3

HANDLING SPATIAL DATA IN R

3.1 OVERVIEW

The aim of this chapter is provide an introduction to the mapping and geographical data handling capabilities of R. It explicitly focuses on developing building blocks for the spatial data analyses in later chapters. These extend the mapping functionality that was briefly introduced in the previous chapter and that will be extended further in Chapter 5. It includes an introduction to the `GISTools` package and its functions, describes methods for producing choropleth maps – from basic to quite advanced outputs – and introduces some methods for generating descriptive statistics. These skills are fundamental to the analyses that will be developed in later in this book. This chapter will:

- Introduce the `GISTools` package
- Describe how to compile maps based on multiple layers
- Describe how to set different shading schemes
- Describe how to plot spatial data with different parameters
- Describe how to develop basic descriptive statistical analyses of spatial data

3.2 INTRODUCTION: GISTools

The previous chapters introduced some basic analytical and graphical techniques using R. However, few of these were particularly geographical. A number of packages are available in R that allow sophisticated visualisation, manipulation and analysis of spatial data. Some of this functionality will be demonstrated in this chapter in conjunction with some mapping tools and specific data types to create different examples of mapping in R. Remember that a package in R is a set of prewritten functions (and possibly data items as well) that are not available when you initially start R running, but can be loaded from the R library at the command line. To illustrate these techniques, the chapter starts by developing some elementary maps, building to more sophisticated mapping.

3.2.1 Installing and Loading `GISTools`

You will use different methods and tools contained within the `GISTools` package to draw maps and to handle spatial information. You should have installed the `GISTools` package onto your computer as you ran the code in Chapter 2 using the `install.packages` command. Once you have downloaded and installed a package on your computer, you can simply load the package when you use R subsequently. To load `GISTools` into the R session that you have just started, simply enter:

```
library(GISTools)
```

It is possible to inspect the functionality and tools available in `GISTools` or any other package by examining the documentation:

```
help(GISTools)
```

or

```
?GISTools
```

This provides a general description of the package. At the bottom of the help window, there is a hyperlink to the index which, if you click on it, will open a page with a list of all the tools available in the package. The CRAN website also has full documentation for each package – for `GISTools`, see `http://cran.r-project.org/web/packages/GISTools/index.tml`.

3.2.2 Spatial Data in `GISTools`

`GISTools`, similar to many other R packages, comes with a number of embedded datasets that can be loaded from the command line after the package is installed. Two datasets will be used in this chapter: polygon and line data for New Haven, Connecticut, and counties in the state of Georgia, both in the USA. The New Haven data include crime statistics, roads, census blocks (including demographic information), railway lines and place names. The data come from two sources, both of which have made the data freely available. The crime data are obtained from the New Haven Crime Log website (`http://www.newhavencrimelog.org`) provided by the *New Haven Independent* newspaper (`http://www.newhavenindependent.org`) – the data may be extracted from the HTML source code of the crime map web pages. The remaining data are obtained from the University of Connecticut's Map and Geographical Information Center (MAGIC: `http://magic.lib.uconn.edu/`). These data can be downloaded in MapInfo MIF or ESRI E00 formats, and with a public domain program called `ogr2ogr` it is possible to convert them into ESRI Shapefiles. The Georgia data include outlines of counties in Georgia (from `http://www.census.gov/geo/`) with a number of attributes relating to the 1990 census including population (`TotPop90`), the percentage of the population that are rural (`PctRural`),

that have a college degree (PctBach), that are elderly (PctEld), that are foreign born (PctFB), that are classed as being in poverty (PctPov), that are black (PctBlack) and the median income of the county (MedInc). The two datasets are shown in Figure 3.1.

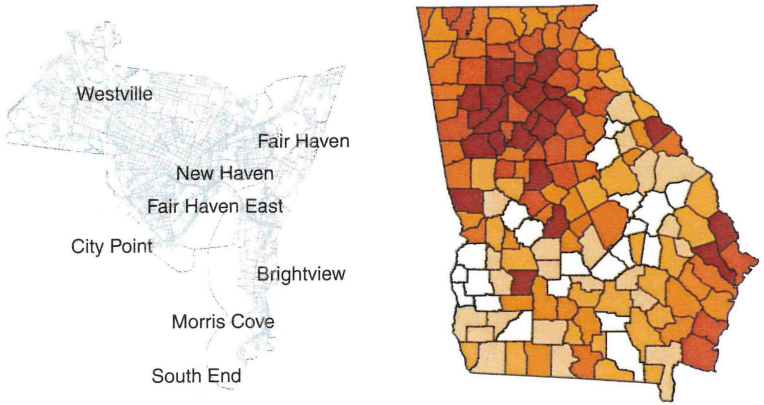

Figure 3.1 The New Haven census blocks with roads in blue, and the counties in the state of Georgia shaded by median income

The first thing you should do with any dataset is examine it. For spatial data, this often means a visual examination of its spatial properties and perhaps a more formal consideration of its attributes. Initially you will use the New Haven datasets to draw your first map in R. Load the New Haven data:

```
data(newhaven)
```

To examine the data that have been loaded enter:

```
ls()
```

```
## [1] "blocks"   "breach"   "burgres.f"     "burgres.n"
## [5] "famdisp" "places"   "roads"         "tracts"
```

This gives the printout above (blocks, breach, roads,...) showing all of the datasets that are loaded. A number of these, including the data called roads, are geographical data. Enter:

```
plot(roads)
```

and a map of roads in New Haven appears.

> The plot command has been used earlier for graphics - but now, after loading the GISTools package, R has learned a new plot method to apply when the term between the brackets refers to geographical data. This is an example of defining and using classes in R as described in Chapter 2.

To determine the class of the roads dataset, enter:

```
class(roads)
```

This shows that roads is a variable of class SpatialLinesDataFrame, defined in the sp package that was automatically loaded in with GISTools. You should investigate the class of blocks, tracts and breach in the same way. The sp package defines a number of classes as summarised in the table below.

Without Attributes	With attributes	ArcGIS Equivalent
SpatialPoints	SpatialPointsDataFrame	Point shapefiles
SpatialLines	SpatialLinesDataFrame	Line shapefiles
SpatialPoints	SpatialPolygonsDataFrame	Polygon shapefiles

So, for example, the breach data are a SpatialPoints class that simply describes locations, with no attributes, whereas the blocks data are of the SpatialPolygonsDataFrame class as they include some census variables associated with each census block. Thus spatial data with attributes defined in this way hold their attributes in the data.frame and you can see this by looking at the first few lines of the blocks data.frame using the head function:

```
head(data.frame(blocks))
```

This prints the first six lines of attributes associated with the census blocks data. A formal consideration of spatial attributes and how to analyse and map them is given later in this chapter. The census blocks in New Haven can be plotted:

```
plot(blocks)
```

Now suppose we want to plot another variable with the census blocks which shows the roads for the area. Entering:

```
plot(roads)
```

will draw a map of the roads. However, one problem is that this has now overwritten the blocks that were drawn before. To stop this from happening, an extra parameter called `add` can be included in the second plot call to ensure that the first set of data that was plotted is not overwritten. This overlays (rather than overwrites) the information. The `add=TRUE` part sets a parameter in the `plot` command to the logical value `TRUE`, instructing R not to overwrite, but to add the new information to the existing plot. A number of other parameters can also be included. For example, enter:

```
par(mar = c(0,0,0,0))
plot(blocks)
plot(roads, add=TRUE, col="red")
```

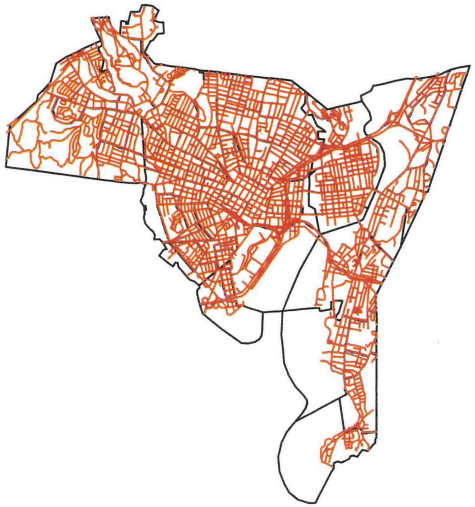

Figure 3.2 The New Haven census blocks and road data

The first line plots the `blocks` data. Again, because there are no other parameters, this starts a new plot. The next line adds the `roads` data, but the extra parameter tells R to draw them in red. Note that where quotes are needed these can be either single or double quotes, but not mixed and not formatted into curly quotes. Your map should look like the one in Figure 3.2.

The `plot` commands above generates maps of the census blocks and the roads in New Haven, regardless of whether the data describe area or linear features. Those familiar with GIS will recognise these as two of the commonly referred to vector features often cited in a GIS context of 'points, lines and areas'. In R, the command `plot` can also be used to plot point features. For example:


```
plot(blocks)
plot(breach, col = "red", add = TRUE)
```

This draws a map of locations where breaches of the peace have occurred in New Haven, overlaying the census blocks. Note that the default plot character is a cross and that some areas have a greater density of incidents than others. Remember from Chapter 2 that `pch` can be used to change the plot character for graphs. It can be used in the same way to map point spatial data. You should experiment with different `pch` and colour settings to get the best visualisation of the breaches of the peace and the census blocks data together. For example:

```
plot(blocks, lwd = 0.5, border = "grey50")
plot(breach, col = "red", pch = 1, add = TRUE)
```

It is possible to display a list of colours by entering:

```
colors()
```

And remember that help for `plot` describes many plot options, including different plot characters.

3.2.3 Embellishing the Map

You have now drawn your first maps in R showing the roads, breaches of peace and census blocks in New Haven. However, maps generally need more information than this. In particular, someone who had no prior knowledge of New Haven may not realise the geographical extent (in miles or kilometres) of the area. Assuming you still have the map window from the previous section, you can add a scale using the `map.scale` command. This command has a large number of parameters. Enter:

```
map.scale(534750,152000,miles2ft(2),"Miles",4,0.5)
```

A scale bar is drawn (overplotted) on the map. The first two parameters are the location at which the scale bar appears, in the coordinate system of the map. Currently, this is in US survey feet, using the State Plane Coordinate System for Connecticut. The coordinates specify the centre of the scale. The third parameter is the length (in projected map units) of the scale. In this case it is 2 miles. Since this distance also has to be specified in the coordinate system of the map, this quantity must be converted into feet, using the `miles2ft` function. The fourth parameter is a text string specifying the name of the units for the scale (miles in this case). The fifth parameter gives the number of gradations in the scale, and the sixth gives the fraction of the units that each gradation on the scale represents. Here, there are four gradations, and each one represents 0.5 miles.

A second embellishment is a north-pointing arrow. To add this, enter:

```
north.arrow(534750,154000,miles2ft(0.25),col= "lightblue")
```

This adds (overplots) the north arrow to the map. Again, the first two parameters here specify the location of the arrow: they give the coordinates, in map units, of the centre of the base of the arrow. The third parameter specifies the length of the arrow's base (here 0.25 miles in map units) and the `col` parameter specifies the colour that the arrow is filled with. If left unspecified, the arrow will be filled in white. Remember the `locator()` function introduced in Chapter 2 – this can be very useful for determining where to place items such scale bars and north arrows in plot windows. A final decoration to the map is a map title. Suppose the map is to be called 'New Haven, CT.'. Then enter the code below to add the title to the map.

```
title('New Haven, CT.' )
```

It should now look like the map in Figure 3.3.

Figure 3.3 An embellished map of the New Haven census blocks and breaches of the peace

3.2.4 Saving Your Map

Having created a map in a window on the screen, you may now want to save the map for either printing, or incorporating in a document. There are a number of

ways that this can be done. The simplest is to right-click with the mouse on the map window, select **copy** or **copy image to clipboard** (Mac) or **Copy as metafile** or **Copy as bitmap** (Windows), and then paste it into a word-processing document (for example, one being created in either OpenOffice or MS Windows). Another is to highlight the window and to use **File > Save as** to save the map as an image file, with a name that you give it. However, it is also possible to save images by using the R commands that were used to create the map. This takes more initial effort, but has the advantage that it is possible to make minor changes (such as altering the position of the scale, or drawing the census block boundaries in a different colour) and to easily rerun the code. Finally, it is also possible to save your map in a number of file formats, such as PDF or PNG.

One way to create a file of commands is to edit a text file with a name ending in .R – note the capital letter. In R, open a new document on a Windows machine by going to File > New script and **File > New script,** or on a Mac by selecting **File > New Document**. Then type in the following:

```
# load package and data
library(GISTools)
data(newhaven)
# plot spatial data
plot(blocks)
plot(roads,add=TRUE,col= 'red')
# embellish the map
map.scale(534750,152000,miles2ft(2), "Miles",4,0.5)
north.arrow(530650,154000,miles2ft(0.25),col= 'lightblue')
title('New Haven, CT')
```

Save the file as 'newhavenmap.R' in the directory in which you started up R.

> When you start an R session you should set the working directory to be the folder that you wish to use to write and read data to and from, to store your command files, such as the `newhavenmap.R` file, and any workspace files or `.RData` files that you save. In Windows this is **File > Change dir...**, and on a Mac it is **Misc > Set Working Directory**.

Now, go back to the R command line and enter:

```
source("newhavenmap.R")
```

and your map will be redrawn. The file contains all of the commands to draw the map, and 'sourcing' it makes R run through these in sequence. Suppose you now

wish to redraw the map, but with the roads drawn in blue, rather than red. In the file editor, go to the second line, and edit the line to become:

```
plot(roads,add=TRUE,col= 'blue')
```

and save the file again. Re-entering `source("newhavenmap.R")` now draws the map, but with the roads drawn in blue. Another parameter sometimes used in map drawing is the line width parameter, `lwd`. This time, edit the first `plot` command in the file to become:

```
plot(blocks, lwd=3)
```

and re-enter the source command. The map is redrawn with thicker boundaries around the census blocks. The `col` and `lwd` parameters can of course be used in combination – edit the file again, so that the second line becomes:

```
plot(roads,add=TRUE,col= "red",lwd=2)
```

and source the file again. This time the roads are thicker, and drawn in red. Another advantage of saving command files, as noted earlier, is that it is possible to place the graphics created into various graphics file formats. To create a PDF, for example, the command:

```
pdf(file= 'map.pdf')
```

can be placed before the first line containing a `plot` command in the `newhavenmap.R` file. This tells R that after this command, any graphics will not be drawn on the screen, but instead written to the file `map.pdf` (or whatever name you choose for the file). When you have written all of the commands you need to create your map, then enter:

```
dev.off()
```

which is short for `device off`, and tells R to close the PDF file, and go back to drawing graphics in windows on the screen in the future. To test this out, insert a new first line at the beginning of `newhavenmap.R` and a new last line at the end. Then re-source the file. This time, no new graphics are drawn but you have now created a set of commands to write the graphic into a PDF file called `map.pdf`. This file will be created in the folder in which you are working. To check that this has worked, open your working directory folder in Windows Explorer, Mac Finder, etc., and there should be a file called `map.pdf`. Click on it and whatever PDF reader you use should open, and your map displayed as a PDF file. This file can be incorporated into presentations, word-processing documents and so on. A similar command, for producing PNG files, is:

```
png(file= 'map.png')
```


which writes all subsequent R graphics into a PNG file, until a `dev.off()` is issued. To test this, replace the first line of `newhavenmap.R` with the above command, and re-source it from the R command line. A new file will appear in the folder called `map.png` which may be incorporated into documents as with the PDF file.

3.3 MAPPING SPATIAL OBJECTS

3.3.1 Introduction

The first part of this chapter has outlined basic commands for plotting data and for producing maps and graphics using R. This next section will now concentrate on developing and expanding these basic techniques, will introduce some new plot parameters and will show you how to extract and download Google Maps data as background context. As you develop more sophisticated analyses in later sections you may wish to return to some of the examples used in this section. It will develop mapping of vector spatial data (points, lines and areas) and will also introduce some new R commands and techniques to help put all of this together. To begin with, you will need some predetermined data and, as ever, you may wish to think about creating a workspace folder in which you can store any results you generate.

3.3.2 Data

In this section you will practise your mapping and plotting techniques. The code in this section will examine the `georgia` dataset, select a subset of specific counties and display these using an OpenStreetMap backdrop. You will need to make use of the `GISTools` package to draw maps and handle spatial information.

You should start a new R session or clear your workspace to remove all the variables and datasets you have created and opened using the previous code and commands. You can clear your workspace via the menu **Misc > Remove all objects** (on a Mac, select **Workspace > Clear Workspace**) or by entering:

```
rm(list=ls())
```

Then you should make sure the `GISTools` package and the `georgia` datasets are loaded by entering:

```
library(GISTools)
data(georgia)
```

Check that the data has loaded correctly using `ls()`. There should be three Georgia datasets: `georgia`, `georgia2` and `georgia.polys`.

3.3.3 Plotting Options

A number of plot parameters exist in addition to the ones that have previously been used, including different window sizes, multiple plots in the same window,

polygon or area shading, hatching, boundary thickness, boundary colour and labelling. Many of these plot parameters are described in the help for `par`. First, plot `georgia` with a single shade and a background colour:

```
plot(georgia, col = "red", bg = "wheat")
```

It is also possible to generate an outline of the area using the `gUnaryUnion` function as in the code below, with the results shown in Figure 3.4. The manipulation of spatial data using overlay, union and intersection functions will be covered in more depth in Chapter 5 later in this book.

```
# do a merge
georgia.outline <- gUnaryUnion(georgia, id = NULL)
# plot the spatial layers
plot(georgia, col = "red", bg = "wheat", lty = 2,
    border = "blue")
plot(georgia.outline, lwd = 3, add = TRUE)
# add titles
title(main = "The State of Georgia", font.main = 2,
    cex.main = 1.5, sub = "and its counties",
    font.sub = 3, col.sub = "blue")
```

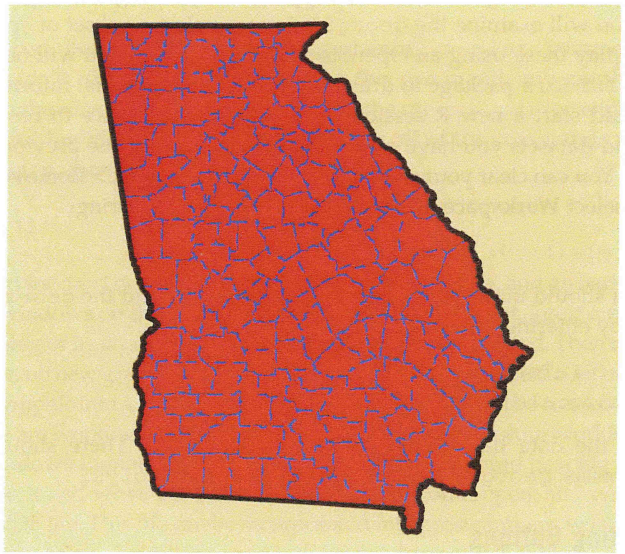

Figure 3.4 An example of applying different plot parameters

In the above code there are two plot commands: the first plots the `georgia` dataset, specifying a dashed blue line to show the county boundaries, a red colour for the objects and a map background colour of wheat. The second overlays the outline created by the union operation with a thicker line width, before the title and subtitle are added. The plot window can be expanded to include multiple plots using the `mfrow` plot parameter. This takes as its arguments the number of rows and the number of columns. Note in the code below the explicit call to create two maps in the plot window and their order using `par(mfrow=c(1,2))` and to adjust the plot margins (`mar`) to accommodate the plots.

```
# set some plot parameters
par(mfrow=c(1,2))
par(mar = c(2,0,3,0))
# 1st plot
plot(georgia, col = "red", bg = "wheat")
title("georgia")
# 2nd plot
plot(georgia2, col = "orange", bg = "lightyellow3")
title("georgia2")
# reset par(mfrow)
par (mfrow=c(1,1))
```

Thus different plot parameters can be used for different subsets of the data such that they are plotted in ways that are different from the default. Note that the parameters can be manually reset for plot windows that are open, for example by entering `par(mfrow=c(1,1))`, or the defaults are reset when a new window is opened.

Sometimes we would like to label the features in our maps. Have a look at the names of the counties in the `georgia` dataset. These are held in the 13th attribute column and `names(georgia)` will return a list of the names of all attributes:

```
data.frame(georgia)[,13]
```

It would be useful to display these on the map, and this can be done using the `pointLabel` function in the `maptools` package that is loaded with `GISTools`. Notice the `col` is set to `NA`. The result is shown in Figure 3.5.

```
# assign some coordinates
Lat <- data.frame(georgia)[,1]   #Y or North/South
Lon <- data.frame(georgia)[,2]   #X or East/West
# assign some label
Names <- data.frame(georgia)[,13]
# set plot parameters, plot and label
par(mar = c(0,0,0,0))
plot(georgia, col = NA)
pl <- pointLabel(Lon, Lat, Names, offset = 0, cex =.5)
```

Figure 3.5 Adding feature labels to the map

Perhaps we are interested in a specific sub-region of the data, for example the area to the east of the state covered by the counties of Jefferson, Jenkins, Johnson, Washington, Glascock, Emanuel, Candler, Bulloch, Screven, Richmond and Burke. A subset of these counties can be selected and plotted in the following way.

```
# the county indices below were extracted from the data.
frame
county.tmp <- c(81, 82, 83, 150, 62, 53, 21, 16, 124,
    121, 17)
georgia.sub <- georgia[county.tmp,]
```

and then plotted:

```
par(mar = c(0,0,3,0))
plot(georgia.sub, col = "gold1", border = "grey")
```



```
plot(georgia.outline, add = TRUE, lwd = 2)
title("A subset of Georgia", cex.main = 2, font.main = 1)
pl <- pointLabel(Lon[county.tmp], Lat[county.tmp],
                 Names[county.tmp], offset = 3, cex = 1.5)
```

Notice how the `county.tmp` variable is used to index the `georgia` data. It is possible to select individual areas or polygons from spatial datasets using the bracket notation as used in matrices and vectors.

Finally, we can bring the different spatial data that have been created together in a single map. You should note that the plot window extent is set by the first plot call and when subsequent plots are 'added' (overplotted), then only the portions of them within that window are displayed.

```
plot(georgia, border = "grey", lwd = 0.5)
plot(georgia.sub, add = TRUE, col = "lightblue")
plot(georgia.outline, lwd = 2, add = TRUE)
title("Georgia with a subset of counties")
```

3.3.4 Adding Context

Finally, a map with context may be more informative. Fortunately at the time of writing this can be done by adding OpenStreetMap tiles.[1] This requires some additional packages to be downloaded and installed in R and a connection to the internet:

```
install.packages(c("OpenStreetMap"),depend=T)
library(OpenStreetMap)
```

The approach is to define the area of interest, to download and plot the map tile from OpenStreetMap and then to plot your data over the tiles. In this case the area for the background map data is defined from the Georgia subset, as created above, which is used to identify the data to download from OpenStreetMap. The results of the code below are shown in Figure 3.6. Note the `spTransform` function in the last line of the code. This transforms the `georgia.sub` data to the same projection as the OpenStreetMap data layer.

```
# define upper left, lower right corners
ul <- as.vector(cbind(bbox(georgia.sub)[2,2],
    bbox(georgia.sub)[1,1]))
```

1 At the time of writing there can be some compatibility issues with the `rJava` package required by `OpenStreetMap`. These relate to the use of 32-bit and 64-bit programes, especially on Windows PCs. If you experience problems installing `OpenStreetMap`, then, it is suggested that you use the 32-bit version of R, which is also installed as part of R for Windows.

```
lr <- as.vector(cbind(bbox(georgia.sub)[2,1],
    bbox(georgia.sub)[1,2]))
# download the map tile
MyMap <- openmap(ul,lr,9, 'mapquest')
# now plot the layer and the backdrop
par(mar = c(0,0,0,0))
plot(MyMap, removeMargin=FALSE)
plot(spTransform(georgia.sub, osm()), add = TRUE, lwd = 2)
```

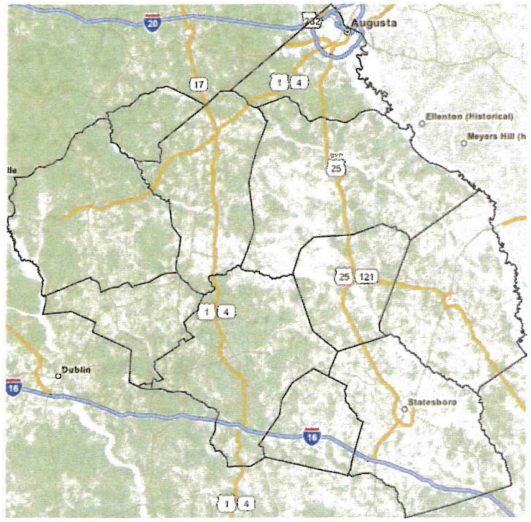

Figure 3.6 A subset of Georgia with an OpenStreetMap backdrop

Google Maps can also be downloaded and used as context as in Figure 3.7. Again, this requires packages to be downloaded and installed and a connection to the internet.

```
install.packages(c("RgoogleMaps", "PBSmapping"), depend=T)
```

Then the area for the background map data is defined to identify the tiles to be downloaded from Google Maps. Some of the plotting commands are specific to the packages installed. Note the first step to convert the subset to `PolySet` format and the last line that defines a polygon plot over Google Maps:

```
# load the package
library(RgoogleMaps)
library(PBSmapping)
```



```
# convert the subset
shp <- SpatialPolygons2PolySet(georgia.sub)
# determine the extent of the subset
bb <- qbbox(lat = shp[,"Y"], lon = shp[,"X"])
# download map data and store it
MyMap <- GetMap.bbox(bb$lonR, bb$latR, destfile = "DC.jpg")
# now plot the layer and the backdrop
par(mar = c(0,0,0,0))
PlotPolysOnStaticMap(MyMap, shp, lwd=2,
    col = rgb(0.25,0.25,0.25,0.025), add = F)
# reset the plot margins
par(mar=c(5,4,4,2))
```

Figure 3.7 A subset of Georgia with a Google Maps backdrop

3.4 MAPPING SPATIAL DATA ATTRIBUTES

3.4.1 Introduction

This section describes some approaches for displaying spatial data attributes. Some of these ideas and commands have already been used in the proceding illustrations, but this section provides a more formal and comprehensive description.

All of the maps that you have generated thus far have simply displayed data (for example, the roads in New Haven and the counties in Georgia). This is fine if the aim is simply to map the locations of different features. However, we are often interested in identifying and analysing the properties or attributes associated with different spatial features. The New Haven and Georgia datasets introduced above both contain areas or regions within them. In the case of the New Haven these are the census reporting areas (census blocks or tracts), and in Georgia the counties within the state. These areas have census attributes which provide population census information for each spatial unit. These attributes are held in the data frame of the spatial object. For example, in the code above you examined the data frame of the Georgia dataset and listed the attributes of individual objects within the dataset. Figure 3.1 actually maps the median income of each county in Georgia, although this code was not shown.

3.4.2 Attributes and Data Frames

The attributes associated with individual features (lines, points, areas in vector data and cell values in raster data) provide the basis for spatial analyses and geographic investigation. Before examining attributes directly, it is important to reconsider the data structures that are commonly used to hold and manipulate spatial data in R.

Clear your workspace and load the New Haven data. Then examine in turn `blocks`, `breach` and `tracts` using the `summary` function:

```
# load & list the data
data(newhaven)
ls()
# have a look at the attributes
summary(blocks)
summary(breach)
summary(tracts)
```

You should notice a number of things from these summaries:

- That each of the datasets is spatial: `blocks` and `tracts` are `SpatialPolygonsDataFrame` objects and `breach` is a `SpatialPoints` object;
- That `blocks` and `tracts` have *data frames* attached to them that contain attributes whose values are summarised by the `summary` function;
- That `breach` does not have any attributes (i.e. it has no *data frame*), it just records locations.

The data frame of these spatial objects needs to accessed in order to examine, manipulate or classify the attribute data. Each row in the data frame contains attribute

values associated with one of the spatial objects – polygons in `blocks` – and each column describes the values associated with a particular attribute for all of the objects. Accessing the data frame allows you to read, alter or compute new attributes. Entering

```
data.frame(blocks)
```

prints all of the attribute information for each census block in New Haven in R console window, whilst

```
head(data.frame(blocks))
```

prints out the first six rows. The attributes can be individually identified using their names. To see the list of column names enter:

```
colnames(data.frame(blocks))
```

One is called `P_VACANT` and describes the percentage of households that are unoccupied (i.e. vacant) in each of the blocks. To access the variable itself, enter:

```
data.frame(blocks)$P_VACANT
```

The $ operator works as it would on a standard data frame to access individual variables (columns) in the data frame. For the data frames of spatial objects a shorthand exists to access this variable. Enter:

```
blocks$P_VACANT
```

A third option is to attach the data frame. Enter:

```
attach(data.frame(blocks))
```

All of the attribute variables now appear as ordinary R variables. For example, to draw a histogram of the percentage vacant housing for each block, enter:

```
hist(P_VACANT)
```

Finally, it is good practice to detach any objects that have been attached after you have finished using them. It is possible to attach many data frames simultaneously, and this can lead to problems if you are not careful. Enter:

```
detach(data.frame(blocks))
```

You can try a similar set of commands with the `tracts` data, but the `breach` dataset has no attributes: it simply records the locations of breaches of the peace. However, the breaches of the peace data can be used to create a raster dataset:

```
# use kde.points to create a kernel density surface
breach.dens = kde.points(breach,lims=tracts)
summary(breach.dens)
```

The `breach.dens` dataset is of class `SpatialPixelsDataFrame` and similarly its attributes are held in a *data frame* which can be examined:

```
head(data.frame(breach.dens))
```

Notice that this has three attributes: the kernel density estimation and two locational attributes that describe the x and y locations. Other raster formats include `SpatialGridDataFrame` into which `SpatialPixelsDataFrame` objects can be coerced:

```
# use 'as' to coerce this to a SpatialGridDataFrame
breach.dens.grid <- as(breach.dens, "SpatialGridDataFrame")
summary(breach.dens.grid)
```

3.4.3 Mapping Polygons and Attributes

A *choropleth* is a thematic map in which areas are shaded in proportion to their attributes. The `GISTools` package includes a choropleth mapping function. Enter:

```
choropleth(blocks, blocks$P_VACANT)
```

This produces a map of the census block in New Haven, shaded by the proportions of vacant properties. Adding a legend to the map allows the map to be interpreted in terms of the levels of vacancy associated with each of the different colour shades in the map. The `choro.legend` command requires information about the variables and the shading scheme used in the map. The shading scheme is a list of class interval boundaries for the quantity being mapped, together with the colour that is used to shade each class interval. There is always one more colour than there are class interval boundaries. In R, shading schemes can be assigned to the variable, and this is passed on to `choro.legend`, and sometimes other functions. In the simplest use of the `choropleth` function, the shading scheme is computed automatically from the variable that is passed to it to be mapped, using a function called `auto.shading`. To compute the shading scheme for `P_VACANT` and store it in a variable called `vacant.shades`, enter

```
vacant.shades = auto.shading(blocks$P_VACANT)
```


and have a look at what is created by entering:

```
vacant.shades
```

You will notice that the `auto.shading` command creates a list with two elements: `$breaks` and `$cols`. These respectively describe the break points between classes and the shading colours used. This information about the shading scheme used can be passed on to `choro.legend`:

```
choro.legend(533000,161000,vacant.shades)
```

This places a legend in the plot window at the coordinates specified by the first two arguments, using the shading scheme specified by the third. The default shading scheme (`auto.shading`) returns five classes, but it is possible to use more. Enter:

```
# set the shading
vacant.shades = auto.shading(blocks$P_VACANT,n=7)
# plot the map
choropleth(blocks,blocks$P_VACANT,shading=vacant.shades)
choro.legend(533000,161000,vacant.shades)
```

The first line of code above firstly derives a shading scheme with seven class intervals (n=7), the next draws the choropleth map – the new argument here is `shading=vacant.shades`, which tells the map-drawing routine to use this shading scheme rather than the default. The final line adds the legend to the map, as before.

It is also possible to alter the colours used in a shading scheme. The default colour scheme uses increasing intensities of red. Graduated lists of colours like this are generated using the `RColorBrewer` package, which is automatically loaded with `GISTools`. This package makes use of a set of colour palettes designed by Cynthia Brewer and intended to optimise the perceptual difference between each shade in the palette, so that visually each shading colour is distinct even when converted to a greyscale. The palettes available in this package are displayed with the command:

```
display.brewer.all()
```

This displays the various colour palettes and their names in a plot window. To generate a list of colours from one of these palettes, for example, enter the following:

```
brewer.pal(5,'Blues')

## [1] "#EFF3FF" "#BDD7E7" "#6BAED6" "#3182BD" "#08519C"
```

This is a list of colour codes used by R to specify the palette. The arguments to `brewer.pal` specify that a five-stage palette based on shades of blue is required. The output of `brewer.pal` can be fed into `auto.shading` to give alternative colours in shading schemes. For example, enter the code below and a choropleth map shaded in green is displayed with its legend (see Figure 3.8). The `cols` argument in `auto.shading` specifies the new colours in the shading scheme.

```
vacant.shades = auto.shading(blocks$P_VACANT,
    cols=brewer.pal(5,"Greens"))
choropleth(blocks, blocks$P_VACANT,shading=vacant.shades)
choro.legend(533000,161000,vacant.shades)
```

A final adjustment to the `auto.shading` command is to change the way the class interval boundaries are computed. As a default, they are based on quantiles of the attribute being mapped, but they can be changed to equal-sized intervals or standard deviations. For example, specifying the option `cutter=rangeCuts` to the `auto.shading` function changes the mapped class intervals as in Figure 3.8 (right).

```
vacant.shades = auto.shading(blocks$P_VACANT, n=5,
    cols=brewer.pal(5,"Blues"), cutter=rangeCuts)
choropleth(blocks,blocks$P_VACANT,shading=vacant.shades)
choro.legend(533000,161000,vacant.shades)
```

In summary, the `choropleth` function maps attributes held in `SpatialPolygons DataFrame` data variables. It automatically shades the variables using five intervals

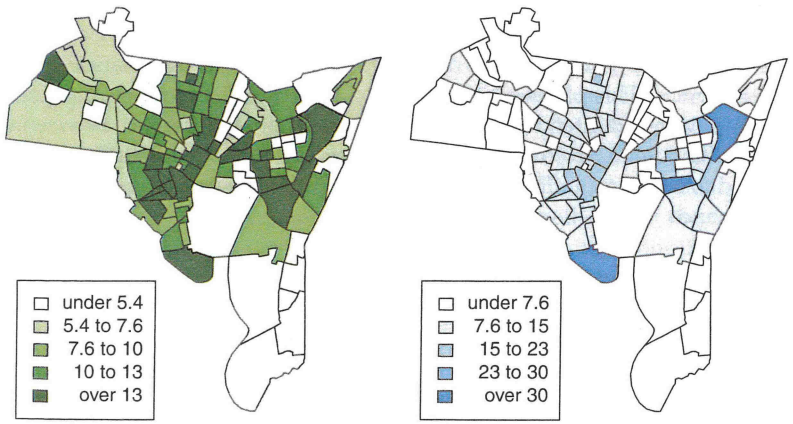

Figure 3.8 Different choropleth maps of vacant properties in New Haven using different shades and cutters

and the 'Reds' palette from the `RColorBrewer` package. The shading colours, their number and the way the intervals between them are determined can all be adjusted. In order to better understand how these functions operate together you should examine the different functions. Enter:

```
choropleth
```

The function code detail is displayed in the R console window. You will see that choropleth calls the `auto.shading` function if no shading parameter is specified. Enter:

```
auto.shading
```

Notice that this in turn specifies a number of default parameters (two digits, five colours and the `RColorBrewer` 'Reds' palette) and calculates the class intervals using quantiles. In addition to using the R help system to understand functions, examining functions in this way can also provide you with insight into their operation.

3.4.4 Mapping Points and Attributes

Point data can be mapped in R as well as polygons and lines. In the New Haven crime dataset, the point locations of reports of breach of the peace events are available. These events are essentially public disorder incidents, on many occasions requiring police intervention. The data are stored in a variable called `breach`. Plotting this variable works in the same way as plotting polygons or lines, using the plot command:

```
plot(breach)
```

This plots the locations of each of the breach of the peace incidents with a + symbol. Usually it is more helpful to plot these on top of another map. As with the `roads` data earlier, this can be done with the `add` option:

```
plot(blocks)
plot(breach, add=TRUE)
```

The `pch` argument (plot character) allows the plotting symbol to be altered. Entering `pch='@'`, for example, replaces the plot symbol with an @ sign:

```
plot(blocks)
plot(breach,add=TRUE,pch='@')
```

Also, as well as text characters, there are a number of special plotting symbols that can be used. A list of plot character options can be found on the help pages for

points (enter `?points`) and are denoted by numbers. These are specified by entering things like `pch=16` and so on. Try entering

```
plot(blocks)
plot(breach,add=TRUE,pch=16)
```

and

```
plot(blocks)
plot(breach,add=TRUE,pch=1,col='red')
```

In the last example, you can see that the `col` option specifying the colour of the plot symbols also works with point data.

If you have very dense point data then one point may obscure another. Adding some transparency to the points can help visualise dense point data. The `add.alpha` function adds transparency to colour palettes. For example, to add transparency to the Brewer 'Reds' palette, enter:

```
# examine the Brewer "Reds" colour palette
brewer.pal(5, "Reds")
# then add a 50% transparency
add.alpha(brewer.pal(5, "Reds"),.50)
```

This prints out a list of five red colours with a transparency term added to them. One of these can be used to display the breaches of the peace as in Figure 3.9, where the density of points is shown more clearly shown.

```
par(mar= c(0,0,0,0))
# plot the blocks and then the breaches of the peace
plot(blocks, lwd = 0.7, border = "grey40")
plot(breach,add=TRUE, pch=1, col= "#DE2D2680")
```

Commonly, point data come in a tabular format rather than as an R spatial object (i.e. of class `sp`) with attributes that include the latitude and longitude or easting and northing of the individual data points. For example, the `quakes` dataset is included as part of R. It provides the locations of 1000 seismic events (earthquakes) near Fiji. To load and examine the data enter:

```
# load the data
data(quakes)
# look at the first 6 records
head(quakes)
```


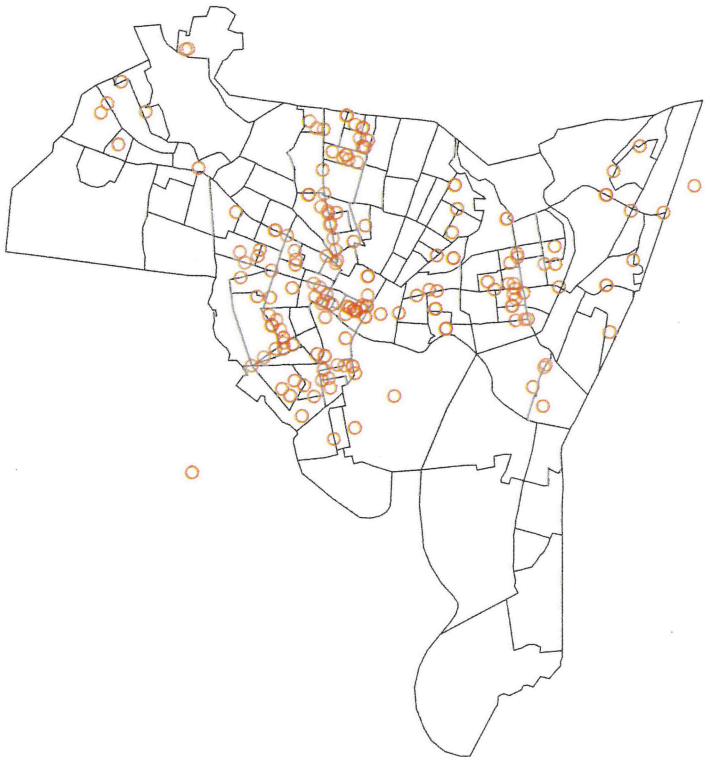

Figure 3.9 Breaches of the peace with a transparency added to the colour

You will see that the dataset comes with a number of attributes: `lat`, `long`, `depth`, `mag` and `stations`. Here you will use the lat and long attributes to create a spatial points dataset, and because we want to include the attributes this will be a `SpatialPointsDataFrame` object. The results of running the code below are shown in Figure 3.10, which shows the spatial context of the data in the Pacific Ocean, to the north of New Zealand.

```
# define the coordinates
data(quakes)
coords.tmp <- cbind(quakes$long, quakes$lat)
# create the SpatialPointsDataFrame
quakes.spdf <- SpatialPointsDataFrame(coords.tmp,
    data = data.frame(quakes))
# set the plot parameters to show 2 maps
par(mar = c(0,0,0,0))
```

```
par(mfrow=c(1,2))
# 1st plot with default plot character
plot(quakes.spdf)
# then with a transparency term
plot(quakes.spdf, pch = 1, col = '#FB6A4A80')
# reset par(mfrow)
par(mfrow=c(1,1))
```

Figure 3.10 Two plots of the Fiji quake data

```
##Not Run##
# you could load the 'maps' package for some context
# install.packages("maps", dep = T)
# library(maps)
# map('world2', fill = F, add = TRUE)
##End Not Run##
```

The last bit of code nicely illustrates how to create a spatial dataset in R. Essentially the sequence is:

- Define the coordinates for the spatial object;
- Assign these to a `SpatialPoints`, `SpatialLines` or `SpatialPolygons` object;
- If the object has attributes, then the dataframe needs to be specified for the `SpatialPointsDataFrame`, `SpatialLinesDataFrame` or `SpatialPolygonsDataF` object.

You should examine the help for these classes of objects. Points just need coordinate pairs, but polygons and lines need lists of coordinates for each object.


```
help("SpatialPolygons-class")
```

This can be illustrated using the `georgia.polys` dataset:

```
data(georgia)
# select the polys of interest
tmp <- georgia.polys[c(1,3,151,113)]
# convert to Polygon and the Polygons object
t1 <- Polygon(tmp[1]); t1 <- Polygons(list(t1), "1")
t2 <- Polygon(tmp[2]); t2 <- Polygons(list(t2), "2")
t3 <- Polygon(tmp[3]); t3 <- Polygons(list(t3), "3")
t4 <- Polygon(tmp[4]); t4 <- Polygons(list(t4), "4")
# create a SpatialPolygons object
tmp.Sp <- SpatialPolygons(list(t1,t2,t3,t4), 1:4)
plot(tmp.Sp, col = 2:5)
# create an attribute
names <- c("Appling", "Bacon", "Wayne", "Pierce")
# now create an SPDF object
tmp.spdf <- SpatialPolygonsDataFrame(tmp.Sp,
    data=data.frame(names))
# plot the data to examine (code not run)
# data.frame(tmp.spdf)
# plot(tmp.spdf, col = 2:5)
```

Note the use of the semi-colon `(;)` to combine commands on the same line. Note also the way that `t1`, `t2`, etc. are created and then overwritten.

You will have noticed that the `quakes` dataset has an attribute describing the magnitude of each earthquake. We can visualise the magnitudes in a number of ways – for example, by using `choropleth` (which will take any `sp` spatial dataset), by selecting data according to different criteria and then plotting these in particular ways, or by plotting all the data points but specifying the size of each data point to be proportional to the attribute magnitude. These are shown in the code blocks below and in the results in Figures 3.11 and 3.12. As a reminder, when you run this code and the other code in this book, you should try manipulating and changing the parameters that are used to explore different mapping approaches. First, choropleth and different sizes of plot characters can be used to indicate the magnitude of the variable being considered (Figure 3.11):

```
# set some plot parameters
par(mfrow=c(2,2))
par(mar = c(0,0,0,0)) # set margins
## 1. Plot using choropleth
choropleth(quakes.spdf, quakes$mag)
```

```
## 2. Plot with a different shading scheme & pch
shades = auto.shading(quakes$mag,n=6,
    cols=brewer.pal(6,'Greens'))
choropleth(quakes.spdf, quakes$mag, shades, pch = 1)
## 3. Plot with a transparency
shades$cols <- add.alpha(shades$cols, 0.5)
choropleth(quakes.spdf, quakes$mag, shading = shades,
pch = 20)
## 4. Plot character size determined by attribute magnitude
tmp <- quakes$mag      # assign magnitude to tmp
tmp <- tmp - min(tmp) # remove minimum
tmp <- tmp / max(tmp) # divide by maximum
plot(quakes.spdf, cex = tmp*3, pch = 1, col = '#FB6A4A80')
```

Figure 3.11 Different choropleth point maps

Next a threshold can be used to define classes in a number of different ways. The plots in Figure 3.12 map these classes using different pot characters and colours.


```
# Set the plot parameters
par(mfrow=c(1,2))
par(mar = c(0,0,0,0))
## 1. Apply a threshold to categorise the data
tmp2 <- cut(quakes$mag, fivenum(quakes$mag), include.lowest = T)
class <- match(tmp2,levels(tmp2))
# specify 4 plot characters to use
pch.var <- c(0,1,2,5)
# Plot the classes
plot(quakes.spdf, pch = pch.var[class], cex = 0.7,
    col = "#252525B3")
## 2. Thresholds for classes can be specified
# logical operations help to define 3 classes
# note the terms such as '+ 0' convert TRUE / FALSE to numbers
index.1 <- (quakes$mag >= 4 & quakes$mag < 5) + 0
index.2 <- (quakes$mag >=5 & quakes$mag < 5.5) * 2
index.3 <- (quakes$mag >=5.5) * 3
class <- index.1 + index.2 + index.3
# specify 3 plot colours to use
col.var <- (brewer.pal(3, "Blues"))
plot(quakes.spdf, col = col.var[class], cex = 1.4, pch = 20)
# reset par(mfrow)
par(mfrow=c(1,1))
```

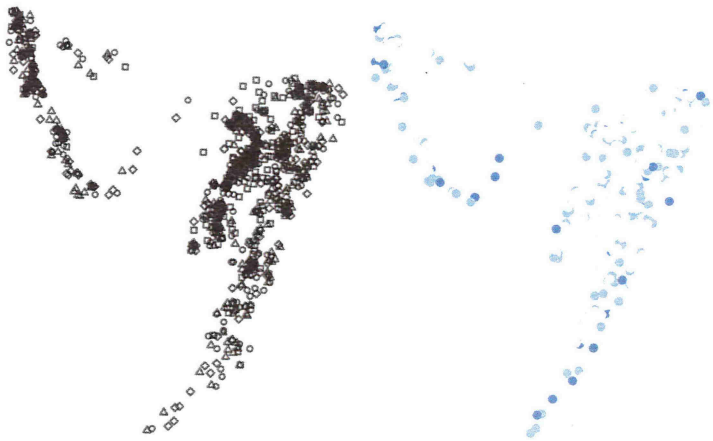

Figure 3.12 Different ways of classifying and mapping point attributes

The code used above includes logical operators and illustrates how they can be used to select elements that satisfy some condition. These can be used singularly or in combination to select in the following way:

```
data <- c(3, 6, 9, 99, 54, 32, -102)
index <- (data == 32 | data <= 6)
data[index]

## [1]    3  6  32  -102
```

These are described in greater detail in Chapter 4.

Finally, it is possible to use the `PlotOnStaticMap` function from the `RgoogleMaps` package to plot the earthquake locations with some context from Google Maps. This is similar to Figure 3.6, which mapped a subset of Georgia counties against an OpenStreetMap backdrop, except that this time points rather than polygons are being displayed and different Google Maps backdrops are used as context, as in Figures 3.13 and 3.14.

```
library(RgoogleMaps)
Lat <- as.vector(quakes$lat)
Long <- as.vector(quakes$long)
MyMap <- MapBackground(lat=Lat, lon=Long, zoom = 10)
# note the use of the tmp variable defined earlier to set
# the cex value
PlotOnStaticMap(MyMap,Lat,Long,cex=tmp+0.3,pch=1,
    col= '#FB6A4A80')
```

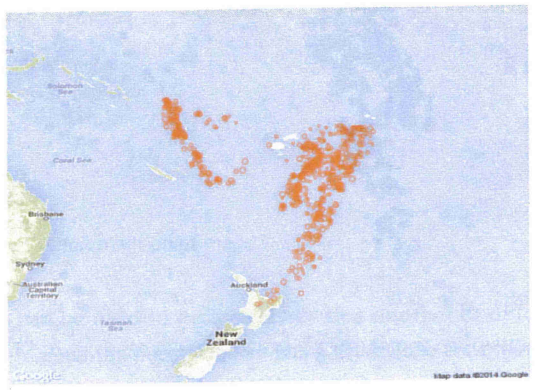

Figure 3.13 Plotting points with a Google Maps context


```
MyMap <- MapBackground(lat=Lat, lon=Long, zoom = 10,
    maptype = "satellite")
PlotOnStaticMap(MyMap,Lat,Long,cex=tmp+0.3,pch=1,
    col='#FB6A4A80')
```

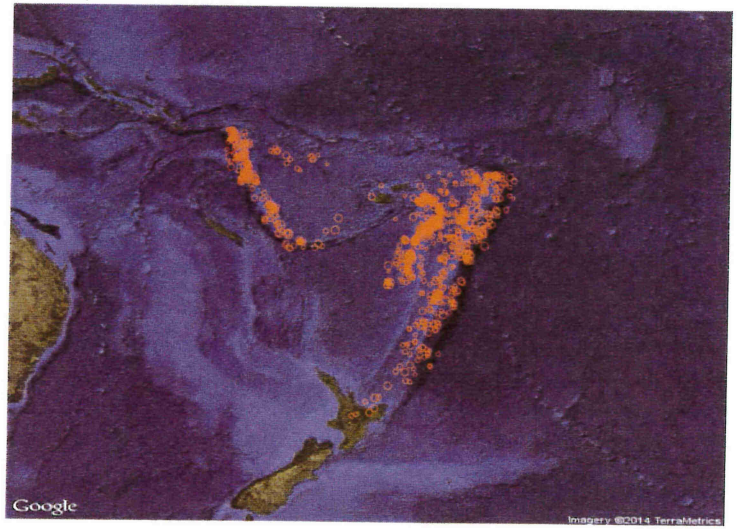

Figure 3.14 Plotting points with Google satellite image context

3.4.5 Mapping Lines and Attributes

This section considers line data spatial objects. These can be defined in a number of ways and typically describe different network features such as roads. In the example below, a subset of the roads in New Haven is extracted. This involves defining a polygon to clip the road data and converting it to a `SpatialPolygonsDataFrame` object before doing so.

```
data(newhaven)
# 1. create a clip area
xmin <- bbox(roads)[1,1]
ymin <- bbox(roads)[2,1]
xmax <- xmin + diff(bbox(roads)[1,]) / 2
ymax <- ymin + diff(bbox(roads)[2,]) / 2
xx = as.vector(c(xmin, xmin, xmax, xmax, xmin))
yy = as.vector(c(ymin, ymax, ymax, ymin, ymin))
# 2. create a spatial polygon from this
crds <- cbind(xx,yy)
```

```
Pl <- Polygon(crds)
ID <- "clip"
Pls <- Polygons(list(Pl), ID=ID)
SPls <- SpatialPolygons(list(Pls))
df <- data.frame(value=1, row.names=ID)
clip.bb <- SpatialPolygonsDataFrame(SPls, df)
# 3. clip out the roads and the data frame
roads.tmp <- gIntersection(clip.bb, roads, byid = T)
tmp <- as.numeric(gsub("clip", "", names(roads.tmp)))
tmp <- data.frame(roads)[tmp,]
# 4. finally create the SLDF object
roads.tmp <- SpatialLinesDataFrame(roads.tmp,
    data = tmp, match.ID = F)
```

Having prepared the roads data subset in this way, a number of methods for mapping spatial lines can be illustrated. These include maps based on classes and continuous variables or attributes contained in the data frame. As before, we can start with a straightforward map which is then embellished in different ways: shading by road type (the AV_LEGEND attribute) and line thickness defined by road segment length (the attribute LENGTH_MI). The maps are shown in Figure 3.15.

```
par(mfrow=c(1,3)) # set plot order
par(mar = c(0,0,0,0)) # set margins
# 1. simple map
plot(roads.tmp)
# 2. mapping an attribute variable
road.class <- unique(roads.tmp$AV_LEGEND)
# specify a shading scheme from the road types
shades <- rev(brewer.pal(length(road.class), "Spectral"))
tmp <- roads.tmp$AV_LEGEND
index <- match(tmp, as.vector(road.class))
plot(roads.tmp, col = shades[index], lwd = 3)
# 3. using an attribute to specify the line width
plot(roads.tmp, lwd = roads.tmp$LENGTH_MI * 10)
# reset par(mfrow)
par(mfrow=c(1,1))
```

3.4.6 Mapping Raster Attributes

The spatial object type considered in this section relates to raster data. Earlier in this chapter a simple raster dataset was created using a kernel density function. This generated a SpatialPixelsDataFrame object which was converted to a SpatialGridDataFrame object. In this section the Meuse dataset, included as part of the sp package, will be used to illustrate how raster attributes can be mapped.

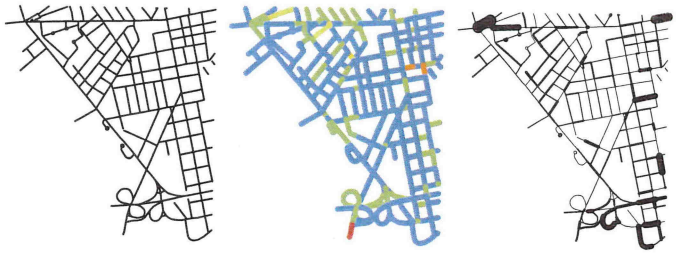

Figure 3.15 A subset of the New Haven roads data, plotted in different ways: simple; shaded using an attribute; line width based on an attribute

Load the meuse.grid data and examine its properties using the class and summary functions.

```
data(meuse.grid)
class(meuse.grid)
summary(meuse.grid)
```

You should notice that meuse.grid is a data.frame object and that it has seven attributes, including an easting (x) and a northing (y). These are described in the meuse.grid help pages (enter ?meuse.grid). The spatial properties of the dataset can be examined by plotting the easting and northing attributes:

```
plot(meuse.grid$x, meuse.grid$y, asp = 1)
```

And it can be converted to a SpatialPixelsDataFrame object (enter ?SpatialPixelsDataFrame for a description of this type of object):

```
meuse.grid = SpatialPixelsDataFrame(points =
    meuse.grid[c("x", "y")], data = meuse.grid)
```

It is possible to map different attributes held in the data.frame of the SpatialPixelsDataFrame object. Essentially these work by specifying the raster dataset and the attribute to be mapped. You should note that the raster datasets passed to image and spplot can be SpatialGridDataFrame or SpatialPixelsDataFrame objects. A number of examples of mapping routines with different shading schemes using image (Figure 3.16) and using spplot (Figure 3.17) are shown below. You may have to close the plot window after the first raster plot before entering the code for the second.

```
par(mfrow=c(1,2)) # set plot order
par(mar = c(0.25, 0.25, 0.25, 0.25)) # set margins
# map the dist attribute using the image function
image(meuse.grid, "dist", col = rainbow(7))
image(meuse.grid, "dist", col = heat.colors(7))
```

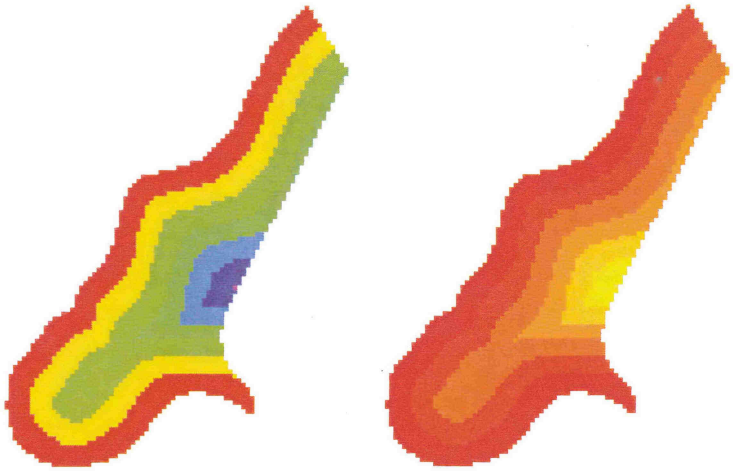

Figure 3.16 Maps of the Meuse raster data using the `image` function

```
# using spplot from the sp package
par(mar = c(0.25, 0.25, 0.25, 0.25)) # set margins
p1 <- spplot(meuse.grid, "dist",
    col.regions=terrain.colors(20))
# position in c(xmin, ymin, xmax, ymax)
print(p1, position = c(0,0,0.5,0.5), more = T)
p2 <- spplot(meuse.grid, c("part.a", "part.b", "soil",
    "ffreq"), col.regions=topo.colors(20))
print(p2, position = c(0.5,0,1,0.5), more = T)
```

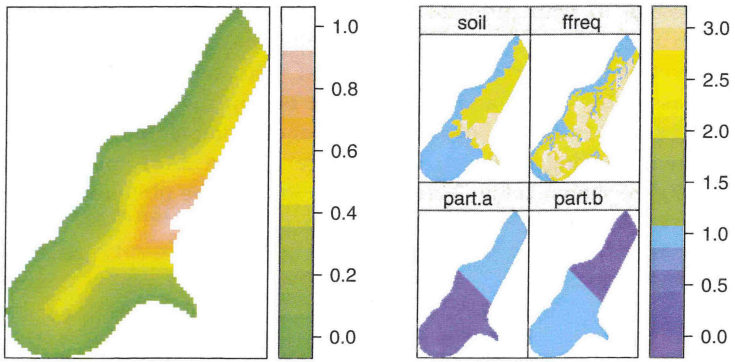

Figure 3.17 Maps of the Meuse raster data using the `spplot` function

3.5 SIMPLE DESCRIPTIVE STATISTICAL ANALYSES

The final section of this chapter before the self-test questions describes how to develop some basic descriptive statistical analyses of attributes held in R `data.frame` objects. These are intended to provide an introduction to methods for analysing the properties of spatial data attributes which will be extended in more formal treatments of statistical and spatial analyses in later chapters. This section first describes approaches for examining the properties of data variables using histograms and boxplots, and then extends this to consider some simple ways of analysing data variables in relation to each other using scatter plots and simple regressions, before showing how mosaic plots can be used to visualise relationships between variables.

3.5.1 Histograms and Boxplots

You should make sure the New Haven data is loaded. There are number of ways of generating simple summaries of any variable. The function `table` can be used to summarise the counts of categorical or discrete data, and `hist`, `summary` and `fivenum` provide summaries of continuous variables. You should use these to explore the `$P_VACANT` variable in `blocks`. For example, typing `hist(blocks$P_VACANT)` will draw a histogram of the percentage vacant housing for each census block in New Haven. Similarly, typing `summary(blocks$P_VACANT)` or `fivenum(blocks$P_VACANT)` will produce other summaries of the distribution of the variable. As with all plot functions, it is possible to adjust the histogram bin sizes and the plot labels as in the example below.

```
data(newhaven)
hist(blocks$P_VACANT, breaks = 20, col = "cyan",
    border = "salmon",
    main = "The distribution of vacant property
    percentages",
    xlab = "percentage vacant", xlim = c(0,40))
```

A further way to provide visual descriptive summaries of variables is to use box-and-whisker plots (or boxplots) via the `boxplot` function. This can be used to display a single variable or multiple variables together. In order to illustrate this the `blocks` dataset can be split into high- and low-vacancy areas based on whether the proportion of properties vacant is greater than 10%.

```
index <- blocks$P_VACANT > 10
high.vac <- blocks[index,]
low.vac <- blocks[!index,]
```

Then `boxplot` can be used to visualise the differences between these two subsets in terms of the distribution of owner occupancy and the proportion of different ethnic groups, as in Figure 3.18.

```
# set plot parameters and shades
cols = rev(brewer.pal(3, "Blues"))
par(mfrow = c(1,2))
par(mar = c(2.5,2,3,1))
# attach the data frame
attach(data.frame(high.vac))
# create a boxplot of 3 variables
boxplot(P_OWNEROCC,P_WHITE,P_BLACK,
    names=c("OwnerOcc", "White", "Black"),
    col=cols, cex.axis = 0.7, main = "High Vacancy")
# detach the data frame
detach(data.frame(high.vac))
# do the same for the second boxplot & variables
attach(data.frame(low.vac))
boxplot(P_OWNEROCC,P_WHITE,P_BLACK,
    names=c("OwnerOcc","White", "Black"),
    col=cols, cex.axis = 0.7, main = "Low Vacancy")
```

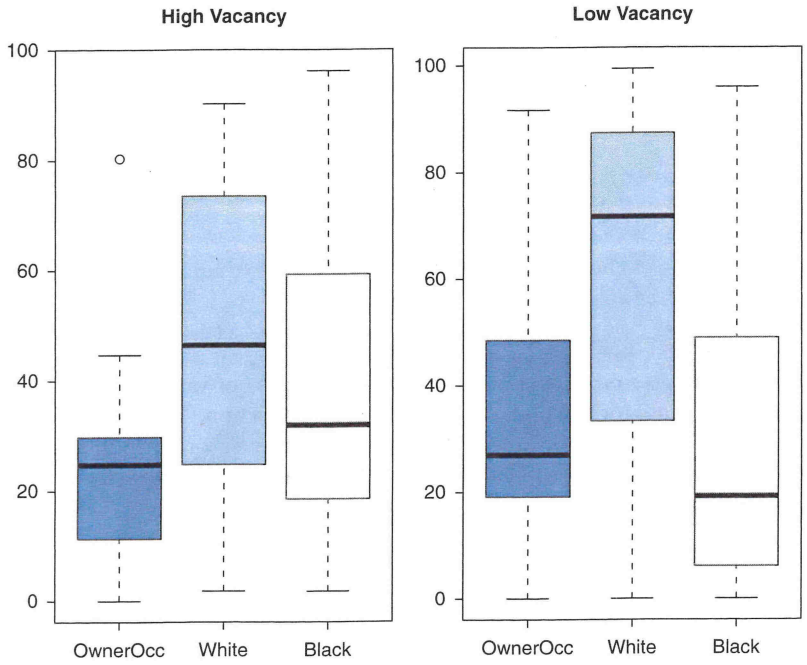

Figure 3.18 Box-and-whisker plots of the `blocks` dataset split into high- and low-vacancy areas


```
detach(data.frame(low.vac))
# reset par(mfrow)
par(mfrow=c(1,1))
# reset the plot margins
par(mar=c(5,4,4,2))
```

3.5.2 Scatter Plots and Regressions

The differences in the two subgroups suggest that there may be some statistical association between the amount of vacant properties and the proportions of different ethnic groups, typically due to well-known socio-economic inequalities and power imbalances. First, we can plot the data to see if we can visually identify any trends:

```
plot(blocks$P_VACANT/100, blocks$P_WHITE/100)
plot(blocks$P_VACANT/100, blocks$P_BLACK/100)
```

The scatter plots suggest that there *may* be a negative relationship between the proportion of white people in a census block and the proportion of vacant properties and that there *may* be a positive association with the proportion of black people. It is difficult to be confident in these statements, but can be examined more formally by using a simple regression model as estimated by the `lm` function and then plotting the coefficient estimates or slopes:

```
# assign some variables
p.vac <- blocks$P_VACANT/100
p.w <- blocks$P_WHITE/100
p.b <- blocks$P_BLACK/100
# fit regressions
mod.1 <- lm(p.vac ~ p.w)
mod.2 <- lm(p.vac ~ p.b)
```


The function `lm` is used in R to fit regression models (`lm` stands for 'linear models'). The models to be fitted are specified in a special notation in R. Effectively a model description is an R variable of its own. Although we do not go into detail about the modelling language in this book, more can be found in, for example, de Vries and Meys (2012: Chapter 15); for now, it is sufficient to know that the R notation `y ~ x` suggests the model $y = ax + b$. The notation is sufficiently rich to allow the specification of very broad set of linear models.

The coefficients can be inspected and it is evident that the proportion of white people is a weak negative predictor of the proportion of vacant properties in a census block and that the proportion of black people is a weak positive predictor.

Specifically, the model suggests relationships that indicate that the amount of vacant properties in a census block decreases by 1% for each 3.5% increase in the proportion of white people and that it increases by 1% for each 3.7% increase in the proportion of black people in the census block. However, when a multi-variate analysis model is computed neither are found to be significant predictors of vacant properties. The models can be examined using the `summary` command:

```
summary(mod.1)

##
## Call:
## lm(formula = p.vac ~ p.w)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.1175  -0.0373  -0.0120  0.0171  0.2827
##
## Coefficients:
##             Estimate  Std. Error  t value   Pr(>|t|)
## (Intercept)   0.1175      0.0109    10.75   <2e-16 ***
## p.w          -0.0355      0.0172    -2.06    0.042 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.062 on 127 degrees of freedom
## Multiple R-squared: 0.0323,Adjusted R-squared: 0.0247
## F-statistic: 4.24 on 1 and 127 DF, p-value: 0.0415

# not run below
# summary(mod.2)
# summary(lm(p.vac ~ p.w + p.b))
```

The trends can be plotted with the data as in Figure 3.19.

```
# define a factor for the jitter function
fac = 0.05
# define a colour palette
cols = (brewer.pal(6, "Spectral"))
# plot the points with small random term added
# this is to help show densities
# 1st properties vacant against p.w
plot(jitter(p.vac, fac), jitter(p.w, fac),
    xlab= "Proportion Vacant",ylab = "Proprtion White /
    Black", col = cols[1], xlim = c(0, 0.8))
```



```
# then properties vacant against p.b
points(jitter(p.vac, fac), jitter(p.b, fac), col = cols[6])
# fit some trend lines from the 2 regression model coefficients
abline(a = coef(mod.1)[1], b= coef(mod.1)[2],
       lty = 1, col = cols[1]); #white
abline(a = coef(mod.2)[1], b= coef(mod.2)[2],
       lty = 1, col = cols[6]); #black
# add some legend items
legend(0.71, 0.19, legend = "Black", bty = "n", cex = 0.8)
legend(0.71, 0.095, legend = "White", bty = "n", cex = 0.8)
```

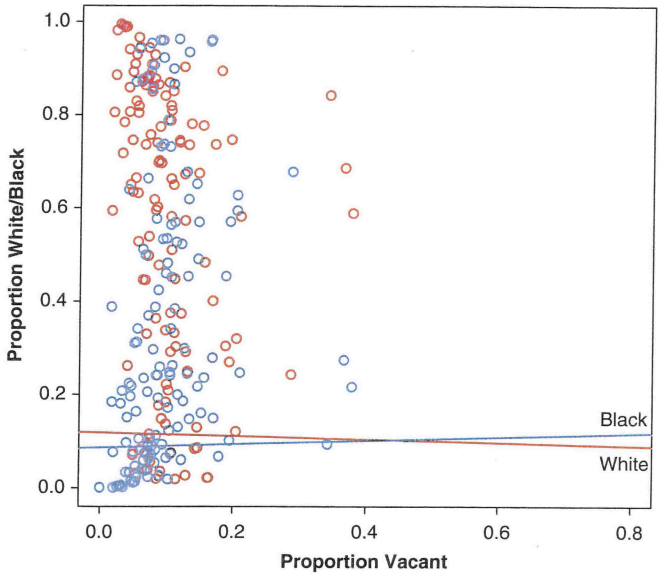

Figure 3.19 Plotting regression coefficient slopes

3.5.3 Mosaic Plots

For data where there is some kind of true or false statement, mosaic plots can be used to generate a powerful visualisation of the statistical properties and relationships between variables. What they seek to do is to compare crosstabulations of counts (hence the need for true or false statements) against a model where proportionally equal counts are expected, in this case of vacant housing across ethnic groups. The mosaic plot in Figure 3.20 shows that the distribution of census blocks

with vacancy levels higher than 10% is *not* evenly distributed amongst different ethnic groups: the tiles in the mosaic plot have areas proportional to the counts (in this case the number of people affected) and their colours show which groups are under- or over-represented, when compared against a model of expected equality. The blue tiles show combinations of property vacancy and ethnicity that are higher than would be expected, with the tiles shaded deep blue corresponding to combinations whose residuals are greater than +4, when compared to the model, indicating a much greater frequency in those cells than would be found if the model of equality were true. The tiles shaded deep red correspond to the residuals less than 4, indicating much lower frequencies than would be expected. Thus the white ethnic group is significantly more strongly associated with areas where vacant properties make up less than 10%, and the other ethnic groups are significantly more strongly associated with areas where vacant properties make up less than 10%, than would be expected in a model of equal distribution.

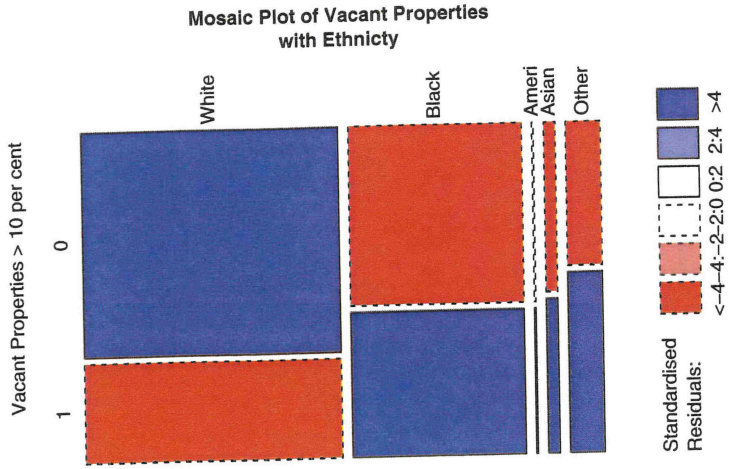

Figure 3.20 An example of a mosaic plot

```
# populations of each group in each census block
pops <- data.frame(blocks[,14:18]) * data.frame(blocks)[,11]
pops <- as.matrix(pops/100)
colnames(pops) <- c("White", "Black", "Ameri", "Asian",
    "Other")
# a true / false for vacant properties
vac.10 <- (blocks$P_VACANT > 10) + 0
# the crosstabulations
mat.tab <- xtabs(pops ~vac.10)
```



```
# mosaic plot
ttext = sprintf("Mosaic Plot of Vacant Properties
                with ethnicty")
mosaicplot(t(mat.tab),xlab='',
           ylab= 'Vacant Properties > 10 percent',
           main=ttext,shade=TRUE,las=3,cex=0.8)
```

3.6 SELF-TEST QUESTIONS

This chapter has introduced a number of commands and functions for mapping spatial data and visualising spatial data attributes. The questions in this section present a series of tasks for you to complete that build on the methods illustrated in the preceding sections. The answers at the end of the chapter present snippets of code that will complete the tasks but, as ever, you may find that your code differs from the answers provided. This is to be expected and is not something that should concern you as there are usually many ways to achieve the same objectives. The tasks seek to extend the mapping that you should have already done (as a reminder, the expectation is that you run the code embedded in the text throughout the book), and in places greater detail and explanation of the specific techniques is given. Four general areas are covered:

- Plots and maps: working with map data
- Misrepresentation of continuous variables: using different cut functions for choropleth mapping
- Selecting data: creating variables and subsetting data using logical statements
- Re-projections: transforming data using `spTransfrom`

Self-Test Question 1. Plots and maps: working with map data
Your task is to write code that will produce a map of the counties in Georgia, shaded in a colour scheme of your choice but using 11 classes describing the distribution of median income in thousands of dollars (this is described by the `MedInc` attribute in the data frame). The maps should include a legend and the code should write the map to a TIFF file, with a resolution of 300 dots per inch and a map size of 7 × 7 inches.

```
# Hints
display.brewer.all() # to show the Brewer palettes
?locator # to identify coordinates in the plot window
cex = 0.75          # sets the character size in choro.legend
# Tools
library(GISTools) # for the mapping tools
```

```
data(georgia)          # to load the Georgia data
choropleth()           # to create the maps
choro.legend()         # to display the legend
```

Self-Test Question 2. Misrepresentation of continuous variables: using different cutters for choropleth mapping

It is well known that it is very easy to *lie with maps* (see Monmonier, 1996). One of the very commonly used tricks for misrepresenting the spatial distribution of phenomena relates to the inappropriate categorisation of continuous variables. Your task in this exercise is produce three maps that represent the same feature, and in so doing you will investigate the impact of different *cut* functions when used to generate maps.

Write code to create three maps in the same window of the numbers of houses in the New Haven census blocks. Apply different cut functions to divide the `HSE_UNITS` in the `blocks` dataset into five classes in different ways based on quantiles, absolute ranges, and standard deviations. You need not add legends, scale bars, etc. but should include map titles.

```
# Hints
?auto.shading      # the help for autoshading tool
?par               # the help for plot parameters
par(mfrow = c(1,2)) # set the plot order to be 1 row & 2 columns
# run the code below to specify a 10 by 8 inch plot window
if (.Platform$GUI == "AQUA") {
      quartz(w=10,h=8) } else {
      x11(w=10,h=8) }
# Tools
library(GISTools)          # for the mapping tools
data(newhaven)             # to load the New Haven data
```

Self-Test Question 3. Selecting data: creating variables and subsetting data using logical statements.

In the previous sections on mapping polygon attributes and mapping lines, different methods for selecting or subsetting the spatial data were introduced. These applied an overlay of spatial data using the `gIntersection` function to select roads within the extent of a `SpatialPolygons` object, and a series of logical operators were used to select and classify earthquake locations that satisfied specific criteria. Additionally, logical operators were introduced in the previous chapter. When applied to a variable they return true or false statements or, more correctly, `logical` data types. In this exercise, the objective is to create a secondary attribute and then to use a logical statement to select data objects when applied to the attribute you create.

A company wishes to market a product to the population in rural areas. They have a model that says that they will sell one unit of their product for every 20 people in rural areas that are visited by one of their sales team, and they would like to know which counties have a rural population density of 20 people per square kilometre. Using the Georgia data, you should develop some code that selects counties based on a rural population density measure. You will need to calculate for each county a *rural population density* score and map the counties in Georgia that have a score of greater than 20 rural people per square kilometre.

```
# Hints
locator()          # to identify locations in the plot window
rect()             # to draw a rectangle for a legend
legend()           # to indicate the rural and non-rural areas
help("!")          # to examine logic operators
# Tools
library(GISTools)  # for the mapping tools
data(georgia)      # use georgia2 as it has a geographic projection
library(rgeos)     # you may need to use install.packages()
gArea()            # a function in rgeos
```

Self-Test Question 4. Re-projections: transforming data using spTransfrom

Spatial data come with projections, which define an underlying geodetic model over which the spatial data are projected. Different spatial datasets need to be aligned over the same projection for the spatial features they describe to be compared and analysed together. National grid projections typically represent the world as a flat surface and allow distance and area calculations to be made, which cannot be done using models that use degrees and minutes. World geodetic systems such as WGS84 provide a standard model provide standard reference system. In the previous exercise you worked with the `georgia2` dataset which is projected in metres, whereas `georgia` is projected in degrees in WGS84. A range of different projects are described in formats for different packages and software are described at the Spatial Reference website.[2] A typical re-projection would be something like

```
new.spatial.data <- spTransform(old.spatial.data,
    new.Projection)
```

You should note that data need to have a projection in order to be transformed. Projections can be assigned if you know what the projection is. Recall the code from earlier in this chapter using the Fiji earthquake data:

2 `http://www.spatialreference.org`

```
library(GISTools)
library(rgdal)
data(quakes)
coords.tmp <- cbind(quakes$long, quakes$lat)
# create the SpatialPointsDataFrame
quakes.spdf <- SpatialPointsDataFrame(coords.tmp,
    data = data.frame(quakes))
```

You can examine the projection properties of this `SpatialPointsDataFrame` object by entering:

```
summary(quakes.spdf)
```

You will see that the `Is projected` and `proj4string` properties are empty. These can be populated if you know the spatial reference system and then the data can be transformed.

```
proj4string(quakes.spdf) <- CRS("+proj=longlat +ellps=WGS84")
```

The objective of this exercise is to re-project the New Haven `blocks` and `breach` datasets from their original reference system to WGS84. Recall that at the start of this chapter the description of these datasets was that they had a local projections system, using the State Plane Coordinate System for Connecticut, in US survey feet. You should transform the breaches of the peace and the census blocks data to latitude and longitude using the `CRS` statement above and the `spTransfrom` function in the `rgdal` package. Then, having transformed the datasets, you should extend the context mapping that used the `RgoogleMaps` package in earlier sections to map the locations of the breaches of peace and the census blocks with a Google Maps backdrop.

```
# Hints
# use the help and the example code they include
?PlotOnStaticMap # for the points
?PlotPolysOnStaticMap # for the census block
# adjust the polygon shading using rgb and transparency
?rgb
# Tools
library(GISTools) # for the mapping tools
library(rgdal) # this has the spatial reference tools
library(RgoogleMaps)
library(PBSmapping)
data(newhaven) # for the breach point dataset
```


ANSWERS TO SELF-TEST QUESTIONS

Q1. Plots and maps: working with map data

```
# load the data and the package
library(GISTools)
data(georgia)
# open the tif file and give it a name
tiff("Quest1.tiff", width=7,height=7,units='in',res=300)
# define the shading scheme
shades <- auto.shading(georgia$MedInc/1000,n=11,
    cols=brewer.pal(11, "Spectral"))
# plot the map
choropleth(georgia,georgia$MedInc/1000,shading=shades)
# add the legend & keys
choro.legend(-81.7, 35.1, shades,
      title ="Median Income (1000s $)", cex = 0.75)
# close the file
dev.off()
```

Your map should look something like Figure 3.21:

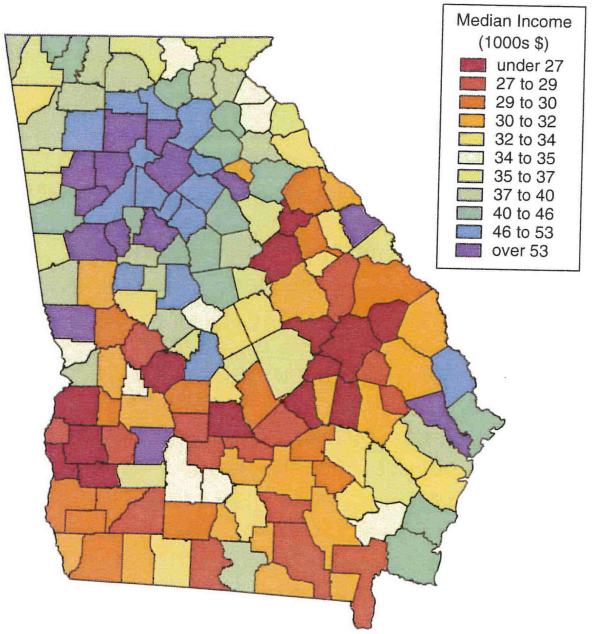

Figure 3.21 The map produced by the code for Self-Test Question 1

Q2. Misrepresentation of continuous variables: using different cutters for choropleth mapping

```
# Code
library(GISTools)
data(newhaven)
attach(data.frame(blocks))
# 1. Initial investigation
# You could start by having a look at the data
hist(HSE_UNITS, breaks = 20)
# You should notice that it has a normal distribution
# but with some large outliers
# have a look at the impacts of different cut schemes
quantileCuts(HSE_UNITS, 5)
rangeCuts(HSE_UNITS, 5)
sdCuts(HSE_UNITS, 5)
# 2. Do the task
# define the plot window
if (.Platform$GUI == "AQUA") {
  quartz(w=10,h=6) } else {
    x11(w=10,h=6) }
# set the plot parameters
par(mar = c(0.25,0.25,2, 0.25))
par(mfrow = c(1,3))
par(lwd = 0.7)
# a) mapping classes defined by quantiles
shades <- auto.shading(HSE_UNITS, cutter = quantileCuts,
    n = 5, cols = brewer.pal(5, "RdYlGn"))
choropleth(blocks,HSE_UNITS,shading=shades)
choro.legend(533000,161000,shades)
title("Quantile Cuts", cex.main = 2)
# b) mapping classes defined by absolute ranges
shades <- auto.shading(HSE_UNITS, cutter = rangeCuts,
  n = 5, cols = brewer.pal(5, "RdYlGn"))
choropleth(blocks,HSE_UNITS,shading=shades)
choro.legend(533000,161000,shades)
title("Range Cuts", cex.main = 2)
# c) mapping classes defined by standard deviations
shades <- auto.shading(HSE_UNITS, cutter = sdCuts,
  n = 5, cols = brewer.pal(5, "RdYlGn"))
choropleth(blocks,HSE_UNITS,shading=shades)
choro.legend(533000,161000,shades)
```



```
title("St. Dev. Cuts", cex.main = 2)
# 3. Finally detach the data frame
detach(data.frame(blocks))
# reset par(mfrow)
par(mfrow=c(1,1))
```

Your map should look something like Figure 3.22:

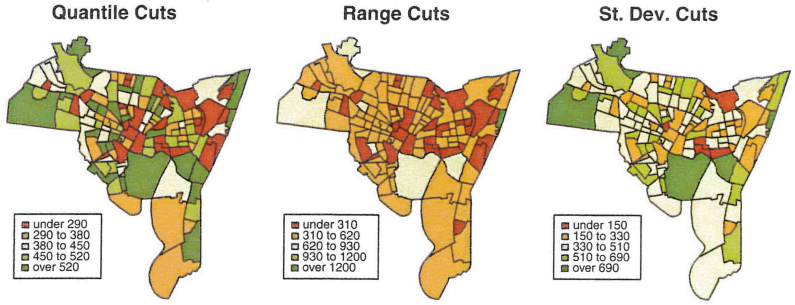

Figure 3.22 The map produced by the code for Self-Test Question 2

Q3. Selecting data: creating variables and subsetting data using logical statements

```
# attach the data frame
attach(data.frame(georgia2))
# calculate rural population
rur.pop <- PctRural * TotPop90 / 100
# calculate county areas in km^2
areas <- gArea(georgia2, byid = T)
areas <- as.vector(areas / (1000* 1000))
# calculate rural density
rur.pop.den <- rur.pop/areas
# detach the data frame
detach(data.frame(georgia2))
# select counties with density > 20
index <- rur.pop.den > 20
# plot them
plot(georgia2[index,], col = "chartreuse4")
# plot the non-rural counties
plot(georgia2[!index,], col = "darkgoldenrod3", add = TRUE)
# add some fancy bits
title("Counties with a rural population density
      of >20 people per km^2", sub = "Georgia, USA")
rect(850000, 925000, 970000, 966000, col = "white")
legend(850000, 955000, legend = "Rural",
```

```
  bty = "n", pch = 19,          col = "chartreuse4")
legend(850000, 975000, legend = "Not Rural",
  bty = "n", pch = 19,          col = "darkgoldenrod3")
```

Your map should look something like Figure 3.23:

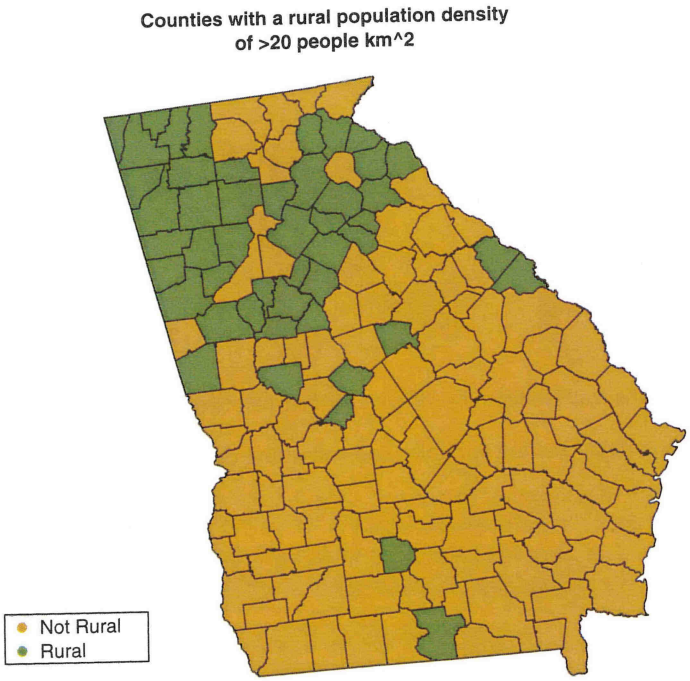

Figure 3.23 The map produced by the code for Self-Test Question 3

Q4. Re-projections: transforming data using `spTransfrom`

```
library(GISTools)  # for the mapping tools
library(rgdal)     # this has the spatial reference tools
library(RgoogleMaps)
library(PBSmapping)
data(newhaven)
# define a new projection
newProj <- CRS("+proj=longlat +ellps=WGS84")
# transform blocks and breach
breach2 <- spTransform(breach, newProj)
blocks2 <- spTransform(blocks, newProj)
# extract coordinates to pass to Google
coords <- coordinates(breach2)
```



```
Lat <- coords[,2]
Long <- coords[,1]
# download map
MyMap <- MapBackground(lat=Lat, lon=Long, zoom = 20)
# convert polys to PBS format
shp <- SpatialPolygons2PolySet(blocks2)
# plot polys on map with shading
PlotPolysOnStaticMap(MyMap, shp, lwd=0.7,
        col = rgb(0.75,0.25,0.25,0.15), add = F)
# now plot points
PlotOnStaticMap(MyMap,Lat,Long,pch=1,col='red', add = TRUE)
```

Your map should look something like Figure 3.24:

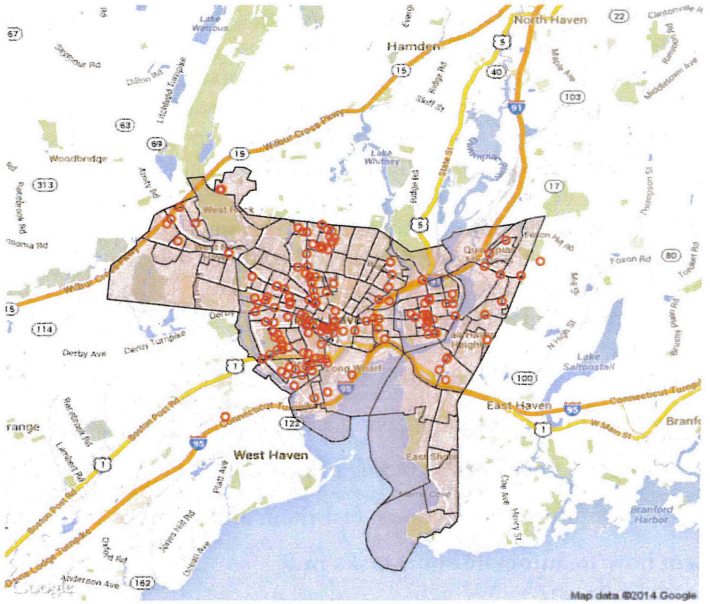

Figure 3.24 The map produced by the code for Self-Test Question 4

REFERENCES

de Vries, A. and Meys, J. (2012) *R for Dummies*. Chichester: John Wiley & Sons.
Monmonier, M. (1996) *How to Lie with Maps*, 2nd edn. Chicago: University of Chicago Press.