Introduction to visualising spatial data in R

Robin Lovelace (R.Lovelace@Leeds.ac.uk) and James Cheshire (james.cheshire@ucl.ac.uk)

July, 2014

Contents

Part I: Introduction **2**

Prerequisites and packages . . . 2

Typographic conventions and getting help . . . 2

Part II: Spatial data in R **4**

Starting the tutorial . . . 4

Downloading the data . . . 4

Loading the spatial data . . . 5

Basic plotting . . . 5

Attribute data . . . 7

Part III: Manipulating spatial data **9**

Changing projection . . . 9

Attribute joins . . . 9

Clipping and spatial joins . . . 12

Spatial aggregation . . . 14

Optional task: aggregation with gIntersects . . . 18

Part IV: Map making with ggplot2 **20**

"Fortifying" spatial objects for ggplot2 maps . . . 21

Adding base maps to ggplot2 with ggmap . . . 23

Advanced Task: Faceting for Maps . . . 25

Part V: Taking spatial data analysis in R further **28**

R quick reference **29**

Further information **30**

Aknowledgements **30**

References **31**

Part I: Introduction

This tutorial is an introduction to spatial data in R and map making with R's 'base' graphics and the popular graphics package **ggplot2**. It assumes no prior knowledge of spatial data analysis but prior understanding of the R command line would be beneficial. If you have not used R before, we recommend working through an 'Introduction to R' type tutorial, such as "A (very) short introduction to R" (Torfs and Brauer, 2012) or the more geographically inclined "Short introduction to R" (Harris, 2012).

Building on such background material, the following tutorial is concerned with specific functions for spatial data and visualisation. It is divided into five parts:

- Introduction, which provides a guide to R's syntax and preparing for the tutorial
- Spatial data in R, which describes basic spatial functions in R
- Manipulating spatial data, which includes changing projection, clipping and spatial joins
- Map making with **ggplot2**, a graphics package for producing beautiful maps quickly
- Taking spatial analysis in R further, a compilation of resources for furthering your skills

Prerequisites and packages

For this tutorial you need a copy of R. If you need to install R, the latest version can be downloaded from http://cran.r-project.org/. We also suggest that you use an R editor, such as RStudio, for this tutorial as this will improve the user-experience and help with the learning process.

R has a huge and growing number of spatial data packages, some of which we will come across in this tutorial. We recommend taking a quick browse on R's main website to see which other spatial packages are available: http://cran.r-project.org/web/views/Spatial.html.

The spatial packages we will use in the tutorial are:

- **ggmap**: extends the plotting package **ggplot2** for maps
- **rgdal**: R's interface to the popular C/C++ GIS library gdal
- **rgeos**: R's interface to the powerful vector processing library geos
- **maptools**: provides various mapping functions
- **dplyr**: fast and concise data manipulation package
- **tidyr**: for creating 'tidy' datasets

To test whether a package is installed, try to load it using `library`. To test if **ggplot2** is available, for example, enter `library(ggplot2)`. If there is no output from R, this is good news: it means that the library has already been installed on your computer.

If you get an error message, it needs to be installed using `install.packages("ggplot2")`. The package will download from CRAN (the Comprehensive R Archive Network); if you are prompted to select a 'mirror', select one that is close to your home. If you have not done so already, install these packages on your computer now. A quick way to do this in one go is to enter the following lines of code:

```
x <- c("ggmap", "rgdal", "rgeos", "maptools", "dplyr", "tidyr") # the packages
install.packages(x) # warning: this may take a number of minutes
lapply(x, library, character.only = TRUE) # load the required packages
```

Typographic conventions and getting help

It is a good idea to get into the habit of consistent and clear writing in any language, and R is no exception. Adding comments to your code is good practice, so you remember at a later date what you've done, aiding

the learning process. There are two main ways of commenting code using the `#` symbol: above a line of code or directly following it, as illustrated in the block of code presented below, which should create figure 1 if typed correctly into the R command line.

```
# Generate data
x <- 1:400
y <- sin(x / 10) * exp(x * -0.01)

plot(x, y) # plot the result
```

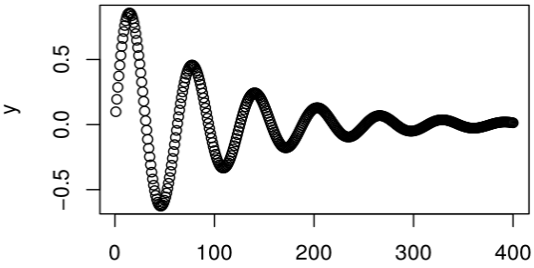

Figure 1: Basic plot of x and y

In the first line of code a new *object* called `x` is created. Any name could have been used, like `x_bumkin`, but `x` is concise and works just fine here. It is good practice to give your objects meaningful names. Note `<-`, the directional "arrow" assignment symbol. This creates new objects. We will be using this symbol a lot in this tutorial.[1]

To distinguish between prose and code, please be aware of the following typographic conventions used in this document: R code (e.g. `plot(x, y)`) is written in a `monospace` font and package names (e.g. **rgdal**) are written in **bold**.

A double hash (`##`) at the start of a line of code indicates that this is output from R. Lengthy outputs have been omitted from the document to save space, so do not be alarmed if R produces additional messages: you can always look up them up online.

As with any programming language, there are often many ways to produce the same output in R. The code presented in this document is not the only way to do things. We encourage you to play with the code to gain a deeper understanding of R. Do not worry, you cannot 'break' anything using R and all the input data can be re-loaded if things do go wrong. As with learning to skateboard, you learn by falling and getting an `Error:` message in R is much less painful than falling onto concrete! We encourage `Error:`s - it means you are trying new things.

If you require help on any function, use the `help` command, e.g. `help(plot)`. Because R users love being concise, this can also be written as `?plot`. Feel free to use it at any point you would like more detail on a specific function (although R's help files are famously cryptic for the un-initiated). Help on more general terms can be found using the `??` symbol. To test this, try typing `??regression`. For the most part, *learning by doing* is a good motto, so let's crack on and download some packages and then some data.

[1] Tip: typing `Alt -` on the keyboard will create the arrow in RStudio. The equals sign `=` also works but is not recommended by R developers.

Part II: Spatial data in R

Starting the tutorial

Now that we have taken a look at R's syntax and installed the necessary packages, we can start looking at some real spatial data. This second part introduces some spatial files that we will download from the internet. Plotting and interrogating spatial objects are central spatial data analysis in R, so we will focus on these elements in the next two parts of the tutorial, before focussing on creating attractive maps in Part IV.

Downloading the data

Firstly, download the data for this tutorial from : https://github.com/Robinlovelace/Creating-maps-in-R. Click on the "Download ZIP" button on the right hand side of the screen and once it is downloaded, unzip this to a new folder on your computer.

For this tutorial, we suggest working on the 'Creating-maps-in-R' project which has already been created for you. To open this project, navigate to `File -> Open File...` in the top menu and navigate to the unzipped `Creating-maps-in-R` folder. Double click on the `Creating-maps-in-R.Rproj` project file to open this project.

Alternatively, you can use the *project menu* to open the project or create a new one. It is *highly recommended* that you use RStudio's projects to organise your R work and that you organise your files into sub-folders (e.g. `code`, `input-data`, `figures`) to avoid digital clutter (Figure 2). The RStudio website contains an overview of the software: rstudio.com/products/rstudio/.

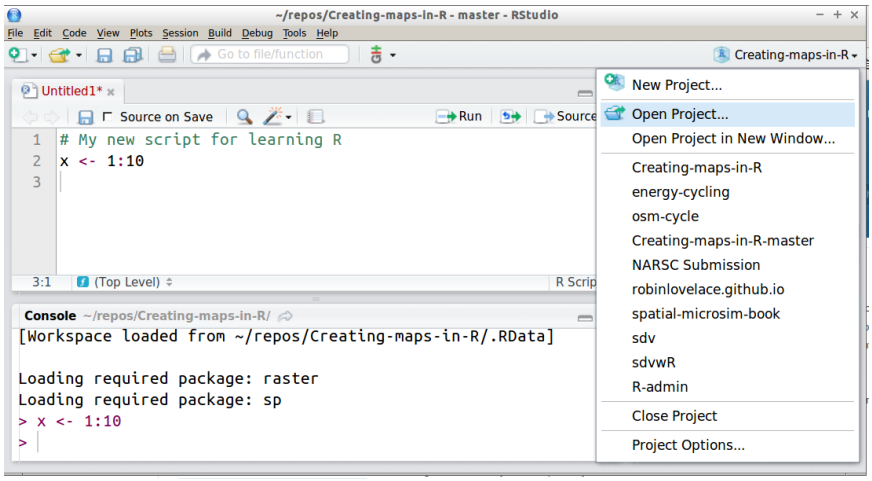

Figure 2: The RStudio environment with the project tab poised to open the Creating-maps-in-R project.

Opening a project sets the current working directory to the project's parent folder, the `Creating-maps-in-R` folder in this case. If you ever need to change your working directory, you can use the 'Session' menu at the top of the page or use the `setwd` command.

It is also worth taking a look at the input dataset in your file browser before opening them in R. You could try opening the file "london_sport.shp"(located within the "data" folder of the project), in a GIS program

such as QGIS (which can be freely downloaded from the internet) to explore the attribute and geometry information in a familiar environment. Also note that .shp files are composed of several files for each object: you should be able to open "london_sport.dbf" in a spreadsheet program such as LibreOffice Calc. Once you've understood something of this input data and where it lives, it's time to open it in R.

Loading the spatial data

One of the most important steps in handling spatial data with R is the ability to read in spatial data, such as shapefiles (a common geographical file format). There are a number of ways to do this, the most commonly used and versatile of which is `readOGR`. This function, from the **rgdal** package, automatically extracts information about the projection and the attributes of data. **rgdal** is R's interface to the "Geospatial Abstraction Library (GDAL)" which is used by other open source GIS packages such as QGIS and enables R to handle a broader range of spatial data formats. If you've not already *installed* and loaded the **rgdal** package (see the 'prerequisites and packages' section) do so now:

```
library(rgdal)
lnd_sport <- readOGR(dsn = "data", layer = "london_sport")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "data", layer: "london_sport"
## with 33 features and 4 fields
## Feature type: wkbPolygon with 2 dimensions
```

In the second line of code above the `readOGR` function is used to load a shapefile and assign it to a new object called "lnd_sport". `readOGR` is a *function* which accepts two *arguments*: `dsn` which stands for "data source name" and specifies the location where the file is stored, and `layer` which specifies the file name. Note that each new argument is separated by a comma and there is no need to specify the file extension (e.g. .shp) when providing the file name. Both arguments in this case are *character strings* (indicated by quote marks like this one "). R functions have a default order of arguments, so `dsn =` and `layer =` do not actually have to be typed for the command to run, for example, `readOGR("data", "london_sport")` would work just as well. For clarity, it is good practice to include argument names such as `dsn` and `layer` when learning new functions and we continue this tradition below.

The files beginning `london_sport` in the `data/` directory contain the population of London Boroughs in 2001 and the percentage of the population participating in sporting activities. This data originates from the active people survey. The boundary data is from the Ordnance Survey.

For information about how to load different types of spatial data, see the help documentation for `readOGR`. This can be accessed by typing `?readOGR`. For another worked example, in which a GPS trace is loaded, please see Cheshire and Lovelace (2014).

Basic plotting

We have now created a new spatial object called "lnd_sport" from the "london_sport" shapefile. Spatial objects are made up of a number of different *slots*, mainly the attribute *slot* and the geometry *slot*. The attribute *slot* can be thought of as an attribute table and the geometry *slot* is where the spatial object (and its attributes) lie in space. Let's now analyse the sport object with some basic commands:

```
head(lnd_sport@data, n = 2)
```

```
##   ons_label                name Partic_Per Pop_2001
## 0      00AF             Bromley       21.7   295535
## 1      00BD Richmond upon Thames       26.6   172330
```

```
mean(lnd_sport$Partic_Per)
```

```
## [1] 20.05455
```

Take a look at this output and notice the table format of the data and the column names. There are two important symbols at work in the above block of code: the `@` symbol in the first line of code is used to refer to the attribute *slot* of the object. The `$` symbol refers to a specific attribute (a variable with a column name) in the `data` *slot*, which was identified from the result of running the first line of code. If you are using RStudio, test out the auto-completion functionality by hitting `tab` before completing the command - this can save you a lot of time in the long run.

The `head` function in the first line of the code above simply means "show the first few lines of data", i.e. the head. It's default is to output the first 6 rows of the dataset (try simply `head(lnd_sport@data)`), but we can specify the number of lines with `n = 2` after the comma. The second line of the code above calculates the mean value of the variable `Partic_Per` (sports participation per 100 people) for each of the zones in the `lnd_sport` object. To explore `lnd_sport` object further, try typing `nrow(lnd_sport)` and record how many zones the dataset contains. You can also try `ncol(lnd_sport)`.

Now we have seen something of the attribute *slot* of the spatial object, let us look at its *geometry*, which describes where the polygons are located in space:

```
plot(lnd_sport) # not shown in tutorial - try it on your computer
```

`plot` is one of the most useful functions in R, as it changes its behaviour depending on the input data (this is called *polymorphism* by computer scientists). Inputting another object such as `plot(lnd_sport@data)` will generate an entirely different type of plot. Thus R is intelligent at guessing what you want to do with the data you provide it with.

R has powerful subsetting capabilities that can be accessed very concisely using square brackets, as shown in the following example:

```
# select rows of lnd_sport@data where sports participation is less than 15
lnd_sport@data[lnd_sport$Partic_Per < 15, ]
```

```
##    ons_label           name Partic_Per Pop_2001
## 17      00AQ         Harrow       14.8   206822
## 21      00BB         Newham       13.1   243884
## 32      00AA City of London        9.1     7181
```

The above line of code asked R to select rows from the `lnd_sport` object, where sports participation is lower than 15, in this case rows 17, 21 and 32, which are Harrow, Newham and the city centre respectively. The square brackets work as follows: anything before the comma refers to the rows that will be selected, anything after the comma refers to the number of columns that should be returned. For example if the data frame had 1000 columns and you were only interested in the first two columns you could specify `1:2` after the comma. The ":" symbol simply means "to", i.e. columns 1 to 2. Try experimenting with the square brackets notation (e.g. guess the result of `lnd_sport@data[1:2, 1:3]` and test it): it will be useful.

So far we have been interrogating only the attribute *slot* (`@data`) of the `lnd_sport` object, but the square brackets can also be used to subset spatial objects, i.e. the geometry *slot*. Using the same logic as before try to plot a subset of zones with high sports participation.

```
# Plot zones where sports participation is greater than 25 %
plot(lnd_sport[lnd_sport$Partic_Per > 25, ]) # output not shown here
```

This plot is useful, but it only shows the areas which meet the criteria. To see the sporty areas in context with the other areas of the map simply use the `add = TRUE` argument after the initial plot. (`add = T` would also work, but we like to spell things out in this tutorial for clarity). What does the `col` argument refer to in the below block - it should be obvious (see figure 3).

If you wish to experiment with multiple criteria queries, use the '&' sign.

```
plot(lnd_sport) # plot the london_sport object
sel <- lnd_sport$Partic_Per > 25 # select the zones with high sports participation
plot(lnd_sport[ sel , ], col = "blue", add = TRUE) # add selected zones to existing map
```

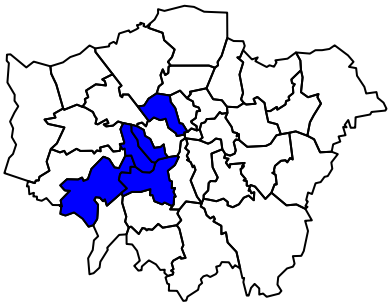

Figure 3: Preliminary plot of London with areas of high sports participation highlighted in blue

Congratulations! You have just interrogated and visualised a spatial object: where are areas with high levels of sports participation in London? The map tells us. Do not worry for now about the intricacies of how this was achieved: you have learned vital basics of how R works as a language; we will cover this in more detail in subsequent sections.

While we are on the topic of loading data, it is worth pointing out that R can save and load data efficiently into its own data format (`.RData`). Try `save(lnd_sport, file = "sport.RData")` and see what happens. If you type `rm(lnd_sport)` (which removes the object) and then `load("sport.RData")` you should see how this works. `lnd_sport` will disappear from the workspace and then reappear.

Attribute data

All shapefiles have both attribute table and geometry data. These are automatically loaded with `readOGR`. The loaded attribute data can be treated the same as an R data frame.

R deliberately hides the geometry of spatial data unless you print the entire object (try typing `print(lnd_sport)`). Let's take a look at the headings of sport, using the following command: `names(lnd_sport)` Remember, the attribute data contained in spatial objects are kept in a 'slot' that can be

accessed using the `@` symbol: `lnd_sport@data`. This is useful if you do not wish to work with the spatial components of the data at all times.

Type `summary(lnd_sport)` to get some additional information about the data object. Spatial objects in R contain much additional information:

```
summary(lnd_sport)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
## min max
## x 503571.2 561941.1
## y 155850.8 200932.5
## Is projected: TRUE
## proj4string :
## [+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 ....]
```

The above output tells us that `lnd_sport` is a special spatial class, in this case a `SpatialPolygonsDataFrame`, meaning it is composed of various polygons, each of which has attributes. This is the typical class of data found in administrative zones. The coordinates tell us what the maximum and minimum x and y values are, for plotting. Finally, we are told something of the coordinate reference system with the `Is projected` and `proj4string` lines. In this case, we have a projected system, which means it is a Cartesian reference system, relative to some point on the surface of the Earth. We will cover reprojecting data in the next part of the tutorial.

Part III: Manipulating spatial data

It is all very well being able to load and interrogate spatial data in R, but to compete with modern GIS packages, R must also be able to modify these spatial objects (see 'using R as a GIS'). R has a wide range of very powerful functions for this, many of which reside in additional packages alluded to in the introduction.

This course is introductory so only commonly required data manipulation tasks, *reprojecting* and *joining/clipping* are covered here. We will look at joining non-spatial data to our spatial object. We will then cover spatial joins, whereby data is joined to other dataset based on spatial location.

Changing projection

Before undertaking spatial queries of an object, it is useful to know the *coordinate reference system* (CRS) it uses. You may have noticed the word `proj4string` in the summary of the `lnd_sport` object above. This represents it CRS mathematically. In some spatial data files, no CRS is specified or worse, an incorrect CRS value is given. Provided the correct CRS is known, this can be righted with a single line:

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

R issues a warning when changing the CRS in this way to ensure the user knows that they are simply changing the CRS, not *reprojecting* the data. R uses EPSG codes to refer to different coordinate reference systems. `27700` is the code for British National Grid. A commonly used geographical ('lat/lon') CRS is 'WGS84', whose EPSG code is `4326`. The following code shows how to search the list of available EPSG codes and create a new version of `lnd_sport` in WGS84:[2]

```
EPSG <- make_EPSG() # create data frame of available EPSG codes
EPSG[grepl("WGS 84$", EPSG$note), ] # search for WGS 84 code
```

```
##     code     note                                  prj4
## 249 4326 # WGS 84 +proj=longlat +datum=WGS84 +no_defs
```

```
lnd_sport_wgs84 <- spTransform(lnd_sport, CRS("+init=epsg:4326")) # reproject
```

The above code uses the function `spTransform`, from the **sp** package, to convert the `lnd_sport` object into a new form, with the Coordinate Reference System (CRS) specified as WGS84. The different EPSG codes are a bit of hassle to remember but you can search for them at spatialreference.org.

Attribute joins

Attribute joins are used to link additional pieces of information to our polygons. in the `lnd_sport` object, for example, we have 5 attribute variables - that can be found by typing `names(lnd_sport)`. But what happens when we want to add more variables from an external source? We will use the example of recorded crimes by London boroughs to demonstrate this.

To reaffirm our starting point, let's re-load the "london_sport" shapefile as a new object and plot it. This is identical to the `lnd_sport` object in the first instance, but we will give it a new name, in case we ever need to re-use `lnd_sport`. We will call this new object `lnd`, short for London:

```
library(rgdal) # ensure rgdal is loaded
# Create new object called "lnd" from "london_sport" shapefile
lnd <- readOGR(dsn = "data", "london_sport")
```

[2]Note: entering `projInfo()` will provide additional CRS options available from **rgdal**.

```
## OGR data source with driver: ESRI Shapefile
## Source: "data", layer: "london_sport"
## with 33 features and 4 fields
## Feature type: wkbPolygon with 2 dimensions
```

```
plot(lnd) # plot the lnd object
```

Figure 4: Plot of London

```
nrow(lnd) # return the number of rows
```

```
## [1] 33
```

The non-spatial data we are going to join to the `lnd` object contains records of crimes in London. This is stored in a comma separated values (`.csv`) file called "mps-recordedcrime-borough". Viewing the file locally shows that each row represents a single reported crime. We are going to use a function called `aggregate` to aggregate the crimes at the borough level, ready to join to our spatial `lnd` dataset. A new object called `crime_data` is created to store this data.

```
# Create and look at new crime_data object
crime_data <- read.csv("data/mps-recordedcrime-borough.csv")

head(crime_data, 3) # display first 3 lines
summary(crime_data$CrimeType) # summary of crime type

# Extract "Theft & Handling" crimes and save
crime_theft <- crime_data[crime_data$CrimeType == "Theft & Handling", ]
head(crime_theft, 2) # take a look at the result (replace 2 with 10 to see more rows)
```

```
# Calculate the sum of the crime count for each district and save result as a new object
crime_ag <- aggregate(CrimeCount ~ Borough, FUN = sum, data = crime_theft)
# Show the first two rows of the aggregated crime data
head(crime_ag, 2)
```

There is a lot going on in the above block of code and you should not expect to understand all of it upon first try: simply typing the commands and thinking briefly about the outputs is all that is needed at this stage to improve your intuitive understanding of R. Here are a few things that you may not have seen before that will likely be useful in the future:

- In the first line of code we specify the location of the file (check in your file browser to be sure).
- The == function is used to select only those observations that meet a specific condition, in this case all crimes involving "Theft and Handling".
- The ~ symbol means "by": we aggregated the `CrimeCount` variable by the district name.

Now that we have crime data at the borough level, the challenge is to join it to the `lnd` object. We will base our join on the `Borough` variable from the `crime_ag` object and the `name` variable from the `lnd` object. It is not always straight forward to join objects based on names as the names do not always match. Let us see which names in the `crime_ag` object match the spatial data object, `lnd`:

```
# Compare the name column in lnd to Borough column in crime_ag to see which rows match.
lnd$name %in% crime_ag$Borough
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

```
# Return rows which do not match
lnd$name[!lnd$name %in% crime_ag$Borough]
```

```
## [1] City of London
## 33 Levels: Barking and Dagenham Barnet Bexley Brent Bromley ... Westminster
```

The first line of code above uses the `%in%` command to identify which values in `lnd$name` are also contained in the names of the crime data. The results indicate that all but one of the borough names matches. The second line of code tells us that it is City of London, row 25, that is named differently in the crime data. Look at the results (not shown here) on your computer.

```
levels(crime_ag$Borough)[1:2] # names of boroughs
```

```
## [1] "Barking and Dagenham" "Barnet"
```

```
# Select the problematic row - the one which *does not* (via the ! symbol) match
sel <- !lnd$name %in% crime_ag$Borough # create selection
# Rename row 25 in crime_ag to match row 25 in lnd
levels(crime_ag$Borough)[25] <- as.character(lnd$name[sel])
summary(lnd$name %in% crime_ag$Borough) # now all columns match
```

```
##    Mode    TRUE    NA's
## logical      33       0
```

The above code block first identified the row with the faulty name and then renamed the level to match the `lnd` dataset. Note that we could not rename the variable directly, as it is stored as a factor.

We are now ready to join the datasets. It is recommended to use the `left_join` function **dplyr** but the `merge` function could equally be used. Note that when we ask for help for a function that is not loaded, nothing happens, indicating we need to load it:

```
?left_join # error flagged
??left_join
library(dplyr) # load the powerful dplyr package (use plyr if unavailable)
?left_join # should now be loaded (use join if unavailable)
```

The above code demonstrates how to search for functions that are not currently loaded in R, using the `??` notation (short for `help.search` in the same way that `?` is short for `help`). Note, you will need to have the **dplyr** package, which provides fast and intuitive functions for processing data, installed for this to work. After **dplyr** is loaded, a single question mark is sufficient to load the associated help.

The documentation for the `left_join` function will be displayed if the plyr package is available (if not, use `install.packages()` to install it). We use `left_join` because we want the length of the data frame to rename unchanged, with variables from new data appended in new columns. Or, in R's rather terse documentation: "return all rows from x, and all columns from x and y." The `*join` commands (including `inner_join` and `anti_join`) are more concise when joining variables have the same name across both datasets. **dplyr** is also helpful here as it contains a useful function to `rename` variables:

```
head(lnd$name) # dataset to add to (results not shown)
head(crime_ag$Borough) # the variables to join
crime_ag <- rename(crime_ag, name = Borough) # rename the Borough heading
# head(join(lnd@data, crime_ag)) # test it works
lnd@data <- left_join(lnd@data, crime_ag)
```

```
## Joining by: "name"
```

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

Challenge: create a map of additional variables in London

With the attribute joining skills you have learned in this section, you should now be able to take datasets from many sources and join them to your geographical data. To test your skills, try to join additional borough-level variables to the `lnd` object. An excellent dataset on this can be found on the the data.gov.uk website.

Using this dataset and the methods developed above, Figure 5 was created: the proportion of council seats won by conservatives in the 2014 local elections. **The challenge** is to create a similar map of a different variable (you may need to skip to Part IV to plot continuous variables).

Clipping and spatial joins

In addition to joining by zone name, it is also possible to do spatial joins in R. There are three main varieties: many-to-one, where the values of many intersecting objects contribute to a new variable in the main table, one-to-many, or one-to-one. Because boroughs in London are quite large, we will conduct a many-to-one spatial join. We will be using transport infrastructure points such as tube stations and roundabouts as the spatial data to join, with the aim of finding out about how many are found in each London borough.

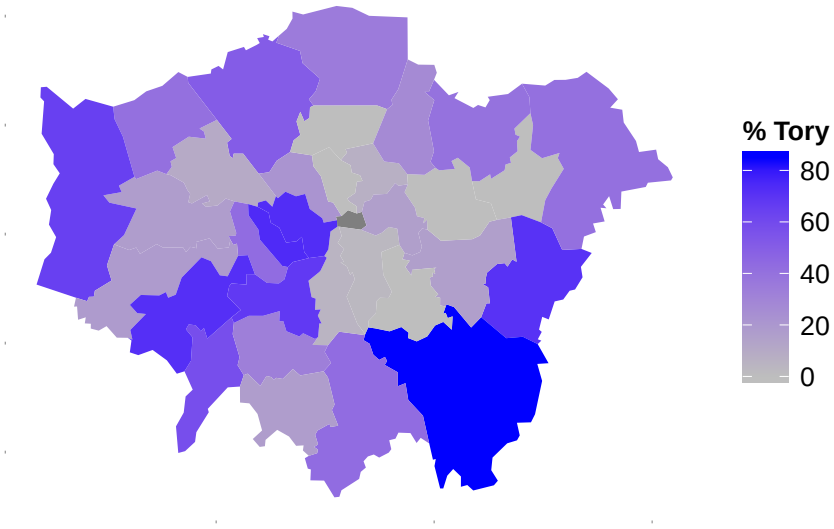

Figure 5: Proportion of council seats won by conservatives in the 2014 local elections using data from data.london.gov and joined using the methods presented in this section

```
library(rgdal)
# create new stations object using the "lnd-stns" shapefile.
stations <- readOGR(dsn = "data", layer = "lnd-stns")
proj4string(stations) # this is the full geographical detail.
proj4string(lnd) # what's the coordinate reference system (CRS)
bbox(stations) # the extent, 'bounding box' of stations
bbox(lnd) # return the bounding box of the lnd object
```

The above code loads the data correctly, but also shows that there are problems with it: the Coordinate Reference System (CRS) of `stations` differs from that of our `lnd` object. OSGB 1936 (or EPSG 27700) is the official CRS for the UK, so we will convert the object to this:

```
# Create reprojected stations object
stations27700 <- spTransform(stations, CRSobj = CRS(proj4string(lnd)))
stations <- stations27700 # overwrite the stations object
rm(stations27700) # remove the stations27700 object to clear up
plot(lnd) # plot London for context (see Figure 6)
points(stations) # overlay the station points
```

Now we can clearly see that the `stations` points overlay the boroughs. The problem is that the spatial extent of `stations` is great than that of `lnd`. We will take a spatially determined subset of the stations object that fall inside greater London. This is *clipping*.

Two functions can be used to clip `stations` so that only those falling within London boroughs are retained: `sp::over`, and `rgeos::gIntersects` (the word preceding the `::` symbol refers to the package which the function is from). Use `?` followed by the function to get help on each. Whether `gIntersects` or `over` is needed depends on the spatial data classes being compared (Bivand et al. 2013).

In this tutorial we will use the `over` function as it is easiest to use. In fact, it can be called just by using square brackets:

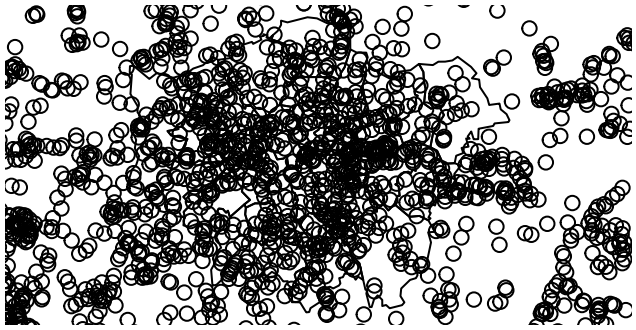

Figure 6: Sampling and plotting stations

```
stations <- stations[lnd, ]
plot(stations) # test the clip succeeded (see figure 7)
```

The above line of code says: "output all `stations` within the `lnd` object bounds". This is an incredibly concise way of clipping and has the added advantage of being consistent with R's syntax for non-spatial clipping. To prove it worked, only stations within the London boroughs appear in the plot.

`gIntersects` can achieve the same result, but with more lines of code (see www.rpubs.com/RobinLovelace for more on this) . It may seem confusing that two different functions can be used to generate the same result. However, this is a common issue in R; the question is finding the most appropriate solution.

In its less concise form (without use of square brackets), `over` takes two main input arguments: the target layer (the layer to be altered) and the source layer by which the target layer is to be clipped. The output of `over` is a data frame of the same dimensions as the original object (in this case `stations`), except that the points which fall outside the zone of interest are set to a value of `NA` ("no answer"). We can use this to make a subset of the original polygons, remembering the square bracket notation described in the first section. We create a new object, `sel` (short for "selection"), containing the indices of all relevant polygons:

```
sel <- over(stations, lnd)
stations <- stations[!is.na(sel[,1]),]
```

Typing `summary(sel)` should provide insight into how this worked: it is a data frame with 1801 NA values, representing zones outside of the London polygon. Note that the preceding two lines of code is equivalent to the single line of code, `stations <- stations[lnd, ]`. The next section demonstrates spatial aggregation, a more advanced version of spatial subsetting.

Spatial aggregation

As with R's very terse code for spatial subsetting, the base function `aggregate` (which provides summaries of variables based on some grouping variable) also behaves differently when the inputs are spatial objects.

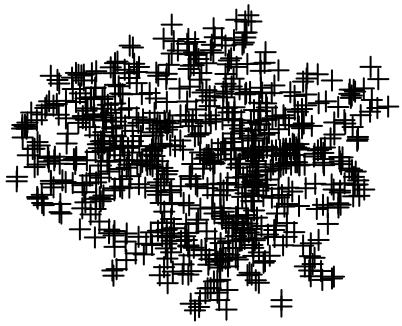

Figure 7: The clipped stations dataset

```
stations_agg <- aggregate(x = stations["CODE"], by = lnd, FUN = length)
head(stations_agg@data)
```

```
##   CODE
## 0   48
## 1   22
## 2   43
## 3   18
## 4   12
## 5   13
```

The above code performs a number of steps in just one line:

- `aggregate` identifies which `lnd` polygon (borough) each `station` is located in and groups them accordingly. The use of the syntax `stations["CODE"]` tells R that we are interested in the spatial data from `stations` and its `CODE` variable (any variable could have been used here as we are merely counting how many points exist).
- It counts the number of `stations` points in each borough, using the function `length`.
- A new spatial object is created, with the same geometry as `lnd`, and assigned the name `stations_agg`, the count of stations.

It may seem confusing that the result of the aggregated function is a new shape, not a list of numbers - this is because values are assigned to the elements within the `lnd` object. To extract the raw count data, one could enter `stations_agg$CODE`. This variable could be added to the original `lnd` object as a new field, as follows:

```
lnd$n_points <- stations_agg$CODE
```

As shown below, the spatial implementation of `aggregate` can provide summary statistics of variables, as well as simple counts. In this case we take the variable `NUMBER` and find its mean value for the stations in each ward.

```
lnd_n <- aggregate(stations["NUMBER"] , by = lnd, FUN = mean)
```

For an optional advanced task, let us analyse and plot the result.

```
q <- cut(lnd_n$NUMBER, breaks= c(quantile(lnd_n$NUMBER)), include.lowest=T)

summary(q) # check what we've created

## [1.82e+04,1.94e+04] (1.94e+04,1.99e+04] (1.99e+04,2.05e+04]
##                   9                   8                   8
##  (2.05e+04,2.1e+04]
##                   8

greys <- paste0("grey", seq(from = 20, to = 80, by = 20)) # create grey colours
clr <- as.character(factor(q, labels = greys)) # convert output of qs to greys
plot(lnd_n, col = clr) # plot (not shown in printed tutorial)
legend(legend = paste0("q", 1:4), fill = greys, "topright")
```

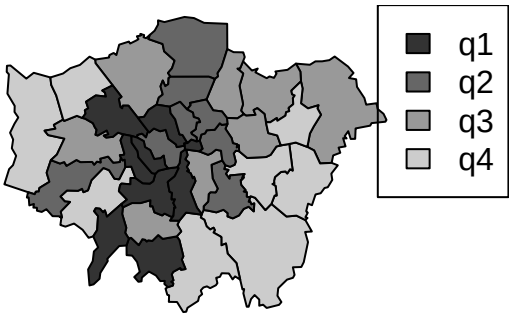

Figure 8: Choropleth map of mean values of stations in each borough

```
areas <- sapply(lnd_n@polygons, function(x) x@area)
```

This results in a simple choropleth map and a new vector containing the area of each borough (the basis for Figure 8). As an additional step, try comparing the mean area of each borough with the mean value of `stations` points within it: `plot(lnd_n$NUMBER, areas)`.

Adding different symbols for tube stations and train stations

Imagine that we want to now display all tube and train stations on top of the previously created choropleth map. How would we do this? The shape of points in R is determined by the `pch` argument, as demonstrated by the result of entering the following code: `plot(1:10, pch=1:10)`. To apply this knowledge to our map, we could add the following code to the chunk added above (see Figure 9):

```
levels(stations$LEGEND) # we want A roads and rapid transit stations (RTS)
sel <- grepl("A Road Sing|Rapid", stations$LEGEND) # selection for plotting
sym <- as.integer(stations$LEGEND[sel]) # symbols
points(stations[sel,], pch = sym)
legend(legend = c("A Road", "RTS"), "bottomright", pch = unique(sym))
```

```
## [1] "Railway Station"
## [2] "Rapid Transit Station"
## [3] "Roundabout, A Road Dual Carriageway"
## [4] "Roundabout, A Road Single Carriageway"
## [5] "Roundabout, B Road Dual Carriageway"
## [6] "Roundabout, B Road Single Carriageway"
## [7] "Roundabout, Minor Road over 4 metres wide"
## [8] "Roundabout, Primary Route Dual Carriageway"
## [9] "Roundabout, Primary Route Single C'way"
```

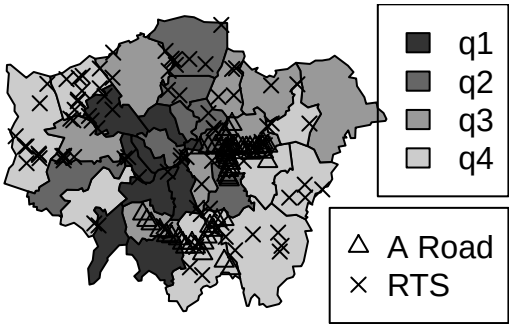

Figure 9: Symbol levels for train station types in London

In the above block of code, we first identified which types of transport points are present in the map with `levels` (this command only works on factor data, and tells us the unique names of the factors that the vector can hold). Next we select a subset of `stations` using a new command, `grepl`, to determine which points we want to plot. Note that `grepl`'s first argument is a text string (hence the quote marks) and that the second is a factor (try typing `class(stations$LEGEND)` to test this). `grepl` uses *regular expressions* to match whether each element in a vector of text or factor names match the text pattern we want. In this case, because we are only interested in roundabouts that are A roads and Rapid Transit systems (RTS). Note the use of the vertical separator `|` to indicate that we want to match `LEGEND` names that contain either "A Road" *or* "Rapid". Based on the positive matches (saved as `sel`, a vector of `TRUE` and `FALSE` values), we subset the stations. Finally we plot these as points, using the integer of their name to decide the symbol and add a legend. (See the documentation of `?legend` for detail on the complexities of legend creation in R's base graphics.)

This may seem a frustrating and un-intuitive way of altering map graphics compared with something like QGIS. That's because it is! It may not worth be stressing too much about base graphics because there is another option — **ggplot**. Please skip to Section IV if you're itching to see this more intuitive alternative to graphics in R.

Optional task: aggregation with gIntersects

As with clipping, we can also do spatial aggregation with the **rgeos** package. In some ways, this method makes explicit the steps taken in `aggregate` 'under the hood'. The code is quite involved and intimidating, so feel free to skip this stage. Working through and thinking about it this alternative method may, however, yield dividends if you intend to perform more sophisticated spatial analysis in R.

```
library(rgeos)
int <- gIntersects(stations, lnd, byid = TRUE) # re-run the intersection query
head(apply(int, MARGIN = 2, FUN = which))
b.indexes <- which(int, arr.ind = TRUE) # indexes that intersect
summary(b.indexes)
b.names <- lnd$name[b.indexes[, 1]]
b.count <- aggregate(b.indexes ~ b.names, FUN = length)
head(b.count)
```

The above code first extracts the index of the row (borough) for which the corresponding column is true and then converts this into names. The final object created, `b.count` contains the number of station points in each zone. According to this, Barking and Dagenham should contain 12 station points. It is important to check the output makes sense at every stage with R, so let's check to see this is indeed the case with a quick plot:

```
plot(lnd[grepl("Barking", lnd$name),])
points(stations)
```

Now the fun part: count the points in the polygon and report back how many there are!

We have now seen how to load, join and clip data. The second half of this tutorial is concerned with *visualisation* of the results. For this, we will use **ggplot2** and begin by looking at how it handles non-spatial data.

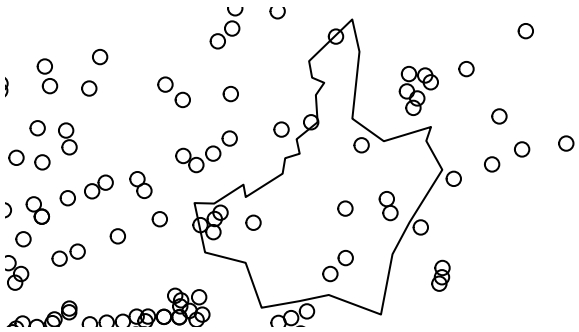

Figure 10: Transport points in Barking and Dagenham

Part IV: Map making with ggplot2

This third part introduces a slightly different method of creating plots in R using the ggplot2 package, and explains how it can make maps. The package is an implementation of the Grammar of Graphics (Wilkinson 2005) - a general scheme for data visualisation that breaks up graphs into semantic components such as scales and layers. **ggplot2** can serve as a replacement for the base graphics in R (the functions you have been plotting with today) and contains a number of default options that match good visualisation practice.

The maps we produce will not be that meaningful - the focus here is on sound visualisation with R and not sound analysis (obviously the value of the former diminished in the absence of the latter!) Whilst the instructions are step by step you are encouraged to deviate from them (trying different colours for example) to get a better understanding of what we are doing.

ggplot2 is one of the best documented packages in R. The full documentation for it can be found online and it is recommended you test out the examples on your own machines and play with them: http://docs.ggplot2.org/current/ .

Good examples of graphs can also be found on the website cookbook-r.com.

Load the package with

It is worth noting that the base `plot()` function requires less data preparation but more effort in control of features. `qplot()` and `ggplot()` from **ggplot2** require some additional work to format the spatial data but select colours and legends automatically, with sensible defaults.

As a first attempt with **ggplot2** we can create a scatter plot with the attribute data in the `lnd` object created above. Type:

```
p <- ggplot(lnd@data, aes(Partic_Per, Pop_2001))
```

What you have just done is set up a ggplot object where you say where you want the input data to come from. `lnd@data` is actually a data frame contained within the wider spatial object `lnd` (the `@` enables you to access the attribute table of the shapefile). The characters inside the `aes` argument refer to the parts of that data frame you wish to use (the variables `Partic_Per` and `Pop_2001`). This has to happen within the brackets of `aes()`, which means, roughly speaking 'aesthetics that vary'.

If you just type p and hit enter you get the error `No layers in plot`. This is because you have not told `ggplot` what you want to do with the data. We do this by adding so-called "geoms", in this case `geom_point()`.

```
p + geom_point()
```

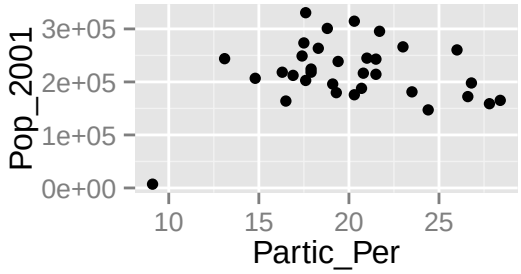

Figure 11: A simple graphic produced with **ggplot2**

Within the brackets you can alter the nature of the points. Try something like `p + geom_point(colour = "red", size=2)` and experiment.

If you want to scale the points by borough population and colour them by sports participation this is also fairly easy by adding another `aes()` argument.

```
p + geom_point(aes(colour=Partic_Per, size=Pop_2001)) # not shown
```

The real power of **ggplot2** lies in its ability to add layers to a plot. In this case we can add text to the plot.

```
p + geom_point(aes(colour = Partic_Per, size = Pop_2001)) +
  geom_text(size = 2, aes(label = name))
```

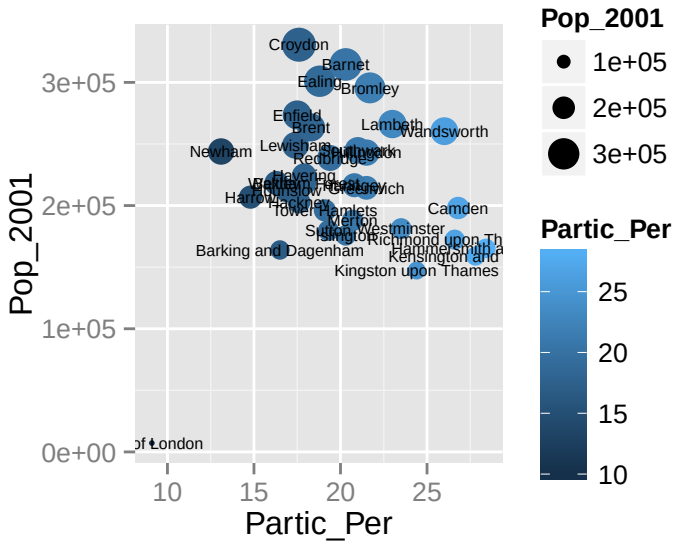

Figure 12: ggplot for text

This idea of layers (or geoms) is quite different from the standard plot functions in R, but you will find that each of the functions does a lot of clever stuff to make plotting much easier (see the documentation for a full list).

The following steps will create a map to show the percentage of the population in each London Borough who regularly participate in sports activities.

"Fortifying" spatial objects for ggplot2 maps

To get the shapefiles into a format that can be plotted we have to use the `fortify()` function. Spatial objects in R have a number of slots containing the various items of data (polygon geometry, projection, attribute information) associated with a shapefile. Slots can be thought of as shelves within the data object that contain the different attributes. The "polygons" slot contains the geometry of the polygons in the form of the XY coordinates used to draw the polygon outline. The generic plot function can work out what to do with these, **ggplot2** cannot. We therefore need to extract them as a data frame. The fortify function was written specifically for this purpose. For this to work, either **maptools** or **rgeos** packages must be installed.

```
lnd_f <- fortify(lnd) # you may need to load maptools
```

```
## Regions defined for each Polygons
```

This step has lost the attribute information associated with the lnd object. We can add it back using the `left_join` function (this performs a data join). To find out how this function works look at the output of typing `?left_join`.

```
head(lnd_f, n = 2) # peak at the fortified data
```

```
##       long      lat order  hole piece group id
## 1 541177.7 173555.7     1 FALSE     1   0.1  0
## 2 541872.2 173305.8     2 FALSE     1   0.1  0
```

```
lnd$id <- row.names(lnd) # allocate an id variable to the sp data
head(lnd@data, n = 2) # final check before join (requires shared variable name)
```

```
##   ons_label                name Partic_Per Pop_2001 CrimeCount
## 1      00AF             Bromley       21.7   295535      15172
## 2      00BD Richmond upon Thames       26.6   172330       9715
##   conservative n_points id
## 1           85       48  0
## 2           72       22  1
```

```
lnd_f <- left_join(lnd_f, lnd@data) # join the data
```

```
## Joining by: "id"
```

Take a look at the `lnd_f` object to see its contents (which stands for "London, fortified"). You should see a large data frame containing the latitude and longitude (they are actually Easting and Northing as the data are in British National Grid format) coordinates alongside the attribute information associated with each London Borough. If you type `print(lnd_f)` you will see just how many coordinate pairs are required! To keep the output to a minimum, take a peek at the first 2 lines of the object using just the `head` command:

```
lnd_f[1:2, 1:8]
```

```
##   id     long      lat order  hole piece group ons_label
## 1  0 541177.7 173555.7     1 FALSE     1   0.1      00AF
## 2  0 541872.2 173305.8     2 FALSE     1   0.1      00AF
```

It is now straightforward to produce a map using all the built in tools (such as setting the breaks in the data) that **ggplot2** has to offer. `coord_equal()` is the equivalent of asp=T in regular plots with R:

```
map <- ggplot(lnd_f, aes(long, lat, group = group, fill = Partic_Per)) +
  geom_polygon() +
  coord_equal() +
  labs(x = "Easting (m)", y = "Northing (m)", fill = "% Sports\nParticipation") +
  ggtitle("London Sports Participation")
```

Now, just typing `map` should result in your first ggplot-made map of London! There is a lot going on in the code above, so think about it line by line: what have each of the elements of code above been designed to do? Also note how the `aes()` components can be combined into one set of brackets after `ggplot`, that has relevance for all layers, rather than being broken into separate parts as we did above. The different plot

functions still know what to do with these. The `group=group` points ggplot to the group column added by `fortify()` and it identifies the groups of coordinates that pertain to individual polygons (in this case London Boroughs).

The default colours are really nice but we may wish to produce the map in black and white, which should produce a map like that shown below (and try changing the colors):

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

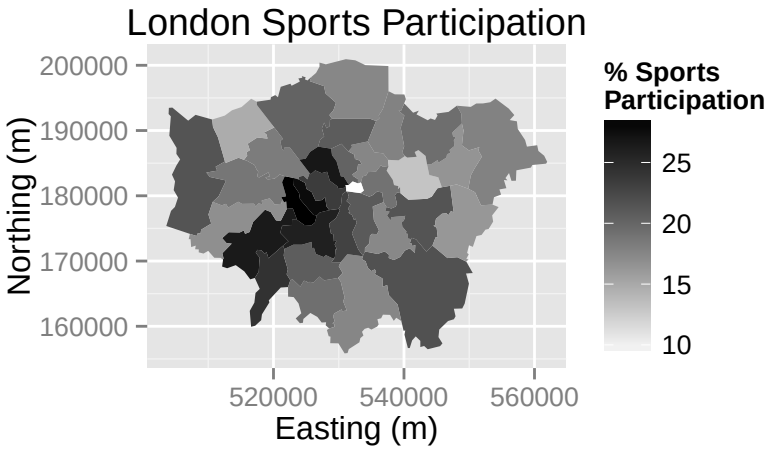

Figure 13: Greyscale map

Saving plot images is also easy. You just need to use `ggsave` after each plot, e.g. `ggsave("my_map.pdf")` will save the map as a pdf, with default settings. For a larger map, you could try the following:

```
ggsave("large_plot.png", scale = 3, dpi = 400)
```

Adding base maps to ggplot2 with ggmap

ggmap is a package that uses the **ggplot2** syntax as a template to create maps with image tiles taken from map servers such as Google and OpenStreetMap:

```
library(ggmap) # install.packages("ggmap") if not installed
```

The `lnd` object loaded previously is in British National Grid but the ggmap image tiles are in WGS84. We therefore need to use the `lnd_sport_wgs84` object created in the reprojection operation (see Section III).

The first job is to calculate the bounding box (bb for short) of the `lnd_sport_wgs84` object to identify the geographic extent of the image tiles that we need.

```
b <- bbox(lnd_sport_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.

```
lnd_b1 <- ggmap(get_map(location = b)) # create basemap for london
```

In much the same way as we did above we can then layer the plot with different geoms.

First fortify the `lnd_sport_wgs84` object and then merge with the required attribute data (we already did this step to create the `lnd_f` object).

```
lnd_wgs84_f <- fortify(lnd_sport_wgs84, region = "ons_label")
lnd_wgs84_f <- left_join(lnd_wgs84_f, lnd_sport_wgs84@data,
  by = c("id" = "ons_label"))
```

We can now overlay this on our base map.

```
lnd_b1 +
  geom_polygon(data = lnd_wgs84_f,
  aes(x = long, y = lat, group = group, fill = Partic_Per), alpha = 0.5)
```

The code above contains a lot of parameters. Use the **ggplot2** help pages to find out what they are. The resulting map looks okay, 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:

```
# download basemap (use load("data/lnd_b2.RData") if you have no internet)
lnd_b2 <- ggmap(get_map(location = b, source = "stamen",
          maptype = "toner", crop = TRUE))
```

We can then produce the plot as before:

```
library(mapproj) # mapproj library needed - install.packages("mapproj")
```

```
## Loading required package: maps
```

```
lnd_b2 +
  geom_polygon(data = lnd_wgs84_f,
    aes(x = long, y = lat, group = group, fill = Partic_Per),
    alpha = 0.5)
```

Finally, to increase the detail of the base map, we can use `get_map`'s `zoom` argument (result not shown)

```
# download basemap (try load("data/lnd_b3.RData") if you lack internet)
lnd_b3 <- ggmap(get_map(location = b, source = "stamen",
  maptype = "toner", crop = TRUE, zoom = 11))
```

```
lnd_b3 +
  geom_polygon(data = lnd_wgs84_f,
    aes(x = long, y = lat, group = group, fill = Partic_Per), alpha = 0.5)
```

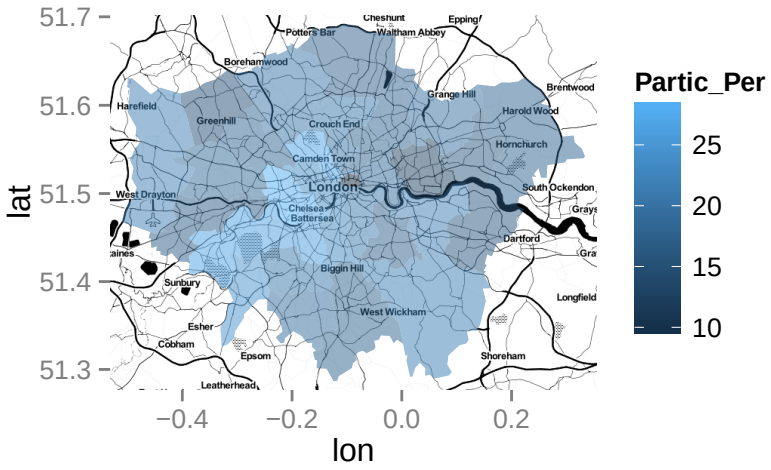

Figure 14: Basemap 2

Advanced Task: Faceting for Maps

Load the data - this shows historic population values between 1801 and 2001 for London, again from the London data store.

`tidy` the data so that the columns become rows. In other words, convert the data from 'flat' to 'long' format, which is the form required by **ggplot2** for faceting graphics: the date of the population survey become a variable in its own right, rather than being strung-out over many columns.

```
london_data <- read.csv("data/census-historic-population-borough.csv")
library(tidyr) # if not install it, or skip the next two steps
ltidy <- gather(london_data, date, pop, -Area.Code, -Area.Name)
head(ltidy, 2) # check the output (not shown)
```

In the above code we take the `london_data` object and create the column names 'date' (the date of the record, previously spread over many columns) and 'pop' (the population which varies). The minus (`-`) symbol in this context tells gather not to include the `Area.Name` and `Area.Code` as columns to be removed. In other words, "leave these columns be". Data tidying is an important subject: more can be read on the subject in Wickham (2014) or in a `vignette` about the package, accessed from within R by entering `vignette("tidy-data")`.

Merge the population data with the London borough geometry contained within our `lnd_f` object, using the `left_join` function from the **dplyr** package:

```
head(lnd_f, 2) # identify shared variables with ltidy
```

```
##   id     long      lat order  hole piece group ons_label    name
## 1  0 541177.7 173555.7     1 FALSE     1   0.1      00AF Bromley
## 2  0 541872.2 173305.8     2 FALSE     1   0.1      00AF Bromley
##   Partic_Per Pop_2001 CrimeCount conservative n_points
## 1       21.7   295535      15172           85       48
## 2       21.7   295535      15172           85       48
```

```
ltidy <- rename(ltidy, ons_label = Area.Code) # rename Area.code variable
lnd_f <- left_join(lnd_f, ltidy)
```

```
## Joining by: "ons_label"
```

Rename the date variable (use `?gsub` and Google 'regex' to find out more).

```
lnd_f$date <- gsub(pattern = "Pop_", replacement = "", lnd_f$date)
```

We can now use faceting to produce one map per year (this may take a little while to appear as displayed in Figure 15).

```
ggplot(data = lnd_f, # the input data
  aes(x = long, y = lat, fill = pop/1000, group = group)) + # define variables
  geom_polygon() + # plot the boroughs
  geom_path(colour="black", lwd=0.05) + # borough borders
  coord_equal() + # fixed x and y scales
  facet_wrap(~ date) + # one plot per time slice
  scale_fill_gradient2(low = "green", mid = "grey", high = "red", # colors
    midpoint = 150, name = "Population\n(thousands)") + # legend options
  theme(axis.text = element_blank(), # change the theme options
    axis.title = element_blank(), # remove axis titles
    axis.ticks = element_blank()) # remove axis ticks


# ggsave("figure/facet_london.png", width = 9, height = 9) # save figure
```

Again there is a lot going on here so explore the documentation to make sure you understand it. Try out different colour values as well.

Try experimenting with the above code block to see what effects you can produce. For bonus points, try creating an animation of London's evolving population over time using the excellent **animation** package.

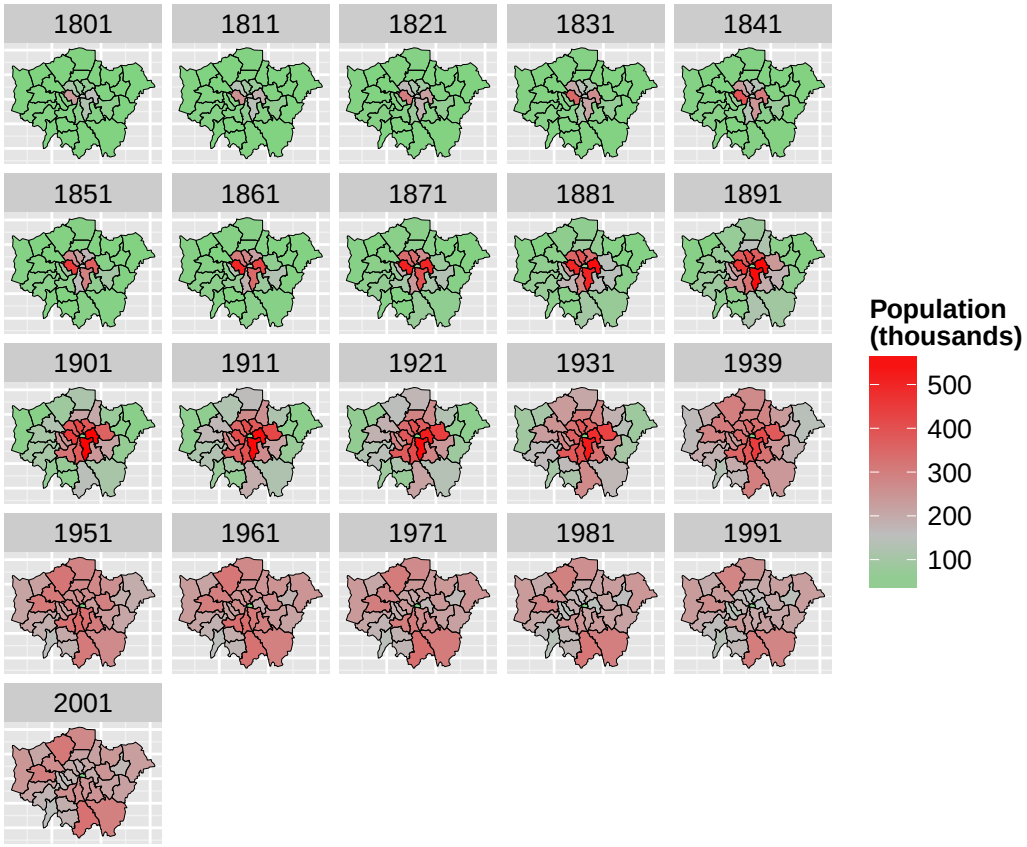

Figure 15: Faceted plot of the distribution of London's population over time

Part V: Taking spatial data analysis in R further

The skills taught in this tutorial are applicable to a very wide range of situations, spatial or not. Often experimentation is the most rewarding learning method, rather than just searching for the 'best' way of doing something (Kabakoff, 2011). We recommend you play around with your data.

If you would like to learn more about R's spatial functionalities, including more exercises on loading, saving and manipulating data, we recommend a slightly longer and more advanced tutorial (Cheshire and Lovelace, 2014). An up-to-date repository of this project, including example data and all the code used, can be found on its GitHub page: github.com/geocomPP/sdvwR. There are also a number of bonus 'vignettes' associated with the present tutorial. These can be found on the vignettes page of the project's repository.

Another advanced tutorial is "Using spatial data", which has example code and data that can be downloaded from the useR 2013 conference page. Such tutorials are worth doing as they will help you understand R's spatial 'ecosystem' as a cohesive whole rather than as a collection of isolated functions. In R the whole is greater than the sum of its parts.

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 a passive consumer (Ramsey & Dubovsky, 2013).

This does not necessarily mean writing a new package or contributing to R's 'Core Team' - it can simply involve helping others use R. We therefore conclude the tutorial with a list of resources that will help you further sharpen you R skills, find help and contribute to the growing online R community:

- R's homepage hosts a wealth of official and contributed guides.
- Stack Exchange and GIS Stack Exchange groups - try searching for "[R]". If your issue has not been not been addressed yet, you could post a polite question.
- R's mailing lists - the R-sig-geo list may be of particular interest here.

Books: despite the strength of R's online community, nothing beats a physical book for concentrated learning. We would particularly recommend the following:

- ggplot2: elegant graphics for data analysis (Wickham 2009).
- Bivand et al. (2013) Provide a dense and detailed overview of spatial data analysis.
- Kabacoff (2011) is a more general R book; it has many fun worked examples.

R quick reference

`#`: comments all text until line end

`df <- data.frame(x = 1:9, y = (1:9)^2`: create new object of class `data.frame`, called df, and assign values

`help(plot)`: ask R for basic help on function, the same as `?plot`. Replace `plot` with any function (e.g. `spTransform`).

`library(ggplot2)`: load a package (replace **ggplot2** with your package name)

`install.packages("ggplot2")`: install package - note quotation marks

`setwd("C:/Users/username/Desktop/")`: set R's *working directory* (set it to your project's folder)

`nrow(df)`: count the number of rows in the object `df`

`summary(df)`: summary statistics of the object `df`

`head(df)`: display first 6 lines of object `df`

`plot(df)`: plot object `df`

`save(df, "C:/Users/username/Desktop/" )`: save df object to specified location

`rm(df)`: remove the `df` object

`proj4string(df)`: query coordinate reference system of `df` object

`spTransform(df, CRS("+init=epsg:4326")`: reproject `df` object to WGS84

Further information

An up-to-date version of this tutorial is maintained at https://github.com/Robinlovelace/Creating-maps-in-R. The source files used to create this tutorial, including the input data can be downloaded as a zip file, as described below. The entire tutorial was written in RMarkdown, which allows R code to run as the document compiles, ensuring reproducibility.

Any suggested improvements or new vignettes are welcome, via email to Robin or by forking the master version of this document.

The colourful syntax highlighting in this document is thanks to RMarkdown. We try to follow best practice in terms of style, roughly following Google's style guide, an in-depth guide written by Johnson (2013) and a chapter from *Advanced R* (Wickham, in press).

Aknowledgements

The tutorial was developed for a series of Short Courses funded by the National Centre for Research Methods (NCRM), via the TALISMAN node (see geotalisman.org). Thanks to the ESRC for funding applied methods research. Many thanks to Rachel Oldroyd, Alistair Leak and Phil Jones who helped demonstrate these materials on the NCRM short courses for which this tutorial was developed. Amy O'Neill organised the course and encouraged feedback from participants. The final thanks is to all users and developers of open source software for making powerful tools such as R accessible and enjoyable to use.

If you have found this tutorial useful in your work, please cite it:

Lovelace, R., & Cheshire, J. (2014). Introduction to visualising spatial data in R. National Centre for Research Methods Working Papers, 14(03). Retrieved from https://github.com/Robinlovelace/Creating-maps-in-R

The bibtex entry for this item can be downloaded from here.

References

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

Cheshire, J. & Lovelace, R. (2014). Manipulating and visualizing spatial data with R. Book chapter in Press.

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

Johnson, P. E. (2013). R Style. An Rchaeological Commentary. The Comprehensive R Archive Network.

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

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

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

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

Wickham, H. (in press). Advanced R. CRC Press.

Wickham, H. (2014). Tidy data. The Journal of Statistical Software, 14(5), Retrieved from http://www.jstatsoft.org/v59/i10

Wilkinson, L. (2005). The grammar of graphics. Springer.