BIOM4051 R Tutorial 1: Vector basics

Figure 1: Common dolphins in Falmouth Bay

Introduction

In this first tutorial we are going to introduce some basic operations working with spatial vector data in R, building on tasks that you have been routinely performing in QGIS. The dataset we will be using are some fictitious sightings of marine mammals gathered during MSc boat trips out into Falmouth Bay. Our goal is to simply plot out the distribution of sightings data, merge data collected from a few different trips, and export the result, as well as doing some simple calculations of distance travelled and distance from shore. By the end of this tutorial you should be able to:

1. Import vector data into R from a variety of spatial and non-spatial file formats
2. Interact with and manipulate attribute tables (or dataframes).
3. Plot and visualize spatial vector data in R
4. Reproject vector data into different coordinate reference systems
5. Calculate some standard geometric measurements (distance, length and area)
6. Merge vector datasets from different sources
7. Save/export spatial vector data

The annotated code in this sheet is intended to be used alongside the accompanying video for Tutorial 1

Setting the working directory and loading required packages

Before we launch into working with data there is some basic setup we need to do which will be the same for all of our R sessions.

The first thing we need to do is to set a path to our working directory which tells R where to look for the data files we will be using and where to save any outputs. You set the working directory using the `setwd` function:

```
setwd('~/Tutorial 1')
```

Next we need to load any external packages that are used by R. We are going to be using package `sf` for working with spatial vector data and we will also load package `tidyverse` for some useful data manipulation functions.

```
library(sf)
library(tidyverse)
```

You can find help on package `sf` by typing `help(package = 'sf')` which will take you to an online manual with all of the different functions that the package contains, along with links to vignettes giving an accessible overview to some of the features. The package cheat sheet is also a really good place to start.

1. Loading spatial vector data using `st_read`

There are various ways of importing vector data into R. One of the most common is to read data from a spatial vector file format like an ESRI shapefile or a geopackage. The function for reading these file types is `st_read`. So go ahead and load in the `sightings.shp` file from the Tutorial 1 data folder (which contains our marine mammal sightings) and name the object 'sightings':

```
sightings = st_read('data/sightings.gpkg')
```

Note that the filepath we put in `st_read` is relative to our working directory, which in this case is the main Tutorial 1 folder. So we tell the function that the file we want is in the 'data' sub-directory.

Let's inspect the data we have imported by printing a summary, which we do just by typing the object's name:

```
sightings
```

```
## Simple feature collection with 40 features and 3 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 179572.3 ymin: 18198.6 xmax: 190863.3 ymax: 33215.13
## Projected CRS: OSGB36 / British National Grid
## First 10 features:
##                   date          species N                      geom
## 1  2021-09-16 10:00:00        Grey seal 1 POINT (181424.9 33162.07)
## 2  2021-09-16 10:06:09        Grey seal 1  POINT (181891.5 33121.7)
## 3  2021-09-16 10:12:18        Grey seal 1 POINT (182036.3 30942.98)
## 4  2021-09-16 10:18:27        Grey seal 1 POINT (179971.7 27830.65)
## 5  2021-09-16 10:24:36        Grey seal 1 POINT (179718.5 27052.75)
## 6  2021-09-16 10:30:46        Grey seal 1 POINT (179610.3 26789.06)
## 7  2021-09-16 10:36:55 Harbour porpoise 1 POINT (179572.3 26523.56)
## 8  2021-09-16 10:43:04        Grey seal 1  POINT (179983.3 25585.8)
## 9  2021-09-16 10:49:13        Grey seal 1 POINT (180389.1 25290.84)
## 10 2021-09-16 10:55:23        Grey seal 1 POINT (180670.6 24661.72)
```

There are a few things to notice from the summary. This object is a simple feature collection. It contains 40 features - which are our sightings - and if we look at the geometry we can see these are POINT features, with projected coordinates on the British National Grid.

Looking at the structure of you can also see that an `sf` object is basically a special type of dataframe (if you type `class(sightings)` you will see this). Each row consists of one observation (or feature) and each column contains the attributes of that feature. In this case the summary tells us that there are three fields or attribute columns (the date of the observation, the species, and the number observed). There is also a special `geometry` column which is unique to `sf` objects and contains the geometry of each feature. In this case it is the coordinates of the points, but it could also be sequences of vertices making up lines or polygons.

2. Manipulating attribute tables

Because `sf` objects are essentially a special class of `dataframe`, most of the standard `dataframe` subsetting and processing functions you may have learned in R will work with it, including base functions and more sophisticated ones from packages like `tidyverse`. Here are some examples of some common dataframe (or attribute table) manipulations that we can perform. Have a go at running each of them in turn and inpsect the outputs. It is not intended to be an exhaustive list so experiment with others too.

```r
# Find the names of the attribute columns
names(sightings)

# Find the number of rows or observations in the dataset
nrow(sightings)

# Just get the first 2 observations (rows) using square bracket notation
sightings[1:2,]

# Extract just the species column
sightings[,'species']
subset(sightings, select = species)

# Remove the species column
subset(sightings, select = -species)

# Extract the species column as a simple character vector
sightings$species

# Find the number of different species
unique(sightings$species)

# Extract only records for Grey seal
subset(sightings, species == 'Grey seal')

# Extract records where not Grey seal and without the date column
subset(sightings, species != 'Grey seal', select = -date)

# Add a column (using the tidyverse function 'mutate')
```

```r
sightings$vessel = 'Free Spirit'

# Rename columns (using the tidyverse function 'rename')
rename(sightings,number = N)

# Reorder columns
sightings[,c('date','vessel','species','N')]
```

You'll notice that in most of these subsetting operations the geometry column comes along without you having to explicitly tell R that you want it, which is standard behaviour for an sf object. You can also access the coordinates of the geometries in the dataframe directly using `st_coordinates`:

```r
st_coordinates(sightings)
```

3. Plotting spatial vector data using ggplot

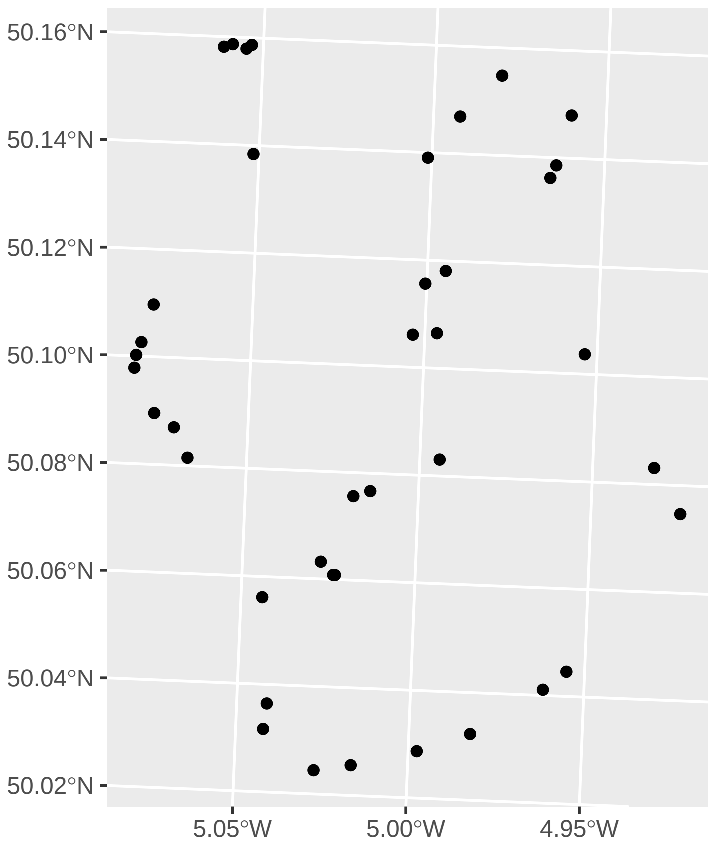
Now that we've inspected our sightings data we need to visualise it. There is a default plotting method for `sf` objects that you can call with `plot()`:

```r
plot(sightings)
```

However, it is quite limited. Instead we are going to use the much more flexible plotting options offered by the package `ggplot2`. `ggplot2` is now widely used for creating publication quality graphics in R, so it makes sense to use it for visualizing spatial data too.

In essence, `ggplot2` works by drawing a plotting canvas using the `ggplot()` function and then adding `geoms` to it. These can be `geom_line` or `geom_point` layers for plotting standard dataframes. However, for spatial vector data, `sf` ships with its own special kind of `geom` called `geom_sf` which you use for plotting all `sf` objects, whether they contain point, line, or polygon geometries.

```r
library(ggplot2)

ggplot() + geom_sf(data = sightings)
```



You'll notice that `geom_sf` applies no layer styling at all and plots the data on a standard ggplot canvas. It has also by default added a coordinate system, called `coord_sf`, which fixes the aspect ratio and has plotted longitude and latitude coordinates, even though our data was on the British National Grid. If you want to change this you need to specify the datum settings in `coord_sf` like this:

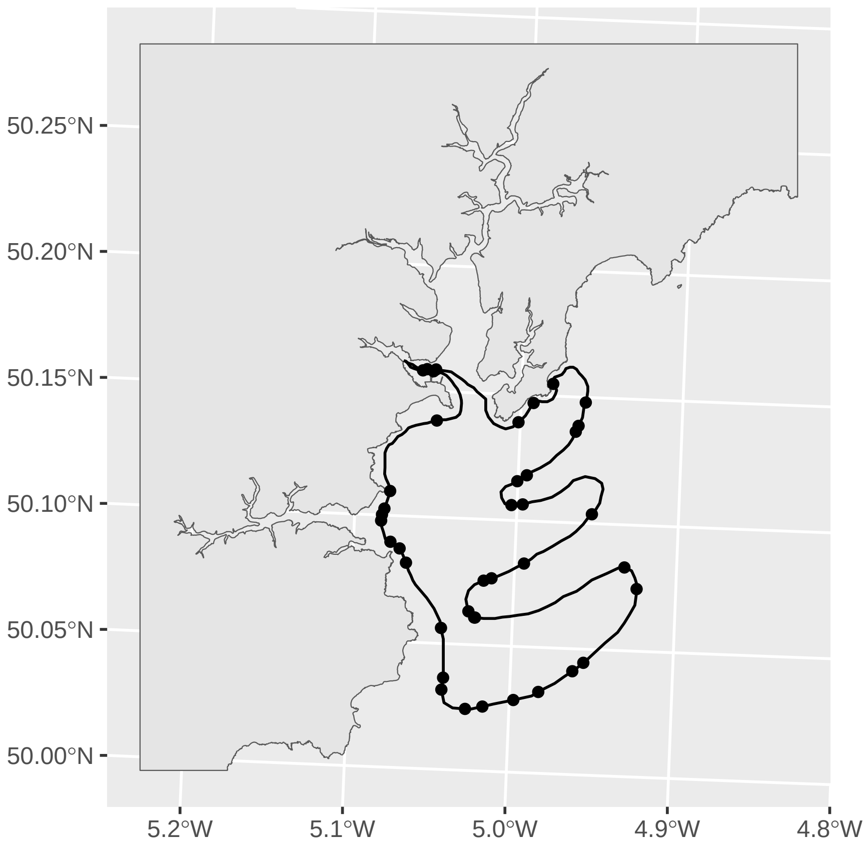
```r
# 27700 is the EPSG code of the British National Grid - more on EPSG codes later
ggplot() + geom_sf(data = sightings) + coord_sf(datum = 27700)
```

We need to add a bit more data to put this in geographic context, so let's load a background land polygon of Falmouth Bay and the route taken during our boat trip from the Tutorial 1 data folder.

```r
land = st_read('data/falmouth_bay.shp')
trip = st_read('data/boat_trip.gpkg')
```

You add layers to a `ggplot` by simply adding `geom_sf` objects to the plotting canvas in the order that they should be plotted:

```r
ggplot() +
  geom_sf(data = land) +
  geom_sf(data = trip) +
  geom_sf(data = sightings)
```



OK, so now we have the beginnings of a simple map. But it isn't very pretty to look at so we might want to do a bit of styling.

4. Basic styling of spatial plots

In ggplot, we control layer aesthetics using seven main arguments:

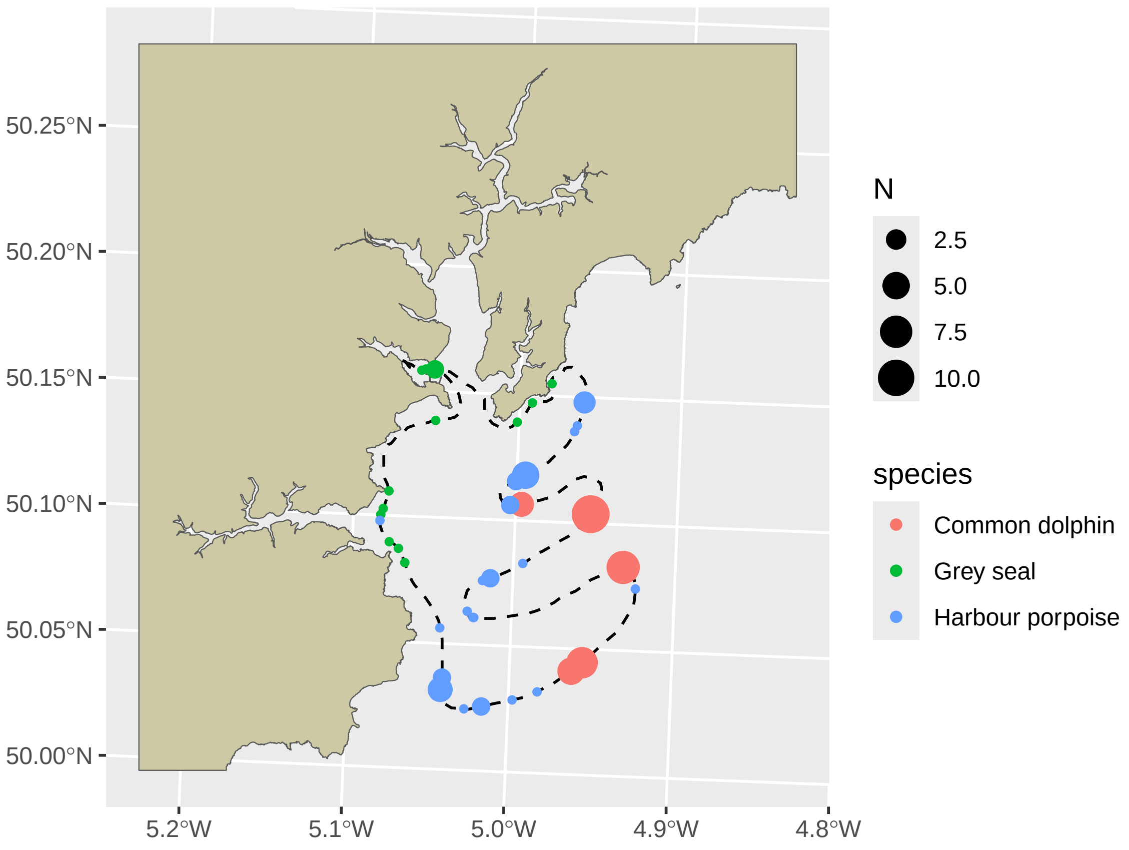
- `col`: colour of lines, points and polygon outlines
- `fill`: fill colour of polygons or some point types
- `alpha`: layer transparency (ranges from 0 - 1; 0 = transparent, 1 = opaque)
- `shape`: shape of point plotting symbols
- `size`: size of plotting symbols
- `linetype`: styling for lines and polygon boundaries
- `linewidth`: line thickness for line geometries and polygon boundaries

The different aesthetics you can set and the options that are available are described in detail here and the names of predefined colours that are recognised by R are summarised here. These aesthetics are set inside the `geom_sf()` call for each layer:

```
ggplot() +
  geom_sf(data = land, fill = 'lemonchiffon3',linewdith = 2, col = 'forestgreen') +
  geom_sf(data = trip ,col='blue',linetype='dashed',linewidth=1) +
  geom_sf(data = sightings, shape = 17, col='red',size=2)
```

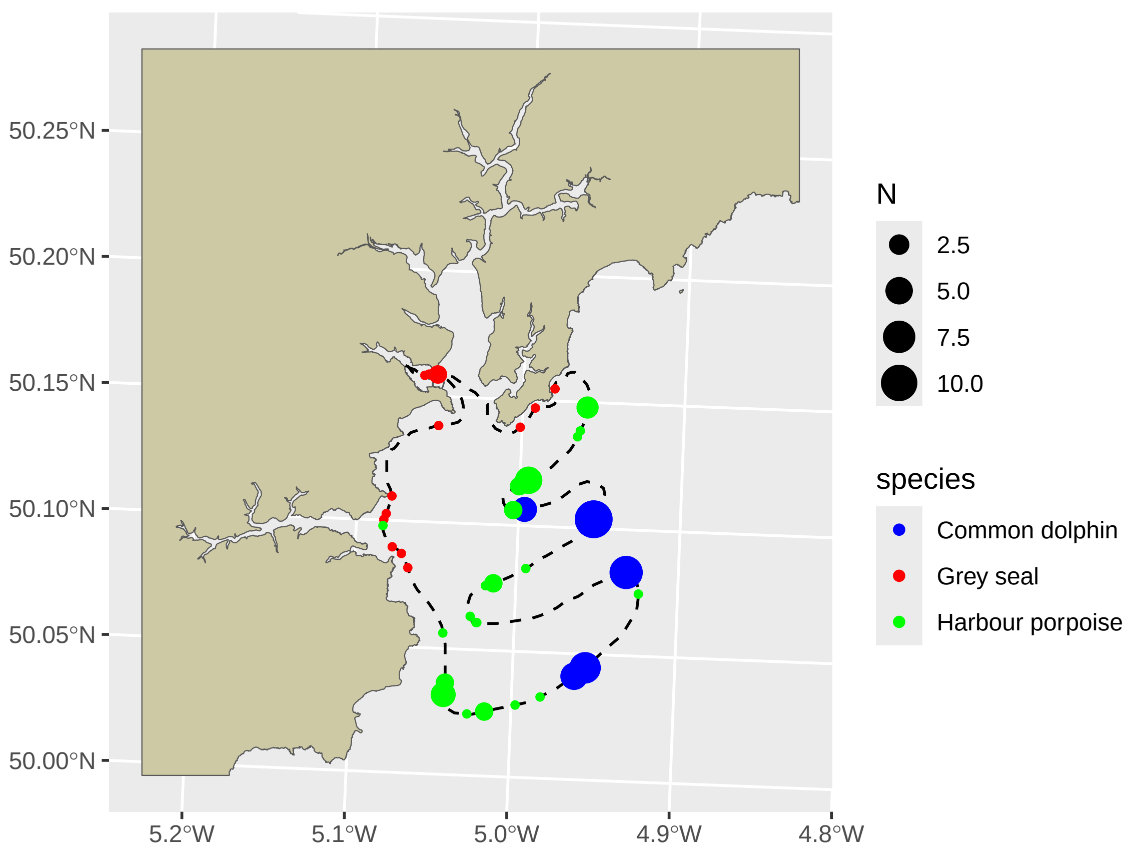
This changes the styling of all features in a layer. In many cases though we want to condition our styling based on some attribute in the dataset. For example, colouring points differently by species or changing their size based on the number of individuals sighted. We do that using the `mapping` argument in `geom_sf`, which involves wrapping up a series of statements inside a little function called `aes( )` (short for 'aesthetics') which tells the function how variables in the data map to the aesthetic properties described above. So, to condition the colour of our sightings points by species and the size by the number of individuals observed we would do:

```r
sightings.plot =
    ggplot() +
    geom_sf(data = land, fill = 'lemonchiffon3') +
    geom_sf(data = trip, linetype = 'dashed') +
    geom_sf(data = sightings, mapping = aes(colour = species, size = N))

sightings.plot
```



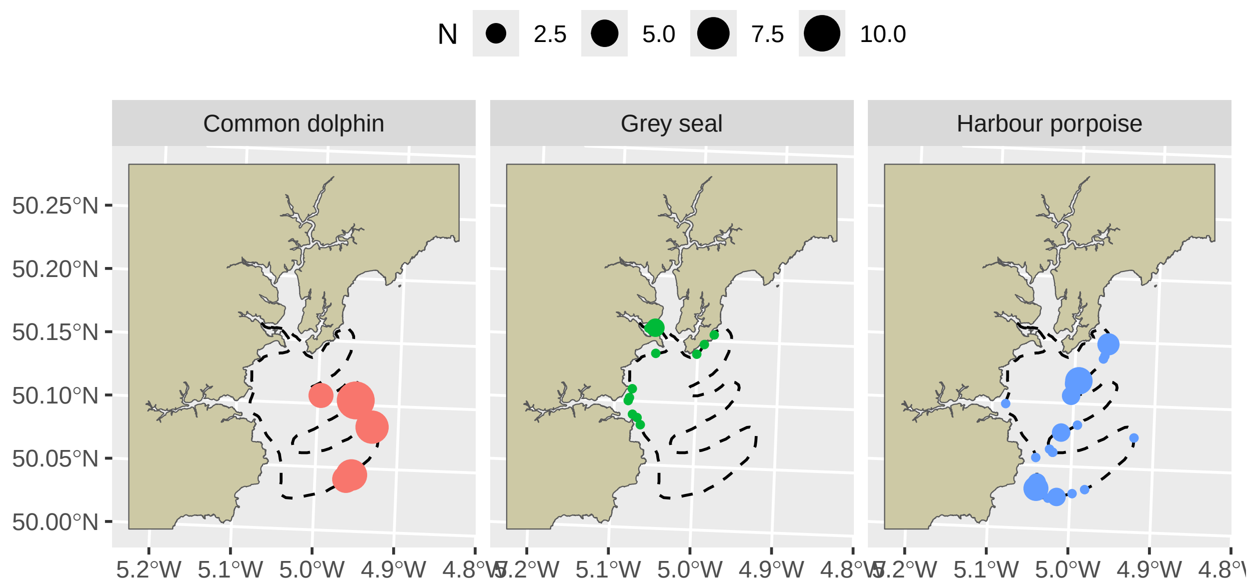
When applying conditional formatting, R automatically selects the colours to be used for each level. We can change these though by adding a manual colour scale using `scale_colour_manual()` with a lookup table that specifies the palette for each level (or species in this case)

```r
pal = c('Grey seal' = 'red',
        'Common dolphin' = 'blue',
        'Harbour porpoise' = 'green')

sightings.plot + scale_colour_manual(values = pal)
```



We can also make a separate plot (or 'facet') for each species using the `facet_wrap()` function:

```
# Note the tilde (~) added before the attribute we want to facet on:
sightings.plot + facet_wrap(~species)
```



We will look at some more styling options for ggplot maps in later tutorials. There are also some really good online guides, like this and this.

5. Converting XY data into spatial format using `st_as_sf`:

So far, we've imported vector data from spatial file formats like .shp and .gpkg. But what if we have data that isn't in a spatial format? Say a second boat has done another trip and sent us a spreadsheet containing coordinates of their sightings downloaded from a handheld GPS. Read in the data from this trip using `read.csv`:

```
sightings2 = read.csv("data/sightings2.csv")
sightings2
```

To do any spatial operations with this data or combine it with our original sightings data we need to convert this into an `sf` object. The function for converting anything into an `sf` object is `st_as_sf`. Because this excel file contains no spatial information, you need to tell the function which columns contain the X and Y coordinates using the `coords` argument, and what coordinate reference system (CRS) they are in using the `crs` argument. In this case, we know that our handheld GPS uses the WGS84 geographic coordinate system (latitude/longitude). The easiest way to specify the CRS is to pass `st_as_sf` the EPSG code, which you can look up here https://epsg.io/. For WGS84, the code is 4326:

```
sightings2 = st_as_sf(sightings2, coords = c('lon','lat'), crs = 4326)
```

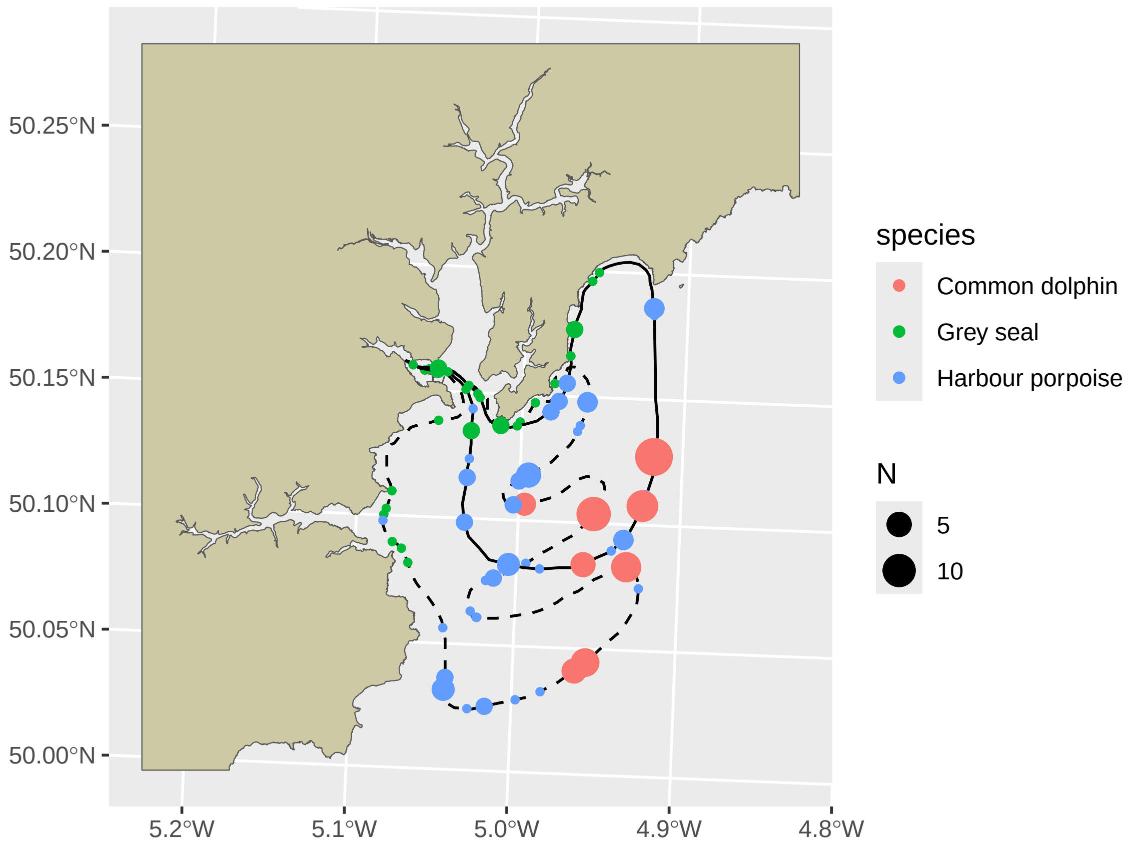
This creates an `sf` object with 32 point features.

We can also load in the track log for the second trip and plot it. In this case it is stored in a geopackage so we read it with `st_read()`.

```
trip2 = st_read('data/boat_trip2.gpkg')
```

Let's add both to our map using the same styling for sightings as previously:

```
sightings.plot +
  geom_sf(data = trip2) +
  geom_sf(data = sightings2, mapping = aes(colour = species, size = N))
```



6. Transforming coordinate systems using `st_transform`

So far we haven't been paying much attention to the coordinate reference systems of the data we have been bringing. All of the original layers we imported were projected on the British Standard Grid. However, the new layers we have just created from XY data are in the WGS84 geographic system. You can extract the coordinate references systems using the `st_crs` function and check if they are the same

```
st_crs(sightings2)
st_crs(sightings)
st_crs(sightings) == st_crs(sightings2)
```

We haven't noticed the difference because `ggplot` re-projects the data 'on-the-fly' (like QGIS) onto whatever datum is set in `coord_sf` (EPSG:4326 by default). However, if we try to do any spatial operations that involve combining these different datasets we are going to get error messages telling us that the coordinate reference systems don't match. So we need to transform some of our objects so that they all match. We do that using `st_transform`. There are two ways to specify the CRS that you want to transform to. You can either pass the EPSG code, or you can extract it from another object:

```
# These are both equivalent:

sightings2 = st_transform(sightings2,crs=27700)
sightings2 = st_transform(sightings2,crs=st_crs(sightings))
```

7. Merging spatial objects using `rbind`:

Now that our data are in the same coordinate system, we can merge the features from the two boat trips together into a single `sf` object (equivalent to a 'Merge shapefile' operation in QGIS). In R this is a simple case of binding the two dataframes together using the base function `rbind`, just as you would with any dataframe (but with the caveat that the CRS and geometry types also need to be the same):

```
sightings  = rbind(sightings,sightings2)
```

This throws a deliberate error to illustrate some of the common formatting problems encountered when dealing with real data. In this case, the dataframes won't bind because the column names don't match exactly: `sightings2` does not have a `vessel` column; and the geometry column has a different name in each dataset ('geom' and 'geometry'). We need to add a `vessel` column and rename the geometry column before binding. You can rename the geometry column using the `st_geometry` function:

```
sightings2$vessel = 'Free spirit'
st_geometry(sightings2) = 'geom'

sightings   = rbind(sightings,sightings2)
```

we can also merge our two boat trip tracks in the same way:

```
trips = rbind(trip,trip2)
```

8. Computing geometric measurements using `st_distance`, `st_length` and `st_area`

Now that we have our merged sightings and trip data, we might want do some simple geometric calculations. For example, we might want to measure how far we have traveled in total during our two trips. You measure the length of line features in `sf` using the function `st_length`:

```
st_length(trips)
```

```
## Units: [m]
## [1] 72075.11 46606.88
```

This function returns a `units` object containing the length of each line feature in metres. We probably want to add this as a column in the attribute table for our merged `trip` object:

```r
trips$distance.travelled = st_length(trips)
```

We might also want to calculate the distance of each of our sightings from shore e.g. to do some simple comparisons of habitat preferences for our species. We do that using `st_distance()`:

```r
st_distance(sightings, land)
```

This function also returns a `units` object, but this time it is a matrix with rows and columns. That's beacuse `st_distance(x,y)` measures the distance between each feature in x (rows) and each feature in y (the columns). In our case there is only one feature in `land` so only 1 column, and each row is a sighting. So, if we want to add a column called `shore.distance` to our sightings table we need specify that we want column 1 from the output of `st_distance`:

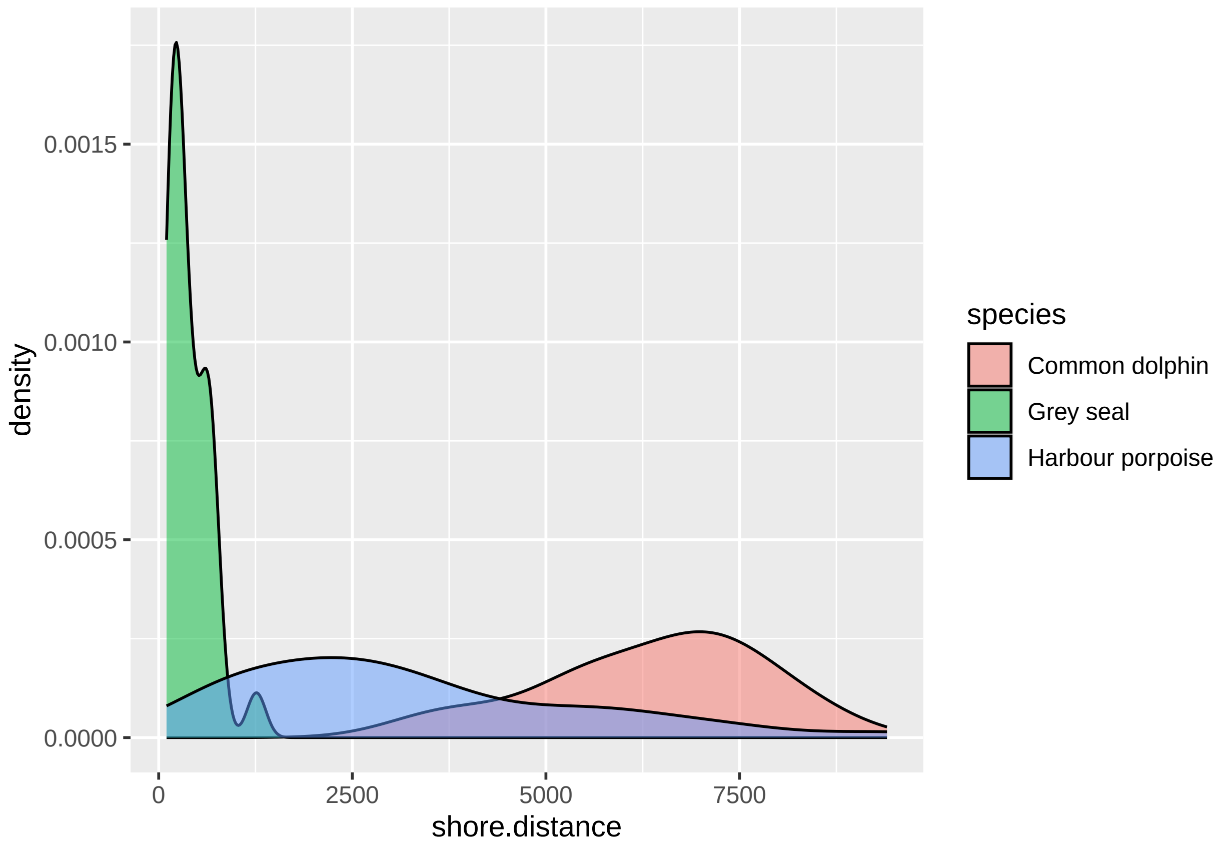
```r
sightings$shore.distance = st_distance(sightings, land)[,1]
```

Note that, while the `units` format is useful for confirming the units of measurement of the values returned by these functions, it is not handled that well by some other calculations or packages (including `ggplot`). So it is often worth converting it to a simple numeric field using `as.numeric`:

```r
sightings$shore.distance = as.numeric(sightings$shore.distance)
```

Finally, to show you how useful R can be at combining spatial processing and analysis in one workflow, let's plot how the distance of sightings from shore varies between our study species. We will use `ggplot` again, this time using `geom_density` to plot a smoothed histogram (or density plot) of `shore.distance` observations for each species:

```r
# We set transparency using `alpha` so each curve can be seen when superimposed
ggplot(sightings) +
  geom_density(mapping = aes(fill = species, x = shore.distance), alpha=0.5)
```



This suggests there are clear differences in the distributions of these species in terms of distance from land (which is unsurprising because data were simulated this way!)..

9. Exporting spatial objects using `st_write()`

Finally, we can export our merged sightings and survey tracks for future use using the `st_write` function. We can either export them as individual files:

```r
st_write(trips, dsn = 'merged_tracks.gpkg')
st_write(sightings, dsn = 'merged_sightings.gpkg')
```

Or write them as different layers in the same geopackage file:

```r
st_write(trips, dsn = 'merged_tracks.gpkg', layer = 'trips')
st_write(sightings, dsn = 'merged_tracks.gpkg', layer = 'sightings')
```