Introduction to visualising spatial data in R

Robin Lovelace (R.Lovelace@leeds.ac.uk), James Cheshire and others

V. 1.2, June, 2015 — see github.com/Robinlovelace/Creating-maps-in-R for latest version

Contents

Preface

This tutorial is an introduction to analysing spatial data in R, specifically through map-making with R's 'base' graphics and various dedicated map-making packages for R including **ggmap** and **leaflet**. It teaches the basics of using R as a fast, user-friendly and extremely powerful command-line Geographic Information System (GIS).

The tutorial is practical in nature: you will load-in, visualise and manipulate spatial data. We assume no prior knowledge of spatial data analysis but some experience with R will help. If you have not used R before, it may be worth following an introductory tutorial, such as "A (very) short introduction to R" (Torfs and Brauer, 2012). This is how the tutorial is organised.

1. Introduction: provides a guide to R's syntax and preparing for the tutorial
2. Spatial data in R: describes basic spatial functions in R
3. Creating and manipulating spatial data: includes changing projection, clipping and spatial joins
4. Map making with **tmap**, **ggplot2** and **leaflet**: this section demonstrates map making with more advanced visualisation tools
5. Taking spatial analysis in R further: a compilation of resources for furthering your skills

Part I: Introduction

Prerequisites and packages

For this tutorial you need a copy of 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, as this will improve the user-experience and help with the learning process. This can be downloaded from http://www.rstudio.com.

R has a huge and growing number of spatial data packages. We recommend taking a quick browse on R's main website to see the spatial packages available: http://cran.r-project.org/web/views/Spatial.html.

In this tutorial we will use:

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

To test whether a package is installed, try to load it using `library`. For example, to test if **ggplot2** is available, 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", "tmap")
# install.packages(x) # warning: uncommenting 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
```

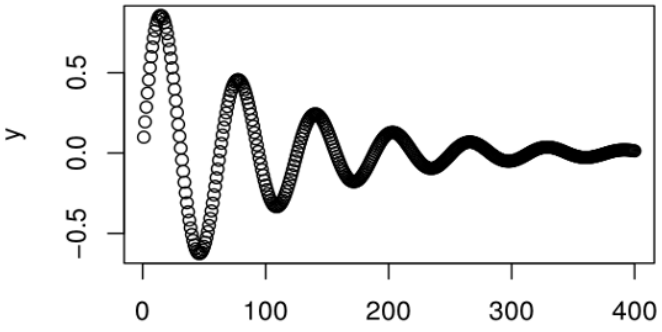

Figure 1: Basic plot of x and y (right) and code used to generate the plot (right).

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 on-line.

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 data.

[1] Tip: typing `Alt -` on the keyboard will create the arrow in RStudio. The equals sign `=` also works but is rarely used.

Part II: Spatial data in R

Starting the tutorial and downloading the data

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.

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/.

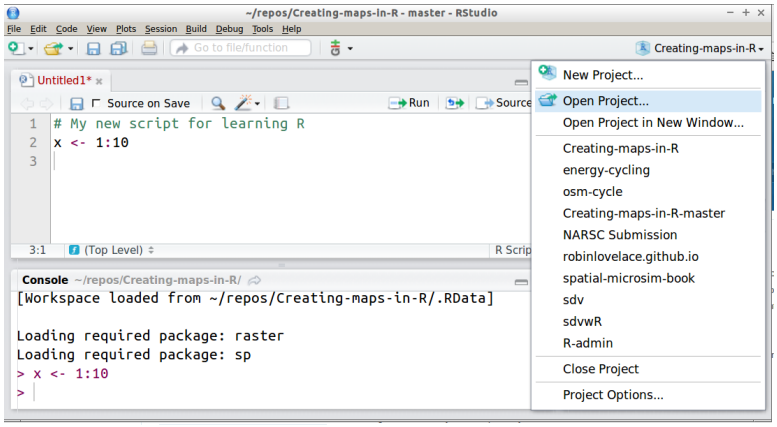

Figure 2: The RStudio environment with the project tab poised to open the Creating-maps-in-R project.

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 worth taking a look at the input dataset in your file browser before opening them in R. You could also try opening the file "london_sport.shp"(located within the "data" folder of the project), in mapping software such as QGIS, which can be freely downloaded from the internet. Note that .shp files are composed of several files for each object such as .prj, .sbn and .dbf files; you should be able to open "london_sport.dbf" in a spreadsheet program such as LibreOffice Calc. Once you think you understand the input data, it's time to open it in R.

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 the information regarding the 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 <- readOGR(dsn = "data", layer = "london_sport")
```

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". `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 for 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).

The structure of spatial data in R

We have now created a new spatial object called `lnd` from the "london_sport" shapefile. Spatial objects are made up of a number of different *slots*, the key ones being `@data` and `@polygons` (or `@lines` for line data) geometry data. The data *slot* can be thought of as an attribute table and the geometry *slot* is where the vertices of the object lie in space. Let's now analyse the sport object with some basic commands:

```
head(lnd@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$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 (see Figure 3).

To display the raw coordinates of the first polygon, for example, we must select a slot within a slot within a slot. The following code, for example, selects the first polygon of `lnd` and then selects the first Polygon within this spatial unit (there is usually only one) and then returns the coordinates of this. Note the plot in Figure 4 has an incorrect projection.

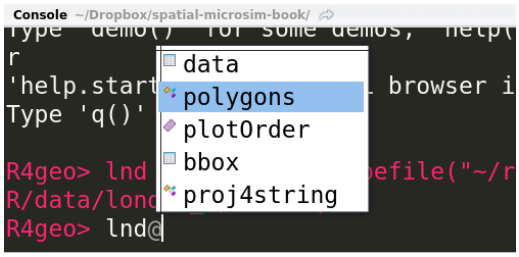

Figure 3: Tab-autocompletion in action: display from RStudio after typing `lnd@` then `tab` to see which slots are in `lnd`

```
head(lnd@polygons[[1]]@Polygons[[1]]@coords, 3)
```

```
##          [,1]     [,2]
## [1,] 541177.7 173555.7
## [2,] 541872.2 173305.8
## [3,] 543441.5 171429.9
```

```
plot(lnd@polygons[[1]]@Polygons[[1]]@coords)
```

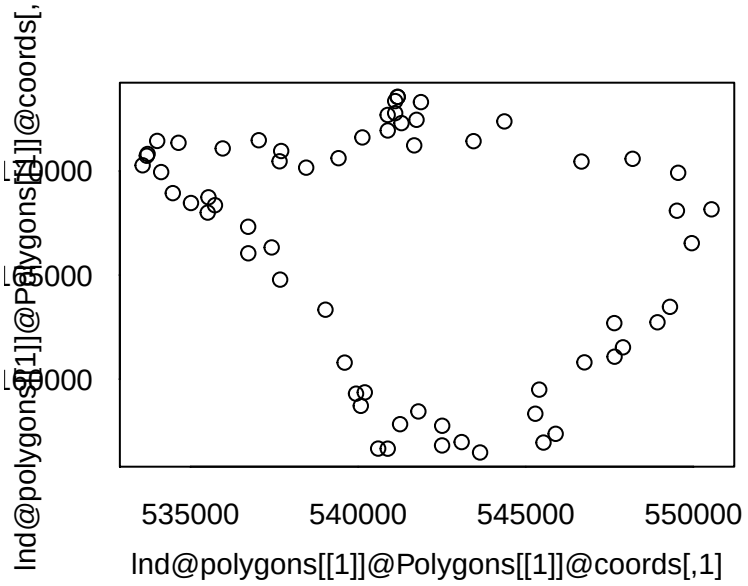

Figure 4: Raw coordinate data displayed and plotted in R.

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@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` object. To explore `lnd` object further, try typing `nrow(lnd)` and record how many zones the dataset contains. You can also try `ncol(lnd)`.

Basic plotting

Now we have seen something of the structure of spatial objects in R, let us look at plotting them. Note, that plots use the *geometry* data, contained primarily in the `@polygons` slot in the above example, to draw the

shape of the zones. The information in `@data` will typically be used to colour in the map. which describes where the polygons are located in space:

```
plot(lnd) # 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@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@data where sports participation is less than 15
lnd@data[lnd$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` 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@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` 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.

```
# Select zones where sports participation is between 20 and 25%
sel <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[sel, ]) # output not shown here
head(sel) # test output of previous selection (not shown)
```

This plot is could be 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 do you think the `col` argument refers to in the below block? (see Figure 5).

If you wish to experiment with multiple criteria queries, use `&`.

```
plot(lnd, col = "lightgrey") # plot the london_sport object
sel <- lnd$Partic_Per > 25
plot(lnd[ sel, ], col = "turquoise", add = TRUE) # add selected zones to map
```

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.

As a bonus stage, select and plot only zones that are close to the centre of London (see Fig. 6). Programming encourages rigorous thinking and it helps to define the problem more specifically:

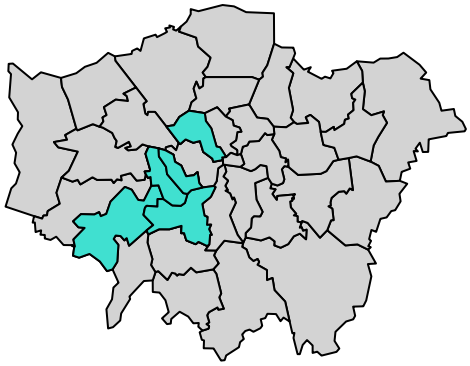

Figure 5: Simple plot of London with areas of high sports participation highlighted in blue

Challenge: Select all zones whose geographic centroid lies within 10 km of the geographic centroid of inner London.[2]

Figure 6: Zones in London whose centroid lie within 10 km of the geographic centroid of the City of London. Note the distinction between zones which only touch or 'intersect' with the buffer (light blue) and zones whose centroid is within the buffer (darker blue).

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)`). Let's take a look at the headings of sport, using the following command: `names(lnd)` Remember, the attribute data contained in spatial objects are kept in a 'slot' that can be accessed using the `@` symbol: `lnd@data`. This is useful if you do not wish to work with the spatial components of the data at all times.

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

```
summary(lnd)
```

```
## Object of class SpatialPolygonsDataFrame
```

[2]To see how this map was created, see the code in intro-spatial.Rmd. This may be loaded by typing `file.edit("intro-spatial.Rmd")` or online at github.com/Robinlovelace/Creating-maps-in-R/blob/master/intro-spatial.Rmd.

```
## 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 first line of the above output tells us that `lnd` is a special spatial class, in this case a `SpatialPolygonsDataFrame`. This means it is composed of various polygons, each of which has attributes. (Summaries of the attribute data are also provided, but are not printed above.) This is the ideal class of data to represent 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: Creating and manipulating spatial data

Beyond visualisation and interrogation GIS software must also be able to create and modify spatial data. R excels in this regard. R's spatial packages provide a very wide and powerful suite of functionality for processing spatial data, and its strict class system make it easy to create new spatial datasets.

Reprojecting and *joining/clipping* are fundamental operations in the GIS analyst's tool-kit, so we introduce these, before looking at joining non-spatial data to spatial objects. Finally we cover spatial joins, whereby information from two spatial objects is combined based on spatial location.

Creating new spatial data

R objects can be created by entering the name of the class we want to make. `vector` and `data.frame` objects for example, can be created as follows:

```
vec <- vector(mode = "numeric", length = 3)
df <- data.frame(x = 1:3, y = c(1/2, 2/3, 3/4))
```

We can check the class of these new objects using `class()`:

```
class(vec)
```

```
## [1] "numeric"
```

```
class(df)
```

```
## [1] "data.frame"
```

The same logic applies to spatial data, but the input requirements are stricter. To create a `SpatialPoints` object, for example, the input coordinates *must* be supplied in a matrix:

```
mat <- as.matrix(df) # create matrix object with as.matrix
sp1 <- SpatialPoints(coords = mat)
```

Note in the code above the use of `as.matrix()` to change the class of an existing object. We then create a spatial points object, one of the fundamental data types for spatial data. (The others are lines, polygons and pixels, which can be created by `SpatialLines`, `SpatialPolygons` and `SpatialPixels`, respectively.) Each type of spatial data has a corollary that can accepts non-spatial data, created by adding `DataFrame`. `SpatialPointsDataFrame()`, for example, creates points with an associated `data.frame`. The number of rows in this dataset must equal the number of features in the spatial object, which in the case of `sp1` is 3.

```
class(sp1)
```

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

```
spdf <- SpatialPointsDataFrame(sp1, data = df)
class(spdf)
```

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

The above code extends the pre-existing object `sp1` by adding data from `df`. To see how strict spatial classes are, try replacing `df` with `mat` in the above code: it causes an error. All spatial data classes can be created in a similar way, although `SpatialLines` and `SpatialPolygons` are much more complicated (Bivand et al. 2013). More frequently your spatial data will be read-in from an externally-created file, e.g. using `readOGR()`. Unlike the spatial objects we created above, most spatial data comes with an associate 'CRS'.

Projections: setting and transforming CRS in R

The *Coordinate Reference System* (CRS) of spatial objects defines where they are placed on the Earth's surface. You may have noticed '`proj4string` 'in the summary of `lnd` above: the information that follows represents its CRS. Spatial data should always have a CRS. If no CRS information is provided, and the correct CRS is known, it can be set as follow:

```
proj4string(lnd) <- NA_character_ # remove CRS information from lnd
proj4string(lnd) <- CRS("+init=epsg:27700") # assign a new CRS
```

R issues a warning when the CRS is changed. This is so the user knows that they are simply changing the CRS, not *reprojecting* the data. An easy way to refer to different projections is via EPSG codes.

Under this system `27700` represents the British National Grid. 'WGS84' (`epsg:4326`) is a very commonly used CRS worldwide. The following code shows how to search the list of available EPSG codes and create a new version of `lnd` in WGS84:[3]

```
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
```

```
lnd84 <- spTransform(lnd, CRS("+init=epsg:4326")) # reproject
```

Above, `spTransform` converts the coordinates of `lnd` into the widely used WGS84 CRS. Now we've transformed `lnd` into a more widely used CRS, it is worth saving it. R stores data efficiently in `.RData` or `.Rds` formats. The former is more restrictive and maintains the object's name, so we use the latter.

```
# Save lnd84 object (we will use it in Part IV)
saveRDS(object = lnd84, file = "data/lnd84.Rds")
```

Now we can remove the `lnd84` object with the `rm` command. It will be useful later. (In RStudio, notice it also disappears from the Environment in the top right panel.)

```
rm(lnd84) # remove the lnd object
```

[3] Note: entering `projInfo()` provides additional CRS options. spatialreference.org provides more information about EPSG codes.

Attribute joins

Attribute joins are used to link additional pieces of information to our polygons. In the `lnd` object, for example, we have 4 attribute variables — that can be found by typing `names(lnd)`. 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:

```
library(rgdal) # ensure rgdal is loaded
# Create new object called "lnd" from "london_sport" shapefile
lnd <- readOGR(dsn = "data", "london_sport")
plot(lnd) # plot the lnd object (not shown)
nrow(lnd) # return the number of rows (not shown)
```

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". If you open the file in a separate spreadsheet application first, we can see 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",
  stringsAsFactors = FALSE)

head(crime_data, 3) # display first 3 lines
head(crime_data$CrimeType) # information about 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, save result
crime_ag <- aggregate(CrimeCount ~ Borough, FUN = sum, data = crime_theft)
# Show the first two rows of the aggregated crime data
head(crime_ag, 2)
```

You should not expect to understand all of this upon first try: simply typing the commands and thinking briefly about the outputs is all that is needed at this stage. 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 when we read in the file we specify its location (check in your file browser to be sure).
- The `==` function is used to select only those observations that meet a specific condition i.e. where it is equal to, 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's 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  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  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$name` are also contained in the Borough names of the aggregated crime data. The results indicate that all but one of the borough names matches. The second line of code tells us that it is 'City of London'. This does not exist in the crime data. This may be because the City of London has it's own Police Force.[4] (The borough name in the crime data does not match `lnd$name` is 'NULL'. Check this by typing `crime_ag$Borough[!crime_ag$Borough %in% lnd$name]`.)

Challenge: identify the number of crimes taking place in borough 'NULL', less than 4,000.

Having checked the data found that one borough does not match, we are now ready to join the spatial and non-spatial datasets. It is recommended to use the `left_join` function from the **dplyr** package 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 remain 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 to 'name'
# head(left_join(lnd@data, crime_ag)) # test it works
lnd@data <- left_join(lnd@data, crime_ag)
```

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

```
## Warning in left_join_impl(x, y, by$x, by$y): joining character vector and
## factor, coercing into character vector
```

[4] See www.cityoflondon.police.uk/.

Take a look at the new `lnd@data` object. You should see new variables added, meaning the attribute join was successful. Congratulations! You can now plot the rate of theft crimes in London by borough (see Fig 8).

```
library(tmap) # load tmap package (see Section IV)
qtm(lnd, "CrimeCount") # plot the basic map
```

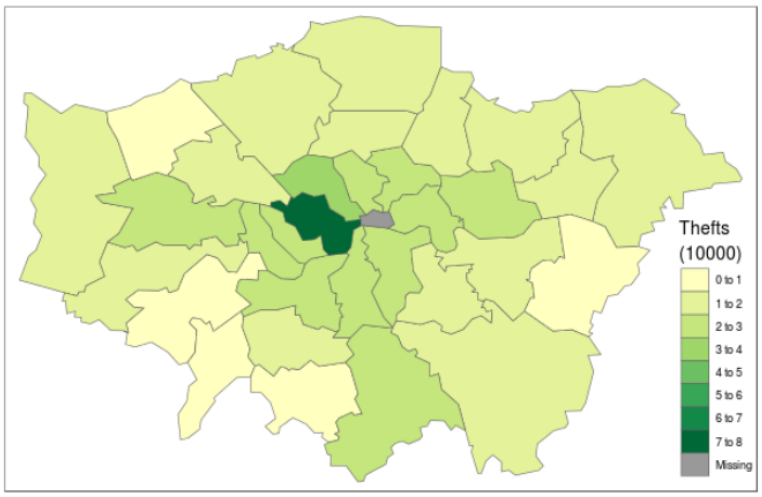

Figure 7: Number of thefts per borough.

Optional 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.london.gov.uk website.

Using this dataset and the methods developed above, Figure 8 was created: the proportion of council seats won by the 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).[5]

Clipping and spatial joins

In addition to joining by attribute (e.g. Borough name), it is also possible to do spatial joins in R. There are three main types: 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. We will be conducting a many-to-one spatial join and 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.

```
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
```

[5]Hint: the solution relies on the **rgeos** function `gCentroid()`. To see how this map was created, see the code in intro-spatial.Rmd at github.com/Robinlovelace/Creating-maps-in-R/blob/master/intro-spatial.Rmd.

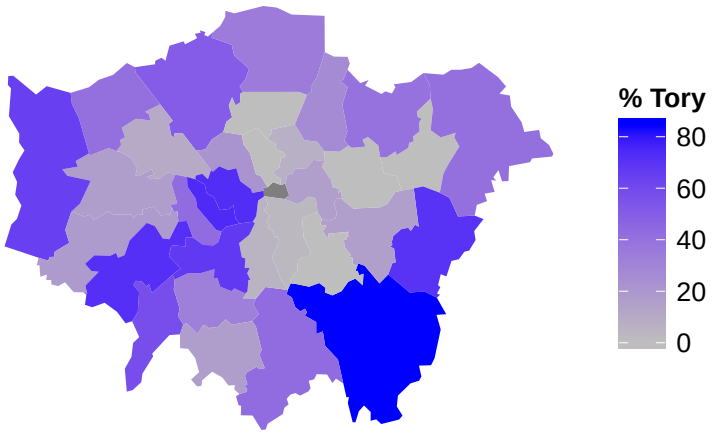

Figure 8: Proportion of council seats won by Conservatives in the 2014 local elections using data from data.london.gov and joined using the methods presented in this section

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 'stations' 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 9)
points(stations) # overlay the station points
```

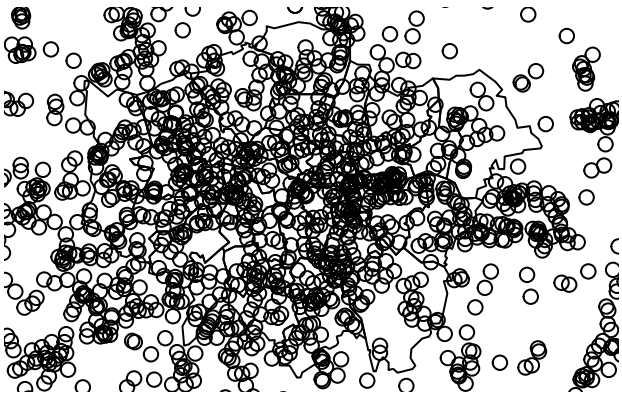

Figure 9: Sampling and plotting stations

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 create 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:

```
stations_backup <- stations # backup the stations object
stations <- stations_backup[lnd, ]
plot(stations) # test the clip succeeded (see Figure 10)
```

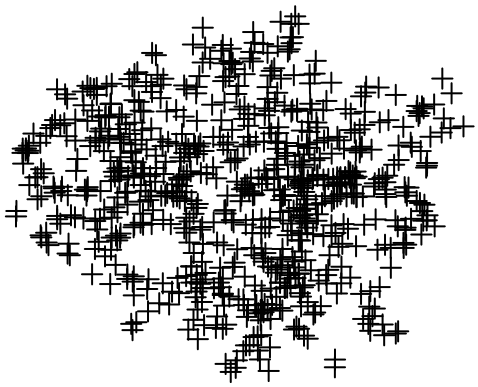

Figure 10: The clipped stations dataset

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_backup, lnd)
stations2 <- stations_backup[!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.

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

```
##   CODE
## 0   54
```

```
## 1   22
## 2   43
## 3   18
## 4   12
## 5   12
```

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.[6]

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

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

```
brks <- quantile(lnd_n$NUMBER)
labs <- grey.colors(n = 4)
q <- cut(lnd_n$NUMBER, brks, labels = labs,
  include.lowest = T)
summary(q) # check what we've created
```

```
## #4C4C4C #969696 #C3C3C3 #E6E6E6
##       9       8       8       8
```

```
qc <- as.character(q) # convert to character class to plot
plot(lnd_n, col = qc) # plot (not shown in printed tutorial)
legend(legend = paste0("Q", 1:4), fill = levels(q), "topright")
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 Fig. 11). 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

[6] See the miniature Vignette 'Clipping and aggregating spatial data with gIntersects' for more information on this: http://rpubs.com/RobinLovelace/83834 .

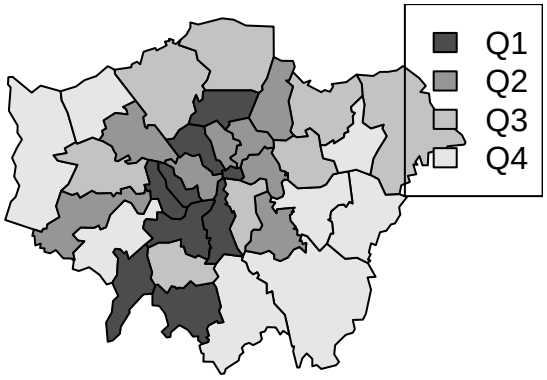

Figure 11: Choropleth map of mean values of stations in each borough

Imagine now that we want to 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, try adding the following code to the chunk above (output not shown):

```
levels(stations$LEGEND) # see A roads and rapid transit stations (RTS) (not shown)
sel <- grepl("A Road Sing|Rapid", stations$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))
```

The above block of code first identifies which types of transport points are present in the map with `levels` (this command only works on factor data). 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 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 unintuitive way of altering map graphics compared with something like QGIS. That's because it is! It may not be worth stressing too much about base graphics because there are intuitive alternatives for quickly creating beautiful maps in R, including **tmap**, **ggplot** and **leaflet**.

Part IV: Making maps with tmap, ggplot2 and leaflet

This fourth part introduces alternative methods for creating maps with R that overcome some of the limitations and clunkiness of R's base graphics. Having worked hard to manipulate the spatial data, it is now time to display the results clearly, beautifully and, in the case of **leaflet**, interactively.

We try the newest of the three packages, **tmap**, first as it is the easiest to use. Then we progress to explore the powerful **ggmap** paradigm before outlining **leaflet**.

tmap

tmap was created to overcome some of the limitations of base graphics and **ggmap**. The focus is on generating 'thematic maps' (which show a variable change over space) quickly and easily. A concise introduction to **tmap** can be accessed (after the package is installed) by using the vignette function:

```
library(tmap)
```

```
## Loading required package: raster
##
## Attaching package: 'raster'
##
## The following object is masked from 'package:dplyr':
##
##     select
```

```
vignette(package = "tmap") # available vignettes in tmap
vignette("tmap-nutshell")
```

```
## starting httpd help server ... done
```

A couple of basic plots show the package's intuitive syntax and attractive default parameters.

```
# Create our first tmap map (not shown)
qtm(shp = lnd, fill = "Partic_Per", fill.palette = "-Blues")
```

```
qtm(shp = lnd, fill = c("Partic_Per", "Pop_2001"), fill.palette = c("Blues"), ncol = 2)
```

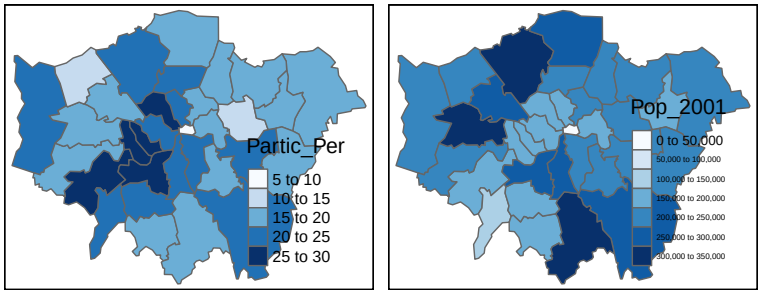

Figure 12: Side-by-side maps of crimes and % voting conservative

The plot above shows the ease with which tmap can create maps next to each other for different variables. Figure 13 illustrates the power of the `tm_facets` command.

```
tm_shape(lnd) +
    tm_fill("Pop_2001", thres.poly = 0) +
tm_facets("name", free.coords=TRUE, drop.shapes=TRUE) +
    tm_layout(legend.show = FALSE, title.position = c("center", "center"), title.size = 20)
```

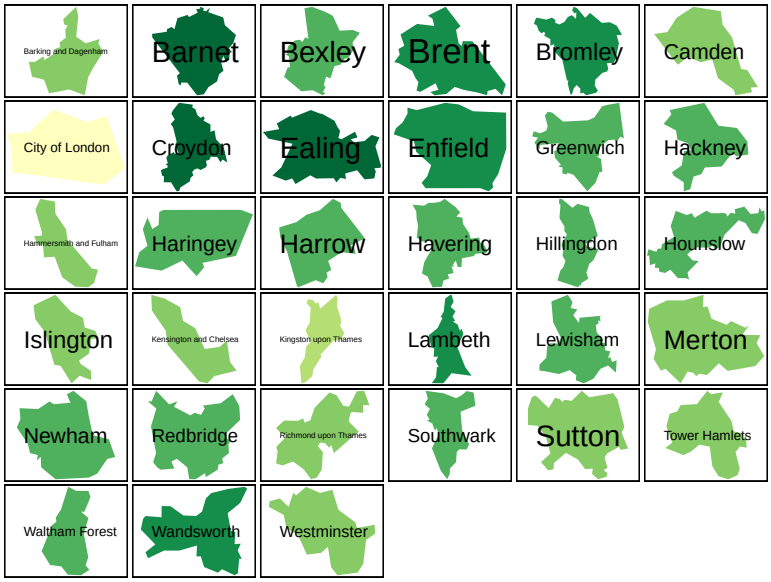

Figure 13: Facetted map of London Boroughs created by tmap

A more established plotting paradigm is **ggplot2** package. This has been subsequently adapted to maps thank to **ggmap**, as described below.

ggmap

ggmap is based on the **ggplot2** package, an implementation of the Grammar of Graphics (Wilkinson 2005). **ggplot2** can replace the base graphics in R (the functions you have been plotting with so far). It contains default options that match good visualisation practice.

ggplot2 is one of the best documented packages in R. This documentation can be found on-line and it is recommended you test out the examples 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 previously. 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()
```

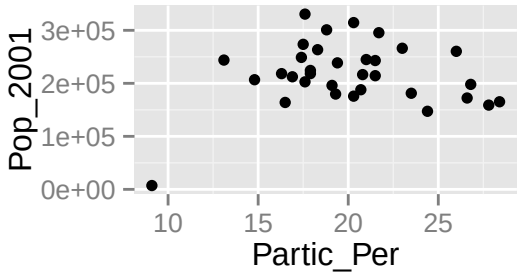

Figure 14: A simple graphic produced with **ggplot2**

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))
```

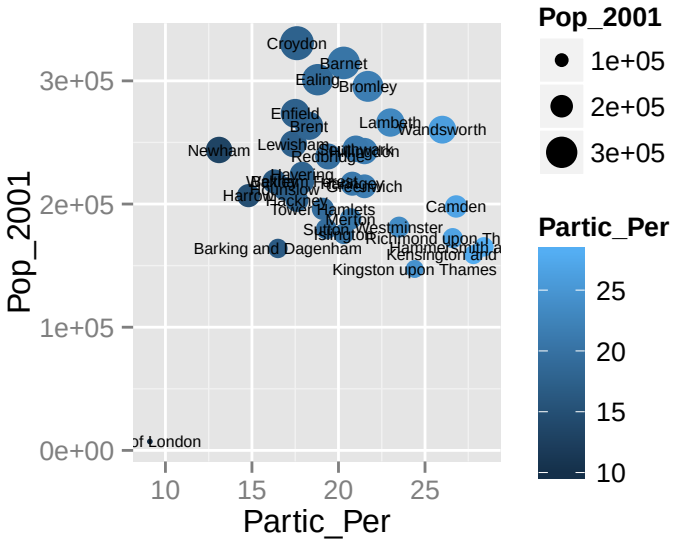

Figure 15: ggplot for text

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).

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

ggmap requires spatial data to be supplied as `data.frame`, using `fortify()`. The generic plot() function can use `Spatial*` objects directly; **ggplot2** cannot. Therefore we need to extract them as a data frame. The fortify function was written specifically for this purpose. For this to work, either the **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 use this function, ensure you have the dplyr package installed and loaded. To find out how it works look at the output of typing `?left_join`. As usual the help documentation is a little sparse here, so let's consider what it means in more detail. The help document explains that `left_join` retains all rows from the original dataset `x`, whereas `inner_join` only retains rows in `x` which have a match in `y`. If there is no match in the joining variable, `inner_join` simply adds `NA` values for the new variables for the affect rows.

```
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 conservative
## 1      00AF              Bromley       21.7   295535      15172           85
## 2      00BD Richmond upon Thames       26.6   172330       9715           72
##   n_points id
## 1       54  0
## 2       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 coordinates (Easting and Northing points as the data are in British National Grid format) 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]
```

```
##       long      lat order  hole piece group id ons_label
## 1 541177.7 173555.7     1 FALSE     1   0.1  0      00AF
## 2 541872.2 173305.8     2 FALSE     1   0.1  0      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 the one shown below. Try changing the colours.

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

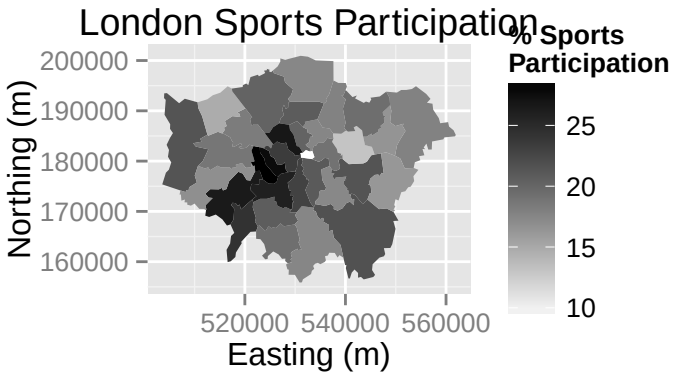

Figure 16: Greyscale map

To save a plot image 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 `lnd84` object created in the reprojection operation (see Part III). Load this with `readRDS()`.

```
lnd84 <- readRDS("data/lnd84.Rds") # load previously saved object
```

The first job is to calculate the bounding box (bb for short) of the `lnd84` object to identify the geographic extent of the image tiles that we need.

```
bb <- bbox(lnd84)
b <- (bb - rowMeans(bb)) * 1.05 + rowMeans(bb)
# 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 `lnd84` object and then merge with the required attribute data (we already did this step to create the `lnd_f` object).

```
lnd_wgs84_f <- fortify(lnd84, region = "ons_label")
lnd_wgs84_f <- left_join(lnd_wgs84_f, lnd84@data,
  by = c("id" = "ons_label"))
```

```
## Warning in left_join_impl(x, y, by$x, by$y): joining factor and character
## vector, coercing into character vector
```

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 many parameters. Use the **ggplot2** help pages (e.g. `?geom_polygon`) to find out what they are. The resulting map could be improved with a simpler basemap in black and white. Stamen provide the tiles we need and they can be brought into the plot with the `get_map` function:[7]

```
# 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)
```

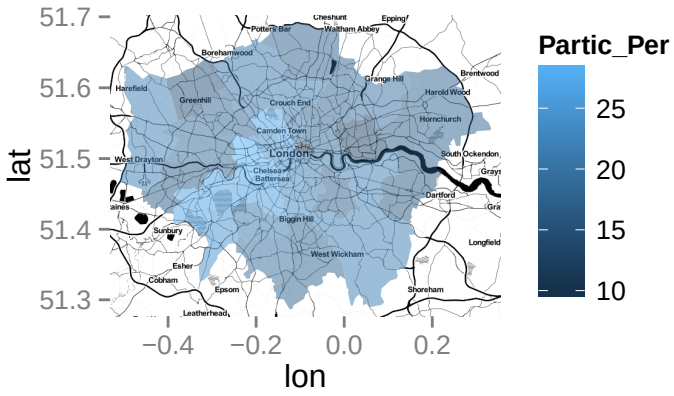

Figure 17: Basemap 2

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)
```

Creating interactive maps with leaflet

Leaflet is probably the world's premier web mapping system, serving hundreds of thousands of maps worldwide each day. The JavaScript library actively developed at github.com/Leaflet/Leaflet, has a strong user community. It is fast, powerful and easy to learn.

A recent package developed by RStudio, called **leaflet**[8] provides R bindings to Leaflet. **rstudio/leaflet** allows the creation and deployment of interactive web maps in few lines of code. One of the exciting things about the package is its tight integration with the R package for interactive on-line visualisation, **shiny**. Used together, these allow R to act as a complete map-serving platform, to compete with the likes of GeoServer! For more information on **rstudio/leaflet**, see rstudio.github.io/leaflet/ and the following on-line tutorial: robinlovelace.net/r/2015/02/01/leaflet-r-package.html.

```
install.packages("leaflet")
library(leaflet)

leaflet() %>%
  addTiles() %>%
  addPolygons(data = lnd84)
```

[7]Note that a wide range of customised on-line basemaps can be used, by modifying 'cloudmade' map styles or customised Google Maps - see r-bloggers.com/creating-styled-google-maps-in-ggmap for details on the latter.

[8]We also refer to this as **rstudio/leaflet**, after the project's on-line home at github.com/rstudio/leaflet, to avoid confusion with the JavaScript library.

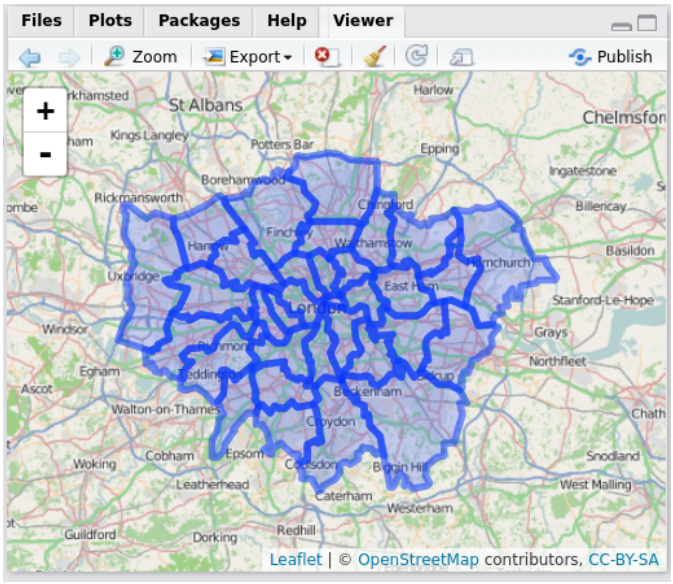

Figure 18: The lnd84 object loaded in rstudio via the leaflet package

Advanced Task: Faceting for Maps

The below code demonstrates how to read in the necessary data for this task and 'tidy' it up. The data file contains historic population values between 1801 and 2001 for London, again from the London data store.

We `tidy` the data so that the columns become rows. In other words, we convert the data from 'flat' to 'long' format, which is the form required by **ggplot2** for faceting graphics: the date of the population survey becomes 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
```

```
##
## Attaching package: 'tidyr'
##
## The following object is masked from 'package:raster':
##
##     extract
```

```
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
```

```
##       long      lat order  hole piece group id ons_label    name Partic_Per
## 1 541177.7 173555.7     1 FALSE     1   0.1  0      00AF Bromley       21.7
## 2 541872.2 173305.8     2 FALSE     1   0.1  0      00AF Bromley       21.7
##   Pop_2001 CrimeCount conservative n_points
## 1   295535      15172           85       54
## 2   295535      15172           85       54
```

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

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

```
## Warning in left_join_impl(x, y, by$x, by$y): joining factor and character
## vector, coercing into character vector
```

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 (Fig. 19).

```
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 = "blue", 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
```

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.

> **Challenge 1:** Try creating this plot for the % of population instead of the absolute population.

> **Challenge 2:** For bonus points, try creating an animation of London's evolving population over time using the excellent **animation** package.

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.

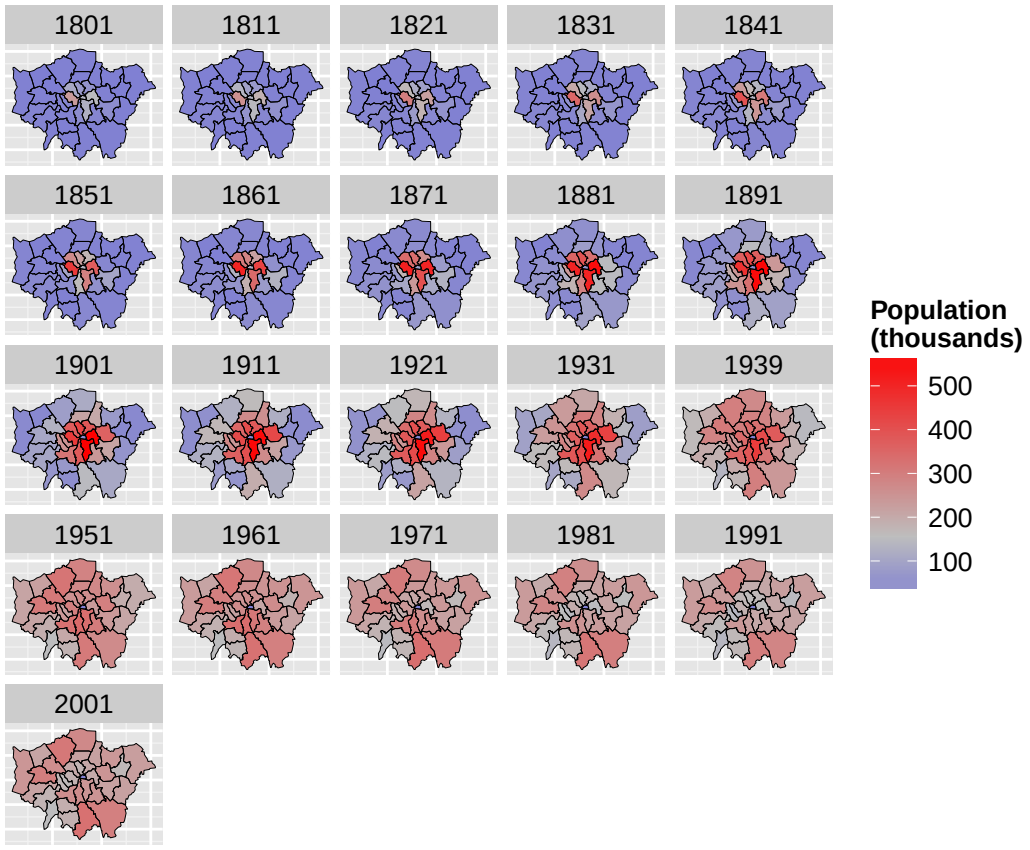

Figure 19: Faceted plot of the distribution of London's population over time

If you enjoyed this tutorial, you may find the book chapter "Spatial Data Visualisation with R" of interest (Cheshire and Lovelace, 2014). The project's repository 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.

Other advanced tutorials include

- "Using spatial data" from the useR 2013 conference page
- "Geographically Weighted PCA", an in-depth and advanced tutorial by Chris Brunsdon -"The Spatial Autoregression Regression Model and Correlation", another advanced tutorial, also by Chris Brunsdon
- "Using R as a GIS", a long and detailed online tutorial by Francisco Rodriguez-Sanchez.
- "solaR: Solar Radiation and Photovoltaic Systems with R", a technical academic paper on the solaR package which contains a number of spatial function (.

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 on-line 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 on-line R community:

- R's homepage hosts a wealth of official and contributed guides. http://cran.r-project.org
- StackOverflow and GIS.StackExchange groups (the "[R]" search term limits the results). If your question has not been answered yet, just ask, preferably with a reproducible example.
- R's mailing lists, especially R-sig-geo. See r-project.org/mail.html.

Books: despite the strength of R's on-line community, nothing beats a physical book for concentrated learning. We recommend the following:

- Everitt and Hothorn (updated 2015): This handbook of statistical analyses is available online as an R package.
- Dorman (2014): detailed exposition of spatial data in R, with a focus on raster data. A free sample of this book is available online.
- Wickham (2009): 'ggplot2: elegant graphics for data analysis'
- Bivand et al. (2013) : 'Applied spatial data analysis with R' - provides a dense and detailed overview of spatial data analysis.
- Kabacoff (2011): 'R in action' is a general R book with 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).

Acknowledgements

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, Hannah Roberts and Phil Jones who helped develop and demonstrate these materials. 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 https://raw.githubusercontent.com/Robinlovelace/Creating-maps-in-R/master/citation.bib or here.

References

Bivand, R. S., Pebesma, E. J., & Rubio, V. G. (2013). Applied spatial data analysis with R. Springer. 2nd ed.

Cheshire, J., & Lovelace, R. (2015). Spatial data visualisation with R. In C. Brunsdon & A. Singleton (Eds.), Geocomputation (pp. 1–14). SAGE Publications. Retrieved from https://github.com/geocomPP/sdv . Full chapter available from https://www.researchgate.net/publication/274697165_Spatial_data_visualisation_with_R

Dorman, M. (2014). Learning R for Geospatial Analysis. Packt Publishing Ltd.

Everitt, B. S., & Hothorn, T. (2015). HSAUR: A Handbook of Statistical Analyses Using R. Retrieved from http://cran.r-project.org/package=HSAUR

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.

Lamigueiro, O. P. (2012). solaR: Solar Radiation and Photovoltaic Systems with R. Journal of Statistical Software, 50(9), 1–32. Retrieved from http://www.jstatsoft.org/v50/i09

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.