm_dots	Draws dots
point	tm_markers	Draws markers

Choropleth maps

Choropleth maps can be created easily by defining the variable that you want to visually display as follows:

```
tm_shape(adm2_utm) +
  tm_polygons("urban")
```

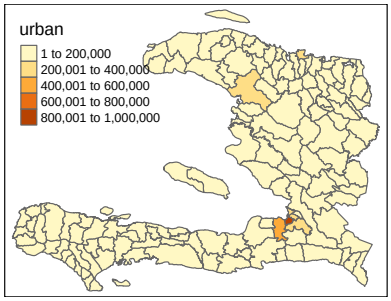

Figure 7: Unscaled choropleth map

A common issue with choropleth maps is that if they are not adjusted for the total population in each area. When this is the case, the maps tend to only represent population density rather than the attribute you are interested in. To resolve this, you can scale the value by the total population. For instance, we can represent the percentage of urban or female individuals in the population. In `tmap` all that is required is that you concatenate the variable names using the `c()` function. The titles for the legend can also be improved by using the title option. Here we also adjust the legend using `tm_legend()` to define that it resides in the top left of the map. You build the plot up by adding extra elements. These elements will depend on the type of spatial data you are using.

```
tm_shape(adm2_utm) +
  tm_polygons(c("urbanpct", "femalepct"), title = c("Urban %", "Female %")) +
    tm_legend(legend.position = c("left", "top"))
```

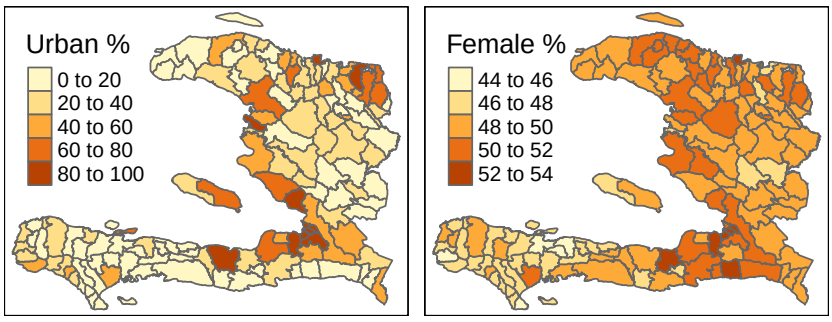

Figure 8: Scaled choropleth for two variables

Adding extra shapes

Extra spatial shapes can be added easily by using the `tm_shape()` command again as follows.

```
tm_shape(adm2_utm) +
  tm_polygons() +
tm_shape(rivers_utm) +
  tm_lines(col = "blue") +
tm_shape(hc_utm) +
  tm_dots()
```

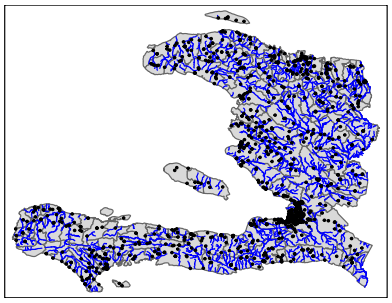

Figure 9: Haiti with river and health centres

Saving maps

To save a map you store the map as an object and then use the `tmap_save()` function. The maps can be stored as a png, jpg, pdf, svg, bmp, or tiff. You can specify the width, height, and the unit type ("cm", "in", "mm").

```
tm <- tm_shape(adm2_utm) +
  tm_polygons(c("urbanpct", "femalepct")) +
    tm_legend(legend.position = c("left", "top"))

tmap_save(tm, "output/Haiti_choro.png", width=10, height=10, units = "cm")
```

For more with `tmap` see the webpage `https://r-tmap.github.io/tmap/index.html`.

Exercise 2

1. Create a new variable for the percentage of under 18s
2. Create a choropleth of the percentage of under 18s using ggplot2
3. Create a choropleth of the percentage of under 18s using tmap
4. Type ?`tmap-element` into the console and see if you can figure out how to add a scale bar.
5. Save the map in the output folder as a png.
6. See if the image exists.

Solution

```
adm2_utm <- adm2_utm %>%
  mutate(under18pct = undr_18/total)

## ggplot2
ggplot(adm2_utm) +
  geom_sf(aes(fill = under18pct))

## tmap
tm_shape(adm2_utm) +
  tm_polygons("under18pct", title="% of under 18's ") +
  tm_legend(legend.position = c("left", "top")) +
  tm_scale_bar(position = c("left", "center")) -> tm

tmap_save(tm, "output/Haiti_under18.png", width=10, height=10, units = "cm")
```

Spatial data manipulation

dplyr verbs

The `dplyr` commands can be used on the spatial data frame. For instance, we can use the `filter()` command to create a subset of the data. This can be useful when we want to highlight a specific area of a map.

```
west_utm <- adm2_utm %>%
  filter(adm1_en == "West")

tm_shape(adm2_utm) +
  tm_polygons() +
tm_shape(west_utm) +
  tm_polygons(col = "red")
```

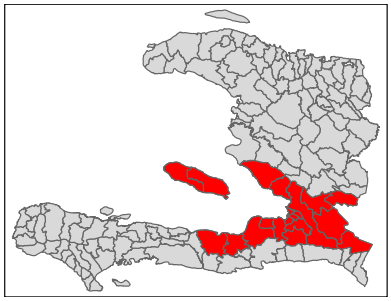

Figure 10: Haiti map highlighting West department

Checking spatial data

First let's load the required packages and load some data.

```
library(mapview); library(sf);

adm2 <- st_read('Data/haiti/adm2.shp')
adm2_utm <- adm2 %>%
  st_transform(crs = 32618)
hc_unprojected <- read_csv('Data/haiti/haiti-healthsites.csv') %>%
        st_as_sf(coords = c("x", "y"))  # Turn into spatial data
hc<- hc_unprojected %>% st_set_crs(4326)
north_utm <- adm2_utm %>% filter(adm1_en == "North")
```

Making maps with mapview

The mapview package provides an easy way to check your data interactively. To create an interactive map you give the `mapview()` a projected spatial file. Multiple objects is also easy, just add them together[5]. If you do not have any projection, the data will still be plotted but the basemaps won't. You can also combine different types of spatial objects and zoom to different areas. The map can be viewed in the browser and a html file can be saved using RStudio. Export>Save as webpage.

[5] Note that mapview will automatically reproject data to match existing layers, so even though adm2 and north_utm have different projections they are in the same place.

```
mapview(adm2) # Single map
mapview(adm2) + mapview(north_utm, col.regions = "red") # Multiple objects
mapview(hc_unprojected) # Not projected - no base map
mapview(adm2) + mapview(north_utm, col.regions = "red") + mapview(hc)
```

This command can be really useful to check whether your coordinates are based in the same place. It should be noted that since mapview will automatically project data to match existing layers, it won't tell you if the spatial data have a different crs.[6]

[6] For spatial data that has been read directly into R, the `st_crs()` command can be used to compare the projections of the data. If all you receive is the coordinates as columns in a csv file, then it is good to check long and lat, followed by the UTM of the area, following by mercator. If none of these work, then you'll need to contact who sent you the data and see if they can help.

Exercise 3

1. Load rivers.shp and adm2.shp
2. Create an interactive map with mapview.
3. Filter adm2 to the South area.
4. Add the South area to the map.

Solution

```
rivers <- st_read('Data/haiti/rivers.shp')
adm2 <- st_read('Data/haiti/adm2.shp')

mapview(adm2) + mapview(rivers)

adm2_south <- adm2 %>% filter(adm1_en == "South")

mapview(adm2) + mapview(rivers) +
  mapview(adm2_south, col.regions = "pink")
```

sf spatial functions

The `sf` package also provides functions for manipulating spatial data. There are a lot of spatial function included in the `sf` package but we will first focus on a few of them. Nearly all of the functions have the prefix `st_`. When using RStudio this means you can take advantage of the auto-complete. Typing `st_` then tab gives a list of possible commands that will appear alongside a description of what they do.

```
capital <- st_read('Data/haiti/capital.shp')
st_crs(capital)

# Create a 30k buffer around the capital
capital_buffer30k <- st_buffer(capital, dist = 30000)
# Extract the centroids of a polygon
west_centroids <-  st_centroid(west_utm)
# Unify into one shape
west_oneshape <- st_union(west_utm)
# Find the intersection between the west and capital buffer
intersect30k_utm <- st_intersection(west_utm, capital_buffer30k)
# Find the difference between west and capital buffer
difference30k_utm <- st_difference(west_utm, capital_buffer30k)
```

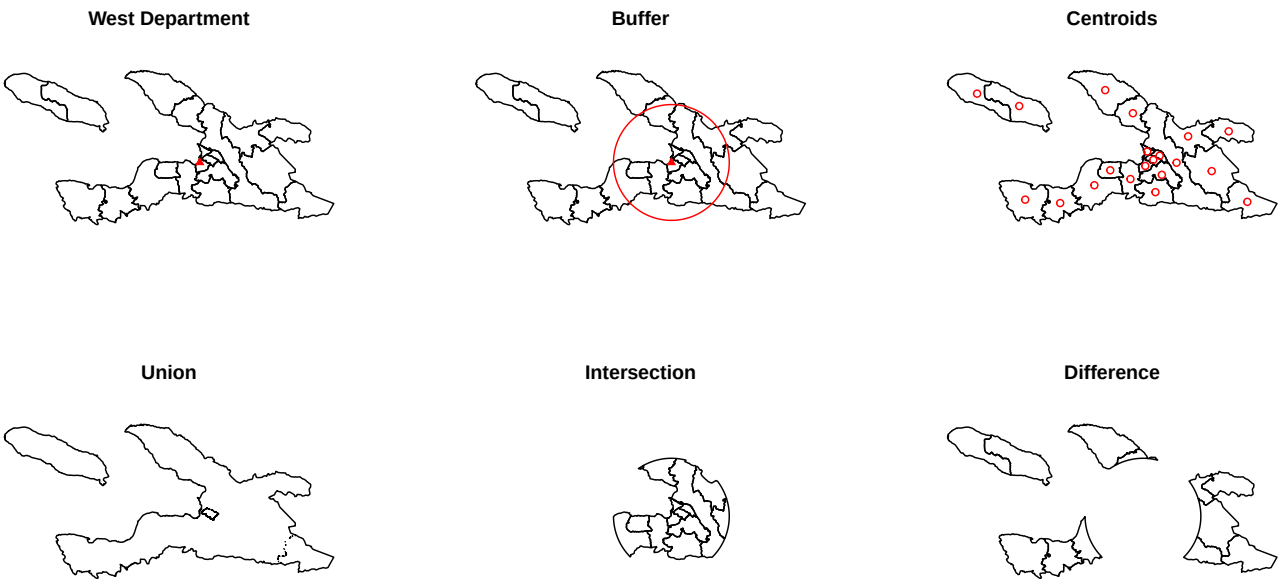

Figure 11: Spatial manipulations using sf

Now we will look at how to use the spatial functions to perform a basic spatial analysis and visualisation in the practical session.

Exercise 4

1. Create a 10k buffer around the capital and plot the 30 and 10k buffer on the same map
2. Turn the 10k buffer into a single shape using `st_union` and plot.
3. Check the intersection between the Haiti and the 10km buffer and plot.
4. Try some further manipulations of your choice.

More with the sf package

Distances

The `st_distance()` command can be used to calculate straight line distances. It can be used to calculate the distance between two single points, a single point and multiple points, and between mutiple points.

This function always returns a matrix structure, where each element represents a distance. For distances from a single point to a single/multiple point(s), this matrix will have only one row which we may redefine as a vector.

```
capital <- st_read('Data/haiti/capital.shp') # Load data point for capital
hc_utm <- hc %>% st_transform(crs = st_crs(capital)) # project hc

hc_dist <- st_distance(capital, hc_utm) %>% as.vector()

# We can use min() to find the closest health centre to the capital.
min(hc_dist)

# The distance can be added to the data frame using mutate.
hc_utm %>% mutate(hcdist = hc_dist)


# Use filter to find out which health centre is closest
hc_utm %>% filter(hc_dist == min(hc_dist)) %>%
  select(name)
```

Calculating the distance between two objects with multiple points will return a matrix with multiple rows.

```
adm2_centroids <- st_centroid(adm2_utm)

adm2_hc_dist <- st_distance(adm2_centroids, hc_utm)

# We can still use min() to find the smallest distance
min(adm2_hc_dist)
```

But we'll need to learn a bit more R to interact with this object. We can subset a matrix using square bracket notation `matrix[rows, columns]`. We can get all the elements in the first row by writing `matrix[1,]` all elements in the first column with `matrix[,1]`. In R there is a helper operator `:` which can be used to get all the integers between two numbers. So 1:4 gives 1, 2, 3, 4. We can take advantage of this select the first 5 rows and the first 3 columns[7].

[7] This syntax can also be used for data frames

```
# 5 rows and 3 columns
adm2_hc_dist[1:5,1:3]
```

When we had a single column we could use `min()` to get the closest health centre. However, now we have multiple locations we are more likely to want to know the distance from each centroid to their nearest health centre. This requires finding the minimum distance of each column or row, depending on the order we put the objects into the `st_distance()` function.

There are several ways to calculate the row or column minimums but I'll present one.[8] Again we're going to learn a little more base R. This time we'll use the `apply()` command. This command applies a function to each row or column of an object such as a matrix. The option `MARGIN` tells R whether it is a column or a row. A value of 1 is row, and 2 is the column.

[8] If you're interested then `do.call()`, `pmap_dbl()`, or map_dbl() functions can also be used

```
# Row minimum
nearest_centroid <- apply(adm2_hc_dist, MARGIN = 1, FUN = min)
# Column minimum
nearest_healthcentre <- apply(adm2_hc_dist, MARGIN = 2, FUN = min)
```

Exercise 5

1. Load the capital.shp and adm1.shp
2. Reproject adm1.shp to UTM. (Use the `st_crs()` of capital)
3. Calculate the centroid of adm1.shp
4. Calculate this distance of the centroids from the capital.
5. What's the maximum distance?

Calculating distance, especially the nearest distance, is a common task in spatial analyses.

Solution

```
capital <- st_read('Data/haiti/capital.shp')
adm1 <- st_read('Data/haiti/adm1.shp')

adm1_utm <- st_transform(adm1, crs = st_crs(capital))

adm1_centroid <- st_centroid(adm1_utm)

adm1_dist <- st_distance(adm1_centroid, capital)

max(adm1_dist)
```