Manipulating and visualizing spatial data with R

Cheshire, James
james.cheshire@ucl.ac.uk

Lovelace, Robin
r.lovelace@leeds.ac.uk

January 5, 2014

Introduction

What is R?

R is a free and open source computer program that runs on all major operating systems. It relies primarily on the *command line* for data input: instead of interacting with the program by clicking on different parts of the screen, so users enter commands via the keyboard. This will seem to strange to people accustomed to relying on a graphical user interface (GUI) for most of their computing, yet the approach has a number of benefits, as highlighted by Gary Sherman (2008, p. 283), developer of the popular Geographical Information System (GIS) QGIS:

> With the advent of "modern" GIS software, most people want to point and click their way through life. That's good, but there is a tremendous amount of flexibility and power waiting for you with the command line. Many times you can do something on the command line in a fraction of the time you can do it with a GUI.

The joy of this, when you get accustomed to it, is that any command is only ever a few keystrokes away, and the order of the commands sent to R can be stored and repeated in scripts, saving time in the long-term and ensuring reproducible results (see "R and reproducible research").

Another important attribute of R, related to its command line interface, is that it is a fully fledged *programming language*. Other GIS programs are written in lower level languages such as C++ which are kept at a safe distance from the users by the GUI. In R, by contrast, the user inputs is the same as what R sees when it processes the request. Access to R's source code and openness about how it works has enabled a veritable army of programmers to improve R over time and add an incredible number of extensions to its capabilities. There are now more than 4000 official packages for R, allowing it to tackle almost any computational or numerical problem one could imagine.

Although writing R source code and creating new packages will not appeal to most R users, it inspires confidence to know that there is a strong and highly skilled community of R developers. If there is a useful function that R cannot currently perform, there is a reasonable chance that someone is working on a solution that will become available at a later date. One area where extension of R's basic capabilities has been particularly successful is the addition of a wide variety of spatial tools.

Why R for spatial data visualisation?

Aside from confusion surrounding its one character name [1] and uncertainty about how to search for help [2], R may also seem a strange choice for a tutorial on *spatial* data visualisation specifically. "I thought R was just for statistics?" and "Why not use a proper GIS package like ArcGIS?" are valid questions.

R was conceived - and is still primarily known - for its capabilities as a "statistical programming language" (Bivand and Gebhardt 2000). Statistical analysis functions remain core to the package but there is a broadening of functionality to reflect a growing user base across disciplines. R has become "an integrated suite of software facilities for data manipulation, calculation and graphical display" (Venables et al. 2013). Spatial data analysis and visualisation is an important growth area within this increased functionality. In recent years R has really made its mark as a data visualisation tool. The map of Facebook friendships produced by Paul Butler is iconic in this regard, and reached a global audience. He mapped the linkages between friends by calculating the great circle arcs between them (using the `geosphere` package) and plotted the result, displayed in figure 1. The secret to the success of this map was the time taken to select the appropriate colour palette, line widths and transparency for the plot. As we discuss in Section 3 the importance of these cannot be understated and are the difference between a stunning graphic and an impenetrable mess.

Figure 1: Iconic plot of Facebook freindship networks worldwide, by Paul Butler

The impact of the graphic was to inspire the R community to produce more ambitious graphics; a process fuelled by the increased demand for data visualisation and the development of sophisticated packages, such as ggplot2, that augment the basic plot functions of R. It is now the case that R has become a key analysis and visualisation tool used by the likes of Twitter, the New York Times and Facebook and thousands of consultants, design houses and journalists. It is not longer the preserve of academic research, with many graduate jobs listing R as a desirable skill.

Finally, it is worth noting that while dedicated GIS programs handle spatial data by default and display the results in a single way, there are various options in R that must be decided by the user, for example whether to use R's base graphics or a dedicated graphics package such as ggplot2. On the other hand, the main benefits of R for spatial data visualisation lie in the *reproducibility* of its outputs, a feature that we will be using to great effect in this tutorial.

R and reproducible research

There is a drive towards transparency in data and methods datasets in academic publishing. R encourages truly transparent and reproducible research by enabling anyone with an R installation reproduce results described in a previous paper. This process is eased by the RStudio integrated development environment (IDE) that allows 'live' R code and results to be embedded in documents. In fact, this tutorial was written in RStudio and can be recompiled on any computer by downloading the project's GitHub repository.

Getting started with the tutorial

The first stage with this tutorial is to download the data from GitHub, where an updated version is stored: github.com/geocomPP/sdvwR. Click on the "Download ZIP" button on the right, and unpack the folder to a sensible place on your computer (e.g. the Desktop). Explore the folder and try opening some of the files, especially those from the sub-folder entitled "data": these are the input datasets we'll be using.

For the purpose of this practical we will be using the RStudio software installed on a server. This marks a real step forward in terms of the use of R "in the cloud" by enabling users to access the software from their web browser. This is the first time we have tried this so lets hope it goes smoothly!

In Firefox go to http://marlin.casa.ucl.ac.uk/rstudio (this is only accessible if you are part of the UCL network). Enter the log in details you have been given and you should see the RStudio interface. In the bottom right window you should see an "Upload" button. Click on this and upload the zipfile you have just downloaded from github. RStudio will automatically uncompress this and you should see the list of files and folders appear. At the end of today you can tick the boxes next to the files/ folders you want and then under the "More" menu export them to a zipfile that you can email to yourself or take on a memory stick.

R and Spatial Data

Preliminaries

R has a unique syntax that is worth learning in basic terms before loading spatial data: to R spatial and non-spatial data are treated in the same way, although they have different underlying data structures. Try typing and running (by pressing `ctl-Enter` in an RStudio script) the following calculations to see how R works and plot the result.

```
t <- seq(from = 0, to = 20, by = 0.1)
x <- sin(t) * exp(-0.2 * t)
plot(t, x)
```

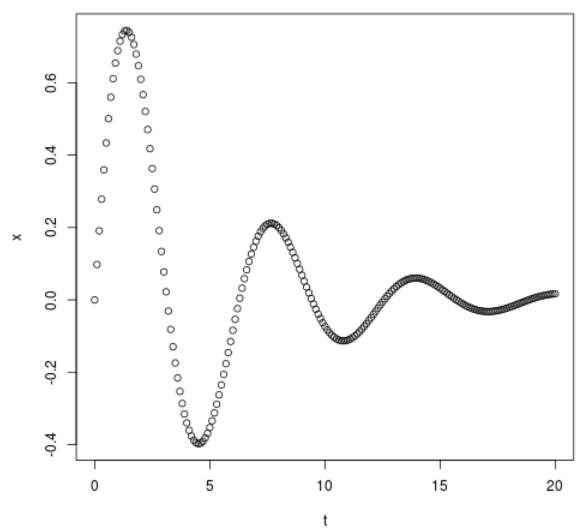

Figure 2: A preliminary plot

R code consists of *functions*, usually proceeded by brackets (e.g. `seq`) and *objects* (`d`, `t` and `x`). Each function contains *arguments*, the names of which often do not need to be stated: the function `seq(0, 20, 0.1)`, for example, would also work because `from`, `to` and `by` are the *default* arguments. Knowing this is important as it can save typing. In this tutorial, however, we generally spell out each of the argument names, for clarity.

Note the use of the assignment arrow `<-` to create new objects. Objects are entities that can be called to by name in R and can be renamed through additional assignements (e.g `y <- x` if y seems a more appropriate name). This is an efficient way of referring to large data objects or sets of commands.

Spatial Data in R

In any data analysis project, spatial or otherwise, it is important to have a strong understanding of the dataset before progressing. This section will therefore begin with a description of the input data. We will see how data can be loaded into R and exported to other formats, before going into more detail about the underlying structure of spatial data in R.

Loading spatial data in R

In most situations, the starting point of a spatial analysis project is to load in a datasets. These may originate from government agencies, remote sensing devices or 'volunteered geographical information' (Goodchild 2007). R is able to import a very wide range of spatial data formats thanks to its interface with the Geospatial Data Abstraction Library (GDAL), which is enabled by the package `rgdal`. Below we will load data from two spatial data formats: GPS eXchange (`.gpx`) and ESRI's Shapefile.

`readOGR` is in fact capable of loading dozens more file formats, so the focus is on the *method* rather than the specific formats. Let's start with a `.gpx` file, a tracklog recording a bicycle ride from Sheffield to Wakefield uploaded OpenStreetMap [3].

```
# download.file('http://www.openstreetmap.org/trace/1619756/data', destfile
# = 'data/gps-trace.gpx')
library(rgdal)  # load the gdal package
ogrListLayers(dsn = "data/gps-trace.gpx")
shf2lds <- readOGR(dsn = "data/gps-trace.gpx", layer = "tracks")  # load track
plot(shf2lds)
shf2lds.p <- readOGR(dsn = "data/gps-trace.gpx", layer = "track_points")  # load points
points(shf2lds.p[seq(1, 3000, 100), ])
```

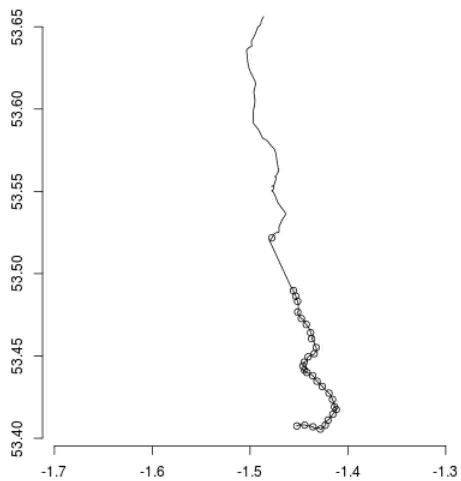

Figure 3: Leeds to Sheffield GPS data

In the code above we first used R to *download* a file from the internet, using the function `download.file` (note this has been *commented out* using the `#` symbol). The two essential arguments of this function are `url` (we could have typed `url =` before the link) and `destfile`, the destination file. As with any function, more optional arguments can be viewed by by typing `?download.file`.

When `rgdal` has successfully loaded, the next task is not to import the file directly, but to find out which *layers* are available to import, with `ogrListLayers`. The output from this command tells us that various layers are available, including `tracks` and `track_points`. These are imported into R's *workspace* using `readOGR`.

Finally, the basic `plot` function is used to visualize the newly imported objects, ensuring they make sense. In the second plot function (`points`), we add points for a subset of the object. To see how to add axes, enter `?axis`

Try discovering more about the function by typing `?readOGR`. The documentation explains that the `dsn =` argument is interpreted differently depending on the type of file used. In the above example, the `dsn` was set to as the name of the file. To load Shapefiles, by contrast, the *folder* containing the data is used:

```
lnd <- readOGR(dsn = "data/", "london_sport")
```

Here, the files reside in a folder entitled `data`, which is in R's current working directory (you can check this using `getwd()`). If the files were stored in the working directory, one would use `dsn = "."` instead. Again, it may be wise to plot the data that results, to ensure that it has worked correctly. Now that the data has been loaded into R's own `sp` format, try interrogating and plotting it, using functions such as `summary` and `plot`.

The london_sport file contains data pertaining to the percentage of people within each London Borough who regularly undertake physical activity and also the 2001 population of each Borough.

The size of spatial datasets in R

Any datasets that have been read into R's *workspace*, which constitutes all objects that can be accessed by name and can be listed using the `ls()` function, can be saved in R's own data storage file type `.RData`. Spatial datasets can get quite large and this can cause problems on computers by consuming all available memory (RAM) or hard disk space. It is therefore wise to understand roughly how large spatial objects are, providing insight into how long certain functions will take to run.

In the absence of prior knowledge, which of the two objects loaded in the previous section would one expect to be larger? One could hypothesize that the London dataset would be larger based on its greater spatial extent, but how much larger? The answer in R is found in the function `object.size`:

```
object.size(shf2lds)

## 107464 bytes

object.size(lnd)

## 125544 bytes
```

In fact, the objects have similar sizes: the GPS dataset is surprisingly large. To see why, we can find out how many *vertices* (points connected by lines) are contained in each dataset:

```
shf2lds.f <- fortify(shf2lds)
nrow(shf2lds.f)

## [1] 6085


lnd.f <- fortify(lnd)

## Regions defined for each Polygons

nrow(lnd.f)

## [1] 1102
```

In the above block of code we performed two functions for each object: 1) *flatten* the dataset so that each vertice is allocated a unique row 2) use `nrow` to count the result.

It is clear that the GPS data has almost 6 times the number of vertices compared to the London data, explaining its large size. Yet when plotted, the GPS data does not seem more detailed, implying that some of the vertices in the object are not needed for visualisation at the scale of the object's *bounding box*.

Simplifying geometries

In many cases the spatial data we have are too detailed for effective data visualisation. Simplifcation can help to make a graphic more readable and less cluttered. Within the 'rgeos' package it is possible to use the `gSimplify` function to simplify spatial R objects:

```
library(rgeos)
shf2lds.simple <- gSimplify(shf2lds, tol = 0.001)
(object.size(shf2lds.simple)/object.size(shf2lds))[1]
```

```
## [1] 0.04608
```

```
plot(shf2lds.simple)
plot(shf2lds, col = "red", add = T)
```

In the above block of code, `gSimplify` is given the object `shf2lds` and the `tol` argument of 0.001 (much larger tolerance values may be needed, for data that is *projected*). Next, we divide the size of the simplified object by the original (note the use of the `/` symbol). The output of `0.04...` tells us that the new object is only around 4% of its original size. We can see how this has happened by again counting the number of vertices. This time we use the `coordinates` and `nrow` functions together:

```
nrow(coordinates(shf2lds.simple)[[1]][[1]])
```

```
## [1] 44
```

The syntax of the double square brackets will seem strange, providing a taster of how R 'sees' spatial data. Do not worry about this for now. Of interest here is that the number of vertices has shrunk, from over 6,000 to only 44, without losing much information about the shape of the line. To test this, try plotting the original and simplified tracks on your computer: when visualized using the `plot` function, object `shf2lds.simple` retains the overall shape of the line and is virtually indistinguishable from the original object.

This example is rather contrived because even the larger object `shf2lds` is only a tenth of a megabyte, negligible compared with the gigabytes of memory available to modern computers. However, it underlines a wider point: for visualizing *small scale* maps, spatial data *geometries* can often be simplified to reduce processing time and use of memory.

Saving and exporting spatial objects

A typical R workflow involves loading the data, processing/analysing the data and finally exporting the data in a new form. `writeOGR`, the logical counterpart of `readOGR` is ideal for this task. This is performed using the following command (in this case we are exporting to an ESRI Shapefile):

```
shf2lds.simple <- SpatialLinesDataFrame(shf2lds.simple, data.frame(row.names = "0",
    a = 1))
writeOGR(shf2lds.simple, layer = "shf2lds", dsn = "data/", driver = "ESRI Shapefile")
```

In the above code, the object was first converted into a spatial dataframe class required by the `writeOGR` command, before being exported as a shapefile entitled shf2lds. Unlike with `readOGR`, the driver must be specified, in this case with "ESRI Shapefile" [4]. The simplified GPS data are now available to other GIS programs for further analysis. Alternatively, `save(shf2lds.simple, file = "data/shf2lds.RData")` will save the object in R's own spatial data format, which is described in the next section.

The structure of spatial data in R

Spatial datasets in R are saved in their own format, defined as `Spatial...` classes within the `sp` package. For this reason, `sp` is the basic spatial package in R, upon which the others depend. Spatial classes range from the basic `Spatial` class to the complex `SpatialPolygonsDataFrame`: the `Spatial` class contains only two required *slots* [5]:

```
getSlots("Spatial")
```

```
##        bbox proj4string
##    "matrix"       "CRS"
```

This tells us that `Spatial` objects must contain a bounding box (`bbox`) and a coordinate reference system (CRS) accessed via the function `proj4string`. Further details on these can be found by typing `?bbox` and `?proj4string`. All other spatial classes in R build on this foundation of a bounding box and a projection system (which is set automatically to `NA` if it is not known). However, more complex classes contain more slots, some of which are lists which contain additional lists. To find out the slots of `shf2lds.simple`, for example, we would first ascertain its class and then use the `getSlots` command:

```
class(shf2lds.simple)  # identify the object's class
```

```
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
```

```
getSlots("SpatialLinesDataFrame")  # find the associated slots
```

```
##         data        lines         bbox  proj4string
## "data.frame"       "list"     "matrix"        "CRS"
```

The same principles apply to all spatial classes including `Spatial* Points`, `Polygons Grids` and `Pixels` as well as associated `*DataFrame` classes. For more information on this, see the `sp` documentation: `?Spatial`.

Manipulating spatial data

Coordinate reference systems

As mentioned in the previous section, all `Spatial` objects in R are allocated a coordinate reference system (CRS). The CRS of any spatial object can be found using the command `proj4string`. In some cases the CRS is not known: in this case the result will simply be `NA`. To discover the CRS of the `lnd` object for example, type the following:

```
proj4string(lnd)
```

```
## [1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +ellps=airy "
```

The output may seem cryptic, but is in fact highly informative: `lnd` has *projected* coordinates, based on the *Transverse Mercator* system (hence `"+proj=tmerc"` in the output) and its origin is at latitude 49N, -2E.

If we *know* that the CRS is incorrectly specified, it can be re-set. In this case, for example we know that `lnd` actually has a CRS of OSGB1936. Knowing also that the code for this is 27700, it can be updated as follows:

```
proj4string(lnd) <- CRS("+init=epsg:27700")
proj4string(lnd)
```

```
## [1] "+init=epsg:27700 +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +datum=O
```

The CRS has now been updated - note that the key details are all the same as before. Note: this method should **never** be used as an attempt to *reproject* data from one CRS to another.

Reprojecting data

Transforming the coordinates of spatial data from one CRS to another (reprojection) is a common task in GIS. This is because data from national sources are generally provided in *projected* coordinates (the location on the cartesian coordinates of a map) whereas data from GPSs and the internet are generally provided in *geographic* coordinates, with latitude and longitude measured in degrees to locate points on the surface of the globe.

Reprojecting data in R is quite simple: all you need is a spatial object with a known CRS and knowledge of the CRS you wish to transform it to. To illustrate why that is necessary, try to plot the objects `lnd` and `shf2lnd.simple` on the same map:

```
combined <- rbind(fortify(shf2lds.simple)[, 1:2], fortify(lnd)[, 1:2])
plot(combined)
```

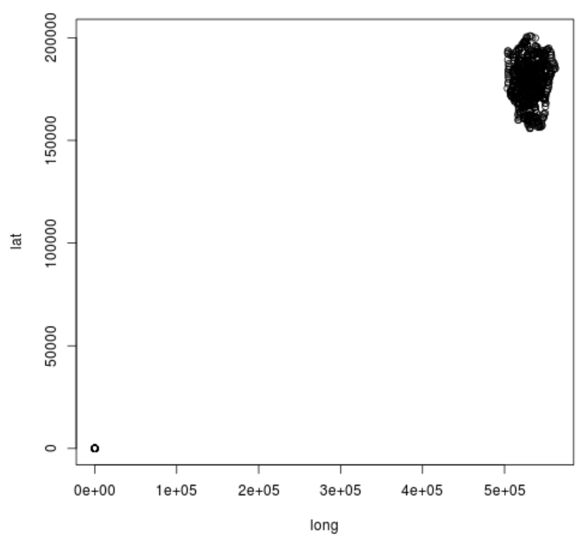

Figure 4: Plot of spatial objects with different CRS

In the above code we first extracted the coordinates of the vertices of each line and polygon using `fortify` and then plotted them using `plot`. The image shows why reprojection is necessary: the `.gpx` data are on a totally different scale than the shapefile of London. Hence the tiny dot at the bottom left of the graph. We will now reproject the data, allowing `lnd` and `shf2lds.simple` to be usefully plotted on the same graphic:

```
lnd.wgs84 <- spTransform(lnd, CRSobj = CRS("+init=epsg:4326"))
```

The above code created a new object,`lnd.wgs84`, that contains the same geometries as the original but in a new CRS using the `spTransform` function. The `CRS` argument was set to `"+init=epsg:4326"`, which represents the WGS84 CRS via an EPSG code [6]. Now `lnd` has been reprojected we can plot it next to the GPS data:

```
combined <- rbind(fortify(shf2lds.simple)[, 1:2], fortify(lnd.wgs84)[, 1:2])
plot(combined)
```

Although the plot of the reprojected data is squashed because the axis scales are not fixed and distorted (*geographic* coordinates such as WGS84 distort space close to the poles), but at least the relative position and shape of both objects can now be seen (making visualisations attractive is covered in the next major section). The presence of the dotted line in the top left of the plot confirms our assumption that the GPS data is from around Sheffield, which is northwest of London.

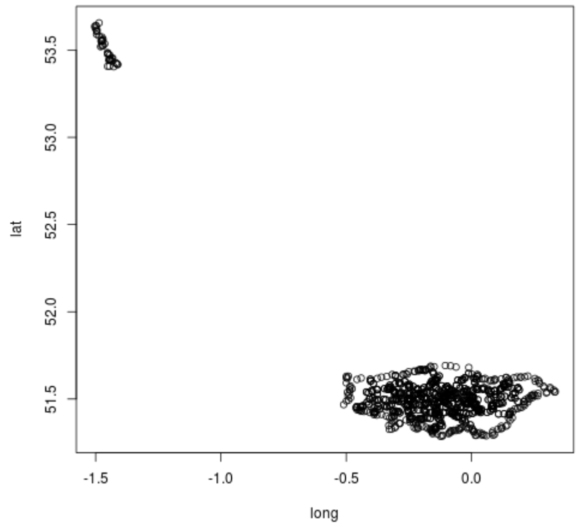

Figure 5: Plot of spatial objects sharing the same CRS

Attribute joins

London Boroughs are official administrative zones so we can easily join a range of other datasets to the polygons in the `lnd` object. We will use the example of crime data to illustrate this data availability, which is stored in the `data` folder available from this project's github page.

```
load("data/crimeAg.Rdata")  # load the crime dataset from an R dataset
```

After the dataset has been explored (e.g. using the `summary` and `head` functions) to ensure compatibility, it can be joined to `lnd`. We will use the the `join` function in the `plyr` package but the `merge` function could equally be used (remember to type `library(plyr)` if needed).

`join` requires all joining variables to have the same name, which has already been done [7].

```
lnd@data <- join(lnd@data, crimeAg)
```

Take a look at the `lnd@data` object. You should see new variables added, meaning the attribute join was successful.

Spatial joins

A spatial join, like attribute joins, is used to transfer information from one dataset to another. There is a clearly defined direction to spatial joins, with the *target layer* receiving information from another spatial layer based on the proximity of elements from both layers to each other. There are three broad types of spatial join: one-to-one, many-to-one and one-to-many. We will focus only the former two as the third type is rarely used.

One-to-one spatial joins

One-to-one spatial joins are by far the easiest to understand and compute because they simply involve the transfer of attributes in one layer to another, based on location. A one-to-one join is depicted in figure 5 below, and can performed using the same technique as described in the section on spatial aggregation.

Figure 6: Illustration of a one-to-one spatial join

Many-to-one spatial joins

Many-to-one spatial joins involve taking a spatial layer with many elements and allocating the attributes associated with these elements to relatively few elements in the target spatial layer. A common type of many-to-one spatial join is the allocation of data collected at many point sources unevenly scattered over space to polygons representing administrative boundaries, as represented in Fig. x.

```
lnd.stations <- readOGR("data/", "lnd-stns", p4s = "+init=epsg:27700")

## OGR data source with driver: ESRI Shapefile
## Source: "data/", layer: "lnd-stns"
## with 2532 features and 6 fields
## Feature type: wkbPoint with 2 dimensions

plot(lnd)
plot(lnd.stations[round(runif(500, 1, nrow(lnd.stations))), ], add = T)
```

The above code reads in a `SpatialPointsDataFrame` consisting of 2532 transport nodes in and surrounding London and then plots a random sample of 500 of these over the previously loaded borough level administrative boundaries. The reason for plotting a sample of the points rather than all of them is that the boundary data becomes difficult to see if all of the points are plotted. It is also useful to see and practice sampling techniques in practice; try to plot only the first 500 points, rather than a random selection, and spot the difference.

The most obvious issue with the point data from the perspective of a spatial join with the borough data we have is that many of the points in the dataset are in fact located outside the region of interest. Thus, the first stage in the analysis is to filter the point data such that only those that lie within London's administrative zones are selected. This in itself is a kind of spatial join, and can be accomplished with the following code.

```
proj4string(lnd) <- proj4string(lnd.stations)
lnd.stations <- lnd.stations[lnd, ]  # select only points within lnd
plot(lnd.stations)  # check the result
```

The station points now clearly follow the form of the `lnd` shape, indicating that the procedure worked. Let's review the code that allowed this to happen: the first line ensured that the CRS associated with each layer is *exactly* the

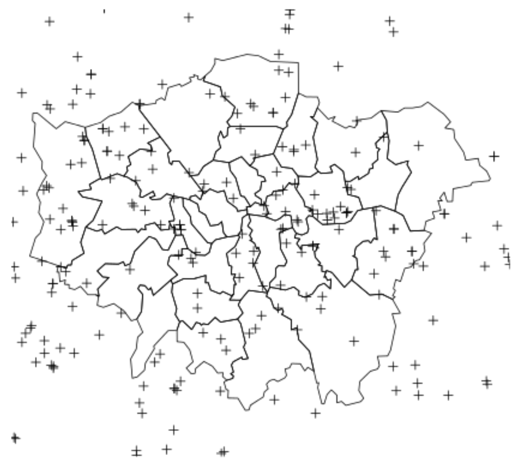

Figure 7: Input data for a spatial join

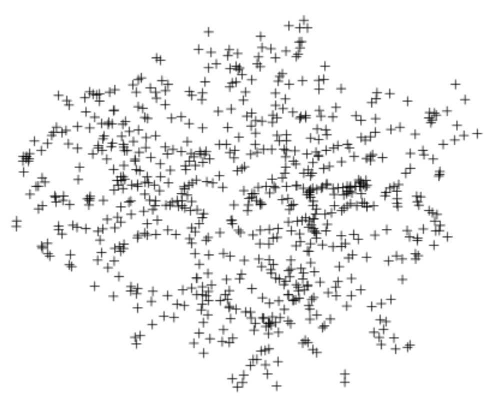

Figure 8: A spatial subset of the points

same: this step should not be required in most cases, but it is worth knowing about. Of course, if the coordinate systems are *actually* different in each layer, the function `spTransform` will be needed to make them compatible. This procedure is discussed in section !!!. In this case, only the name was slightly different hence direct alteration of the CRS name via the function `proj4string`.

The second line of code is where power of R's sp package becomes clear: all that was needed was to place another spatial object in the row index of the points (`[lnd, ]`) and R automatically understood that a subset based on location should be produced. This line of code is an example of R's 'terseness' - only a single line of code is needed to perform what is in fact quite a complex operation.

Spatial aggregation

Now that only stations which *intersect* with the `lnd` polygon have been selected, the next stage is to extract information about the points within each zone. This many-to-one spatial join is also known as *spatial aggregation*. To do this there are a couple of approaches: one using the `sp` package and the other using `rgeos` (see Bivand et al. 2013, 5.3).

As with the *spatial subest* method described above, the developers of R have been very clever in their implementation of spatial aggregation methods. To minimise typing and ensure consistency with R's base functions, `sp` extends the capabilities of the `aggregate` function to automatically detect whether the user is asking for a spatial or a non-spatial aggregation (they are, in essence, the same thing - we recommend learning about the non-spatial use of `aggregate` in R for comparison).

Continuing with the example of station points in London polygons, let us use the spatial extension of `aggregate` to count how many points are in each borough:

```
lndStC <- aggregate(lnd.stations, by = lnd, FUN = length)
summary(lndStC)
plot(lndStC)
```

As with the spatial subset function, the above code is extremely terse. The aggregate function here does three things: 1) identifies which stations are in which London borough; 2) uses this information to perform a function on the output, in this case `length`, which simply means "count" in this context; and 3) creates a new spatial object equivalent to `lnd` but with updated attribute data to reflect the results of the spatial aggregation. The results, with a legend and colours added, are presented in Figure 8 below. The code used below involves a number of steps that we have not yet covered. Copy these to create the map and they will be explained in Section 3 of the tutorial.

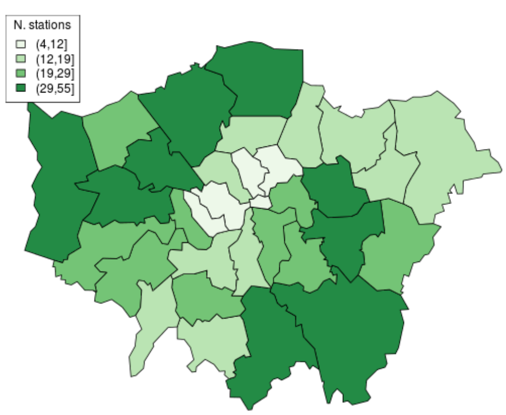

Figure 9: Number of stations in London boroughs

As with any spatial attribute data stored as an `sp` object, we can look at the attributes of the point data using the `@` symbol:

```
head(lnd.stations@data, n = 2)
```

```
##    CODE          LEGEND FILE_NAME NUMBER                  NAME MICE
## 91 5520 Railway Station  gb_south  17607       Belmont Station   19
## 92 5520 Railway Station  gb_south  17608 Woodmansterne Station    5
```

In this case we have three potentially interesting variables: "LEGEND", telling us what the point is, "NAME", and "MICE", which represents the number of mice sightings reported by the public at that point (this is a fictional variable!!). To illustrate the power of the `aggregate` function, let us use it to find the average number of mice spotted in transport points in each London borough, and the standard deviation:

```
lndAvMice <- aggregate(lnd.stations["MICE"], by = lnd, FUN = mean)
summary(lndAvMice)
lndSdMice <- aggregate(lnd.stations["MICE"], by = lnd, FUN = sd)
summary(lndSdMice)
```

In the above code, `aggregate` was used to create entirely new spatial objects that are exactly the same as `lnd`, except with new attribute data. To add the mean mice count to the original object, the following code can be used:

```
lnd$av.mice <- lndAvMice$MICE
```

The above code creates a new variable in the `lnd@data` object entitled "av.mice" and populates it with desired values. Thus `Spatial` objects can behave in the same way as data.frames when refering to attribute variables.

Summary

To summarise this section, we have taken a look inside R's representation of spatial data, learned how to manipulate these datasets in terms of CRS transformations and attribute data and finally explored spatial joins and aggregation.

Fundamentals of Spatial Data Visualisation

Good maps depend on sound analysis and can have an enormous impact on the understanding and communication of results. It has never been easier to produce a map. The underlying data required are available in unprecedented volumes and the technological capabilities of transforming them into compelling maps and graphics are increasingly sophisticated and straightforward to use. Data and software, however, only offer the starting points of good spatial data visualisation since they need to be refined and calibrated by the researchers seeking to communicate their findings. In this section we will run through the features of a good map. It is worth noting that not all good maps and graphics contain all the features below – they should simply be seen as suggestions rather than firm principles.

Effective map making is hard process – as Krygier and Wood (2011) put it "there is a lot to see, think about, and do"" (p6). It often comes at the end of a period of intense data analysis and perhaps when the priority is to get a paper finished or results published and can therefore be rushed as a result. The beauty of R (and other scripting languages) is the ability to save code and simply re-run it with different data. Colours, map adornments and other parameters can therefore be quickly applied, so it is well worth creating a template script that adheres to best practice.

We have selected ggplot2 as our package of choice for the bulk of our maps and spatial data visualisations because it has a number of these elements at its core. The "gg"" in its slightly odd name stands for "Grammar of Graphics"", which is a set of rules developed by Leland Wilkinson (2005) in a book of the same name. Grammar in the context of graphics works in much the same way as it does in language - it provides a structure. The structure is informed by both human perception and also mathematics to ensure that the resulting visualisations are both technically sound and comprehensible. By creating ggplot2, Hadley Wickham, implemented these rules as well as developing ways in which plots can be built up in layers (see Wickham, 2010). This layering component is especially useful in the context of spatial data since it is conceptually the same as map layers in Geographical Information Systems (GIS).

First load the libraries required for this section:

```
library(rgdal)
library(ggplot2)
library(gridExtra)
```

Set your working directory as before:

```
setwd("C:/Users/Uname/Desktop/sdvwR")
```

For this section we are going to use a map of the world to demonstrate some of the cartographic principles as they are introduced. The world map used is available from the Natural Earth website. Because these are already saved in the data folder, we can proceed to load the data.

```
wrld <- readOGR("data/", "ne_110m_admin_0_countries")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "data/", layer: "ne_110m_admin_0_countries"
## with 177 features and 63 fields
## Feature type: wkbPolygon with 2 dimensions
```

```
plot(wrld)
```

Figure 10: A Basic Map of the World

To see the first ten rows of attribute information assocuiated with each of the country boundaries type the following:

```
head(wrld@data)[, 1:5]
```

```
##   scalerank      featurecla labelrank           sovereignt sov_a3
## 0         1 Admin-0 country         3          Afghanistan    AFG
## 1         1 Admin-0 country         3               Angola    AGO
## 2         1 Admin-0 country         6              Albania    ALB
## 3         1 Admin-0 country         4 United Arab Emirates    ARE
## 4         1 Admin-0 country         2            Argentina    ARG
## 5         1 Admin-0 country         6              Armenia    ARM
```

You can see there are a lot of columns associated with this file. Although we will keep all of them, we are only really interested in the population estimate ("pop_est") field. Before progressing it is is worth reprojecting the data in order that the population data can be seen better. The coordinate reference system of the wrld shapefile is currently WGS84. This is the common latitude and longitude format that all spatial software packages understand. From a cartographic perspective the standard plots of this projection, of the kind produced above, are not suitable since they heavily distort the shapes of those countries further from the equator. Instead the Robinson projection provides a good compromise between areal distortion and shape preservation. We therefore project it as follows.

```
wrld.rob <- spTransform(wrld, CRS("+proj=robin"))
plot(wrld.rob)
```

Figure 11: The Robinson Projection

`+proj=robin` refers to the Robinson prjection. You will have spotted from the plot that the countries in the world map are much better proportioned.

We now need to `fortify` this spatial data to convert it into a format that ggplot2 understands, we also use `merge` to re-attach the attribute data that is lost in the fortify operation.

```
wrld.rob.f <- fortify(wrld.rob, region = "sov_a3")

## Loading required package: rgeos
## rgeos version: 0.3-2, (SVN revision 413M)
##  GEOS runtime version: 3.3.8-CAPI-1.7.8
##  Polygon checking: TRUE


wrld.pop.f <- merge(wrld.rob.f, wrld.rob@data, by.x = "id", by.y = "sov_a3")
```

The code below produces a map coloured by the population variable. It demonstrates the sophistication of ggplot2 by first stringing together a series of plot commands and assigning them to a single R object called `map`. If you type `map` into the command line, R will then execute the code and generate the plot. By simple specifing our `fill` variable within the `aes()` part of the code and then using the `geom_polygon()` command ggplot2 will fill colour the countries using a default colour pallette and auto-generated key. As will be shown in the next section these defaults can be easily altered to produce different looking maps.

```
map <- ggplot(wrld.pop.f, aes(long, lat, group = group, fill = pop_est)) + geom_polygon() +
    coord_equal() + labs(x = "Longitude", y = "Latitude", fill = "World Population") +
    ggtitle("World Population")

map
```

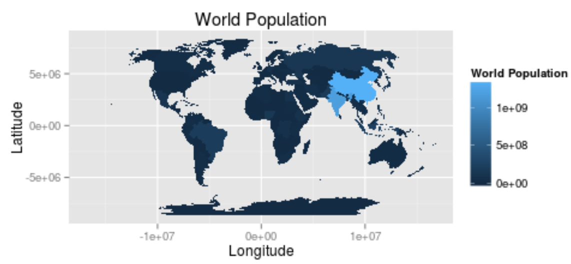

Figure 12: World Population Map

Colour and other aesthetics

Colour has an enormous impact on how people will percieve a graphic. Readers of a map come to it with a range of pre-conceptions about how the world looks.

Choropleth Maps

ggplot2 knows the different between continuous and categorical (nominal) data and will automatically assign the appropriate colour palettes when producing choropleth maps such as the one above. The default colour palettes are generally a good place to start but users may wish to vary them for a whole host of reasons, such as the need to print in black and white. The `scale_fill_` family of commands facilitate such customisation. For categorical data `scale_fill_manual()` is a useful command:

```
# Produce a map of continents
map.cont <- ggplot(wrld.pop.f, aes(long, lat, group = group, fill = continent)) +
    geom_polygon() + coord_equal() + labs(x = "Longitude", y = "Latitude", fill = "World Population") +
    ggtitle("World Continents")

# To see the default colours
map.cont
```

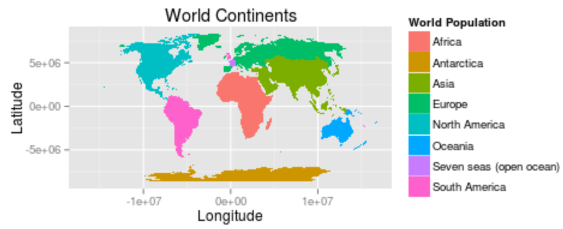

Figure 13: A Map of the Continents Using Default Colours

```
# To change these
map.cont + scale_fill_manual(values = c("yellow", "red", "purple", "white",
    "orange", "blue", "green", "black"))
```

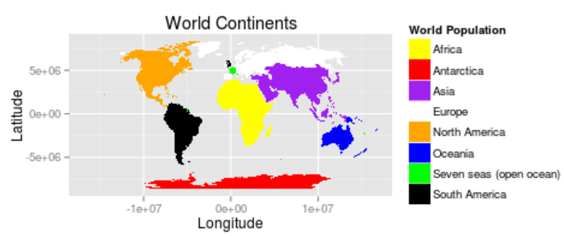

Figure 14: A Map of the Continents Using Default Colours

Whilst, `scale_fill_continuous()` works with continuous datasets:

```
# note the use of the 'map' object created earler

map + scale_fill_continuous(low = "white", high = "black")
```

Figure 15: Black and White Population Map

It is well worth looking at the *Color Brewer* palettes developed by Cynthia Brewer. These are designed to be colour blind safe and perceptually uniform such that no one colour jumps out more than any others. This latter characteristic is important when trying to produce impartial maps. R has a package that contains the colour palettes and these can be easily utlised by ggplot2.

```
library(RColorBrewer)

# look at the help documents to see the palettes available. Also visit
# http://colorbrewer2.org/ for more information
`?`(RColorBrewer)

# note the use of the scale_fill_gradientn() function rather than
# scale_fill_continuous() used above

map + scale_fill_gradientn(colours = brewer.pal(7, "YlGn"))
```

In addition to altering the colour scale used to represent continuous data it may also be desirable to adjust the breaks at which the colour transitions occur. There are many ways to select both the optimum number of breaks (i.e colour transtions) and the locations in the dataset at which they occur. The `classINT` package contains many ways to automatically create these breaks. We use the **`grid.arrange`** function from the gridExtra package to display the maps side by side.

```
library(classInt)

# Specify how many breaks you want - generally this should be fewer than 7.

nbrks <- 6

# Here quantiles are used to identify the breaks (note that we are using the
# original 'wrld.rob' object and not the 'wrld.rob@data£pop_est.f'). USe the
# help files to see the full range of options.
brks <- classIntervals(wrld.rob@data$pop_est, n = nbrks, style = "quantile")

print(brks)
```

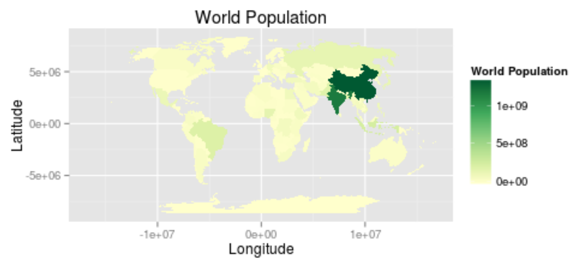

Figure 16: World Map with Yellow Green Colour Brewer Palette

```
## style: quantile
##        [-99,1790208)    [1790208,4579439)    [4579439,9035536)
##                   30                   29                   29
##   [9035536,16639804)  [16639804,40784057) [40784057,1.339e+09]
##                   30                   29                   30
```

```
# Now the breaks can be easily inserted into the code above for a range of
# colour palettes
YlGn <- map + scale_fill_gradientn(colours = brewer.pal(nbrks, "YlGn"), breaks = c(brks$brks))

PuBu <- map + scale_fill_gradientn(colours = brewer.pal(nbrks, "PuBu"), breaks = c(brks$brks))

grid.arrange(YlGn, PuBu, ncol = 2)
```

If you are not happy with the automatic methods of specifying breaks it can also be done manually:

```
nbrks <- 4
brks <- c(1e+08, 2.5e+08, 5e+07, 1e+09)

map + scale_fill_gradientn(colours = brewer.pal(nbrks, "PuBu"), breaks = c(brks))
```

There are many other ways to specify and alter the colours in ggplot2 and these are outlined in the help documentation. There are also many examples online.

If the map's purpose is to clearly communicate data then it is often advisable to conform to conventions so as not to disorientate readers to ensure they can focus on the key messages contained in the data. A good example of this is the use of blue for bodies of water and green for landmasses. The code example below generates two plots with our wrld.pop.f object. The first colours the land blue and the sea (in this case the background to the map) green and the second is more conventional.

```
map2 <- ggplot(wrld.pop.f, aes(long, lat, group = group)) + coord_equal()
```

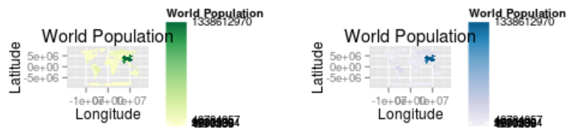

Figure 17: Different Colour Palettes with Bespoke Breaks

Figure 18: unnamed-chunk-5

```
blue <- map2 + geom_polygon(fill = "light blue") + theme(panel.background = element_rect(fill = "dark gree

green <- map2 + geom_polygon(fill = "dark green") + theme(panel.background = element_rect(fill = "light bl

grid.arrange(blue, green, ncol = 2)
```

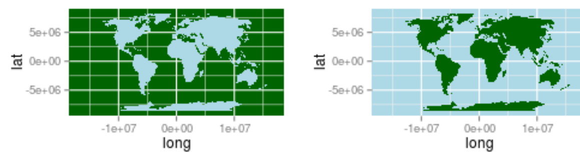

Figure 19: Conforming to Colour Convention

Experimenting with line colour and line widths

In addition to conforming to colour conventions, line colour and width offer important parameters, which are often overlooked tools for increasing the legibility of a graphic. As the code below demonstrates, it is possible to adjust line colour through using the `colour` parameter and the line width using the `lwd` parameter. The impact of different line widths will vary depending on your screen size and resolution. If you save the plot to pdf (or an image) then the size at which you do this will also affect the line widths.

```
map3 <- map2 + theme(panel.background = element_rect(fill = "light blue"))

yellow <- map3 + geom_polygon(fill = "dark green", colour = "yellow")

black <- map3 + geom_polygon(fill = "dark green", colour = "black")

thin <- map3 + geom_polygon(fill = "dark green", colour = "black", lwd = 0.1)

thick <- map3 + geom_polygon(fill = "dark green", colour = "black", lwd = 1.5)

grid.arrange(yellow, black, thick, thin, ncol = 2)
```

There are other parameters such as layer transparency (use the `alpha` parameter for this) that can be applied to all aspects of the plot - both points, lines and polygons. Space does not permit their full exploration here but more information is available from the many online examples and the ggplot2 package documentation.

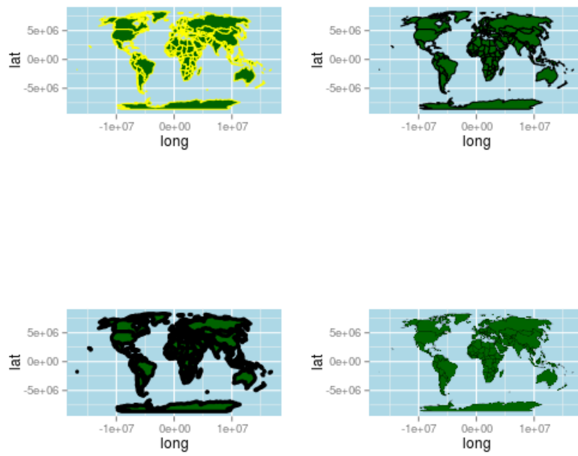

Figure 20: The Impact of Line Width

Map Adornments and Annotations

Map adornments and annotations are essential to orientate the viewer and provide context; they include graticules, north arrows, scale bars and data attribution. Not all are required on a single map, indeed it is often best that they are used sparingly to avoid unecessary clutter (Monkhouse and Wilkinson, 1971). With ggplot2 many of these are added automatically but they can be customised.

North arrow

In the maps created so far, we have defined the *aesthetics* of the map in the foundation function ggplot. The result of this is that all subsequent layers are expected to have the same variables and essentially contain data with the same dimensions as original dataset. But what if we want to add a new layer from a completely different dataset, e.g. to add an arrow? To do this, we must not add any arguments to the `ggplot` function, only adding data sources one layer at a time:

Here we create an empty plot, meaning that each new layer must be given its own dataset. While more code is needed in this example, it enables much greater flexibility with regards to what can be included in new layer contents. Another possibility is to use `geom_segment()` to add a rudimentary arrow (see `?geom_segment` for refinements):

```
library(grid)  # needed for arrow
ggplot() + geom_polygon(data = wrld.pop.f, aes(long, lat, group = group, fill = pop_est)) +
    geom_line(aes(x = c(-1.3e+07, -1.3e+07), y = c(0, 5e+06)), arrow = arrow()) +
    coord_fixed()  # correct aspect ratio
```

Scale bar

ggplot2's scale bar capabilities are perhaps the least satisfactory element of the package. For this example we use the `geom_line()` function to draw a line of approximately 1km in length using the `lnd.f` object containing the London Boroughs discussed in Section 2. The reason for this is that it is in a projected coordinate system - British National Grid - so each map unit is worth 1m. In the case of the world map the distances at the equator in terms of degrees east to west are very different from those further north or south. Any line drawn using the the simple approach below would therefore be inaccurate. For maps covering large areas - such as the entire world - leaving the axis labels on will enable them to act as a graticule which will indicate distance.

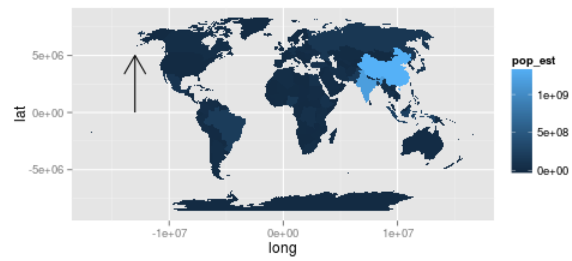

Figure 21: North Arrow Example

```
load("data/lnd.f.RData")
ggplot() + geom_polygon(data = lnd.f, aes(long, lat, group = group)) + geom_line(aes(x = c(505000,
    515000), y = c(158000, 158000)), lwd = 2) + annotate("text", label = "10km",
    x = 510000, y = 160000) + coord_fixed()
```

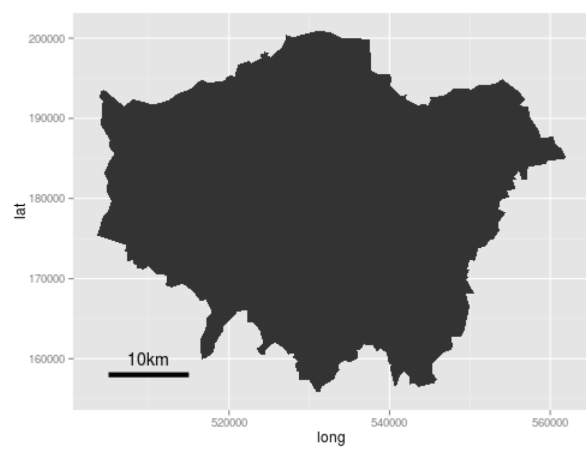

Figure 22: Scale Bar Example

Legends

Legends are added automatically but can be customised in a number of ways. A few examples are included below with more details avaialble in the `ggplot2` documentation.

```
# Position
map + theme(legend.position = "top")
```

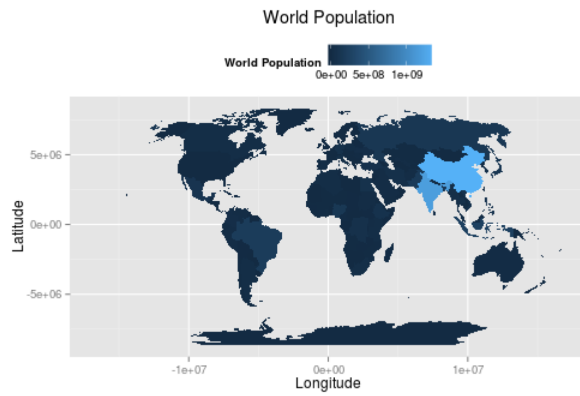

Figure 23: Formatting the Legend

```
# Title
map + theme(legend.title = element_text(colour = "Red", size = 16, face = "bold"))
```

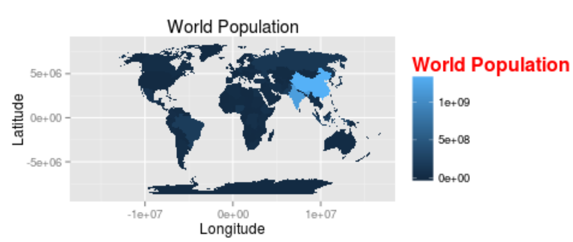

Figure 24: Formatting the Legend

```
# Label Font Size and Colour
map + theme(legend.text = element_text(colour = "blue", size = 16, face = "italic"))
```

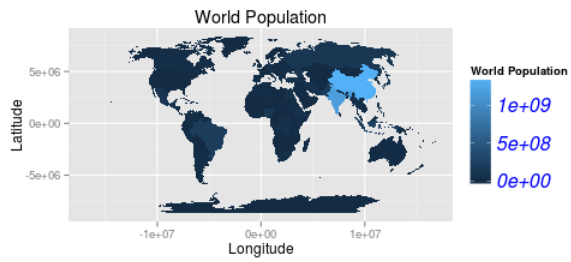

Figure 25: Formatting the Legend

```
# Border and background box
map + theme(legend.background = element_rect(fill = "gray90", size = 0.5, linetype = "dotted"))
```

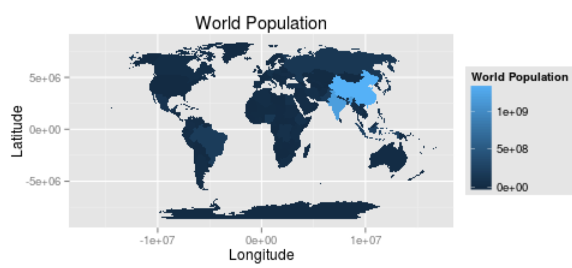

Figure 26: Formatting the Legend

Adding Basemaps To Your Plots

The development of the ggmap package has enabled the simple use of online mapping services such as Google Maps and OpenStreetMap for base maps. Using image tiles from these services spatial data can be placed in context as users can easily orientate themselves to streets and landmarks.

For this example we are going to use the shapefile of London sports participation introduced in Section 2. The data were originally projected to British National Grid (BNG) which is not compatible with the online map services used in the following examples. It therefore needs reprojecting - a step we completed earlier. The reprojected file can be loaded as follows:

```
load("data/lnd.wgs84.RData")
```

The first job is to calculate the bounding box (bb for short) of the `lnd.wgs84` object to identify the geographic extent of the map. This information is used to request the appropriate map tiles from the map service of our choice. This process is conceptually the same as the size of your web browser or smartphone screen when using Google maps for navigation. The first line of code in the snippet below retrieves the bounding box and the two that follow add 5% so there is a little space around the edges of the data to be plotted.

```
b <- bbox(lnd.wgs84)
b[1, ] <- (b[1, ] - mean(b[1, ])) * 1.05 + mean(b[1, ])
b[2, ] <- (b[2, ] - mean(b[2, ])) * 1.05 + mean(b[2, ])
# scale longitude and latitude (increase bb by 5% for plot) replace 1.05
# with 1.xx for an xx% increase in the plot size
```

This is then fed into the `get_map` function as the location parameter. The syntax below contains 2 functions. `ggmap` is required to produce the plot and provides the base map data.

```
library(ggmap)

lnd.b1 <- ggmap(get_map(location = b))

## Warning: bounding box given to google - spatial extent only approximate.
```

`ggmap` follows the same syntax structures as ggplot2 and so can easily be integrated with the other examples included here. First `fortify` the `lnd.wgs84` object and then merge with the required attribute data.

```
lnd.wgs84.f <- fortify(lnd.wgs84, region = "ons_label")
lnd.wgs84.f <- merge(lnd.wgs84.f, lnd.wgs84@data, by.x = "id", by.y = "ons_label")
```

We can now overlay this on our base map using the `geom_polygon()` function.

```
lnd.b1 + geom_polygon(data = lnd.wgs84.f, aes(x = long, y = lat, group = group,
    fill = Partic_Per), alpha = 0.5)
```

The resulting map looks reasonable, but it would be improved with a simpler base map in black and white. A design firm called *stamen* provide the tiles we need and they can be brought into the plot with the `get_map` function:

```
lnd.b2 <- ggmap(get_map(location = b, source = "stamen", maptype = "toner",
    crop = T))  #note the addition of the maptype parameter.
```

We can then produce the plot as before.

```
lnd.b2 + geom_polygon(data = lnd.wgs84.f, aes(x = long, y = lat, group = group,
    fill = Partic_Per), alpha = 0.5)
```

Finally, if we want to increase the detail of the base map, `get_map` has a zoom parameter.

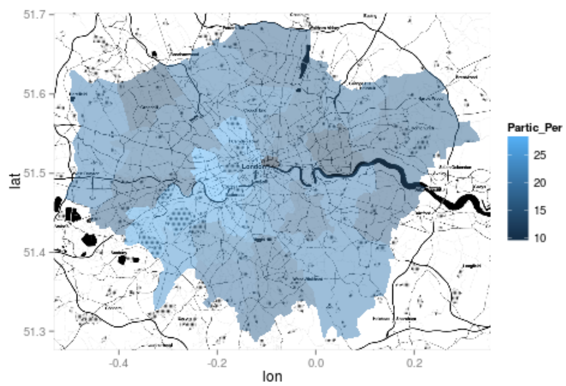

Figure 27: Using the Stamen Toner basemap

```
lnd.b3 <- ggmap(get_map(location = b, source = "stamen", maptype = "toner",
    crop = T, zoom = 11))

lnd.b3 + geom_polygon(data = lnd.wgs84.f, aes(x = long, y = lat, group = group,
    fill = Partic_Per), alpha = 0.5)
```

Spatial polygons are not the only data types compatible with `ggmap` - you can use any plot type and set of parameters available in `ggplot2`, making it an ideal companion package for spatial data visualisation.

Summary

There are an almost infinite number of different combinations colours, adornments and line widths that could be applied to a map, so take inspiration from maps and graphics you have seen and liked. The process is an iterative one, it will take multiple attempts to get right. Show your map to friends and colleagues - all will have an opinion but don't be afraid to stand by the decisions you have taken. To give your maps a final polish you may wish to export them as a pdf using the `ggsave()` function and importing them into a vector graphics package such as Adobe Illustrator or Inkscape.

The beauty of producing maps in a programming environment as opposed to the GUI offered by the majority of GIS software packages lies in the fact that each line of code can be easily adapted to a different dataset. Users can therefore create a series of scripts that act as templates and simply call them when required. This saves a huge amount of time and has the added advantage that all outputs will have a consistent style and thus offer more professional looking publications.

A Final Example

Here we present a final example that draws upon the many advanced concepts discussed in this tutorial to produce a map of 18th Century Shipping flows. The data have been obtained from the CLIWOC project and they represent a sample of digitised ships' logs from the 18th Century. We are using a very small sample of the the full dataset, which is available from here: http://pendientedemigracion.ucm.es/info/cliwoc/. The example has been chosen to demonstrate a range of capabilities within ggplot2 and the ways in which they can be applied to produce high-quality maps with only a few lines of code. We end by showing how the maps can be animated to chart the routes over time and the ability of R to produce many maps very quickly.

As always, the first step is to load in the required packages and datasets. Here we are using the png package to load in a series of map annotations. These have been created in image editing software and will add a historic feel to the map. We are also loading in a World boundary shapefile and the shipping data itself.

```
library(rgdal)
library(ggplot2)
library(png)
wrld <- readOGR("data/", "ne_110m_admin_0_countries")

## OGR data source with driver: ESRI Shapefile
## Source: "data/", layer: "ne_110m_admin_0_countries"
## with 177 features and 63 fields
## Feature type: wkbPolygon with 2 dimensions

btitle <- readPNG("figure/brit_titles.png")
compass <- readPNG("figure/windrose.png")
bdata <- read.csv("data/british_shipping_example.csv")
```

If you look at the first few lines in the `bdata` object you will see there are 7 columns with each row representing a single point on the ships course. The year of the journey and the nationality of the ship are also included. The final 3 columns are identifiers that are used later to group the coordinate points together into the paths that ggplot2 plots.

We first specify some plot parameters that remove the axis labels.

```
xquiet <- scale_x_continuous("", breaks = NULL)
yquiet <- scale_y_continuous("", breaks = NULL)
quiet <- list(xquiet, yquiet)
```

The next step is to `fortify` the World coastlines and create the base plot. This sets the extents of the plot window and provides the blank canvas on which we will build up the layers. The first layer created is the wrld object; the code is wrapped in `c()` to prevent it from executing by simply storing it as the plot's parameters.

```
wrld.f <- fortify(wrld, region = "sov_a3")

## Loading required package: rgeos
## rgeos version: 0.3-2, (SVN revision 413M)
##  GEOS runtime version: 3.3.8-CAPI-1.7.8
##  Polygon checking: TRUE

base <- ggplot(wrld.f, aes(x = long, y = lat))
wrld <- c(geom_polygon(aes(group = group), size = 0.1, colour = "black", fill = "#D6BF86",
    data = wrld.f, alpha = 1))
```

To see the result of this simply type:

```
base + wrld
```

The code snipped below creates the plot layer containing the the shipping routes. The `geom_path()` function is used to string together the coordinates into the routes. You can see within the `aes()` component we have specified long and lat plus pasted together the trp and `group.regroup` variables to identify the unique paths.

```
route <- c(geom_path(aes(long, lat, group = paste(bdata$trp, bdata$group.regroup,
    sep = ".")), colour = "#0F3B5F", size = 0.2, data = bdata, alpha = 0.5,
    lineend = "round"))
```

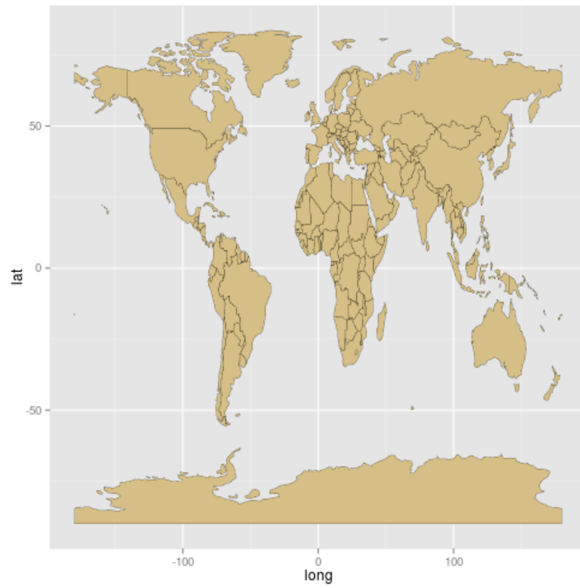

Figure 28: World Map

We now have all we need to generate the final plot by building the layers together with the + sign as shown in the code below. The first 3 arguments are the plot layers, and the parameters within `theme()` are changing the background colour to sea blue. `annotation_raster()` plots the png map adornments loaded in earlier- this requires the bounding box of each image to be specified. In this case we use latitude and longitude (in WGS84) and we can use these paramrters to change the png's position and also its size. The final two arguments fix the aspect ratio of the plot and remove the axis labels.

```
base + route + wrld + theme(panel.background = element_rect(fill = "#BAC4B9",
    colour = "black")) + annotation_raster(btitle, xmin = 30, xmax = 140, ymin = 51,
    ymax = 87) + annotation_raster(compass, xmin = 65, xmax = 105, ymin = 25,
    ymax = 65) + coord_equal() + quiet
```

In the plot example we have chosen the colours carefully to give the appearance of a historic map. An alternative approach could be to use a satellite image as a base map. It is possible to use the readPNG function to import NASA's "Blue Marble" image for this purpose. Given that the route information is the same projection as the image it is very straightforward to set the image extent to span -180 to 180 degrees and -90 to 90 degrees and have it align with the shipping data. Producing the plot is accomplished using the code below. This offers a good example of where functionality designed without spatial data in mind can be harnessed for the purposes of producing interesting maps. Once you have produced the plot, alter the code to recolour the shipping routes to make them appear more clearly against the blue marble background.

```
earth <- readPNG("figure/earth_raster.png")

base + annotation_raster(earth, xmin = -180, xmax = 180, ymin = -90, ymax = 90) +
    route + theme(panel.background = element_rect(fill = "#BAC4B9", colour = "black")) +
    annotation_raster(btitle, xmin = 30, xmax = 140, ymin = 51, ymax = 87) +
    annotation_raster(compass, xmin = 65, xmax = 105, ymin = 25, ymax = 65) +
    coord_equal() + quiet
```

Animating your plots

R is not designed to produce animated graphics and as such it has very few functions that enable straightforward animation. To produce animated graphics users can use a loop to plot and then export a series of images that can

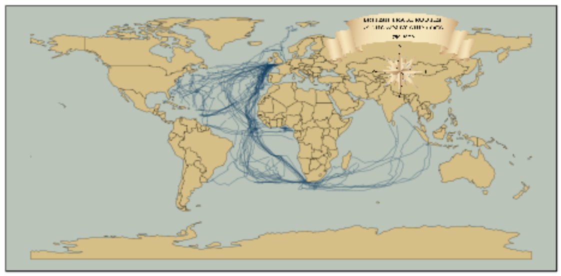

Figure 29: World Shipping

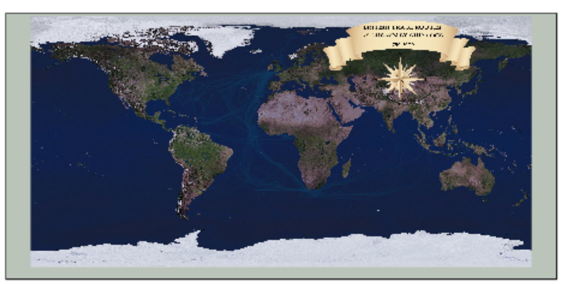

Figure 30: World Shipping with raster background

then be stitched together into a video. There are two approaches to this; the first is to create a loop that fills a folder with the desired images and then utilise third party software to stitch the images together, whilst the second uses R's own animation package. The latter option still requires the installation of an additional software package called ImageMagick but it has the benefit of creating the animation for you within R and faciliting the export to a range of formats, not least HTML and GIF. Here we demonstrate the use of the package to produce an HTML animation of the shipping tracks completed in each year of the bdata object. The code snippet below appears extremely dense, but it only contains a few addtions to the plot code utilised above.

First load the package:

```
library(animation)
```

Then clear any previous animation. Obviously the first time you run this it is unnecessary, but it is a good habit to get into.

```
ani.record(reset = TRUE)
```

We then initiate the "for loop". In this case we are using the `unique()` function to list the unique years within the `bdata` object. The loop will take the first year, in this case 1791, and assign it to the object `i`. The code inside the {} brackets will then run with `i=1791`. You will spot that `i` is used in a number of places- first to subset the data when creating the route plot and then as the title in the `ggtitle()` function. We need to force ggplot to create the graphic within the loop so the entire plot call is wrapped in the `print()` function. Once the plot is called `ani.record()` is used to save the plot still and `dev.off()` used to clear the plot window ready for the next iteration. `i` is then assigned the next year in the list and the code runs again until all years are plotted.

```
for (i in unique(bdata$year)) {
    route <- c(geom_path(aes(long, lat, group = paste(trp, group.regroup, sep = "."))),
        colour = "#0F3B5F", size = 0.2, data = bdata[which(bdata$year == i),
            ], alpha = 0.5, lineend = "round"))
    print(base + route + wrld + theme(panel.background = element_rect(fill = "#BAC4B9",
        colour = "black")) + annotation_raster(btitle, xmin = 30, xmax = 140,
        ymin = 51, ymax = 87) + annotation_raster(compass, xmin = 65, xmax = 105,
        ymin = 25, ymax = 65) + coord_equal() + quiet + ggtitle(i))
    ani.record()
    dev.off()
}
```

The final step in the process is to save the animation to HTML and view it in your web browser. `ani.replay()` retrieves the animation stored by the `ani.record()` function and `outdir=getwd()` ensures the final file is stored in your working directory.

```
saveHTML(ani.replay(), img.name = "record_plot", outdir = getwd())
```

You will note that there is something a little odd about the order in which the years appear. This can be solved by an additional step before the loop code above. Have a think then add this in and then regenerate the animation.

Recap and Conclusions

This tutorial has covered a large number of techniques and approaches for the preparation, analysis and visualisation of spatial data in R. Whilst it only covers the tip of the iceberg in terms of R's capabilities, it does lay the foundations to the use of the multitude of other spatial data packages available. These can be discovered online and through the help documentation and other tutorials provided by the R community. By utilising the data visualisation techniques and examples of best practice we have covered it is hoped that you will be able to communicate your results in a compelling and effective way without the need for the repetitive "pointing and clicking" required of many GIS packages; you can now tweak colours and other aspects of the plots without the need to start from scratch each time an iterative improvement is required. As the R community grows so will its range of applications and available packages so there will be many exciting opportunities ahead to improve on what is presented here.

References

Bivand, R., & Gebhardt, A. (2000). Implementing functions for spatial statistical analysis using the language. Journal of Geographical Systems, 2(3), 307–317.

Bivand, R. S., Pebesma, E. J., & Rubio, V. G. (2008). Applied spatial data: analysis with R. Springer.

Burrough, P. A. & McDonnell, R. A. (1998). Principals of Geographic Information Systems (revised edition). Clarendon Press, Oxford.

Goodchild, M. F. (2007). Citizens as sensors: the world of volunteered geography. GeoJournal, 69(4), 211–221.

Harris, R. (2012). A Short Introduction to R. social-statistics.org.

Kabacoff, R. (2011). R in Action. Manning Publications Co.

Krygier, J. Wood, D. 2011. Making Maps: A Visual Guide to Map Design for GIS (2nd Ed.). New York: The Guildford Press.

Longley, P., Goodchild, M. F., Maguire, D. J., & Rhind, D. W. (2005). Geographic information systems and science. John Wiley & Sons.

Monkhouse, F.J. and Wilkinson, H. R. 1973. Maps and Diagrams Their Compilation and Construction (3rd Edition, reprinted with revisions). London: Methuen & Co Ltd.

Ramsey, P., & Dubovsky, D. (2013). Geospatial Software's Open Future. GeoInformatics, 16(4).

Sherman, G. (2008). Desktop GIS: Mapping the Planet with Open Source Tools. Pragmatic Bookshelf.

Torfs and Brauer (2012). A (very) short Introduction to R. The Comprehensive R Archive Network.

Venables, W. N., Smith, D. M., & Team, R. D. C. (2013). An introduction to R. The Comprehensive R Archive Network (CRAN). Retrieved from http://cran.ma.imperial.ac.uk/doc/manuals/r-devel/R-intro.pdf .

Wickham, H. (2009). ggplot2: elegant graphics for data analysis. Springer.

Wickham, H. 2010. A Layered Grammar of Graphics. American Statistical Association, Institute of Mathematics Statistics and Interface Foundation of North America Journal of Computational and Graphical Statistics. 19, 1: 3-28

Endnotes

1. "What kind of a name is R?" common question. R's name originates from the creators of R, Ross Ihaka and Robert Gentleman. R is an open source implementation of the statistical programming language S, so its name is also a play on words that makes implicit reference to this.

2. R is notoriously difficult to search for on major search engines, as it is such a common letter with many other uses beyond the name of a statistical programming language. This should not be a deterrent, as R has a wealth of excellent online resources. To overcome the issue, you can either be more specific with the search term (e.g. "R spatial statistics") or use an R specific search engine such as rseek.org. You can also search of online help *from within R* using the command `RSiteSearch`. E.g. `RSiteSearch("spatial statistics")`. Experiment and see which you prefer!

3. More information about this ride, and a video from it, can be found on robinlovelace.net.

4. A complete list of drivers for importing and exporting spatial data can be displayed by typing `getGDALDriverNames()`.

5. Slots are elements found 'inside' classes of the S4 object system. While the sub-elements of S3 objects such as `data.frame` are referred to using the `$` symbol, the slots of S4 objects are identified using `@`. Thus, the variable `x` of dataframe `df` can be referred to with `df$x`. In the same way, the data associated with a polygon layer such as `lnd` can be accessed with `lnd@data`. Note that `lnd@data` is itself a dataframe, so can be further specified, e.g. with `lnd@data$name`. For more on spatial data classes, see Bivand et al. (2013).

6. EPSG stands for “European Petroleum Survey Group”, but this is not really worth knowing as the organisation is now defunct (www.epsg.org/). The important thing is that EPSG codes provide a unified way to refer to a wide range of coordinate systems, as each CRS has its own epsg code. These can be found at the website spatialreference.org. To see how this website can be useful, try searching for “osgb”, for example to find the epsg code for the British National Grid.

7. To see how the `crimeAg` dataset was created, please refer to the “Creating-maps-in-R” tutorial (Cheshire and Lovelace, 2014) hosted on GitHub. The file “intro-spatial-rl.pdf” contains this information, in the section on “Downloading addtional data”.

```
source("chapter.R")  # convert chapter to tex
```