Spatial data visualisation with R

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

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

February 4, 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. This means that instead of interacting with the program by clicking on different parts of the screen via a *graphical user interface* (GUI), users type commands for the operations they wish to complete. This seems a little daunting at first but 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. In addition, R facilitates truly transparent and reproducible research by removing the need for expensive software licenses and encouraging documentation of code. It is possible for anyone with the R installed to reproduce all the steps used by others. With the RStudio program it is even possible to include 'live' R code in text documents.

In R what 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 many programmers to improve R over time and add an incredible number of extensions to its capabilities. There are now more than 4000 official add-on *packages* for R, allowing it to tackle almost any numerical problem. If there is a useful function that R cannot currently perform, there is a good 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 in recent years is the addition of a wide variety of spatial analysis and visualisation tools (Bivand et al. 2013). The latter will be the focus of this chapter.

Why R for spatial data visualisation?

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. The map of Facebook friendships produced by Paul Butler, for example, is iconic in this regard and has reached a global audience (Figure 1). This shows linkages between friends as lines across the curved surface of the Earth (using the `geosphere` package). 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 such details cannot be overstated. They can be the difference between a stunning graphic and an impenetrable mess.

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

The map helped inspire the R community to produce more ambitious graphics, a process fuelled by increased demand for data visualisation and the development of packages that augment R's preinstalled 'base graphics'. Thus R has become a key tool for analysis and visualisation used by the likes of Twitter, the New York Times and Google. Thousands of consultants, design houses and journalists also rely on R - it is no longer merely the preserve of academic research and many graduate jobs now list R as a desirable skill.

Finally, it is worth noting a few key differences between R and traditional GIS software such as QGIS. 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). Indeed, it is this flexibility, illustrated by the custom map of shipping routes in Section 4 of this chapter, that makes R so attractive. Another benefit of R compared with traditional approaches to GIS is that it facilitates *transparency* of research, a feature that we will be using to great effect in this chapter: all of the results presented in the subsequent sections can be reproduced (and modified) at home, as described in the next section.

A practical primer on spatial data in R

This section briefly introduces some of the key steps to get started with R. Like the rest of the chapter, it has a large *practical* element, including R code to run on your own computer. For users completely new to R, we recommend beginning with an 'introduction to R' type tutorial, such as Torf and Brauer (2012) or the frequently updated tutorial "Introduction to Visualising Spatial Data in R" (Lovelace and Cheshire 2014). Both are available free online.

The first stage is to obtain and load the data used in the examples. In this case, all the data has been uploaded to an online repository that provides a detailed tutorial to accompany this Chapter: github.com/geocomPP/sdvwR. Upon visiting this page you will see many files. One of these is 'sdv-tutorial.pdf', which offers a comprehensive introductory tutorial - we recommend new R users refer to this to accompany the chapter. To download the data that will allow the examples to be reproduced, click on the "Download ZIP" button on the right, and unpack this to a sensible place on your computer (for example, the Desktop). This should result in a folder called 'sdvwR-master' being created. Explore this and try opening some of the files, especially those from the sub-folder entitled "data", the input datsets.

In any data analysis project, spatial or otherwise, it is important to have a strong understanding of the dataset before progressing. We will see how data can be loaded into R (ready for the next section) and exported to other formats.

Loading spatial data in R

R is able to import a very wide range of spatial data formats thanks to its interface with the Geospatial Data Abstraction Library (GDAL). The `rgdal` package makes this possible: install and load it by by entering `install.packages("rgdal")` followed by `library(rgdal)`, on separate lines. The former only needs to be typed once, as it saves the data from the internet. The latter must be typed for each new R session that requires the package.

The world map we use is available from the Natural Earth website and a slightly modified version of it (entitled "world") is loaded using the following code. A common problem preventing the data being loaded correctly is that R

is not in the correct *working directory*. Please refer to the online tutorial if this is an issue.

```
library(rgdal)  # load the package (needs to be installed)
wrld <- readOGR("data/", "world")
plot(wrld)
```

Figure 2: A Basic Map of the World

The above block of code loaded the rgdal library, created a new *object* called `wrld` and plotted this object to ensure it is as we expect. This operation should be fast on most computers because `wrld` is quite small. Spatial data can get very large indeed, however. It can therefore be useful to know the size spatial objects and simplify them when necessary. R makes this easy, as described in Section 2 of the tutorial that accompanies this Chapter. For now, let us continue with an even more important topic: how R 'sees' spatial data.

How R 'sees' spatial data

Spatial datasets in R are saved in their own format, defined as `Spatial` classes within the `sp` package (Bivand et al. 2013). This data class divides the spatial information into different *slots* so the attribute and geometry data are stored separately. This makes handling spatial data in R efficient. For more detail on this topic, see "The structure of spatial data in R" in the online tutorial. We will see in the next section that this complex data structure can be simplified in R using the `fortify` function.

For now, let us ask some basic questions about the `wrld` object, using functions that would apply to any spatial dataset in R, to gain an understanding of what we have loaded. How many rows of attribute data are there? This query can be answered using `nrow`:

```
nrow(wrld)
```

```
## [1] 175
```

What do the first 2 rows and 5 columns of attribute data contain? To answer this question, we need to refer to the `data` slot of the object using the `@` symbol and use square brackets to define the subset of the data to be displayed. In R, the rows are always referred to before the comma within the square brackets and the column numbers after. Try playing with the following line of code, for example by removing the square brackets entirely:

```
wrld@data[1:2, 1:5]
```

```
##   scalerank      featurecla labelrank  sovereignt sov_a3
## 0         1 Admin-0 country         3 Afghanistan    AFG
## 1         1 Admin-0 country         3      Angola    AGO
```

The output shows that the first country in the `wrld` object is Afghanistan. Now that we have a basic understanding of the attributes of the spatial dataset, and know where to look for more detailed information about spatial data in R via the online tutorial, it is time to move on to the topic of visualisation.

Fundamentals of Spatial Data Visualisation

Good maps depend on sound analysis and can have an enormous impact on the understanding and communication of results. Thanks to new data sources and programs such as R, it has never been easier to produce a map. Spatial datasets are now available in unprecedented volumes and tools for transforming them into compelling maps and graphics are becoming increasingly sophisticated accessible. Data and software, however, only offer the starting points of good spatial data visualisation. Graphics need to be refined and calibrated to best communicate the message contained in the data. This section describes the features of a good map. Not all good maps and graphics *must* contain all the features below: they should be seen as suggestions rather than firm principles.

Effective map making is difficult process, as Krygier and Wood (2011) put it: "there is a lot to see, think about, and do" (p6). Visualisation usually comes at the end of a period of data analysis and, perhaps when the priority is to finish an assignment, is tempting to rush. The beauty of R (and other scripting languages) is the ability to save code and re-run it. Colours, map adornments and other parameters can therefore be quickly applied, so it is worth spending time creating a template script that adheres to best practice.

We use *ggplot2* as the package of choice to produce most of the maps presented in this chapter because it easily facilitates good practice in data visualisation. The "gg" in its name stands for "Grammar of Graphics", a set of rules developed by Wilkinson (2005). 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 technically sound and comprehensible. By creating ggplot2 Hadley Wickham implemented these rules, including a syntax for building graphics in layers using the `+` symbol (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 conventional GIS.

First ensure that the necessary packages are installed and that R is in the correct working directory. Then load the ggplot2 package used in this section.

```
library(ggplot2)
```

We are going to use the previously loaded map of the world to demonstrate some of the cartographic principles as they are introduced. To establish the starting point, find the first 35 column names of the `wrld` object:

```
names(wrld@data)[1:35]
```

```
##  [1] "scalerank"  "featurecla" "labelrank"  "sovereignt" "sov_a3"
##  [6] "adm0_dif"   "level"      "type"       "admin"      "adm0_a3"
## [11] "geou_dif"   "geounit"    "gu_a3"      "su_dif"     "subunit"
## [16] "su_a3"      "brk_diff"   "name"       "name_long"  "brk_a3"
## [21] "brk_name"   "brk_group"  "abbrev"     "postal"     "formal_en"
## [26] "formal_fr"  "note_adm0"  "note_brk"   "name_sort"  "name_alt"
## [31] "mapcolor7"  "mapcolor8"  "mapcolor9"  "mapcolor13" "pop_est"
```

This shows many attribute columns associated with the `wrld` object. Although we will keep all of them, we are only really interested in population (`"pop_est"`). Typing `summary(wrld$pop_est)` provides basic descriptive statistics on population.

Before progressing, we will reproject the data to reduce distortion in the size of countries close to the North and South poles (at the top and bottom of the above plot). The coordinate reference system of the wrld shapefile is currently WGS84, the most common latitude and longitude format that all spatial software packages understand. From a cartographic perspective this projection this is not ideal. Instead, the Robinson projection provides a good compromise between areal distortion and shape preservation:

Figure 3: The Robinson Projection

```
wrld.rob <- spTransform(wrld, CRS("+proj=robin"))  #'+proj=robin' refers to the Robinson projection
plot(wrld.rob)
```

Now the countries in the world map are much better proportioned. The above plots use R's *base graphics*. The function `fortify` must be used to convert the spatial data it into a format that ggplot2 understands. Then the `merge` function is used to re-attach the attribute data lost during the fortify operation.

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

```
## Loading required package: rgeos
## rgeos version: 0.2-19, (SVN revision 394)
##  GEOS runtime version: 3.3.8-CAPI-1.7.8
##  Polygon checking: TRUE
```

```
# Use by.x and by.y arguments to specify the columns that match the two
# dataframes together:
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 syntax 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
specifying the `fill` variable within the `aes()` (short for 'aesthetics') argument, ggplot2 colours the countries using a default colour palette and automatically generates a legend. `geom_polygon()` tells ggplot2 to plot polygons. As will be shown in the next section these defaults can be easily altered to change a map's appearance.

```
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
```

Colour and other aesthetics

Colour has an enormous impact on how people will perceive a graphic. Adjusting a colour palette from yellow to red or from green to blue, for example, can alter the readers' response. In addition, the use of colour to highlight particular regions or de-emphasise others are important tricks in cartography that shouldn't be overlooked. Below we present a few examples of how to create high quality maps with R. For more information about the importance of different features of a map for its interpretation, see Monmonier (1996).

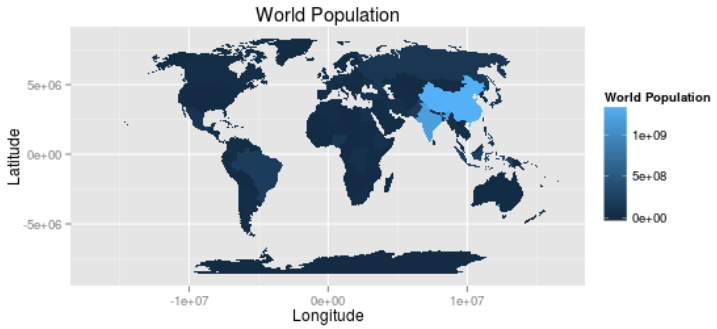

Figure 4: World Population Map

Choropleth Maps

ggplot2 knows the difference between continuous and categorical (nominal) variables and will automatically assign the appropriate colour palettes accordingly. The default colour palettes are generally a sensible 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 enable such customisation. For categorical data, `scale_fill_manual()` can be used:

```
# 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 Continents") +
    ggtitle("World Continents")

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

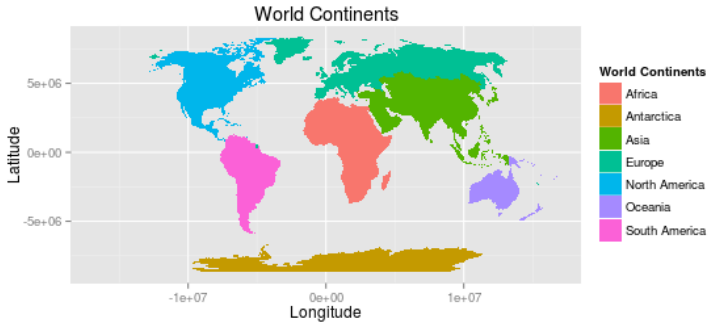

Figure 5: A Map of the Continents Using Default Colours

To change the colour scheme, we can set our own colours:

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

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")
```

It is well worth looking at the *Color Brewer* palettes developed by Cynthia Brewer (see http://colorbrewer2.org). 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 utilised by ggplot2.

```
library(RColorBrewer)
# look at the help documents to see the palettes available.
`?`(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 transitions) and the locations in the dataset at which they occur. This is important for the comprehension of a graphic since it alters the colours associated with each value. 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)
```

```
## Loading required package: class
## Loading required package: e1071
```

```
library(gridExtra)
```

```
## Loading required package: grid
```

```
# Specify how number of breaks - 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$pop_est.f' Use the
# help files (by typing ?classIntervals) to see the full range of options
brks <- classIntervals(wrld.rob@data$pop_est, n = nbrks, style = "quantile")

print(brks)

# 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 for 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))
```

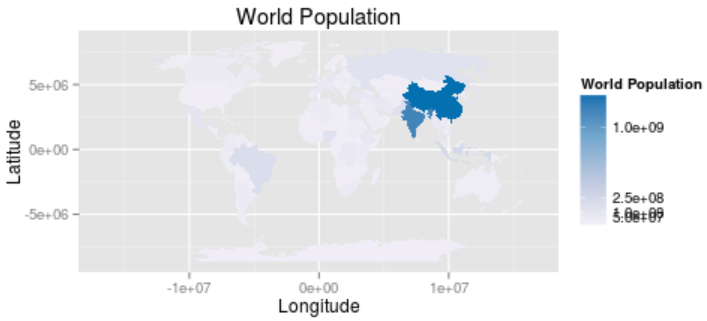

Figure 6: unnamed-chunk-6

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

If the map's purpose is to clearly communicate data then it is advisable to conform to widely used conventions. A good example of this is the use of blue for water and green for landmasses. The code example below generates two plots with the `wrld.pop.f` object. The first colours the land blue and the sea (in this case the background to the map) green. The second plot is more conventional.

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

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

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

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

Experimenting with line colour and widths

Line colour and width are important parameters too often overlooked for increasing the legibility of a graphic. The code below demonstrates it is possible to adjust these using the `colour` and `lwd` arguments. The impact of different line widths will vary depending on your screen size and resolution. If you save the plot to pdf (e.g. using the `ggsave` command), this will also affect the relative 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)
```

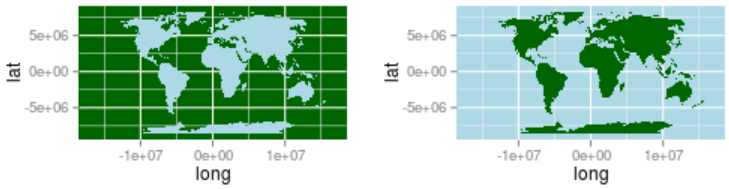

Figure 7: Conforming to Colour Convention

```
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 full exploration here but more information is available from the ggplot2 package documentation (see ggplot2.org).

Map Adornments and Annotations

Map adornments and annotations orientate the viewer and provide context. They include grids (also known as graticules), orientation 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 unnecessary clutter (Monkhouse and Wilkinson 1971). With ggplot2 scales and legends are provided by default, but they can be customised.

North arrow

In the maps created so far, we have defined the *aesthetics* (`aes`) of the map in the foundation function `ggplot()`. The result of this is that all subsequent layers are expected to have the same variables. But what if we want to add a new layer from a completely different dataset, for example to add a north 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)) +
```

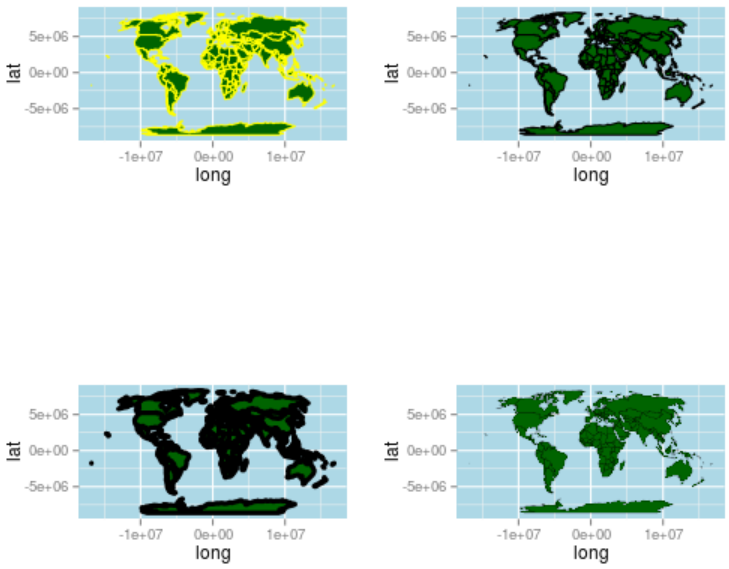

Figure 8: The Impact of Line Width

```
    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 advanced element of the package. This approach will only work if the spatial data are in a projected coordinate system to ensure there are no distortions as a result of the curvature of the earth. 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 to indicate distance. We therefore load in a file containing the geometry of London's Boroughs.

```
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()
```

Legends

Legends are added automatically but can be customised in a number of ways. They are an important adornment of any map since they describe what its colours mean. Try to select colour breaks that are easy to follow and avoid labeling the legend with values that go to a large number of significant figures. A few examples of legend customisation are included below by way of introduction, but there are many more examples available in the ggplot2 documentation.

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

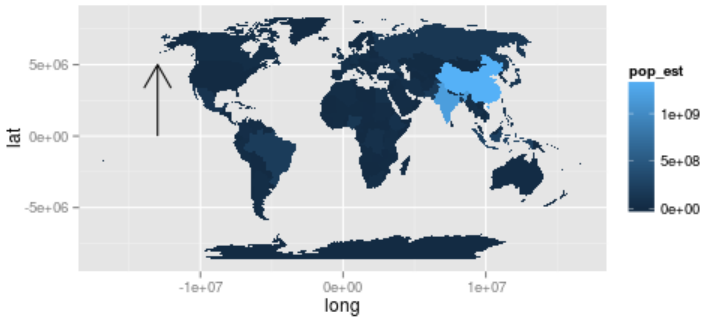

Figure 9: North Arrow Example

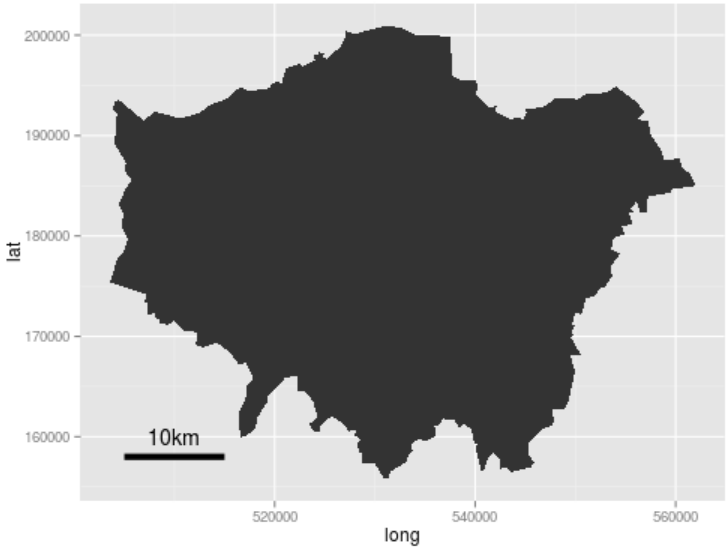

Figure 10: Scale Bar Example

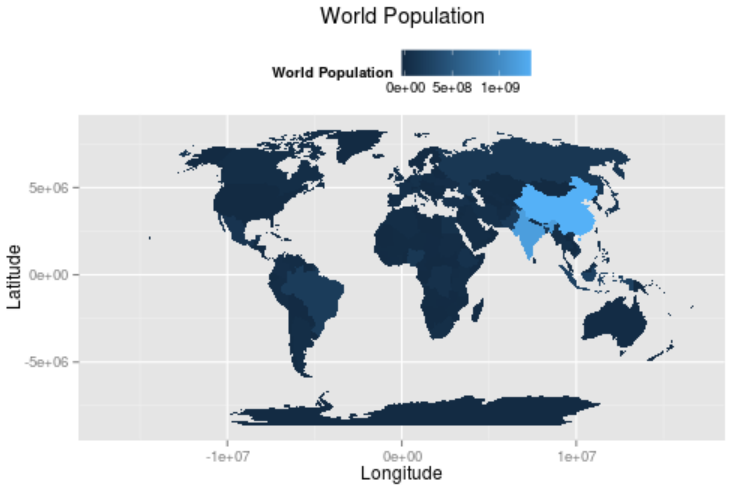

Figure 11: Formatting the Legend

As you can see, this added the legend in a new place. Many more options for customization are available, as highlighted in the examples below.

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

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

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

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 use data on London sports participation. 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 - a process 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)
```

This produces a much clearer map and enables readers to focus on the data rather than the basemap. 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.

A Final Example

Here we present a final example that draws upon the many advanced concepts discussed in this chapter 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
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.

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 ship's 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")
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 + coord_fixed()
```

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"))
```

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 parameters 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
```

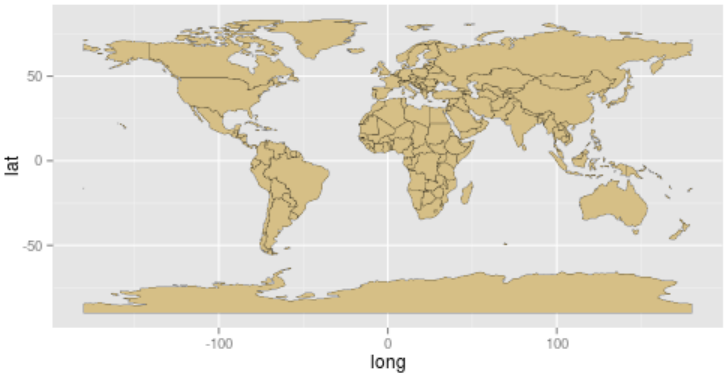

Figure 12: World Map

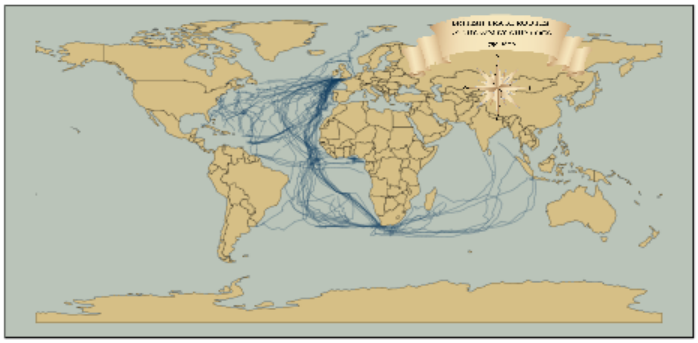

Figure 13: World Shipping

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
```

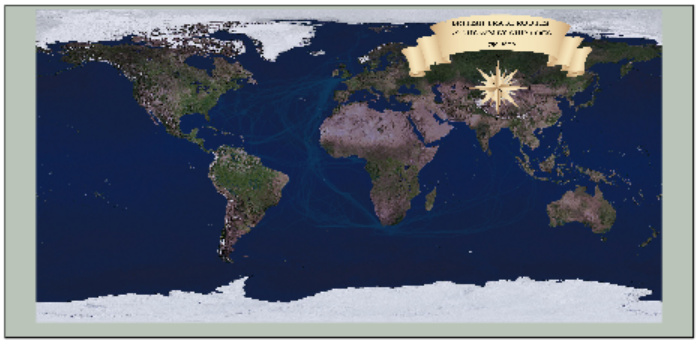

Figure 14: World Shipping with raster background

Conclusions

There are an almost infinite number of different combinations colours, adornments and line widths that could be applied to a map (or any other data visualisation) so do not feel constrained by the examples presented in this chapter. Take inspiration from maps and graphics you have seen and liked, and experiment. The process is iterative, probably taking multiple attempts to get right. Show your map to friends and peers for feedback before you publish them or use them in a report. To give your maps a final polish you may wish to export them as a pdf using `ggsave` function and then add additional customisations using 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 programs lies in the fact that each line of code can be easily adapted to a different purpose. Users can create a series

of scripts that act as templates and simply call them when required. This can save time in the long run and has the added advantage that all outputs will have a consistent style.

This chapter has covered a variety of techniques for the preparation and visualisation of spatial data in R. While this is only the tip of the iceberg in terms of R's spatial capabilities, the simple worked examples lay the foundations for further exploration of spatial data in R, using the multitude of spatial data packages available. These can be discovered online, through R's internal help (we recommend frequent use of R queries such as `?plot`) and other book chapters on the subject. It is hoped that the techniques and examples covered in this chapter will help communicate the results of spatial data analysis to the target audience in a compelling and effective way, without the need for the repetitive "pointing and clicking" described in the chapter's opening quote. As the R community grows, so will its range of applications and available packages. The supportive online communities surrounding large open source programs such as R are one of their greatest assets, so we recommend you become an active "open source" citizen rather than merely a passive consumer of new software (Ramsey & Dubovsky, 2013). As R continues its ascent a as a spatial analysis and data visualisation platform, the opportunities to benefit from it by creating compelling maps are only set to grow.

References

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

Bivand, R. S., Pebesma, E. J., & Rubio, V. G. (2013). Applied spatial data: analysis with R. Springer.

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

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

Lovelace, R. and Cheshire, J. (2014). Introduction to visualising spatial data in R. National Centre for Research Methods Working Paper. Updated pdf version available from github.com/Robinlovelace/Creating-maps-in-R.

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

Monmonier, M. 1996. How to Lie with Maps (2nd Ed.). Chicago: University of Chicago Press.

Ramsey, P., & Dubovsky, D. (2013). Geospatial Software's Open Future. GeoInformatics, 16(4). See also a talk by Paul Ramsey entitled "Being an open source citizen": blog.cleverelephant.ca/2013/10/being-open-source-citizen.

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


```
source("md2pdf.R")  # convert chapter to tex
```