Spatial data visualisation with R

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

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

May 2, 2014

Introduction

What is R?

R is a free and open source computer program for processing data. It runs on all major operating systems and 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. For new users this might seem a little daunting at first, however 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.

A key benefit is that commands sent to R can be stored and repeated from scripts, thus facilitating transparent and reproducible research by removing the need for both software licenses and encouraging documentation of code. Furthermore, access to R's source code and the provision of a framework for extensions has enabled many programmers to improve on the basic, or "base", R functionality. As a result, there are now more than 4000 official add-on *packages*, allowing R to tackle almost any numerical problem. If there is a useful function that R cannot currently perform, it is likely that someone is working on a solution. One area where extension of R's basic capabilities have 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 broadening functionality to reflect a growing user base across disciplines. It 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 pass 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 chart.

Arguably this map helped inspire the R community to produce more ambitious graphics, a process fuelled by an 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.

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

Finally, it is worth noting that there are a few key differences between R and traditional desktop GIS software. 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.

One example of this is the choice between R's base graphics or a dedicated graphics package such as ggplot2. The former option requires no additional packages and can provide very quick feedback about the nature of the dataset in question with the generic `plot` function. The ggplot2 option, by contrast, requires a new package to be loaded but opens up a very wide range of functions for visualising data, beyond the base graphics. ggplot2 also has sensible defaults for grid axes, legends and other adornments, allowing the user to create complex and beautiful graphics with minimal effort. We encourage users to try both but, following the focus on *visualisation*, have used ggplot2 for all but the first two plots presented in this chapter.

An innovative feature of this chapter is that *all* of the graphics presented in it are reproducible (see the next section for how). We encorage users not only to reproduce the graphics presented here but to play around with the code, taking advantage of the wide range of visual analysis uptions opened up by R. Indeed, it is this flexibility, illustrated by the custom map of shipping routes presented later in this chapter, that makes R an attractive visualisation solution.

All of the results presented in this chapter can be reproduced (and modified) by typing the short code snippets that are presented into R. Elsewhere in this book, these principles are extended in the context of reproducible geographic information science.

A practical primer on spatial data in R

This section introduces those steps required to get started with processing spatial data in R. The focuses of the chapter is on the visualisation of so-called vector data (common in socio-economic examples), however, R also provides functionality for the analysis and visualisation of raster data (see supporting materials). For users completely new to R, we would recommend beginning with an introductory tutorial, such as Torf and Brauer (2012) or "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 for the examples into R. These data have been uploaded into an online repository that also provides a detailed tutorial to accompany this Chapter: github.com/geocomPP/sdvwR. [1]

In any data analysis project, spatial or otherwise, it is important to have a strong understanding of the dataset before progressing. R is able to import a very wide range of spatial data formats by linking with the Geospatial Data Abstraction Library (GDAL). An interface to this library is contained in the rgdal package: install and load it by entering `install.packages("rgdal")` followed by `library(rgdal)`, on separate lines. The former only needs to be typed once to install the package, however, the latter must be run for each new R session that requires use of the functions contained within the package.

[1] To download the data that will allow the examples to be reproduced, click on the "Download ZIP" button on the right hand side of the page, and unzip this to a convenient place on your computer (for example, the Desktop). This should result in a folder called 'sdvwR-master' being created.

The world map that we use is available from the *Natural Earth* website and a slightly modified version of it (entitled "world") is loaded using the following code (see Figure 2).[2]

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

The above block of code loads the rgdal library, creates and then plots, in Figure 2, a new *object* called `wrld`. This operation should be fast on most computers because `wrld` has quite a small file size. Spatial data can however get very large as the number and complexity of zones increases. It can therefore be useful to keep track of the size of spatial objects and to simplify them when necessary prior to visualisation [3].

When spatial data are imported into R, they are saved in a `spatial` object class using the sp package (Bivand et al. 2013). The spatial data are divided into a series of different *slots*, storing the attribute and geometry data separately [4]. To view the slot names of an object you can use the function slotNames(), with the object name written within the brackets.

The contents of "slots" within spatial data objects can be accessed using the "`@`" symbol. In the example below, the first two rows of the data slot are displayed, and can be treated as a standard data frame.

```
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
```

Fundamentals of Spatial Data Visualisation

Good maps can have an enormous impact on understanding of spatial patterns, from initial exploratory data analysis, through to the communication of results. Graphics do, however, need to be refined and calibrated, and this section describes such considerations. It should be noted that not all good maps and graphics must contain all the features discussed: they should be seen as suggestions rather than firm principles.

Effective map making is a difficult process, as Krygier and Wood (2011) put it: "there is a lot to see, think about, and do" (p6). We use a ggplot2 as the package of choice to produce most of the maps presented in this chapter because it 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: providing structure to the presented material. The ggplot2 package was developed by Hadley Wickham, and includes 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 a conventional GIS.

In the following analysis, the previously loaded map of the world will be used to demonstrate a series of cartographic principles. This spatial object contains 35 columns of data, however, for our purposes, 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 [5]. The coordinate reference system of the world shapefile (named `wrld`) is WGS84, which is a very common latitude and longitude format. Without projecting the data when plotted this format distorts the size of countries close to the North and South poles (at the top and bottom of the above plot). Instead, the Robinson projection (see Figure 2) can be used to provide a better compromise between areal distortion and shape preservation. Changes of projection can be accomplished using `spTransform()`, with the projection required set with the CRS (coordinate reference system) parameter.

```
library(ggplot2)
wrld.rob <- spTransform(wrld, CRS("+proj=robin"))  #'+proj=robin' refers to the Robinson projection
plot(wrld.rob)
```

Figure 2: A Basic Map of the World in geographic (cartesian) coordinates (left) and the Robinson Projection (right).

Plotting the reprojected spatial object results in a world map that is much better proportioned, and as such, when adding detail to the representation, salient patterns will be presented more clearly to end users. The plots created in the examples thus far use R's base graphics capabilities. However, the remaining examples will use the ggplot2 package introduced above. This requires the data to be in a slightly different format to base R and should be converted using the `fortify` function. This step discards the attribute data associated with the spatial object and so this needs to be reattached using the `merge` function.

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

Now that the R object is in the correct format to be plotted with ggplot2, the code that follows produces a choropleth map coloured by the population variable. This demonstrates the syntax of ggplot2 by first linking 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 shown in Figure 3. 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 later, 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/1000000)) + geom_polygon() +
coord_equal() + labs(x = "Longitude", y = "Latitude", fill = "World Population") +
ggtitle("World Population")
map
```

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. For more information about the importance of different features of a map for its interpretation, see Monmonier (1996).

ggplot2 recognises the difference between continuous and categorical (nominal) variables and will automatically assign an appropriate colour palette accordingly (see Figure 4). The default colour palettes are a good place to start, but users can specify them, for example, if they needed to print a map in black and white. The `scale_fill_` family of commands enable such customisation. For categorical data, `scale_fill_manual()` can be used.

[2] A common problem preventing the data being loaded correctly is that R is not set with the correct working directory. For more information, please refer to the online tutorial hosted at github.com/geocomPP/sdvwR.

[3] R makes this easy, and is described in Section 2 of the tutorial that accompanies this Chapter (github.com/geocomPP/sdvwR).

[4] For more detail on this topic, see "The structure of spatial data in R" in the online tutorial.

[5] For more information on referencing systems, see the links in the supporting materials

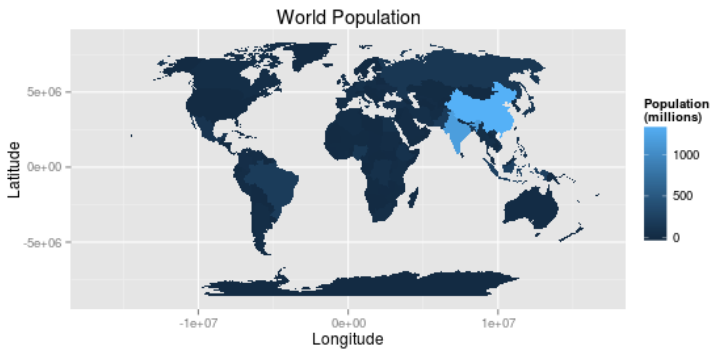

Figure 3: World Population Map

```
# 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
```

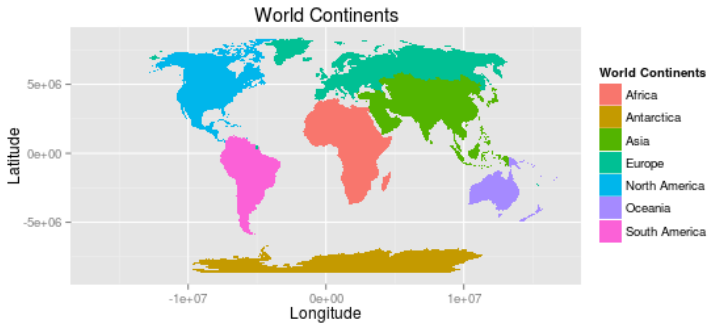

Figure 4: A Map of the Continents Using Default Colours

Colours can be specified manually, either using words, as illustrated in the example, or more flexibly, such as through the use of hexadecimal colour codes:

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

The command `scale_fill_continuous()` is used to set a continuum-based colour scheme:

```
# Note the use of the 'map' object created earler
map + scale_fill_continuous(low = "white", high = "black")
```

Choosing an appropriate colour palette is difficult and there are a variety of considerations such as the intended destination of a graphic (computer screen, print etc), the likely audience and the visual impairments such as colour

blindness. There is a large body of literature associated with colour perception and this forms the basis to 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 these colour palettes and they can be easily accessed by ggplot2.

```
library(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 pallete used to represent a continuous dataset, 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 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 a series of maps side by side, illustrating different break choices.

```
library(classInt)

library(gridExtra)



# 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)


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))
```

Line colour and width are also important parameters for enhancing the legibility of a graphic (see Figure 5). 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 screen size and resolution. Also, if you save the plot to pdf (e.g. using the `ggsave` command), this will also alter the relative line widths. As such, it is often useful to generate and check plots in the desired output format, and then adjust the code until these are appropriate.

```
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)
```

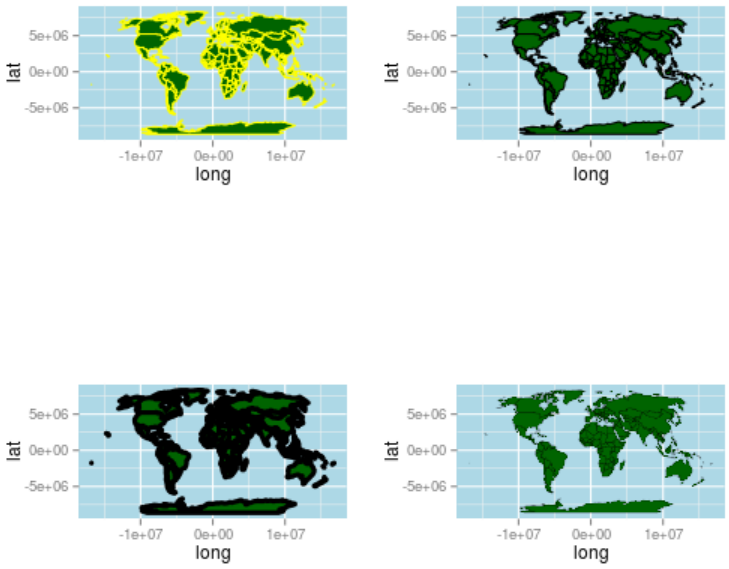

Figure 5: The Impact of Line Width

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 in the ggplot2 package documentation (see ggplot2.org).

Map Adornments

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.

The maps created so far have concerned a single dataset, however, it is possible to layer separate datasets together to create a single representation. This first requires an empty plot, meaning that each new layer must be defined with its own dataset, and as such, adjust the syntax a little from that presented previously. Although more code is needed, this does enable much greater flexibility with regards to what can be included as new layer content. 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
```

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. As such, the following example uses a shapefile pertaining to 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 attributes the colours reference. As a general rule, legends with values that go to a large number of significant figures should be avoided. The following code moves the legend from the default position to the top of the map.

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

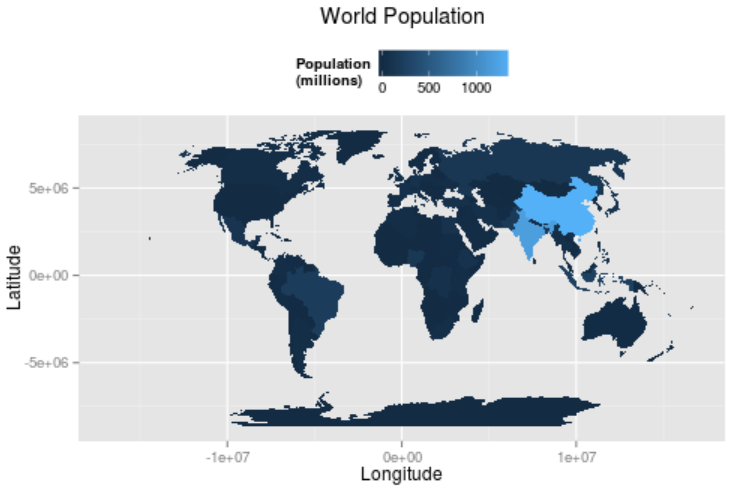

Figure 6: Formatting the Legend

Many more options are available, such as adding a title, adjusting the font size or colour, and controlling other aspects such as the map borders. These are illustrated by the following code.

```
# 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"))
```

Plotting over a base map

The ggmap package extends the ggplot2 package to integrate online mapping services such as Google Maps and OpenStreetMap (OSM) for base cartography. By using image tiles derived from these services, spatial data can be placed in context as users can easily orientate themselves to streets and landmarks. In the following examples, data on London sports participation is used. The data were originally projected in British National Grid (BNG) which pertains to a different referencing system than that used in the Google or OSM online map services. This is a common problem and one that is easily overcome using the reprojection function outlined above.

After importing the boundary data and reprojecting, a bounding box of the `lnd.wgs84` object was calculated to identify the geographic extent of the map. This information is used to request an appropriate base map from a selected map tile service. The first block of code in the snippet below retrieves the bounding box and then adds 5% so there is a little space around the edges of the data to be plotted. This is then fed into the `get_map` function as the location parameter. The code actually utlises two nested functions, `ggmap` and `get_map` which are required to produce the plot and provides the base map data. You will notice from the code snippet below that `ggmap` follows the same syntax structures as ggplot2 and so can easily be integrated with the other examples included here. The example data object contains spatial polygons but spatial points and lines can also 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

library(ggmap)

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



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 thegeom_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. A design firm called *stamen* provide the tiles we need and they can be brought into the plot with the `get_map` function. This produces a much clearer map and enables readers to focus on the data rather than the cartography of the base map. The integration of data and services from third parties is a growing trend within R and one of its key strengths: users are not constrained to proprietary or paid for services, they can make full use of the open data sources that are now available.

```
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)
```

Case Study

As an illustrative example, this final section describes the creation of a map depicting 18th Century shipping flows. The data used in this visualisation have been obtained from the Climatological Database for the World's Oceans (CLIWOC) and 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 plotting capabilities within ggplot2, and illustrate those ways in which they can be applied to produce high-quality maps that are reproducible with only a few lines of code.

The example uses the png package to load in a series of map annotations stored at png graphics files. 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")
```

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

The first few lines in the `bdata` object contain seven columns, with each row reporting a single point on the ship's course. The first step is to specify the format for a number of plot parameters that will 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 prepare the World coastlines for input into ggplot2 with the `fortify` command, and then combined with background data to create the plot. In the following code, this sets the extents of the plot window and provides a blank canvas on which layers can be built. 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 snippet below creates the plot layer containing the the shipping routes. The `geom_path()` function is used to string together the coordinates to form the routes. You can see within the `aes()` that this specifies the 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"))
```

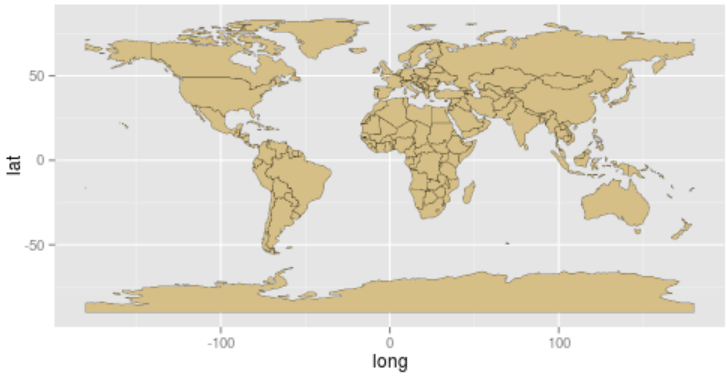

Figure 7: World Map

We now have all we need to generate the final plot, shown in Figure 8, 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
```

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
```

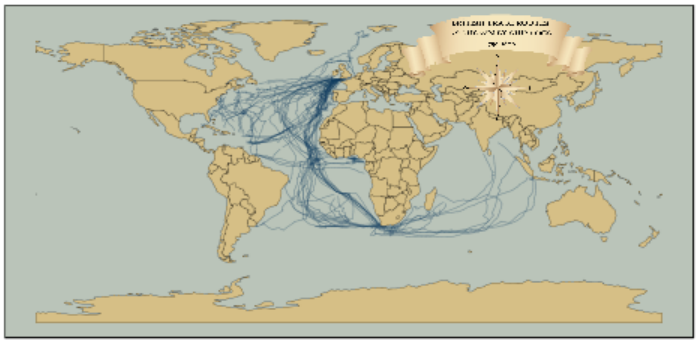

Figure 8: World Shipping

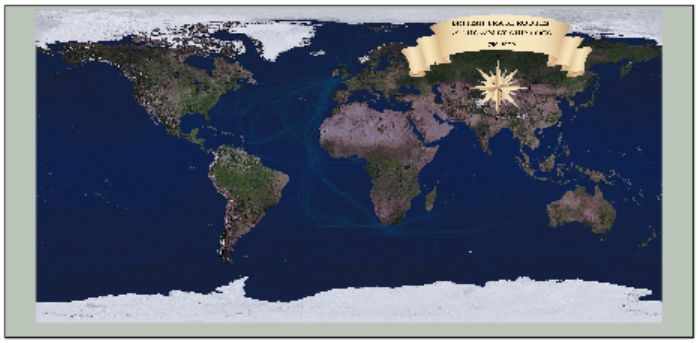

Figure 9: World Shipping with raster background

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. 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. 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.

Supporting Materials

R is a constantly evolving and as its user community grows the number of online resources is set to increase. This chapter has focussed on the use of vector data. For those who would like to use R for processing and visualising raster datasets, we would recommend a vignette from the raster package: cran.r-project.org/web/packages/raster/vignettes/Raster.pdf We also recommend the following tutorial on plotting raster data with ggplot2: tinyurl.com/nencrgf.

Map projections are a complex and important topic that we only touch on here. The following two web pages offer some further context and technical information: spatial.ly/2011/03/flattening-the-earth/ and en.wikipedia.org/wiki/Map_projection.

To stay abreast of current developments and tutorials in R we recommend: r-bloggers.com, a feed aggregator about R, and the rpubs.com/ code repository.

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