An INTRODUCTION to

R

for SPATIAL ANALYSIS & MAPPING

SECOND EDITION

CHRIS BRUNSDON and LEX COMBER

An Introduction to R for Spatial Analysis and Mapping

In the digital age, social and environmental scientists have more spatial data at their fingertips than ever before. But how do we capture this data, analyse and display it, and, most importantly, how can it be used to study the world?

Spatial Analytics and GIS is a series of books that deal with potentially tricky technical content in a way that is accessible, usable and useful. Early titles include *Urban Analytics* by Alex Singleton, Seth Spielman and David Folch, and *An Introduction to R for Spatial Analysis* and *Mapping* by Chris Brunsdon and Lex Comber.

Series Editor: Richard Harris

About the Series Editor

Richard Harris is Professor of Quantitative Social Geography at the School of Geographical Sciences, University of Bristol. He is the lead author on three textbooks about quantitative methods in geography and related disciplines, including *Quantitative Geography: The Basics* (Sage, 2016).

Richard's interests are in the geographies of education and the education of geographers. He is currently Director of the University of Bristol Q-Step Centre, part of a multimillion-pound UK initiative to raise quantitative skills training among social science students, and is working with the Royal Geographical Society (with IBG) to support data skills in schools.

Books in this Series:

Geocomputation, Chris Brunsdon and Alex Singleton

GIS and Agent-Based Modelling and *Geographical Information Systems*,

Andrew Crooks, Nicolas Malleson, Ed Manley and Alison Heppenstall

Modelling Environmental Change, Colin Robertson

An Introduction to Big Data and Spatial Data Analytics in R,

Lex Comber and Chris Brunsdon

Published in Association with this Series:

Quantitative Geography, Richard Harris

An Introduction to R for Spatial Analysis and Mapping

Second Edition

Chris Brunsdon
and Lex Comber

Los Angeles
London
New Delhi
Singapore
Washington DC
Melbourne

SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC | Melbourne

SAGE Publications Ltd

1 Oliver's Yard

55 City Road

London EC1Y 1SP

SAGE Publications Inc.

2455 Teller Road

Thousand Oaks, California 91320

SAGE Publications India Pvt Ltd

B 1/I 1 Mohan Cooperative Industrial Area

Mathura Road

New Delhi 110 044

SAGE Publications Asia-Pacific Pte Ltd

3 Church Street

#10-04 Samsung Hub

Singapore 049483

© Chris Brunsdon and Lex Comber 2019

First edition published 2015. Reprinted 2015 (twice), 2016 (twice) and 2017 (twice)

This edition first published 2019

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act, 1988, this publication may be reproduced, stored or transmitted in any

form, or by any means, only with the prior permission in writing of the publishers, or in the case of reprographic reproduction, in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

Library of Congress Control Number: 2018943836

British Library Cataloguing in Publication data

A catalogue record for this book is available from the British Library

ISBN 978-1-5264-2849-3

ISBN 978-1-5264-2850-9 (pbk)

Editor: Robert Rojek

Assistant editor: John Nightingale

Production editor: Katherine Haw

Copyeditor: Richard Leigh

Proofreader: Neville Hankins

Indexer: Martin Hargreaves

Marketing manager: Susheel Gokarakonda

Cover design: Francis Kenney

Typeset by: C&M Digitals (P) Ltd, Chennai, India

Printed in the UK

At SAGE we take sustainability seriously. Most of our products are printed in the UK using responsibly sourced papers and boards. When we print overseas we ensure sustainable papers are used as measured by the PREPS grading system. We undertake an annual audit to monitor our sustainability.

Praise for *An Introduction to R for Spatial Analysis and Mapping* 2e

‘There’s no better text for showing students and data analysts how to use R for spatial analysis, mapping and reproducible research. If you want to learn how to make sense of geographic data and would like the tools to do it, this is your guide.’

Richard Harris, University of Bristol

‘The future of GIS is open-source! *An Introduction to R for Spatial Analysis and Mapping* is an ideal introduction to spatial data analysis and mapping using the powerful open-source language R. Assuming no prior knowledge, Brunsdon and Comber get the reader up to speed quickly with clear writing, excellent pedagogic material and a keen sense of geographic applications. The second edition is timely and fresh. This book should be required reading for every Geography and GIS student, as well as faculty and professionals.’

Harvey Miller, The Ohio State University

‘While there are many books that provide an introduction to R, this is one of the few that provides both a general and an application-specific (spatial analysis) introduction and is therefore far more useful and accessible. Written by two experts in the field, it covers both the theory and practice of spatial statistical analysis and will be an important addition to the bookshelves of researchers whose spatial analysis needs have outgrown currently available GIS software.’

Jennifer Miller, University of Texas at Austin

‘Students and other life-long learners need flexible skills to add value to

spatial data. This comprehensive, accessible and thoughtful book unlocks the spatial data value chain. It provides an essential guide to the R spatial analysis ecosystem. This excellent state-of-the-art treatment will be widely used in student classes, continuing professional development and self-tuition.'

Paul Longley, University College London

'In this second edition, the authors have once again captured the state of the art in one of the most widely used approaches to spatial analysis. Spanning from the absolute beginner to more advanced concepts and underpinned by a strong "learn by doing" ethos, this book is ideally suited for both students and teachers of spatial analysis using R.'

Jonny Huck, The University of Manchester

'A timely update to the *de facto* reference and textbook for anyone – geographer, planner, or (geo)data scientist – needing to undertake mapping and spatial analysis in R. Complete with self-tests and valuable insights into the transition from sp to sf, this book will help you to develop your ability to write flexible, powerful, and fast geospatial code in R.'

Jonathan Reades, King's College London

'Brunsdon and Comber's 2nd edition of their acclaimed text book is updated with the key developments in spatial analysis and mapping in R and maintains the pedagogic style that made the original volume such an indispensable resource for teaching and research.'

Scott Orford, Cardiff University

Contents

About the authors
1 INTRODUCTION
- 1.1 Introduction to the Second Edition
- 1.2 Objectives of This Book
- 1.3 Spatial Data Analysis in R
- 1.4 Chapters and Learning Arcs
- 1.5 Specific Changes to the Second Edition
- 1.6 The R Project for Statistical Computing
- 1.7 Obtaining and Running the R Software
- 1.8 The R Interface
- 1.9 Other Resources and Accompanying Website
- References

2 DATA AND PLOTS
- 2.1 Introduction
- 2.2 The Basic Ingredients of R: Variables and Assignment
- 2.3 Data Types and Data Classes
- 2.4 Plots
- 2.5 Another Plot Option: ggplot
- 2.6 Reading, Writing, Loading and Saving Data
- 2.7 Answers to Self-Test Questions
- Reference

3 BASICS OF HANDLING SPATIAL DATA IN R
- 3.1 Overview
- 3.2 Introduction to sp and sf: The sf Revolution
- 3.3 Reading and Writing Spatial Data
- 3.4 Mapping: An Introduction to tmap
- 3.5 Mapping Spatial Data Attributes
- 3.6 Simple Descriptive Statistical Analyses
- 3.7 Self-Test Questions
- 3.8 Answers to Self-Test Questions
- References

4 SCRIPTING AND WRITING FUNCTIONS IN R
- 4.1 Overview
- 4.2 Introduction

4.3 Building Blocks for Programs
4.4 Writing Functions
4.5 Spatial Data Structures
4.6 apply Functions
4.7 Manipulating Data with dplyr
4.8 Answers to Self-Test Questions
5 USING R AS A GIS
5.1 Introduction
5.2 Spatial Intersection and Clip Operations
5.3 Buffers
5.4 Merging Spatial Features
5.5 Point-in-Polygon and Area Calculations
5.6 Creating Distance Attributes
5.7 Combining Spatial Datasets and Their Attributes
5.8 Converting between Raster and Vector
5.9 Introduction to Raster Analysis
5.10 Answers to Self-Test Questions
References
6 POINT PATTERN ANALYSIS USING R
6.1 Introduction
6.2 What is Special about Spatial?
6.3 Techniques for Point Patterns Using R
6.4 Further Uses of Kernel Density Estimation
6.5 Second-Order Analysis of Point Patterns
6.6 Looking at Marked Point Patterns
6.7 Interpolation of Point Patterns with Continuous Attributes
6.8 The Kriging Approach
6.9 Concluding Remarks
6.10 Answers to Self-Test Questions
References
7 SPATIAL ATTRIBUTE ANALYSIS WITH R
7.1 Introduction
7.2 The Pennsylvania Lung Cancer Data
7.3 A Visual Exploration of Autocorrelation
7.4 Moran's: An Index of Autocorrelation
7.5 Spatial Autoregression
7.6 Calibrating Spatial Regression Models in R

7.7 Answer to Self-Test Question
References
8 LOCALISED SPATIAL ANALYSIS
8.1 Introduction
8.2 Setting Up the Data Used in This Chapter
8.3 Local Indicators of Spatial Association
8.4 Further Issues with the Above Analysis
References
9 R AND INTERNET DATA
9.1 Introduction
9.2 Direct Access to Data
9.3 Using RCurl
9.4 Working with APIs
9.5 Creating a Statistical 'Mashup'
9.6 Using Specific Packages
9.7 Web Scraping
References
10 EPILOGUE
10.1 The Future of R as a Tool for Geocomputation
10.2 Extensions of R as a Language
10.3 Improvements 'Under the Bonnet'
10.4 Coexistence with Other Software
10.5 Finally…
References
Index

About the authors

Chris Brunsdon

is Professor of Geocomputation and Director of the National Centre for Geocomputation at the National University of Ireland, Maynooth, having worked previously in the Universities of Newcastle, Glamorgan, Leicester and Liverpool, variously in departments focusing on both geography and computing. He has interests that span both of these disciplines, including spatial statistics, geographical information science, and exploratory spatial data analysis, and in particular the application of these ideas to crime pattern analysis, the modelling of house prices, medical and health geography and the analysis of land use data. He was one of the originators of the technique of geographically weighted regression (GWR). He has extensive experience of programming in R, going back to the late 1990s, and has developed a number of R packages which are currently available on CRAN, the Comprehensive R Archive Network. He is an advocate of free and open source software, and in particular the use of reproducible research methods, and has contributed to a large number of workshops on the use of R and of GWR in a number of countries, including the UK, Ireland, Japan, Canada, the USA, the Czech Republic and Australia. When not involved in academic work he enjoys running, collecting clocks and watches, and cooking – the last of these probably cancelling out the benefits of the first.

Alexis Comber

(Lex) is Professor of Spatial Data Analytics at Leeds Institute for Data Analytics (LIDA), University of Leeds. He worked previously at the University of Leicester where he held a chair in geographical information science. His first degree was in plant and crop science at the University of Nottingham, and he completed a PhD in computer science at the Macaulay Institute, Aberdeen (now the James Hutton Institute) and the University of Aberdeen, developing expert systems for land cover monitoring. This brought him into the world of spatial data, spatial analysis, and mapping. Lex’s interests span many different application areas, including land cover/land use, demographics, public health,

agriculture, bio-energy and accessibility, all of which require multi-disciplinary approaches. His research draws from geocomputation, mathematics, statistics and computer science, and he has extended techniques in operations research/location–allocation (what to put where), graph theory (cluster detection in networks), heuristic searches (how to move intelligently through highly dimensional big data), remote sensing (novel approaches for classification), handling divergent data semantics (uncertainty handling, ontologies, text mining) and spatial statistics (quantifying spatial and temporal process heterogeneity). Outside of academic work and in no particular order, Lex enjoys his vegetable garden, walking the dog and playing pinball (he is the proud owner of a 1981 Bally Eight Ball Deluxe).

1 Introduction

1.1 Introduction to the Second Edition

Since the first edition of this book was drafted and subsequently published, there have been a number of developments in the handling of data and spatial data in R. The use of R has exploded, and it is now a common tool taught at undergraduate and postgraduate level in many courses. This is due to a number of interrelated factors. Perhaps the most critical of these from a scientific point of view is that R is free and open source, which means that the code and functions used to manipulate data are transparent and can be integrated by the user, rather than being simply presented as black boxes as is common in many commercial software packages. Additionally, R is underpinned by a core statistical functionality that provides the basis for rigorous analysis and confident package development. Finally, R provides a dynamic analysis environment in which new packages are constantly developed, refined and updated.

One such set of developments is at the heart of the second edition of this book: the emergence of tidy and lazy data formats and structures for spatial and non-spatial data, to improve data manipulations, data wrangling and data handling supporting cleaner data science. The most notable example of this is the tidyverse, which is a collection of R packages designed for data science (https://www.tidyverse.org). These provide a suite of tools for data analysis, linkage and data visualisation, but also augmented data formats such as the tibble and language extending operations using a piping syntax. Similar developments have also occurred in mapping, spatial data and spatial data analysis in R, such as the tmap package for thematic mapping (Tennekes, 2015) and the sf package that includes both new data structures and tools for handling spatial data (Pebesma et al., 2016).

In the same way that the first edition of this book, written in 2013, reflected our practice and how we worked with spatial data in R at that time, so the second edition reflects our current practice and the techniques we now use. In 2013, spatial data analysis was undertaken using data in the sp format, as

defined in the sp package, and using tools drawn from a range of packages underpinned by the sp data format such as rgdal and maptools. The first edition had a strong focus on the GISTools package (Brunsdon and Chen, 2014) which wrapped many functions from other packages with an sp underpinning. Now we work mainly with spatial data in sf format (described more fully in Chapter 3). At the time of writing, the R spatial community is in a period of transition from sp to sf formats and so both are introduced and discussed in this second edition. Many packages with spatial operations and functions for spatial analyses have not yet been updated to work with sf. For these reasons, this edition will, where possible, describe the manipulation and analysis of spatial data using sf format and functions but will switch between (and convert data between) sp and sf formats as needed. The focus is no longer primarily on GISTools, but this package still provides some analytical short-cuts and functionality and will be used if appropriate.

R is dynamic – things do not stay the same, and this is part of its attraction and to be celebrated. New tools, packages and functions are constantly being produced, and they are updated to improve and develop them. In most cases this is not problematic as the update almost always extends the functionality of the package without affecting the original code. However, in a few instances, specific packages are completely rewritten without backward compatibility. If this happens then the R code that previously worked may not work with the new package as the functions may take different parameters, arguments and critical data formats. However, there is usually a period of transition over some package versions before the code stops working altogether. So occasionally a completely new paradigm is introduced, and this has been the case recently for spatial data in R with the release of the sf package (Pebesma et al., 2016) and the tidyverse. The second edition reflects these developments and updates.

1.2 Objectives of this Book

This book assumes no prior knowledge of either R or spatial analysis and mapping. It provides an introduction to the use of R and the increasing number of tools that can be used for explicitly spatial analyses, geocomputation and the statistical analysis of geographical information. The text draws from a number of open source, user-contributed libraries or

‘packages’ that support mapping and cartographic outputs arising from both raster and vector analyses. The book implicitly focuses on vector GIS as other texts cover raster with classic geostatistics (e.g. Bivand et al., 2013), although rasters are implicitly included in some of the exercises, for example the outputs of density surfaces and some of the geographically weighted analyses as described in later chapters.

The original rationale for producing the first edition of this book in 2013 related to a number of factors. First, the increasing use of R as an analytical tool across a range of different scientific disciplines is evident. Second, there are an increasing number of data capture devices that are GPS-enabled: smartphones, tablets, cameras, etc. This has resulted in more and more data (both formal and informal) having location attached to them. Third, there is therefore an associated increase in demand for explicitly spatial analyses of such data, in order to exploit the richness of analysis that location affords. Finally, at the time of writing, there are no books on the market that have a specific focus on spatial analysis and mapping of such data in R that do not require any prior knowledge of GIS, spatial analysis or geocomputation. One of the few textbooks on using R for the analysis of spatial data is Bivand et al. (2013), although this is aimed at advanced users. These have not changed. If anything, the number of R users has increased, and of those more and more are increasingly working with *spatial* data. This is reflected in the number of online tools, functions and tutorials (greatly supported by the functionality of RMarkdown) and the continued development of packages (existing and new) and data formats supporting spatial data analysis. As introduced earlier, an excellent example of the latter is the Simple Features format in the sf package. For these reasons, what we have sought to do is to write a book with a geographical focus and (hopefully) user friendliness and that reflects the latest developments in spatial analyses and mapping in R.

As you work through this book you will learn a number of techniques for using R directly to carry out spatial data analysis, visualisation and manipulation. Although here we focus mostly on vector data (some raster analysis is demonstrated) and on social and economic applications, and the packages that this book uses have been chosen as being the most appropriate for analysing these kinds of data, R also presents opportunities for the analysis of many other kinds of spatial data – for example, relating to climate

and landscape processes. While some of libraries and packages covered in this book may also be useful in the analysis of the physical geographical and environmental data, there will no doubt be other packages that may also play an important role – for example, the PBSMapping package, developed by the Pacific Biological Station in Nanaimo, British Columbia, Canada, offers a number of functions that may be useful for the analysis of biogeographical data.

1.3 Spatial Data Analysis in R

In recent years large amounts of spatial data have become widely available. For example, there are many governmental open data initiatives that make census data, crime data and various other data relating to social and economic processes freely available. However, there is still a need to flexibly analyse, visualise and model data of these kinds in order to understand the underlying patterns and processes that the data describe. While there are many packages and software available that are capable of analysing spatial data, in many situations standard statistical modelling approaches are not appropriate: data observations may not be independent or the relationship between variables may vary across geographical space. For this reason many standard statistical packages provide only inadequate tools for analysis as they cannot account for the complexities of spatial processes and spatial data.

Similarly, although standard GIS packages and software provide tools for the visualisation of spatial data, their analytical capabilities are relatively limited, inflexible and cannot represent the state of the art. On the other hand, many R packages are created by experts and innovators in the field of spatial data analysis and visualisation, and as R is, in fact, a programming language it is a natural testing ground for newly developed approaches. Thus R provides arguably the best environment for spatial data analysis and manipulation. One of the key differences between a standard GIS and R is that many people view GIS as a tool to handle very large geographical databases rather than for more sophisticated modelling and analysis, and this is reflected in the evolution of GIS software, although R is catching up in its ability to easily handle very large datasets. We do not regard R as competing with GIS, rather we see the two kinds of software as having complementary functionality.

1.4 Chapters and Learning arcs

The broad-level content and topics covered by the chapters have not changed. Nor have the associated learning arcs. The revisions for the second edition have focused on updates to visualisation and mapping tools through the ggplot2 and tmap packages and to spatial data structures through sf.

The chapters build in the complexity of the analyses they develop, and by working through the illustrative code examples you will develop skills to create your own routines, functions and programs. The book includes a mix of *embedded exercises*, where the code is provided for you to work through with extensive explanations, and *self-test questions*, which require you to develop an answer yourself. All chapters have self-test questions. In some cases these are included in an explicitly named section, and in others they are embedded in the rest of the text. The final section in each chapter provides model answers to the self-test questions. Thus in contrast to the exercises, where the code is provided in the text for you to work through (i.e. for you to enter and run yourself), the self-test questions are tasks for you to complete, mostly requiring you to write R code yourself, with answers provided in the last section of each chapter. The idea of these questions is to give you some experience with working with different kinds of data structures, functions and operations in R. There is a strong emphasis on solving problems, rather than simply working through the code. In this way, snippets of code are included in each chapter describing commands for data manipulation and analysis and to exemplify specific functionality. It is expected that you will run the R code yourself in each chapter. This can be typed directly into the R console or may be written directly into a script or document as described below. It is also possible to access the code in each chapter from the book's website (again see below). The reasons for running the code yourself are so that you get used to using the R console and to help your understanding of the code's functionality.

In various places *information boxes* are included to develop a deeper understanding of functions and alternative approaches for achieving the same ends.

The book is aimed at both second- and third-year undergraduate and

postgraduate students. Chapters 6–8 go into much more detail about specific types of spatial analysis and are extensively supported by references from the scientific literature in a way that the earlier chapters are not. For these reasons Chapters 2–5 might be considered as introductory and Chapters 6–8 might be considered as advanced. Thus the earlier chapters are suitable for an *Introduction to R* module (Chapters 2–4) or for an *Introduction to Mapping in R* module, and the later ones for a module covering more *Advanced Techniques* (Chapters 6–9). The book could also be used as the basis for a *Geographical Programming* module, drawing from different chapters, especially Chapters 4 and 9, depending on the experience and technical capabilities of the student group.

The formal learning objectives of this book are:

- to apply appropriate data types, arrays, control structures, functions and packages within R code
- to introduce geographical analysis and spatial data handling in R
- to develop programming skills in the R language with particular reference to current geocomputational research and applications
- to exemplify the principles of algorithm and function construction in R
- to design and construct basic graphical algorithms for the analysis and visualisation of spatial information

In terms of learning arcs, each chapter introduces a topic, has example code to run and self-test questions to work through. In a similar way, earlier chapters provide the foundations for later ones. The dependencies and prerequisites for each chapter are listed in Table 1.1, and you should note that these are inherited (i.e. if Chapter 4 is a prerequisite then the prerequisites for Chapter 4 also are relevant).

1.5 Specific Changes to the Second Edition

In Chapter 2 the main changes were to introduce the ggplot2 package alongside the basic plot operation. The code for some figures, maps and plots is shown for both approaches. The other change was to remove the use of deprecated maptools functions for reading and writing spatial data and to replace these with readOGR and writeOGR functions from the rgdal package

and the st_read function in sf. The self-test questions in each chapter reflect these changes.

Table 1.1

Table 1.1 Chapter prerequisites

Chapter	Prerequisite chapters	Comments
Chapter 2	None	Data types and plots – the jumping-off point for all other chapters
Chapter 3	2	The first maps and spatial data types
Chapter 4	2, 3	Coding blocks and functions
Chapter 5	2, 3	GIS-like operations in R
Chapter 6	4, 5	Cluster analysis and mapping of point data
Chapter 7	4, 5	Attribute analysis and mapping of polygon data
Chapter 8	6, 7	Analysis of geographical variation in spatial processes
Chapter 9	3, 4, 5	Spatial analysis of data from the web

Chapter 3 covers the basics of handling spatial data. This chapter now has a focus on operations on sf objects and tools and a much reduced focus on sp formats and the GISTools package, although it still draws from some of the functionality of packages based on sp. The data manipulations now incorporate operations on both sp and sf objects, bridging between the two data formats. In a similar way, the GISTools mapping functions have been replaced by code using the tmap package, and again many simple plot routines have been replaced with ggplot2 operations.

Chapter 4 has a few small changes relating to some data table manipulations using the functions in the dplyr package and demonstrates the use of apply functions as an alternative to potentially slower (but perhaps more transparent) for loop operations.

Chapter 5 goes into sf operations in much more detail and ubiquitously uses tmap. The detailed walk-through coding exercises mix sp and sf formats, using sf where possible, but where we think there is a distinct advantage to using sp then this has been presented.

Chapters 6–9 have been revised much less than the earlier chapters, although a new example has been added to Chapter 9 to reflect changes in web API support in R. This is because they are focused on more advanced topics, the

nuts and bolts of which have not changed much. However, where appropriate the plotting and mapping routines have been updated to use tmap and ggplot2 packages.

Chapter 10, the epilogue, evaluates our 2013 thoughts about the direction of travel in this area and considers the main developments from where we are now in 2018, including the extensions to R, improvements under the bonnet and the coexistence of R with other software arising from the tidyverse, piping syntax, sf formats, Rcpp, the ubiquity of RStudio as the choice of R interface and tools such as RMarkdown. An example of the latter is that the first edition of this book was written in Sweave and the second edition entirely in RMarkdown.

1.6 The R Project for Statistical Computing

R was developed from the S language which was originally conceived at the Lucent Technologies (formerly AT&T) Bell Laboratories in the 1970s and 1980s. Douglas Martin at the company StatSci developed S into the enhanced commercial product known as S+ in the late 1980s and early 1990s (Krause and Olson, 1997). R was initially developed by Robert Gentleman and Ross Ihaka of the Department of Statistics at the University of Auckland. It is becoming widely used in many areas of scientific activity and quantitative research, partly because it is available free in source code form and also because of its extensive functionality, through the continually growing number of contributions of code and functions, in the form of R packages, which when installed can be called as libraries. The background to R, along with documentation and information about packages as well as the contributors, can be found at the R Project website http://www.r-project.org.

1.7 Obtaining and Running the R Software

We assume that most readers will be using the RStudio interface to R. You should download the latest version of R and then RStudio in order to run the code provided in this book. At the time of writing, the latest version of R is version 3.4.3 and you should ensure you have at least this version. There are 32-bit and 64-bit versions available, and we assume you have the 64-bit

version. The simplest way to get R installed on your computer is to go the download pages on the R website – a quick search for 'download R' should take you there, but if not you could try:

- http://cran.r-project.org/bin/windows/base/
- http://cran.r-project.org/bin/macosx/
- http://cran.r-project.org/bin/linux/

for Windows, Mac and Linux, respectively. The Windows and Mac versions come with installer packages and are easy to install, while the Linux binaries require use of a command terminal.

RStudio can be downloaded from https://www.rstudio.com/products/rstudio/download/ and the free version of RStudio Desktop is more than sufficient for this book. RStudio allows you to organise your work into projects, to use RMarkdown to create documents and webpages, to link to your GitHub site and much more. It can be customised for your preferred arrangement of the different panes.

You may have to set a *mirror* site from which the installation files will be downloaded to your computer. Generally you should pick one that is near to you. Once you have installed the software you can run it. On a Windows computer, an R icon is typically installed on the desktop; on a Mac, R can be found in the Applications folder. Macs and Windows have slightly different interfaces, but the protocols and processes for an R session on either platform are similar.

The base installation includes many functions and commands. However, more often we are interested in using some particular functionality, encoded into packages contributed by the R developer community. Installing packages for the first time can be done at the command line in the R console using the install.packages command, as in the example below to install the GISTools library, or via the R menu items.

```
install.packages("tmap", dependencies = T)
```

In Windows, the menu for this can be accessed by **Packages > Load**

Packages and on a Mac via **Packages and Data > Package Installer**. In either case, the first time you install packages you may have to set a mirror site, from which to download the packages. Once the package has been installed then the library can be called as below.

```
library(tmap)
```

Further descriptions of packages, their installation and their data structures are given in later chapters. There are literally thousands of packages that have been contributed to the R project by various researchers and organisations. These can be located by name at http://cran.r-project.org/web/packages/available_packages_by_name.html if you know the package you wish to use. It is also possible to search the CRAN website to find packages to perform particular tasks at http://www.r-project.org/search.html. Additionally, many packages include user guides in the form of a PDF document describing the package and listed at the top of the index page of the help files for the package. The most commonly used packages in this book are listed in Table 1.2.

When you install these packages it is strongly suggested you also install the dependencies – other packages required by the one that is being installed – by either checking the box in the menu or including depend=TRUE in the command line as below:

```
install.packages("GISTools", dep = TRUE)
```

Packages are occasionally completely rewritten, and this can impact on code functionality. Since we started writing the revision for this edition of the book, the read and write functions for spatial data in the maptools package (readShape Poly, writePolyShape, etc.) have deprecated. For instance:

Table 1.2

Table 1.2 R packages used in this book

Name	Description
datasets	A package containing a number of datasets supplied with the standard installation of R
deldir	Functions for Delaunay triangulations, Dirichlet or Voronoi tessellations of point datasets
dplyr	A grammar of data manipulation
e1071	Functions for data mining, latent class analysis, clustering and modelling
fMultivar	Tools for financial engineering but useful for spatial data
ggplot2	Declarative graphics creation, based on *The Grammar of Graphics* (Wilkinson, 2005)
GISTools	Mapping and spatial data manipulation tools
gstat	Functions for spatial and geostatistical modelling, prediction and simulation
GWmodel	Geographically weighted models
maptools	Functions for manipulating and reading geographical data
misc3d	Miscellaneous functions for three-dimensional (3D) plots
OpenStreetMap	High resolution raster maps and satellite imagery from OpenStreetMap
raster	Manipulating, analysing and modelling of raster or gridded spatial data
RColorBrewer	A package providing colour palettes for shading maps and other plots
RCurl	General HTTP requests, functions to fetch uniform resource identifiers (URIs), to get and post web data
reshape2	Flexibly reshape data
rgdal	Geospatial Data Abstraction Library, projection/transformation operations
rgeos	Geometry Engine – Open Source (GEOS), topology operations on geometries
rgl	3D visualisation device (OpenGL)
RgoogleMaps	Interface to query the Google server for static maps as map backgrounds
Rgraphviz	Provides plotting capabilities for R graph objects
rjson	Converts R objects into JavaScript Object Notation (JSON) objects and vice versa
sf	Simple Features for R – a standardised way to encode spatial vector data
sp	Classes and methods for spatial data
SpatialEpi	Performs various spatial epidemiological analyses
spatstat	A package for analysing spatial data, mainly spatial point patterns
spdep	Functions and tests for evaluating spatial patterns and autocorrelation
tibble	A modern reimagining of the data frame
tidyverse	A collection of R packages designed for data science
tmap	A mapping package that allows maps to be constructed in highly controllable layers

```
library(maptools)

?readShapePoly
```

If you examine the help files for these functions you will see that they contain a warning and suggest other functions that should be used instead. The book website will always contain working code snippets for each chapter to overcome any problems caused by function deprecation.

Such changes are only a minor inconvenience and are part of the nature of a dynamic development environment provided by R in which to do research: such changes are inevitable as packages finesse, improve and standardise.

1.8 The R Interface

We expect that most readers of this book and most users of R will be using the RStudio interface to R, although users can of course still use just R. RStudio provides a good interface to the different things that R users will want to know about the R sessions via the four panes: the console where code is entered; the file that is being edited; variables in the working environments; files in the project file space; plot windows, help pages, as well as font type and size, pane colour, etc. Users can set up their personal preferences for how they like their RStudio interface. Similar to straight R, there are few pull-down menus in R, and therefore you will type command lines in what is termed a *command line interface*. Like all command line interfaces, the learning curve is steep but the interaction with the software is more detailed, which allows greater flexibility and precision in the specification of commands.

As you work though the book, the expectation is that you will run all the code that you come across. We cannot emphasise enough the importance of learning by doing – the best way to learn how to write R code is to write and enter it. Some of the code might look a bit intimidating when first viewed, especially in later chapters. However, the only really effective way to understand it is to give it a try.

Beyond this there are further choices to be made. Command lines can be entered in two forms: directly into the *R console* window or as a series of commands into a script window. We strongly advise that all code should be written in scripts (script files have a .R extension) and then run from the script. RStudio includes its own editor (similar to Notepad in Windows or

TextEdit on a Mac). Scripts are useful if you wish to automate data analysis, and have the advantage of keeping a saved record of the relevant R programming language commands that you use in a given piece of analysis. These can be re-executed, referred to or modified at a later date. For this reason, you should get into the habit of constructing scripts for all your analyses. Since being able to edit functions is extremely useful, both the MS Windows and Mac OSX versions of R have built-in text editors. In RStudio you should go to **File > New File**. In R, to start the Windows editor with a blank document, go to **File > New Script**, and to open an existing script, **File > Open Script**. To start the Mac editor, use the menu option **File > New Document** to open a new document and **File > Open Document** to open an existing file.

Once code is written into these files, they can be saved for future use; rather than copy and pasting each line of code, both R and RStudio have their own short-cuts. Lines of code can be run directly by placing the cursor on the relevant line (or highlighting a block) and then using Ctrl-R (Windows) or Cmd-Return (Mac). RStudio also has a number of other keyboard short-cuts for running code, auto-filling when you are typing, assignment, etc. Further tips are described at http://r4ds.had.co.nz/workflow-basics.html.

It is also good practice to set the working directory at the beginning of your R session. This can be done via the menu in RStudio: **Session > Set Working Directory > …**. In Windows R select **File > Change dir…,** and in Mac R select **Misc > Set Working Directory**. This points the R session to the folder you choose and will ensure that any files you wish to read, write or save are placed in this directory.

Scripts can be saved by selecting **File > Save As** which will prompt you to enter a name for the R script you have just created. Chose a name (e.g. test.R) and select save. It is good practice to use the file extension .R.

1.9 Other Resources and Accompanying Website

There are many freely available resources for R users. In order to get some practice with R we strongly suggest that you download the 'Owen Guide' (entitled *The R Guide*) and work through this up to and including Section 5. It

can be accessed via http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf. It does not require any additional libraries or data and provides a gentle introduction to R and its syntax.

There are many guides to the R software available on the internet. In particular, you may find some of the following links useful:

- http://www.r-bloggers.com
- http://stackoverflow.com/ and specifically http://stackoverflow.com/questions/tagged/r

The contemporary nature of R means that much of the R development for processing geographical information is chronicled on social media sites (you can search for information on services such as Twitter, for example #rstats) and blogs (such as the R-bloggers site listed above), rather than standard textbooks. In addition to the above resources, there is a website that accompanies this book:**https://study.sagepub.com/Brunsdon2e**. This site contains all of the code, scripts, exercises and self-test questions included in each chapter, and these are available to download. The scripts for each chapter allow the reader to copy and paste the code into the R console or into their own script. At the time of writing, all of the code in the book is correct. However, R and its packages are occasionally updated. In most cases this is not problematic as the update almost always extends the functionality of the package without affecting the original code. However, in a few instances, specific packages are completely rewritten without backward compatibility. If this happens the code on the accompanying website will be updated accordingly. You are therefore advised to check the website regularly for archival components and links to new resources.

References

Bivand, R.S., Pebesma, E.J. and Gómez-Rubio, V. (2013) Applied Spatial Data: Analysis with R, 2nd edition. New York: Springer.

Brunsdon, C. and Chen, H. (2014) GISTools: Some further GIS capabilities for R. R Package Version 0.7-4. http://cran.r-project.org/package=GISTools.

Krause, A. and Olson, M. (1997) The Basics of S and S-PLUS. New York: Springer.

Pebesma, E., Bivand, R., Cook, I., Keitt, T., Sumner, M., Lovelace, R., Wickham, H., Ooms, J. and Racine, E. (2016) sf: Simple features for R. R Package Version 0.6-3. http://cran.r-project.org/package=sf.

Tennekes, M. (2015) tmap: Thematic maps. R Package Version 1. http://cran.r-project.org/package=tmap.

Wilkinson, L. (2005) The Grammar of Graphics. New York: Springer.

2 Data and Plots

2.1 Introduction

This chapter introduces some of the different data types and data structures that are commonly used in R and how to visualise them. As you work through this book, you will gain experience in using and manipulating these individually and within blocks of code. It sequentially builds on the ideas that are introduced, for example developing your own functions, and tests this knowledge through self-test exercises. As you progress, the exercises will place more emphasis on solving problems, using the different data structures needed, rather than simply working through the example code. As you work though the code, you should use the help available to explore the different functions that are called in the code snippets, such as max, sqrt and length.

This chapter covers a lot of ground – it will:

- Review basic commands in R
- Introduce variables and assignment
- Introduce data types and classes
- Describe how to test for and manipulate data types
- Introduce and compare data frames and tibbles
- Introduce basic plot commands
- Describe how to read, write, load and save different data types

Chapter 1 introduced R, the reasons for using it in spatial analysis and mapping, and described how to install it. It also directed you to some of the many resources and introductory exercises for undertaking basic operations in R. Specifically it advised that you should work through the 'Owen Guide' (entitled *The R Guide*) up to the end of Section 5. This can be accessed via https://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf. This chapter assumes that you have worked your way through this – it does not take long and provides critical introductory knowledge for the more specialised materials that will be covered in the rest of this book.

2.2 The Basic Ingredients of R: Variables and Assignment

The R interface can be used as a sort of calculator, returning the results of simple mathematical operations such as (−5 + −4). However, it is normally convenient to *assign* values to *variables*. The form for doing this is:

R_object <- value

The arrow performs the assignments and is referred to as *gets*. So in this case you would say *R_object gets value*. It is possible to use an equals sign instead of gets, but this only performs a soft assignment (the difference between the arrow and the equals sign relates to how R how stores the R_object). The objects and variables that are created can then be manipulated or subject to further operations.

```
# examples of simple assignment

x <- 5

y <- 4

# the variables can be used in other operations

x+y

[1] 9

# including defining new variables

z <- x + y

z

[1] 9

# which can then be passed to other functions

sqrt(z)

[1] 3
```

```
I
```

The snippet of code above is the first that you have come across in this book. There will be further snippets throughout each chapter. Two key points. First, you are strongly advised to enter and run the code at the R prompt yourself. Our very strong advice is that you write the code into a script or document using the in-built text editor in RStudio. For example, for each chapter you might start a new RStudio session or project and open a new .R file. This script can be used to save the code snippets you enter and to include your comments and annotations. The reasons for doing this are so that you get used to using the R console, and running the code will help your understanding of the code's functionality. Lines of code can be run directly by placing the cursor on the line of code (or highlighting a block of code) and then using Ctrl-R (Windows) or Cmd-Return (Mac). Keeping copies of your code in this way will help you keep a record of it and will allow you to go back and edit it at a later date. Second, we would like to emphasise the importance of learning by doing and getting your hands dirty. Some of the code might look a bit fearsome when first viewed, especially in later chapters, but the only really effective way to understand it is to give it a try. Remember that the code and chapter summaries are available on the book's website **https://study.sagepub.com/Brunsdon2e** so that you can copy and paste these into the R console or your own script. A final point is that in the code, any comments are prefixed by # and are ignored by R when entered into the console.

The basic assignment type in R is to a *vector* of values. Vectors can have single values as in x, y and z above, or multiple values. Note the use of c(4.3,7.1, …) in the code below, where the c instructs R to combine or *concatenate* multiple values:

```
# example of vector assignment

tree.heights <- c(4.3,7.1,6.3,5.2,3.2,2.1)

tree.heights

[1] 4.3 7.1 6.3 5.2 3.2 2.1
```

Remember that **UPPER** and **lower** case matters to R. So tree.heights, Tree.Heights and TREE.HEIGHTS will be treated as referring to different variables by R. Make sure you type in upper and lower case exactly as it is written, otherwise you are likely to get an error.

In the example above, a vector of values has been assigned to the variable tree.heights. It is possible to apply a single assignment to the entire vector, as in the code below that returns tree.heights squared. Note how the operation returns the square of each element in the vector.

```
tree.heights**2
```

```
[1] 18.49 50.41 39.69 27.04 10.24 4.41
```

Other operations or functions can then be applied to these vectors variables:

```
sum(tree.heights)

[1] 28.2

mean(tree.heights)

[1] 4.7
```

And, if needed, the results can be assigned to yet further variables:

```
max.height <- max(tree.heights) max.height

[1] 7.1
```

One of the advantages of vectors and other structures with multiple data elements is that they can be subsetted. Individual elements or subsets of elements can be extracted and manipulated:

```
tree.heights

[1] 4.3 7.1 6.3 5.2 3.2 2.1

tree.heights[1]    # first element

[1] 4.3

tree.heights[1:3] # a subset of elements 1 to 3

[1] 4.3 7.1 6.3

sqrt(tree.heights[1:3])    #square roots of the subset

[1] 2.073644 2.664583 2.509980

tree.heights[c(5,3,2)]    # a subset of elements 5,3,2: note the ordering

[1] 3.2 6.3 7.1
```

In the above examples the numeric values were assigned. However, character or logical values can be also assigned as in the code below. This starts to hint

at the idea of different classes and types of variables which are described in more detail in the next sections.

```
# examples of character variable assignment

name <- "Lex Comber"

name

[1] "Lex Comber"

# these can be assigned to a vector of character variables

cities <- c("Leicester","Newcastle","London","Leeds","Exeter")

cities

[1] "Leicester" "Newcastle" "London"    "Leeds"

[5] "Exeter"

length(cities)

[1] 5

# an example of a logical variable

northern <- c(FALSE, TRUE, FALSE, TRUE, FALSE)

northern

[1] FALSE TRUE FALSE TRUE FALSE

# this can be used to subset other variables

cities[northern]

[1] "Newcastle" "Leeds"
```

2.3 Data Types and Data Classes

This section introduces data classes and data types to a sufficient depth for readers of this book. However, more formal descriptions of basic classes for R data objects can be found in the R Manual on the CRAN website at http://stat.ethz.ch/R-manual/R-devel/library/methods/html/BasicClasses.html.

2.3.1 Data Types in R

Data in R can be considered as being organised into a hierarchy of data types which can then be used to hold data values in different structures. Each of the types is associated with a test and a conversion function. The basic or core data types and associated tests and conversions are shown in Table 2.1.

You should note from the table that each type has an associated test in the form is.xyz, which will return TRUE or FALSE, and a conversion in the form as.xyz. Most of the exercises, methods, tools, functions and analyses in this book work with only a small subset of these data types: character, numeric and logical. These data types can be used to populate different data structures or classes, including vectors, matrices, data frames, lists and factors. The data types are described in more detail below. In each case the objects created by the different classes, conversion functions or tests are illustrated.

Table 2.1

Table 2.1 Data type, tests and conversion functions

Type	Test	Conversion
character	`is.character`	`as.character`
complex	`is.complex`	`as.complex`
double	`is.double`	`as.double`
expression	`is.expression`	`as.expression`
integer	`is.integer`	`as.integer`
list	`is.list`	`as.list`
logical	`is.logical`	`as.logical`
numeric	`is.numeric`	`as.numeric`
single	`is.single`	`as.single`
raw	`is.raw`	`as.raw`

2.3.1.1 Characters

Character variables contain text. By default the function character creates a vector of whatever length is specified. Each element in the vector is equal to "", an empty character element in the variable. The function as.character tries to convert its argument to character type, removing any attributes including, for example, vector element names. The function is.character tests whether

the arguments passed to it are of character type and returns TRUE or FALSE depending on whether its argument is of character type or not. Consider the following examples of these functions and the results when they are applied to different inputs:

```
character(8)

[1] "" "" "" "" "" "" "" ""

# conversion

as.character("8")

[1] "8"

# tests

is.character(8)

[1] FALSE

is.character("8")

[1] TRUE
```

2.3.1.2 Numeric

Numeric data variables are used to hold numbers. The function numeric is used to create a vector of the specified length with each element equal to 0. The function as.numeric tries to convert (coerce) its argument to numeric type. It is identical to as.double and to as.real. The function is.numeric tests whether the arguments passed to it are of numeric type and returns TRUE or FALSE depending on whether its argument is of numeric type or not. Notice how the last test in the code below returns FALSE because not all of the elements are numeric.

```
numeric(8)

[1] 0 0 0 0 0 0 0 0

# conversions

as.numeric(c("1980","−8","Geography"))
```

```
[1] 1980 −8 NA

as.numeric(c(FALSE,TRUE))

[1] 0 1

# tests

is.numeric(c(8, 8))

[1] TRUE

is.numeric(c(8, 8, 8, "8"))

[1] FALSE
```

2.3.1.3 Logical

The function logical creates a logical vector of the specified length and by default each element of the vector is set to equal FALSE. The function as.logical attempts to convert its argument to be of logical type. It removes any attributes including, for example, vector element names. A range of character strings c("T", "TRUE", "True", "true"), as well any number not equal to zero, are regarded as TRUE. Similarly, c("F", "FALSE", "False", "false") and zero are regarded as FALSE. All others are regarded as NA. The function is.logical returns TRUE or FALSE depending on whether the argument passed to it is of logical type or not.

```
logical(7)

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

# conversion

as.logical(c(7,5,0,−4,5))

[1] TRUE TRUE FALSE TRUE TRUE

# TRUE and FALSE can be converted to 1 and 0

as.logical(c(7,5,0,−4,5)) ∗ 1

[1] 1 1 0 1 1

as.logical(c(7,5,0,−4,5)) + 0
```

```
[1] 1 1 0 1 1

# different ways to declare TRUE and FALSE

as.logical(c("True","T","FALSE","Raspberry","9","0", 0))

[1] TRUE TRUE FALSE NA NA NA NA
```

Logical vectors are very useful for indexing and subsetting data, including spatial data, to select the data that satisfy some criteria. For example, consider the following:

```
data <- c(3, 6, 9, 99, 54, 32, −102)

# a logical test

index <- (data > 10)

index

[1] FALSE FALSE FALSE TRUE TRUE TRUE FALSE

# used to subset data

data[index]

[1] 99 54 32

sum(data)

[1] 101

sum(data[index])

[1] 185
```

2.3.2 Data Classes in R

The different data types can be used to populate different data structures or *classes*. This section will describe and illustrate vectors, matrices, data frames, lists and factors, data classes that are commonly used in spatial data analysis.

2.3.2.1 Vectors

All of the commands in R in Section 2.3.1 produced vectors. Vectors are the most commonly used data structure and the standard one-dimensional R variable. You will have noticed that when you specified character or logical, etc., a vector of a given length was produced. An alternative approach is to use the function vector, which produces a vector of the length and type or mode specified. The default is logical, and when you assign values to vectors R will seek to convert them to whichever vector mode is most convenient. Recall that the test is.vector returns TRUE if its argument is a vector of the specified class or mode with no attributes other than names, returning FALSE otherwise, and that the function as.vector seeks to convert its argument into a vector of whatever mode is specified.

```
# defining vectors

vector(mode = "numeric", length = 8)

[1] 0 0 0 0 0 0 0 0

vector(length = 8)

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

# testing and conversion

tmp <- data.frame(a=10:15, b=15:20)

is.vector(tmp)

[1] FALSE

as.vector(tmp)

a    b

1 10 15

2 11 16

3 12 17

4 13 18

5 14 19

6 15 20
```

2.3.2.2 Matrices

The function matrix creates a matrix from the data and parameters that are passed to it. This must include parameters for the number of columns and rows in the matrix. The function as.matrix attempts to turn its argument into a matrix, and again the test is.matrix tests to see whether its argument is a matrix.

```
# defining matrices

matrix(ncol = 2, nrow = 0)

[,1] [,2]

matrix(1:6)

     [,1]

[1,]   1

[2,]   2

[3,]   3

[4,]   4

[5,]   5

[6,]   6

matrix(1:6, ncol = 2)

     [,1] [,2]

[1,]    1    4

[2,]    2    5

[3,]    3    6

# conversion and test

as.matrix(6:3)

[,1]
```

```
[1,]    6
[2,]    5
[3,]    4
[4,]    3
```

```
is.matrix(as.matrix(6:3))
[1] TRUE
```

Matrix rows and columns can be named – note the use of byrow=TRUE in the following.

```
flow <- matrix(c(2000, 1243, 543, 1243, 212, 545,
654, 168, 109), c(3,3), byrow=TRUE)
# Rows and columns can have names, not just 1,2,3,…
colnames(flow) <- c("Leeds", "Maynooth", "Elsewhere")
rownames(flow) <- c("Leeds", "Maynooth", "Elsewhere")
# examine the matrix
flow
```

```
            Leeds  Maynooth  Elsewhere
Leeds        2000      1243        543
Maynooth     1243       212        545
Elsewhere     654       168        109
```

```
# and functions exist to summarise
outflows <- rowSums(flow)
outflows
Leeds    Maynooth    Elsewhere
3786    2000    931
```

However, if the data class is not a matrix then just use names, rather than

rownames or colnames.

```
z <- c(6,7,8)

names(z) <- c("Newcastle","London","Manchester")

z

Newcastle    London    Manchester

                6          7            8
```

R has many additional tools for manipulating matrices and performing matrix algebra functions that are not described here. However, as spatial scientists we are often interested in analysing data that have a matrix-like form, as in a data table. For example, in an analysis of spatial data in vector format, the rows in the attribute table represent specific features (such as polygons) and the columns hold information about the attributes of those features. Alternatively, in a raster analysis environment, the rows and columns may represent specific latitudes and longitudes, or northings and eastings, or raster cells. Methods for analysing data in matrix-like structures will be covered in more detail in later chapters as spatial data objects (Chapter 3) and spatial analyses (Chapter 5) are introduced.

I

You will have noticed in the code snippets that a number of new functions are introduced, For example, early in this chapter, the function sum was used. R includes a number of functions that can be used to generate descriptive statistics such as sum and max. You should explore these as they occur in the text to develop your knowledge of and familiarity with R. Further useful examples are in the code below and throughout this book. You could even store them in your own R script. R includes extensive help files which can be used to explore how different functions can be used, frequently with example snippets of code. An illustration of how to find out more about the sum function and some further summary functions is provided in the code below.

```
?sum

help(sum)

# Create a variable to pass to other summary functions

x <− matrix(c(3,6,8,8,6,1,−1,6,7),c(3,3),byrow=TRUE)
```

```
# Sum over rows
rowSums(x)
# Sum over columns
colSums(x)
# Calculate column means
colMeans(x)
# Apply function over rows (1) or columns (2) of x
apply(x,1,max)
# Logical operations to select matrix elements
x[,c(TRUE,FALSE,TRUE)]
# Add up all of the elements in x
sum(x)
# Pick out the leading diagonal
diag(x)
# Matrix inverse
solve(x)
# Tool to handle rounding
zapsmall(x %*% solve(x))
```

2.3.2.3 Factors

The function factor creates a vector with specific categories, defined in the levels parameter. The ordering of factor variables can be specified and an ordered function also exists. The functions as.factor and as.ordered are the coercion functions. The test is.factor returns TRUE or FALSE depending on whether its argument is of type factor or not, and is.ordered returns TRUE when its argument is an ordered factor and FALSE otherwise.

```
# a vector assignment
```

```
house.type <- c("Bungalow", "Flat", "Flat", "Detached", "Flat", "Terrace", "Terrace")

# a factor assignment

house.type <- factor(c("Bungalow", "Flat", "Flat", "Detached", "Flat", "Terrace",
"Terrace"),

levels=c("Bungalow","Flat","Detached","Semi","Terrace"))

house.type

[1] Bungalow Flat    Flat    Detached Flat Terrace

[7] Terrace

Levels: Bungalow Flat Detached Semi Terrace

# table can be used to summarise

table(house.type)

house.type

Bungalow    Flat Detached    Semi     Terrace

1           3           1           0                    2

# levels controls what can be assigned

house.type <- factor(c("People Carrier", "Flat", "Flat", "Hatchback", "Flat", "Terrace",
"Terrace"),

levels=c("Bungalow","Flat","Detached","Semi","Terrace"))

house.type

[1] <NA>    Flat    Flat    <NA>    Flat    Terrace    Terrace

Levels: Bungalow Flat Detached Semi Terrace
```

Factors are useful for categorical or classified data – that is, data values that must fall into one of a number of predefined classes. It is obvious to see how this might be relevant to geographical analysis, where many features represented in spatial data are labelled using one of a set of discrete classes.

2.3.2.4 Ordering

There is no concept of ordering in factors. However, this can be imposed by

using the ordered function. Ordering allows inferences about preference or hierarchy to be made (lower–higher, better–worse, etc.) and this can be used in data selection or indexing (as above) or in the interpretation of derived analyses.

```
income <-factor(c("High", "High", "Low", "Low", "Low", "Medium", "Low", "Medium"),
levels=c("Low", "Medium", "High"))

income > "Low"

[1] NA NA NA NA NA NA NA NA

# levels in ordered defines a relative order

income <-ordered(c("High", "High", "Low", "Low", "Low", "Medium", "Low",
"Medium"), levels=c("Low", "Medium", "High"))

income > "Low"

[1] TRUE TRUE FALSE FALSE FALSE TRUE FALSE TRUE
```

Thus we can see that ordering is implicit in the way that the levels are specified and allows other, ordering-related functions to be applied to the data.

The functions sort and table are new functions. In the above code relating to factors, the function table was used to generate a tabulation of the data in house.type. It provides a count of the occurrence of each level in house.type. The command sort orders a vector or factor. You should use the help in R to explore how these functions work and try them with your own variables. For example:

```
sort(income)
```

2.3.2.5 Lists

The character, numeric and logical data types and the associated data classes described above all contain elements that must all be of the same basic type. Lists do not have this requirement. Lists have slots for collections of different elements. A list allows you to gather a variety of different data types together

in a single data structure and the *n*th element of a list is denoted by double square brackets.

```
tmp.list <- list("Lex Comber",c(2015, 2018),
"Lecturer", matrix(c(6,3,1,2), c(2,2)))
tmp.list
[[1]]
[1] "Lex Comber"
[[2]]
[1] 2015 2018
[[3]]
[1] "Lecturer"
[[4]]
     [,1] [,2]
[1,]    6    1
[2,]    3    2
# elements of the list can be selected
tmp.list[[4]]
     [,1] [,2]
[1,]    6    1
[2,]    3    2
```

From the above it is evident that the function list returns a list structure composed of its arguments. Each value can be tagged depending on how the argument was specified. The conversion function as.list attempts to coerce its argument to a list. It turns a factor into a list of one-element factors and drops attributes that are not specified. The test is.list returns TRUE if and only if its argument is a list. These are best explored through some examples; note that list items can be given names.

```
employee <- list(name="Lex Comber", start.year = 2015, position="Professor")
employee
$name
[1] "Lex Comber"
$start.year
[1] 2015
$position
[1] "Professor"
```

Lists can be joined together with append:

```
append(tmp.list, list(c(7,6,9,1)))
```

and lapply applies a function to each element of a list:

```
# lapply with different functions
lapply(tmp.list[[2]], is.numeric)
lapply(tmp.list, length)
```

Note that the length of a matrix, even when held in a list, is the total number of elements.

2.3.2.6 Defining Your Own Classes

In R it is possible to define your own data type and to associate it with specific behaviours, such as its own way of printing, drawing. For example, you will notice in later chapters that the plot function is used to draw maps for spatial data objects as well as conventional graphs. Suppose we create a list containing some employee information.

```
employee <- list(name="Lex Comber", start.year = 2015,
```

```
position="Professor")
```

This can be assigned to a new class, called staff in this case (it could be any name, but meaningful ones help).

```
class(employee) <- "staff"
```

Then we can define how R treats that class in the form <existing function>. <class> – for example, how it is printed. Note how the existing function for printing is modified by the new class definition:

```
print.staff <- function(x) {
cat("Name: ",x$name,"\n")
cat("Start Year: ",x$start.year,"\n")
cat("Job Title: ",x$position,"\n")}
# an example of the print class
print(employee)
Name: Lex Comber
Start Year: 2015
Job Title: Professor
```

You can see that R knows to use a different print function if the argument is not a variable of class staff. You could modify how your R environment treats existing classes in the same way, but do this with caution. You can also undo the class assigned by using unclass, and the print.staff function can be removed permanently by using rm(print.staff):

```
print(unclass(employee))
$name
[1] "Lex Comber"
$start.year
[1] 2015
```

```
$position

[1] "Professor"
```

2.3.2.7 Classes in Lists

Variables can be assigned to new or user-defined class objects. The example below defines a function to create a new staff object.

```
new.staff <- function(name,year,post) {

result <- list(name=name, start.year=year, position=post)

class(result) <- "staff"

return(result)}
```

A list can then be defined, which is populated using that function as in the code below (note that functions will be dealt with more formally in later chapters).

```
leeds.uni <- vector(mode='list',3)

# assign values to elements in the list

leeds.uni[[1]] <- new.staff("Heppenstall, Alison", 2017,"Professor")

leeds.uni[[2]] <- new.staff("Comber, Lex", 2015,"Professor")

leeds.uni[[3]] <- new.staff("Langlands, Alan", 2014,"VC")

And the list can be examined by entering:

leeds.uni
```

2.3.2.8 data.frame versus tibble

Data of different types and classes are often held in tabular format. The data.frame and tibble classes of the data table are described in this section.

Generally, in *data tables*, each of the records (rows) relates to some kind of

real-world feature (a person, a transaction, a date, etc.) and the columns represent some attribute associated with that feature. In R data can be in a matrix, but matrices can only hold one *type* of data (e.g. integer, logical and character). However, data.frame and tibble class objects can hold different data types in different columns (or fields). This section introduces these (in fact, the tibble class includes data.frame) because they are used to hold attributes of spatial objects (points, lines, areas, pixels) in the R spatial data formats sf and sp, as introduced in detail in Chapter 3. Thus in *spatial data tables*, each record typically represents some real-world *geographical* feature (a place, a route, a region, etc.) and the fields describe variables or attributes associated with that feature (population, length, area, etc.).

The data.frame class in R is composed of a series of vectors of equal length, which together form a two-dimensional data structure. Each vector records values for a particular theme or attribute. Typically these form the columns in a data frame, and the name of each vector provides the column name or header. They are ordered such that the *n*th element in each vector describes a property for the *n*th record (row) representing the *n*th feature. The data.frame class is the most commonly used method for storing data in R.

A data frame can be created using the data.frame() function:

```
df <- data.frame(dist = seq(0,400, 100), city = c("Leeds", "Nottingham", "Leicester",
"Durham", "Newcastle"))

str(df)

'data.frame': 5 obs. of 2 variables:

$ dist: num    0 100 200 300 400

$ city: Factor w/ 5 levels "Durham","Leeds",..: 2 5 3 1 4
```

The data.frame() function by default encodes character strings into factors. To see this enter:

```
df$city
```

To overcome this the df object can be refined using stringsAsFactors =

FALSE:

```
df <- data.frame(dist = seq(0,400, 100),

city = c("Leeds", "Nottingham", "Leicester", "Durham", "Newcastle"), stringsAsFactors =
FALSE)

str(df)

'data.frame': 5 obs. of 2 variables:

$ dist: num    0 100 200 300 400

$ city: chr    "Leeds" "Nottingham" "Leicester" "Durham" …
```

The tibble class is a reworking of the data.frame class that seeks to retain the operational advantages of data frames and eliminate aspects that have proven to be less effective. Enter the code below to create tb:

```
tb <- tibble(dist = seq(0,400, 100),

city = c("Leeds", "Nottingham", "Leicester", "Durham", "Newcastle"))

Probably the biggest criticism of data.frame is the partial matching behaviour. Enter the
following code:

df$ci

[1] "Leeds" "Nottingham" "Leicester" "Durham"

[5] "Newcastle"

tb$ci

NULL
```

Although there is no variable called ci, the partial matching in the data.frame means that the city variable is returned. This is a bit worrying!

A further problem is what gets returned when a data table is subsetted. A tibble always returns a tibble, whereas a data frame may return a vector or a data frame, depending on the dimensions of the result. For example, compare the outputs of the following code:

```
# 1 column
df[,2]
tb[,2]
class(df[,2])
class(tb[,2])
# 2 columns
df[,1:2]
tb[,1:2]
class(df[,1:2])
class(tb[,1:2])
```

Note that a tibble is a data frame, but tibbles seek to be *lazy* by not changing variable names or types or do partial matching. And they are *surly* because they complain more. This forces cleaner coding by identifying problems earlier in the data analysis cycle.

Finally, the print method for tibble returns the first 10 records by default, whereas for data.frame the head() function is frequently used to display just the first 6 records. The tibble class also includes a description of the class of each field (column) when it is printed.

It is possible to convert between tibbles and data frames using the following functions:

```
data.frame(tb)
as_tibble(df)
```

The following functions work with both tibbles and data frames:

```
names()
colnames()
```

```
rownames()

length() # length of the underlying list

ncol()

nrow()
```

They can be subsetted in the same way as a matrix, using the [row, column] notation as above, and they can both be combined using cbind() and rbind().

```
cbind(df, Pop = c(700,250,230,150,1200))
```

	dist	city	Pop
1	0	Leeds	700
2	100	Nottingham	250
3	200	Leicester	230
4	300	Durham	150
5	400	Newcastle	1200

```
cbind(tb, Pop = c(700,250,230,150,1200))
```

	dist	city	Pop
1	0	Leeds	700
2	100	Nottingham	250
3	200	Leicester	230
4	300	Durham	150
5	400	Newcastle	1200

You could explore the tibble vignette by entering:

```
vignette("tibble")
```

2.3.3 Self-Test Questions

In the next pages there are a number of self-test questions. In contrast to the previous sections where the code is provided in the text for you to work through (i.e. you enter and run it yourself), the self-test questions are tasks for you to complete, mostly requiring you to write R code. Answers to them are provided in Section 2.7. The self-test questions relate to the main data types that have been introduced: factors, matrices, lists (named and unnamed) and classes.

2.3.3.1 Factors

Recall from the descriptions above that factors are used to represent categorical data – where a small number of categories are used to represent some characteristic in a variable. For example, the colour of a particular model of car sold by a showroom in a week can be represented using factors:

```
colours <- factor(c("red","blue","red","white", "silver","red","white","silver",
"red","red","white","silver","silver"), levels=c("red","blue","white","silver","black"))
```

Since the only colours this car comes in are red, blue, white, silver and black, these are the only levels in the factor.

Self-Test Question 1. Suppose you were to enter:

```
colours[4] <- "orange"

colours
```

What would you expect to happen? Why?

Next, use the table function to see how many of each colour were sold. First reassign the colours (as you may have altered this variable in the previous self-test question):

```
colours <- factor(c("red","blue","red","white", "silver","red","white","silver",
"red","red","white","silver","silver"), levels=c("red","blue","white","silver","black"))

table(colours)
```

```
colours

red    blue    white    silver    black

5        1           3           4           0
```

Note that the result of the table function is just a standard vector, but that each of its elements is named – the names in this case are the levels in the factor. Now suppose you had simply recorded the colours as a character variable, in colours2 as below, and then computed the table:

```
colours2 <-c("red","blue","red","white", "silver","red","white","silver",
"red","red","white","silver")

# Now, make the table

table(colours2)

colours2

blue    red    silver    white

      1       5              3           3
```

Self-Test Question 2. What two differences do you notice between the results of the two table expressions?

Now suppose we also record the type of car – it comes in saloon, convertible and hatchback. This can be specified by another factor variable called car.type:

```
car.type <- factor(c("saloon","saloon","hatchback",
"saloon","convertible","hatchback","convertible", "saloon","hatchback","saloon","saloon",
"saloon","hatchback"), levels=c("saloon","hatchback","convertible"))
```

The table function can also work with two arguments:

```
table(car.type, colours)
```

```
                    colours
car.type            red    blue    white    silver    black
saloon                2       1        2         2        0
hatchback             3       0        0         1        0
convertible           0       0        1         1        0
```

This gives a two-way table of counts – that is, counts of red hatchbacks, silver saloons and so on. Note that the output this time is a matrix. For now enter the code below to save the table into a variable called crosstab to be used later on:

```
crosstab <- table(car.type,colours)
```

Self-Test Question 3. What is the difference between table(car.type, colours) and table(colours,car.type)?

Finally in this section, ordered factors will be considered. Suppose a third variable about the cars is the engine size, and that the three sizes are 1.1 litre, 1.3 litre and 1.6 litre. Again, this is stored in a variable, but this time the sizes are ordered. Enter:

```
engine <- ordered(c("1.1litre","1.3litre","1.1litre", "1.3litre","1.6litre","1.3litre","1.6litre",
"1.1litre","1.3litre","1.1litre", "1.1litre", "1.3litre","1.3litre"),
levels=c("1.1litre","1.3litre","1.6litre"))
```

Recall that with ordered variables, it is possible to use comparison operators > (greater than), < (less than), >= (greater than or equal to) and <= (less than or equal to). For example:

```
engine > "1.1litre"

[1] FALSE TRUE FALSE TRUE TRUE TRUE TRUE FALSE TRUE

[10] FALSE FALSE TRUE TRUE
```

Self-Test Question 4. Using the engine, car.type and colours variables, write expressions to give the following:

- The colours of all cars with engines with capacity greater than 1.1 litres.
- The counts of types (hatchback etc.) of all cars with capacity below 1.6 litres.
- The counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litre.

2.3.3.2 Matrices

In the previous section you created a matrix called crosstab. A number of functions can be applied to matrices:

```
dim(crosstab) # Matrix dimensions

[1] 3 5

rowSums(crosstab) # Row sums

saloon    hatchback    convertible

        7         4            2

colnames(crosstab) # Column names

[1] "red"    "blue"    "white"    "silver"    "black"
```

Another important tool for matrices is the apply function. To recap, this applies a function to either the rows or columns of a matrix, giving a single-dimensional list as a result. A simple example finds the largest value in each row:

```
apply(crosstab,1,max)

        saloon    hatchback convertible

      2           3           1
```

In this case, the function max is applied to each row of crosstab. The 1 as the second argument specifies that the function will be applied row by row. If it were 2 then the function would be column by column:

```
apply(crosstab,2,max)

red    blue    white    silver    black

3    1    2       2    0
```

A useful function is which.max. Given a list of numbers, it returns the index of the largest one. For example:

```
example <- c(1.4,2.6,1.1,1.5,1.2)

which.max(example)

[1] 2
```

In this case, the second element is the largest.

Self-Test Question 5. What happens if there is more than one number taking the largest value in a list? Use either the help facility or experimentation to find out.

Self-Test Question 6. The function which.max can be used in conjunction with apply. Write an expression to find the index of the largest value in each row of crosstab.

The function levels returns the levels of a variable of type factor in character form. For example:

```
levels(engine)

[1] "1.1litre" "1.3litre" "1.6litre"
```

The order they are returned in is the one specified in the original factor assignment and the same order as row or column names produced by the table function. This means that levels can be used in conjunction with which.max when applied to matrices to obtain the row or column names instead of an index number:

```
levels(colours)[which.max(crosstab[,1])]
```

```
[1] "blue"
```

Alternatively, the same effect can be achieved by the following:

```
colnames(crosstab)[which.max(crosstab[,1])]

[1] "blue"
```

You should unpick these last two lines of code to make sure you understand what each element is doing.

```
colnames(crosstab)

[1] "red"    "blue"    "white"    "silver" "black"

crosstab[,1]

saloon    hatchback convertible

      2           3           0

which.max(crosstab[,1])

hatchback

2
```

More generally, a function could be written to apply this operation to any variable with names:

```
# Defines the function

which.max.name <- function(x) {

return(names(x)[which.max(x)])}

# Next, give the variable 'example' names for the values

names(example) <- c("Bradford","Leeds","York", "Harrogate","Thirsk")

example

  Bradford  Leeds  York  Harrogate  Thirsk

       1.4    2.6   1.1        1.5     1.2
```

```
which.max.name(example)

[1] "Leeds"
```

Self-Test Question 7. The function which.max.name could be applied (using apply) to a table or matrix to find the name of the row or column with the largest value. If the crosstab table is considered a table of car sales, write an apply expression to determine the best-selling colour for each car type and the best-selling car type in each colour.

Note that in the last code snippet, a function was defined called which.max.name. You have been using functions, but these have all been existing ones as defined in R until now. Functions will be thoroughly dealt with in Chapter 4, but you should note two things about them at this point. First is the form:

```
function name <- function(function inputs) {

variable <- function

actions return(variable)

}
```

Second are the syntactic elements of the curly brackets { } that bound the code, and the return() function that defines the value to be returned.

2.3.3.3 Lists

From the text in this chapter, recall that lists can be named and unnamed. Here we will only consider the named kind. Lists may be created by the list function in the form:

```
var <- list(name1=value1, name2=value2, …)
```

Self-Test Question 8. Suppose you wanted to store both the row- and column-wise apply results (from Question 7) in a list called most.popular with two named elements called colour (containing the most popular colour for each car type) and type (containing the most popular car type for each

colour). Write an R expression that assigns the best-selling colour and car types to a list.

2.3.3.4 Classes

The objective of this task is to create a class based on the list created in the previous section. The class will consist of a list of most popular colours and car types, together with a third element containing the total number of cars sold (called total). Call this class sales.data. A function to create a variable of this class, given colours and car.type, is as follows:

```
new.sales.data <- function(colours, car.type) {

xtab <- table(car.type,colours)

result <- list(colour=apply(xtab,1,which.max.name), type=apply(xtab,2,which.max.name),
total=sum(xtab))

class(result) <- "sales.data"

return(result)}
```

This can be used to create a sales.data object which has the colours and car.type variables assigned to it via the function:

```
this.week <- new.sales.data(colours,car.type)

this.week

$colour

saloon    hatchback convertible

"red"        "red"      "white"

$type

red     blue     white    silver     black

"hatchback" "saloon" "saloon" "saloon" "saloon"

$total

[1] 13
```

```
attr(,"class")

[1] "sales.data"
```

In the above code, a new variable called this.week, of class sales.data, is created. Following the ideas set out in the previous section, it is now possible to create a print function for variables of class sales.data. This can be done by writing a function called print.sales.data that takes an input or argument of the sales.data class.

Self-Test Question 9. Write a print function for variables of class sales.data. This is a difficult problem and should be tackled by those with previous programming experience. Others can try this now but should return to it after the functions have been formally introduced in Chapter 4.

2.4 Plots

There are a number of plot routines and packages in R. In this section some basic plot types will be introduced, followed by some more advanced plotting commands and functions. The aim of this section to give you an understanding of how the basic plot types can be used as building blocks in more advanced plotting routines that are called in later chapters to display the results of spatial analysis.

2.4.1 Basic Plot Tools

The most basic plot is the scatter plot. Figure 2.1 was created from the function rnorm which generates a set of random numbers. Note that each running of the code will generate a slightly different plot as different random numbers are generated.

```
x1 <- rnorm(100)

y1 <- rnorm(100)

plot(x1,y1)
```

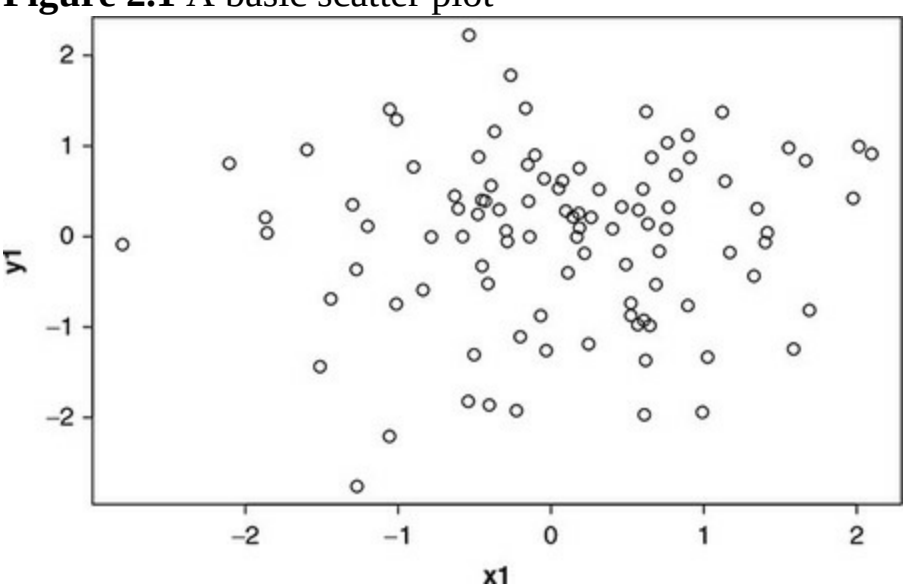
The generic plot function creates a graph of the two variables, plotting them

on the *x*-axis and the *y*-axis. The default settings for the plot function produce a scatter plot and you should note that by default the axes are labelled with expressions passed to the plot function. Many parameters can be set for plot either by defining the plot environment (described later) or when the plot is called. For example, the option col specifies the plot colour and pch the plot character:

```
plot(x1,y1,pch=16, col='red')
```

Other options include different types of plot: type = 'l' produces a line plot of the two variables, and again the col option can be used to specify the line colour and the option lwd specifies the plot line width. You should run the code below to produce different line plots:

Figure 2.1 A basic scatter plot




```
x2 <- seq(0,2*pi,len=100)

y2 <- sin(x2)

plot(x2,y2,type='l')

plot(x2,y2,type='l', lwd=3, col='darkgreen')
```

You should examine the help for the plot command (reminder: type ?plot at the R prompt) and explore different plot types that are available. Having

called a new plot as in the above examples, other data can be plotted using other commands: points, lines, polygons, etc. You will see that plot by default assumes the plot type is point unless otherwise specified. For example, in Figure 2.2 the line data described by x2 and y2 are plotted, after which the points described by x2 and y2r are added to the plot.

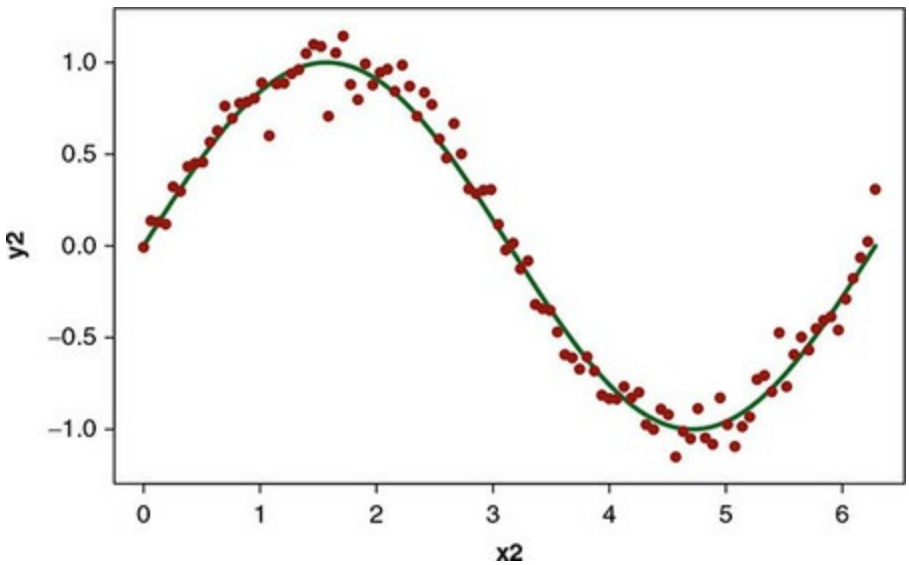
```
plot(x2,y2,type='l', col='darkgreen', lwd=3, ylim=c(−1.2,1.2))

y2r <- y2 + rnorm(100,0,0.1)

points(x2,y2r, pch=16, col='darkred')
```

In the above code, the rnorm function creates a vector of small values which are added to y2 to create y2r. The function points adds points to an existing plot. Many other options for plots can be applied here. For example, note the ylim option. This sets the limits of the *y*-axis, while xlim does the same for the *x*-axis. You should apply the commands below to the plot data.

```
y4 <- cos(x2)

plot(x2, y2, type='l', lwd=3, col='darkgreen')

lines(x2, y4, lwd=3, lty=2, col='darkblue')
```

Notice that, similar to points, the function lines adds lines to an existing plot, and note the lty option as well. This specifies the type of line (dotted, simple, etc.).

Figure 2.2 A line plot with points added



I

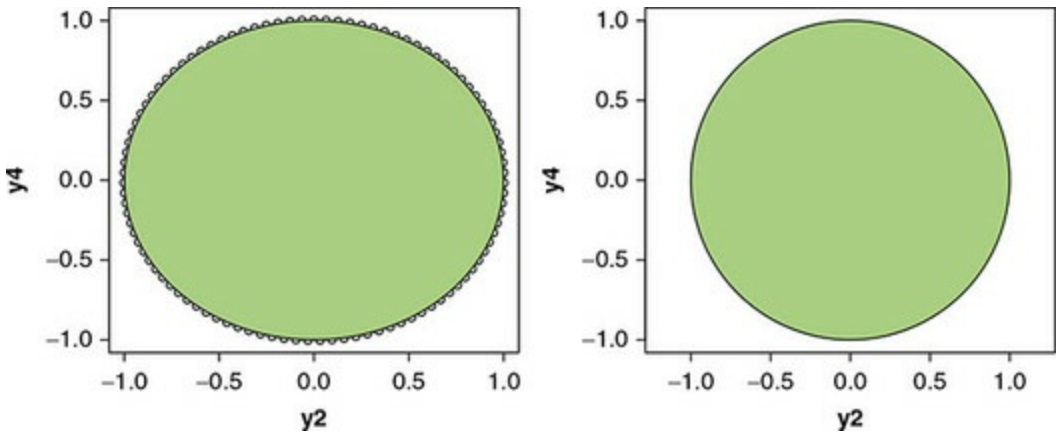
You should examine the different plot types and parameters in par. Enter ?par for the help page to see the full list of different plot parameters. One of these, mfrow, is used below to set a combined plot of one row and two columns. This needs to be reset or the rest of your plots will continue to be printed in this way. To do this enter:

```
par(mfrow = c(1,2))

plot(x2, y2, type='l', lwd=3, col='darkgreen')

plot(x2, y2, type='l', col='darkgreen', lwd=3, ylim=c(−1.2,1.2))

points(x2, y2r, pch=16, col='darkred')

par(mfrow = c(1,1))
```

The last line of code resets par.

The function polygon adds a polygon to an existing plot. The option col sets the polygon fill colour. By default a black border is drawn; however, including the parameter border = NA would result in no border being drawn. In Figure 2.3 two different plots of the same data illustrate the application of these parameters.

Figure 2.3 Points with polygons added




```
x2 <- seq(0,2*pi,len=100)

y2 <- sin(x2)

y4 <- cos(x2)

# specify the plot layout and order

par(mfrow = c(1,2))

# plot #1

plot(y2,y4)

polygon(y2,y4,col='lightgreen')

# plot #2: this time with 'asp' to set the aspect ratio of the axes

plot(y2,y4, asp=1, type='n')

polygon(y2,y4,col='lightgreen')
```

In the second plot, the parameter asp fixes the aspect ratio, in this case to 1 so that the x and y scales are the same, and type = 'n' draws the plot axes to correct scale (i.e. of the y2 and y4 data) but adds no lines or points.

So far the plot commands have been used to plot pairs of x and y coordinates in different ways: points, lines and polygons (this may suggest different vector types in a GIS for some readers). We can extend these to start to consider geographical coordinates more explicitly with some geographical data. You will need to install the GISTools package, which may involve setting a mirror site as described in Chapter 1. The first time you use any package in R it needs to be downloaded before it is installed.

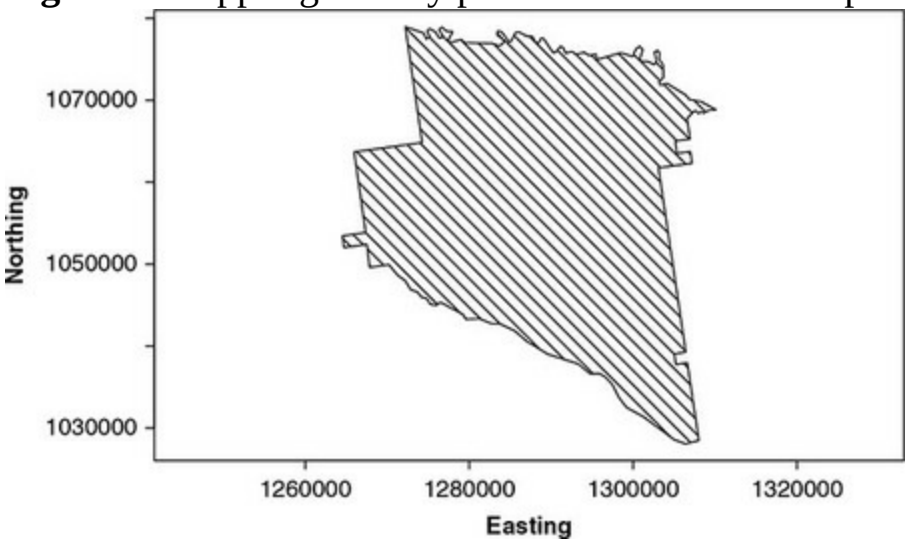
```
install.packages("GISTools", depend = T)
```

Then you can call the package in the R console:

```
library(GISTools)
```

You will then see some messages when you load the package, letting you know that the packages that GISTools makes use of have also been loaded automatically. You only need to *install* a package onto your computer the first time you use it. Once it is installed it can simply be called. That is, there is no need to download it again, you can simply enter library(package).

Figure 2.4 Appling County plotted from coordinate pairs



The code below loads a number of datasets with the data(georgia) command. It then selects the first element from the georgia.polys dataset and assigns it to a variable called appling. This contains the coordinates of the outline of Appling County in Georgia. It then plots this to generate Figure 2.4.

```
# library(GISTools)

data(georgia)

# select the first element

appling <- georgia.polys[[1]]
```

```
# set the plot extent

plot(appling, asp=1, type='n', xlab="Easting", ylab="Northing")

# plot the selected features with hatching

polygon(appling, density=14, angle=135)
```

There are a number of things to note in this bit of code.

1. The call data(georgia) loads three datasets: georgia, georgia2 and georgia.polys.
2. The first element of georgia.polys contains the coordinates for the outline of Appling County.
3. Polygons do not have to be regular; they can, as in this example, be geographical zones. The code assigns the coordinates to a variable called appling and this is a two-column matrix.
4. Thus, with an x and y pairing, the following plot commands all work with data in this format: plot, lines, polygon, points.
5. As before, the plot command in the code below has the type = 'n' parameter, and asp = 1 fixes the aspect ratio. The result is that that the x and y scales are the same but the command adds no lines or points.

The wider point being demonstrated here is how routines for plotting spatial data that we will use subsequently are underpinned by these kinds of data structures and core plotting routines. The code above illustrates the engines of, for example, the mapping and visualisation packages tmap and ggplot.

2.4.2 Plot Colours

Plot colours can be specified names or as red, green and blue (RGB) values. The former can be listed by entering the following:

```
colours()
```

RGB colours are composed of three values in the ranges 0 to 1. Having run the code above, you should have a variable called appling in your workspace. Now try entering the code below:

```
plot(appling, asp=1, type='n', xlab="Easting", ylab="Northing")

polygon(appling, col=rgb(0,0.5,0.7))
```

A fourth parameter can be added to rgb to indicate transparency as in the code below, where the range is from 0 (invisible) to 1 (opaque).

```
polygon(appling, col=rgb(0,0.5,0.7,0.4))
```

Text can also be added to the plot and its placement in the plot window specified. The cex parameter (for *c*haracter *ex*pansion) determines the size of text. Note that parameters like col also work with text and that HTML colours also work (such as "B3B333"). The code below generates two plots. The first plots a set of random points and then plots appling with a transparency shading over the top (Figure 2.5).

```
# set the plot extent

plot(appling, asp=1, type='n', xlab="Easting", ylab="Northing")

# plot the points

points(x = runif(500,126,132)*10000, y = runif(500,103,108)*10000, pch=16, col='red')

# plot the polygon with a transparency factor

polygon(appling, col=rgb(0,0.5,0.7,0.4))
```

The second plots appling, but with some descriptive text (Figure 2.6).

```
plot(appling, asp=1, type='n', xlab="Easting", ylab="Northing")

polygon(appling, col="#B3B333")

# add text, specifying its placement, colour and size

text(1287000,1053000,"Appling County",cex=1.5)

text(1287000,1049000,"Georgia",col='darkred')
```

Figure 2.5 Appling County with transparency

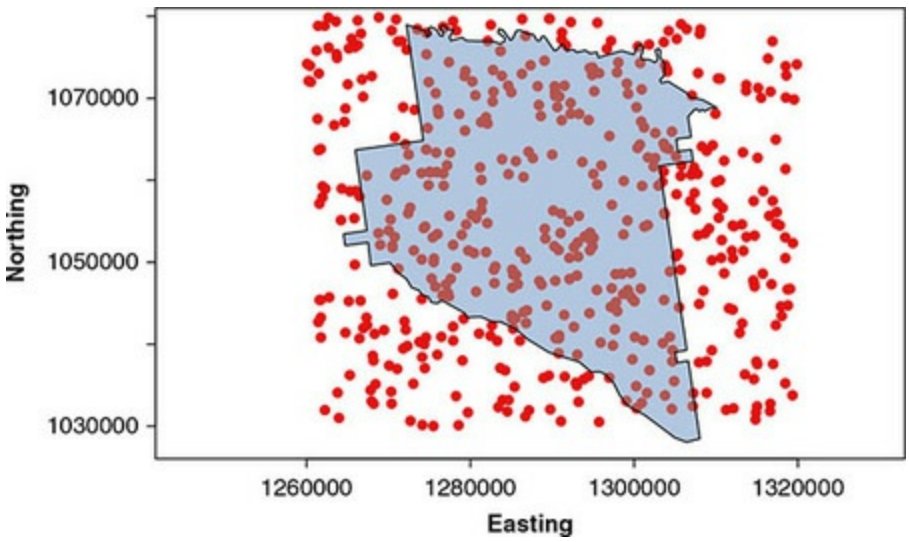
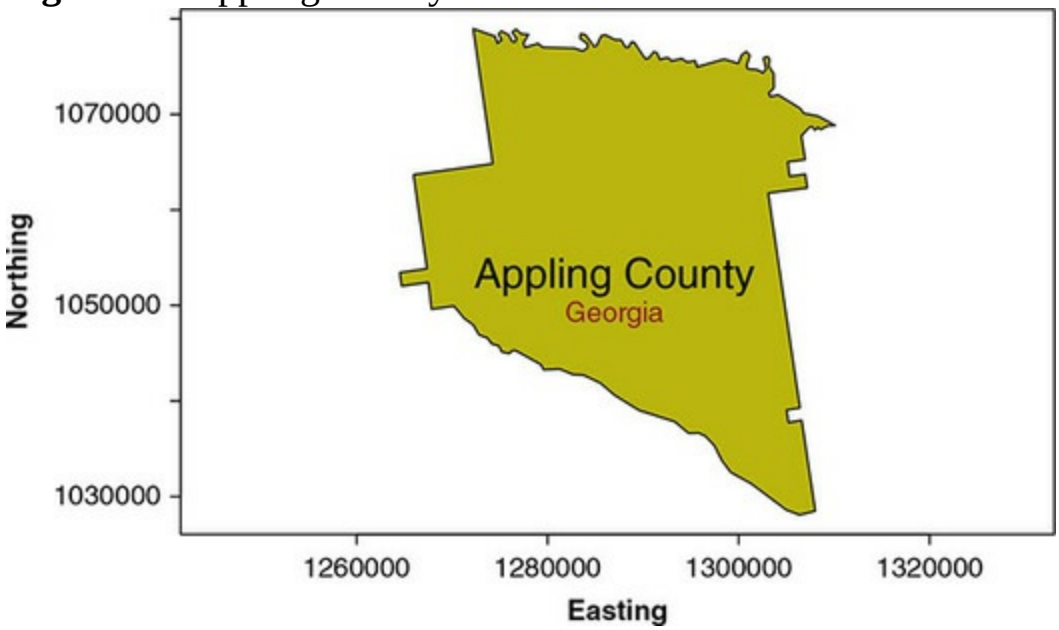


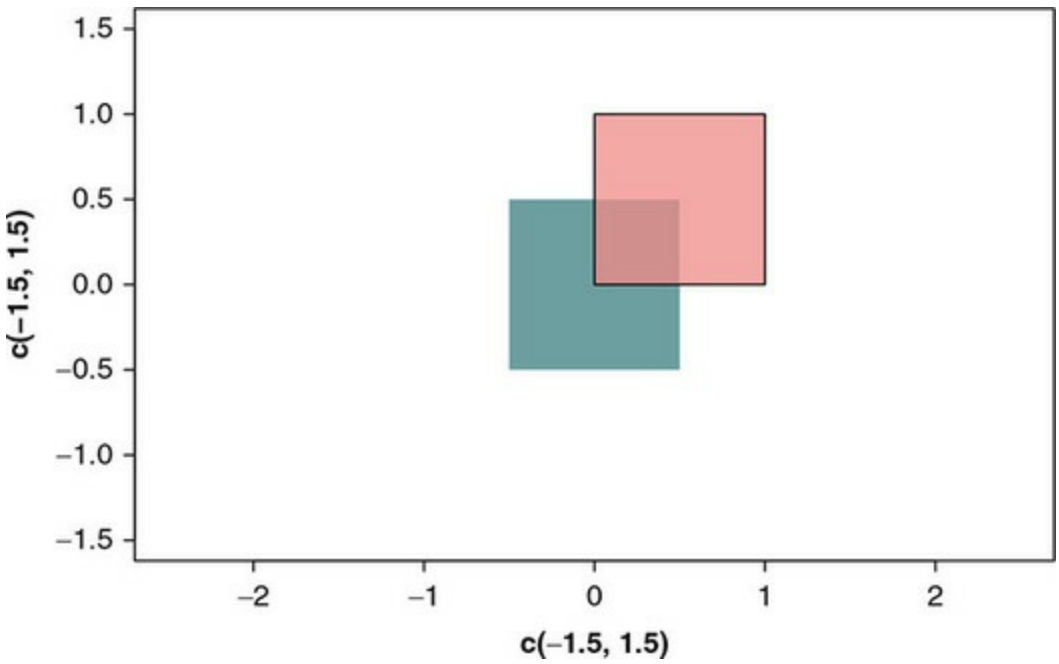
Figure 2.6 Appling County with text



I

In the above code, the coordinates for the text placement need to be specified. The function locator is very useful in this context: it can be used to determine locations in the plot window. Enter locator() at the R prompt, and then left-click in the plot window at various locations. When you right-click, the coordinates of these locations are returned to the R console window.

Figure 2.7 Plotting rectangles

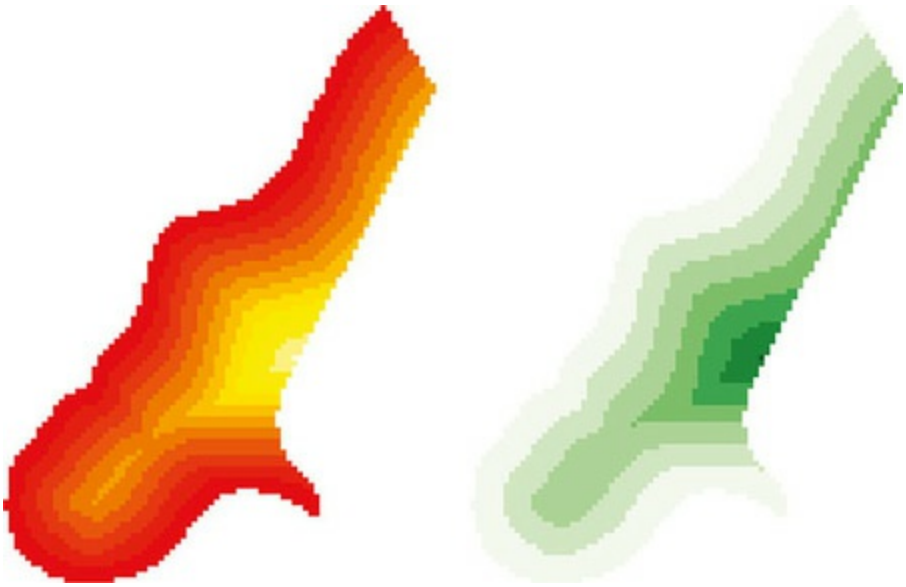


Other plot tools include rect, which draws rectangles. This is useful for placing map legends as your analyses develop. The following code produces the plot in Figure 2.7.

```
plot(c(−1.5,1.5),c(−1.5,1.5),asp=1, type='n')
# plot the green/blue rectangle
rect(−0.5,−0.5,0.5,0.5, border=NA, col=rgb(0,0.5,0.5,0.7))
# then the second one
rect(0,0,1,1, col=rgb(1,0.5,0.5,0.7))
```

The command image plots tabular and raster data as shown in Figure 2.8. It has default colour schemes, but other colour palettes exist. This book strongly recommends the use of the RColorBrewer package, which is described in more detail in Chapter 3, but an example of its application is given below:

Figure 2.8 Plotting raster data



```
# load some grid data
data(meuse.grid)
# define a SpatialPixelsDataFrame from the data
mat = SpatialPixelsDataFrame(points = meuse.grid[c("x", "y")], data = meuse.grid)
# set some plot parameters (1 row, 2 columns)
par(mfrow = c(1,2))
# set the plot margins
par(mar = c(0,0,0,0))
# plot the points using the default shading
image(mat, "dist")
# load the package
library(RColorBrewer)
# select and examine a colour palette with 7 classes
greenpal <- brewer.pal(7,’Greens’)
# and now use this to plot the data
image(mat, "dist", col=greenpal)
# reset par
```

```
par(mfrow = c(1,1))
```

You should note that par(mfrow = c(1,2)) results in one row and two columns and that it is reset in the last line of code.

I

The command contour(mat, "dist") will generate a contour plot of the matrix above. You should examine the help for contour; a nice example of its use can be found in code in the help page for the volcano dataset that comes with R. Enter the following in the R console:

```
?volcano
```

2.5 Another Plot Option: ggplot

2.5.1 Introduction to ggplot

A suite of tools and functions for plotting are available via the ggplot2 package which is included as part of the tidyverse (https://www.tidyverse.org). The ggplot2 package applies principles described in *The Grammar of Graphics* (Wilkinson, 2005) (hence the *gg* in the name of the package) which conceptualises graphics and plots in terms of their theoretical components. The approach is to handle each element of the graphic separately in a series of layers, and in so doing to control each part of the plot. This is different from the basic plot functions used above which apply specific plotting functions based on the type or class of data that were passed to them.

The ggplot2 package can be installed by installing the whole tidyverse:

```
install.packages("tidyverse", dep = T)
```

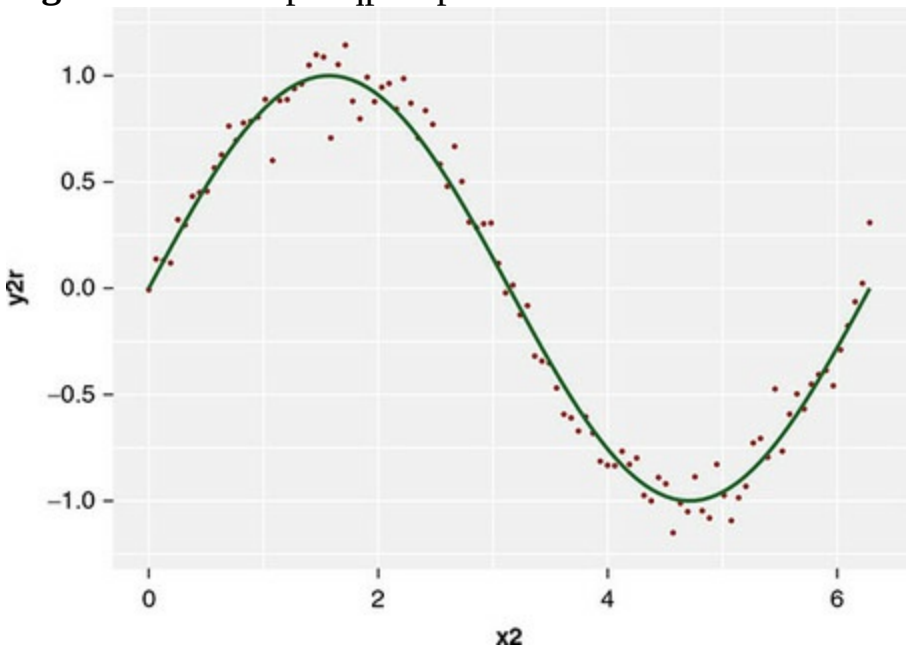
Or it can be installed on its own:

```
install.packages("ggplot2", dep = T)
```

And then loaded into the workspace:

```
library(ggplot2)
```

Figure 2.9 A simple qplot plot



The plots above can be re-created using either the qplot or ggplot functions in the ggplot2 package. The function qplot() is used to produce quick, simple plots in a similar way to the plot function. It takes x and y and a data argument for a data frame containing x and y. Figure 2.9 re-creates Figure 2.2. Notice how the elements in theme are used to control the display.

```
qplot(x2,y2r,col=I('darkred'), ylim=c(−1.2, 1.2)) +

geom_line(aes(x2,y2), col=I("darkgreen"), size = I(1.5)) +
theme(axis.text=element_text(size=20),

                axis.title=element_text(size=20,face="bold"))
```

Notice how the plot type is first specified (in this case qplot()) and then subsequent lines include instructions for what to plot and how to plot it. Here geom_line() was specified followed by some style instructions.

Try adding:

```
theme_bw()
```

or:

```
theme_dark()
```

to the above. Remember that you need to include a + for each additional element in ggplot.

To reproduce the Appling plots, the variable appling has to be converted from a matrix to a data frame whose elements need to be labelled:

```
appling <- data.frame(appling)

colnames(appling) <- c("X", "Y")
```

Then qplot can be called as in Figure 2.10 to re-create Figure 2.5 defined above in stages.

```
# create the first plot with qplot

p1 <- qplot(X, Y, data = appling, geom = "polygon", asp = 1, colour = I("black"),

fill=I(rgb(0,0.5,0.7,0.4))) + theme(axis.text=element_text(size=12),
axis.title=element_text(size=20))

# create a data.frame to hold the points

df <- data.frame(x = runif(500,126,132)*10000, y = runif(500,103,108)*10000)

# now use ggplot to construct the layers of the plot

p2 <- ggplot(appling, aes(x = X, y= Y)) + geom_polygon(fill = I(rgb(0,0.5,0.7,0.4))) +
geom_point(data = df, aes(x, y),col=I('red')) + coord_fixed() +
theme(axis.text=element_text(size=12),

axis.title=element_text(size=20))

# finally combine these in a single plot

# using the grid.arrange function

# NB you may have to install the gridExtra package

library(gridExtra)
```

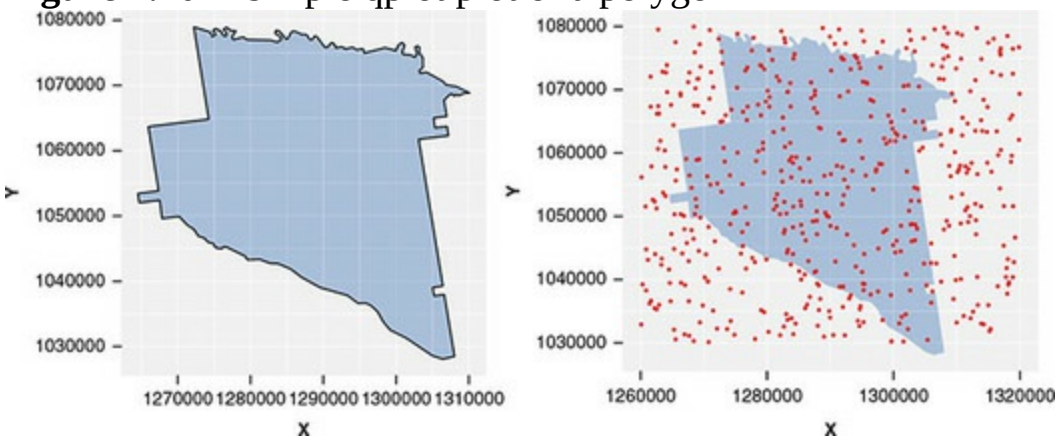
```
grid.arrange(p1, p2, ncol = 2)
```

The result is shown in Figure 2.10, the right-hand part of which re-creates Figure 2.5.

Notice a number of things. First, the structural differences in the way the graphic is called, including the specification of the type with the geom parameter (compared to the geom_line parameter earlier). Second, the assignment of the plot objects to variables p1 and p2. Third, the use of the grid.arrange() function in the gridExtra package that allows two graphics to be included in the plot window. Finally, you will have to install the gridExtra package before the first time you use it:

```
install.packages("gridExtra", dep = T)
```

Figure 2.10 A simple qplot plot of a polygon



2.5.2 Different ggplot Types

This section briefly introduces different kinds of plots using ggplot for different kinds of variables, including scatter plots, histograms and boxplots. In subsequent chapters, different flavours and types of ggplot will be illustrated. But this is a vast package and involves a bit of a learning curve at first. To fully understand all that it can do is beyond the scope of this subsection in this chapter, but there is plenty of help and advice on the internet. You could explore some of this yourself by following some of the links at http://ggplot2.tidyverse.org.

The basic call to ggplot is complemented by an aesthetic prefixed by geom_ and has the following syntax:

```
ggplot(data = <data frame>, aes(x,y,colour)) +     geom_XYZ()
```

To illustrate the different plotting options, we need to create some data and some categorical variables. The code below extracts the data frame from georgia and converts it to a tibble. This is like the attribute table of a shapefile. Note that ggplot will work with any type of data frame.

```
# data.frame

df <- data.frame(georgia)

# tibble

tb <- as.tibble(df)
```

Enter the code below to see the first 10 records:

```
tb
```

You can see that this has attributes for the counties of Georgia, and a number of variables are included. Next, the code below creates an indicator for rural/not-rural, which we set to values using the levels function. Note the use of the + 0 to convert the TRUE and FALSE values to 1s and 0s:

```
tb$rural <- as.factor((tb$PctRural > 50) + 0)

levels(tb$rural) <- list("Non-Rural" = 0, "Rural"=1)
```

Then we create an income category variable around the interquartile range of the MedInc variable (median county income). There are fancier ways to do it, but the code below is tractable:

```
tb$IncClass <- rep("Average", nrow(tb))

tb$IncClass[tb$MedInc >= 41204] = "Rich"
```

```
tb$IncClass[tb$MedInc <= 29773] = "Poor"
```

The distributions can be checked if you wanted using the table() function:

```
table(tb$IncClass)
```

Scatter plots can be used to show two variables together. The data pairs in tb should be examined. For example, consider PctBach and PctEld, representing the percentages of the county populations with bachelor's degrees and who are elderly (whatever that means).

```
ggplot(data = tb, mapping=aes(x=PctBach, y=PctEld)) + geom_point()
```

The plot can be enhanced by passing a grouping variable to the colour parameter in aes:

```
ggplot(data = tb, mapping=aes(x=PctBach, y=PctEld, colour=rural)) + geom_point()
```

Now modify the code above to group by the IncClass variable created earlier. What happens? What do you see? Does this make sense? Are there any trends? It could tentatively be said that the poor areas are more elderly and have fewer people with bachelor's degrees. This might be confirmed by adding a trend line:

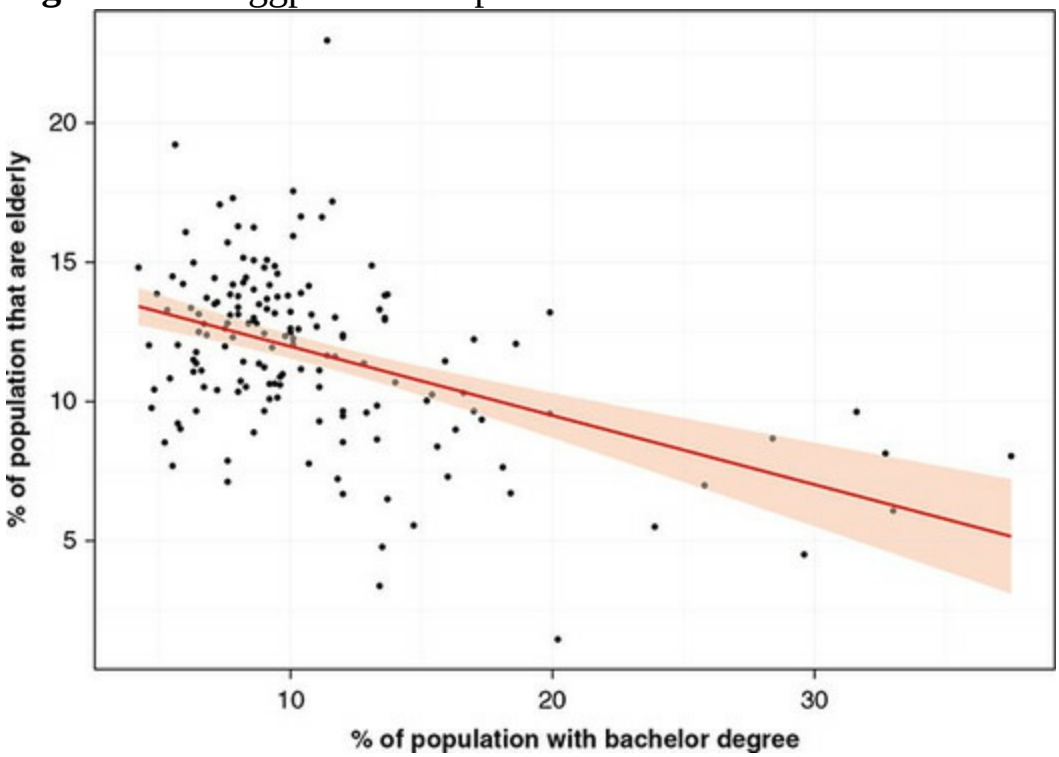
```
ggplot(data = tb, mapping = aes(x = PctBach, y = PctEld)) +
geom_point() + geom_smooth(method = "lm")
```

Also note that style templates can be added and colours changed. Putting this all together generates Figure 2.11:

```
ggplot(data = tb, mapping = aes(x = PctBach, y = PctEld)) + geom_point() +
geom_smooth(method = "lm", col = "red", fill = "lightsalmon") + theme_bw() +
xlab("% of population with bachelor degree") + ylab("% of population that are elderly")
```

You can explore other styles by trying the ones listed under the help for theme_bw.

Figure 2.11 A ggplot scatter plot



Next, *histograms* can be used to examine the distributions of income across the 159 counties of Georgia:

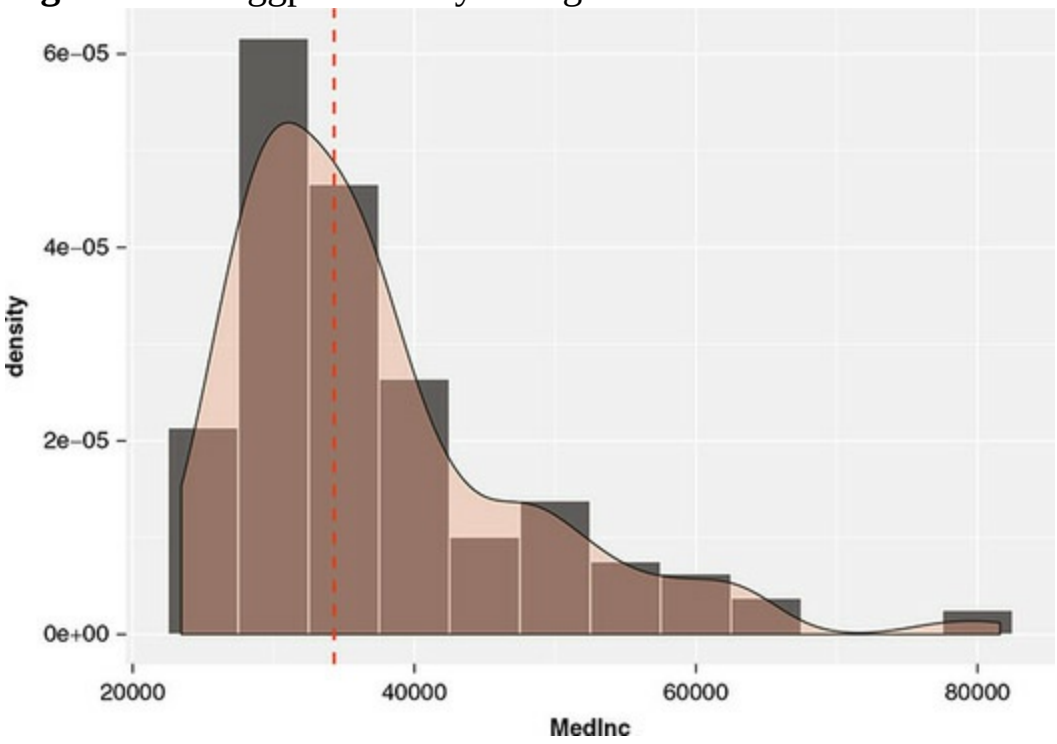
```
ggplot(tb, aes(x=MedInc)) +
geom_histogram(, binwidth = 5000, colour = "red", fill = "grey")
```

The axes can be labelled, the theme set and title included as with the above examples, by including additional elements in the plot. Probability densities can also be plotted as follows, generating Figure 2.12:

```
ggplot(tb, aes(x=MedInc)) + geom_histogram(aes(y=..density..),
binwidth=5000,colour="white") + geom_density(alpha=.4, fill="darksalmon") +
# Ignore NA values for mean
geom_vline(aes(xintercept=median(MedInc, na.rm=T)), color="orangered1",
```

```
linetype="dashed", size=1)
```

Figure 2.12 A ggplot density histogram



Multiple plots can be generated using the facet() options in ggplot. These create separate plots for each group. Here the PctBach variable is plotted and median incomes compared:

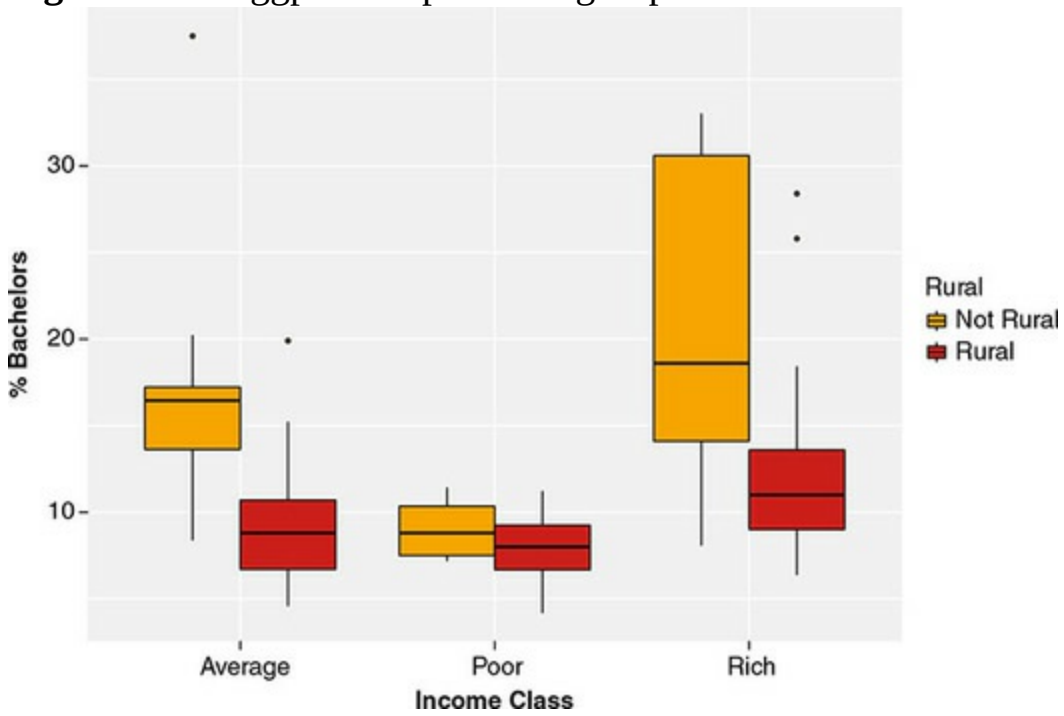
```
ggplot(tb, aes(x=PctBach, fill=IncClass)) +
    geom_histogram(color="grey30",
binwidth = 1) + scale_fill_manual("Income Class",
values = c("orange", "palegoldenrod","firebrick3")) +
facet_grid(IncClass~.) +
xlab("% Bachelor degrees") +
ggtitle("Bachelors degree % in different income classes")
```

Another way of examining distributions is through *boxplots*. Boxplots display the distribution of a continuous variable and can be broken down by a categorical variable. A basic boxplot can be generated with the geom_boxplot

aesthetic:

```
gplot(tb, aes(x = "",PctBach)) +     geom_boxplot()
```

Figure 2.13 A ggplot boxplot with groups



This can be extended with some grouping, as before, and to compare more than one treatment as in Figure 2.13:

```
ggplot(tb, aes(IncClass, PctBach, fill = factor(rural))) +     geom_boxplot() +
   scale_fill_manual(name = "Rural",
              values = c("orange", "firebrick3"),
              labels = c("Non-Rural"="Not Rural","Rural"="Rural")) +
xlab("Income Class") + ylab("% Bachelors")
```

This is only scratching the surface of the capability of ggplot. Additional refinements will be demonstrated throughout this book.

2.6 Reading, Writing, Loading and Saving Data

There are a number of ways of getting data in and out of R, and three methods for reading and writing different formats are briefly considered here: text files, R data files and spatial data.

2.6.1 Text Files

Consider the appling data variable above. This is a matrix variable, containing two columns and 125 rows. You can examine the data using dim and head:

```
# display the first six rows
head(appling)
# display the variable dimensions
dim(appling)
```

You will note that the data fields (columns) are not named; however, these can be assigned.

```
colnames(appling) <- c("X", "Y")
```

The data can be written into a comma-separated variable file using the command write.csv and then read back into a different variable, as follows:

```
write.csv(appling, file = "test.csv")
```

This writes a .csv file into the current working directory. You check where this is by using the getwd() function. You can set the working directory either though the setwd() function or through the menu (**Session > Set Working Directory**). If you open it using a text editor or spreadsheet software, you will see that it has three columns: X and Y as expected plus the index for each record. This is because the default for write.csv includes the default row.names = TRUE. Again examine the help file for this function.

```
write.csv(appling, file = "test.csv", row.names = F)
```

R also allows you to read .csv files using the read.csv function. Read the file you have created into a variable:

```
tmp.appling <- read.csv(file = "test.csv")
```

Notice that in this case what is read from the .csv file is assigned to the variable tmp.appling. Try reading this file without assignment. The default for read.csv is that the file has a header (i.e. the first row contains the names of the columns) and that the separator between values in any record is a comma. However, these can be changed depending on the nature of the file you are seeking to load into R. A number of different types of files can be read into R. You should examine the help files for reading data in different formats. Enter ??read to see some of these listed. You will note that read.table and write.table require more parameters to be specified than read.csv and write.csv.

2.6.2 R Data Files

It is possible to save variables that are in your workspace to a designated file. This can be loaded at the start of your next session. For example, if you have been running the code as introduced in this chapter you should have a number of variables, from x at the start to engine and colours and the appling data above.

You can save this workspace using the drop-down menus in the RStudio interface or using the save function. The RStudio menu route saves everything that is present in your workspace, as listed by ls(), while the save command allows you to specify what variables you wish to save.

```
# this will save everything in the workspace

save(list = ls(), file = "MyData.RData")

# this will save just appling

save(list = "appling", file = "MyData.RData")

# this will save appling and georgia.polys
```

```
save(list = c("appling", "georgia.polys"), file = "MyData.RData")
```

You should note that the .RData file binary format is very efficient at storing data: the Appling .csv file used 4kb of memory, while the .RData file used only 2kb. Similarly, .RData files can be loaded into R using the menu in the R interface or within the R console by writing:

```
load("MyData.RData")
```

This will load the variables in the .RData file into the R console.

2.6.3 Spatial Data Files

It is appropriate to briefly consider how to get spatial data in and out of R, but note that this is covered in more detail in Chapter 3.

The rgdal package includes two generic functions for reading and writing all kinds of spatial data: readOGR() and writeOGR(). Load the rgdal package:

```
library(rgdal)
```

The georgia object in sp format can be written to a shapefile using the writeOGR() function as follows:

```
writeOGR(obj=georgia, dsn=".", layer="georgia", driver="ESRI Shapefile",
overwrite_layer=T)
```

It can be read back into R using the readOGR() function:

```
new.georgia <- readOGR("georgia.shp")
```

Spatial data can be also be read in and written out using the sf functions st_read() and st_write(). For example, to read in and write out the georgia.shp shapefile that was created above (and to overwrite g2) the following code can be used. You will need to install and load the sf package:

```
install.packages("sf", dep = T)

library(sf)

setwd("/MyPath/MyFolder")

g2 <- st_read("georgia.shp")

st_write(g2, "georgia.shp", delete_layer = T)
```

2.7 Answers to Self-Test Questions

Q1: orange is not one of the factor's levels, so the result is an NA.

```
colours[4] <- "orange"

colours

 [1] red    blue   red    <NA>   silver red  white silver
 [9] red    red    white  silver silver

Levels: red blue white silver black
```

Q2: There is no count for black in the character version – table does not know that this value exists, since there is no levels information. Also the order of colours is alphabetical in the character version. In the factor version, the order is based on that specified in the factor function.

Q3: The first variable is tabulated along the rows, the second along the columns.

Q4: Find the colours of all cars with engines with capacity greater than 1.1 litres:

```
# Undo the colour[4] <- 'orange' line used above

colours <- factor(c("red","blue","red","white", " silver","red","white","silver",

"red","red","white","silver"),

levels=c("red","blue","white","silver","black"))

colours[engine > "1.1litre"]
```

```
[1] blue    white <NA>    red    white red silver <NA>

Levels: red blue white silver black
```

Counts of types of all cars with capacity below 1.6 litres:

```
table(car.type[engine < "1.6litre"])

saloon    hatchback convertible

7           4           0
```

Counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litres:

```
table(colours[(engine >= "1.3litre") & (car.type == "hatchback")])

   red    blue    white silver    black

2    0     0     0     0
```

Q5: The index returned corresponds to the *first* number taking the largest value.

Q6: An expression to find the index of the largest value in each row of crosstab using which.max and apply:

```
apply(crosstab,1,which.max)

      saloon    hatchback    convertible

   1        1           3
```

Q7: Use apply functions to return the best-selling colour and car type:

```
apply(crosstab,1,which.max.name)

      saloon    hatchback    convertible

   "red"    "red"     "white"

apply(crosstab,2,which.max.name)
```

```
red       blue       white       silver       black

15 "hatchback" "saloon" "saloon" "saloon" "saloon"
```

Q8: An R expression that assigns the best-selling colour and car types to a list:

```
most.popular <- list(colour=apply(crosstab,1,which.max.name),

type=apply(crosstab,2,which.max.name))

most.popular

$colour

saloon    hatchback    convertible

"red"     "red"     "white"

$type

red       blue       white       silver       black

15 "hatchback" "saloon" "saloon" "saloon" "saloon"
```

Q9: A print function for variables of class data.frame:

```
print.sales.data <- function(x) {

    cat("Weekly Sales Data:\n")

cat("Most popular colour:\n")

    for (i in 1:length(x$colour)) {

cat(sprintf("%12s:%12s\n",names(x$colour)[i],x$colour[i]))}

cat("Most popular type:\n")

for (i in 1:length(x$type)) {

        cat(sprintf("%12s:%12s\n",names(x$type)[i],x$type[i]))} cat("Total Sold = ",x$total)

}

this.week

Weekly Sales Data:
```

```
Most popular colour:
        saloon:        red
    hatchback:        red
    convertible:       white
    Most popular type:
            red:     hatchback
blue:     saloon
        white:     saloon
        silver:     saloon
        black:     saloon
    Total Sold =    13
```

Although the above is one *possible* solution to the question, it is not unique. You may decide to create a very different looking print.sales.data function. Note also that although until now we have concentrated only on print functions for different classes, it is possible to create class-specific versions of *any* function.

Reference

Wilkinson, L. (2005) The Grammar of Graphics. New York: Springer.

3 Basics of Handling Spatial Data In R

3.1 Overview

The aim of this chapter is to provide an introduction to the mapping and geographical data handling capabilities of R. It explicitly focuses on developing the building blocks for the spatial data analyses in later chapters. These extend the mapping functionality that was briefly introduced in the previous chapter and will be extended further in Chapter 5. It includes an introduction to the sp and sf packages and the R spatial data formats they support, and the tmap package. This chapter describes methods for moving between the sp and sf formats and 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 later in the book. This chapter will:

- Introduce the sp and sf R spatial data formats and describe how to use them
- Describe how to compile maps based on multiple layers using both basic plot functions and the tmap package
- Describe how to set different plot parameters and shading schemes
- Describe how to develop basic descriptive statistical analyses of spatial data

3.1.1 Spatial Data

Data are often held in data tables or databases – a bit like a spreadsheet. The rows represent some real-world feature (a person, a transaction, a date, etc.) and the columns represent some attribute associated with that feature. Rows in databases may be referred to as *records* and columns as *fields*. There are some cases where the features can be either a record or a field – for example, a date could belong to a list of daily supermarket transactions (as a record) or be an attribute associated with an event at a location (as a field). For the purposes of much of the practical work in this chapter data will be

conceptualised in this way.

In R there are many data formats and packages for handling and manipulating them. For example, the tibble format defined within the dplyr package as part of the tidyverse is starting to supersede data frames (in fact it includes the data.frame class). This is part of a concerted activity by many package development teams to provide tidy and lazy data formats and processes for data science, mapping and spatial data analysis. Some of the background to this activity can be found on the webpage for tidyverse (https://www.tidyverse.org), which is a collection of R packages designed for data science.

The preceding description of *data*, with records (rows) and fields (columns), can be extended to *spatial data* in which each record typically represents some real-world *geographical* feature – a place, a route, a region, etc. – and individuals fields provide a measurement or attribute associated with that feature. In geographical data, features are typically represented as points, lines or areas.

Why spatial data? Nearly all data are *spatial* – they are collected some*where*. If and when a third edition of this book is written in the future, we expect to extend this argument to the spatio-temporal domain in which all data are spatio-temporal – they are collected some*where* and at some *time*.

3.1.2 Installing and Loading Packages

The previous chapter included a number of 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 pre-written 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.

This chapter uses a number of packages: raster, OpenStreetMap, RgoogleMaps, grid, rgdal, tidyverse, reshape2, ggmosaic, GISTools, sf and tmap. You will have to install them before you use them for the first time. You will have installed the GISTools and sf packages using the install.packages() function if you worked through Chapter 2. Once you have downloaded and installed a package, you can simply load the package when you use R subsequently.

The is.element query combined with the installed.packages() function can be used to check whether a package is installed.

```
is.element("sf", installed.packages())
```

If FALSE is returned then you need to install the package as above:

```
install.packages("sf", dep = TRUE)
```

Note the dep = TRUE parameter. This tells R to load the package with its dependencies (i.e. other packages that it depends on). Then the package can be loaded:

```
library(sf)
```

It is possible to inspect the functionality and tools available in sf or any other package by examining the documentation.

```
help(sf)

# or

?sf
```

This provides the 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 sf see http://cran.r-project.org/web/packages/sf/index.html.

3.2 Introduction to sp and sf: The sf Revolution

As described in Chapter 1, the first edition of this book focused on the sp format for spatial data in R. This format is defined in the sp package. It provides an organised set of spatial data classes, providing a unified way of moving from one package to another, taking advantage of the different tools and the functions they include. However, R is dynamic and sometimes a new paradigm is introduced; this has been the case recently for spatial data in R, with the release of the sf package by Pebesma et al. (2016).

In this chapter, both the sp and sf formats are introduced. The manipulation and analysis of spatial data use, where possible, the sf format and associated tools. However, some packages and operations for spatial analyses have not yet been updated to work with sf. For example, at the time of writing, many of the functions in spdep, such as those for cluster analysis using Moran's *I* (see Anselin, 1995) and the *G*-statistic (described in Ord and Getis, 1995), only work with sp format spatial data. For these reasons, this chapter (and others throughout the book) will, where possible, describe the manipulation and analysis of spatial data using sf format and functions but will switch between (and convert data between) sp and sf formats as needed.

3.2.1 sp data format

The sp package defines a number of classes (or sp objects) for handling points, lines and areas, as summarised in Table 3.1. The sp data formats underpin many of the packages that you will use directly or indirectly (i.e. they are loaded by other packages): they have *dependencies* on sp. An example is the GISTools package by Brunsdon and Chen (2014) which has dependencies on maptools, sp, rgeos and other packages. If you install and load GISTools you will see these packages being loaded.

Table 3.1

Table 3.1 Spatial data formats in R

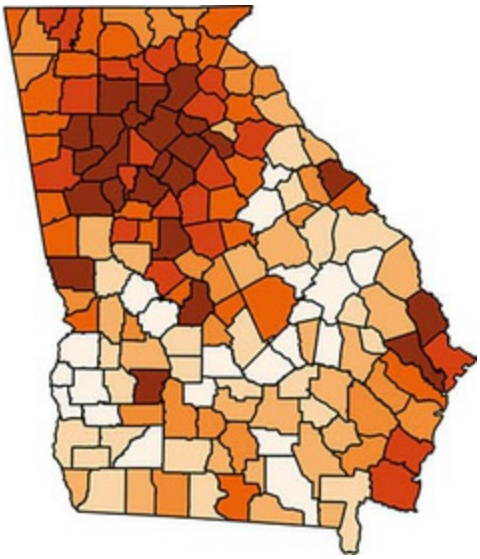
Without attributes	With attributes	ArcGIS equivalent
SpatialPoints	SpatialPointsDataFrame	Point shapefiles
SpatialLines	SpatialLinesDataFrame	Line shapefiles
SpatialPolygons	SpatialPolygonsDataFrame	Polygon shapefiles

Pebesma et al. (2016).
Pebesma et al. (2016).

3.2.1.1 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, to illustrate spatial data manipulation, mapping and analysis in both sf and sp. These are polygon and line data for New Haven, Connecticut and the 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 Georgia data include outlines of the counties in Georgia 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



Having installed GISTools, you can load the newhaven data or georgia data using the data() function. Load the newhaven data and then examine what is loaded and the types (or classes) of data that are loaded:

```
data(newhaven)
ls()
[1] "blocks" "breach" "burgres.f" "burgres.n"
[5] "famdisp" "places" "roads" "tracts"
class(breach)
[1] "SpatialPoints"
attr(,"package")
[1] "sp"
class(blocks)
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

The breach data are of the SpatialPoints class and simply describe locations, with no attributes. The blocks data, on the other hand, 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 in sp 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))
```

Note that the data frame of sp objects can also be accessed using the @data parameter of the blocks data frame using the head function:

```
head(blocks@data)
```

Both of these code snippets print 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 using the R plot function:

```
plot(blocks)
```

The default plot function for the sp class of objects can be used to generate maps, and this was the focus of the first edition of this book using the GISTools packages. It described how different plot commands could be combined to created plot layers. For example, to draw a map of the roads in red, with the blocks in black (the plot default colour) as in Figure 3.2, the code below could be entered:

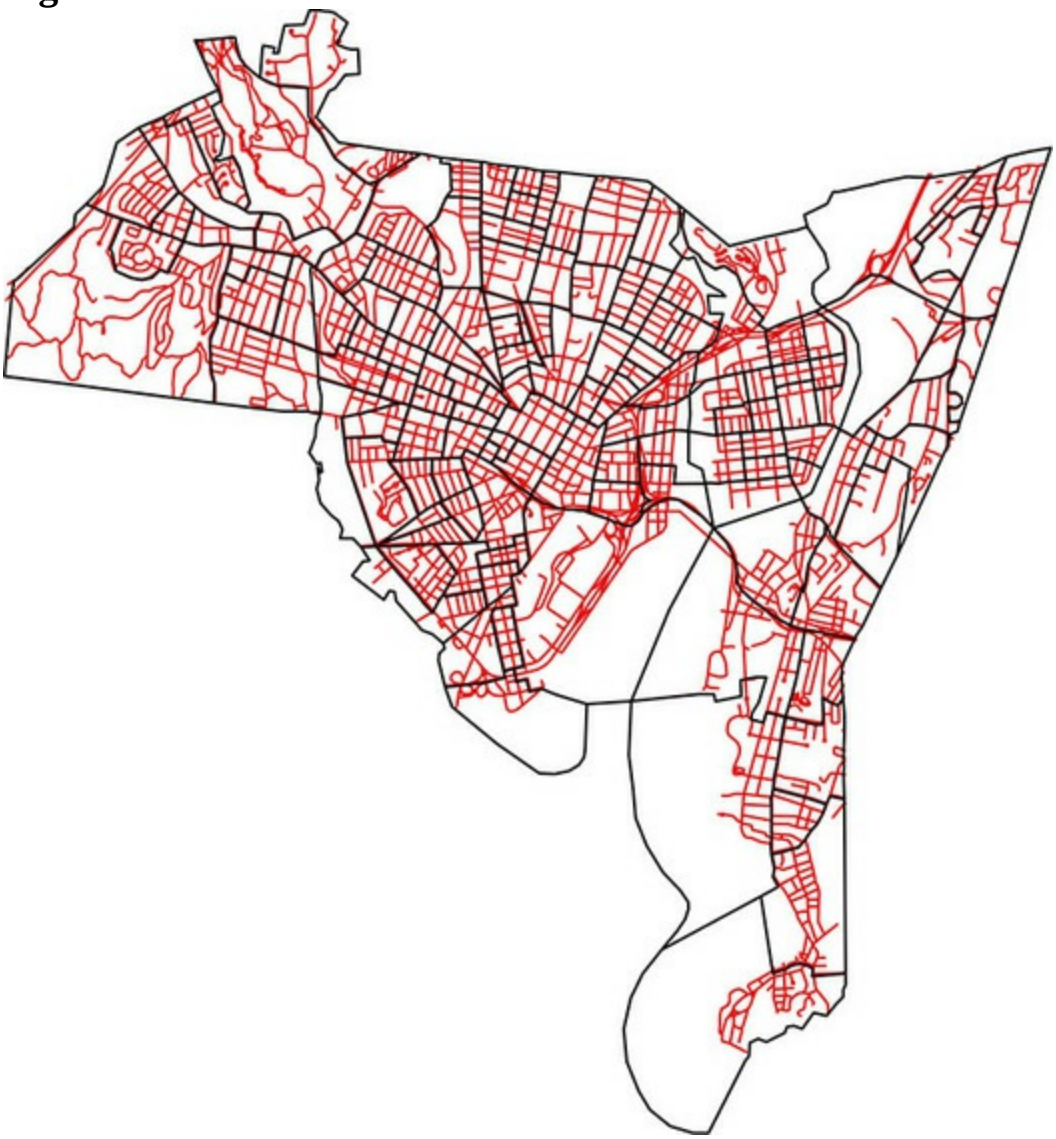
```
par(mar = c(0,0,0,0))

plot(roads, col="red")

plot(blocks, add = T)
```

3.2.2 sf Data Format

Recently a new class of R spatial objects has been defined and released as a package called sf, which stands for ‘simple features’ (Pebesma et al., 2016).

It seeks to encode spatial data in a way that conforms to a formal standard (ISO 19125-1:2004). This emphasises the spatial geometry of objects, the way that objects are stored in databases. In brief, the aim of the team developing sf (actually many of them are the same people who developed sp, so they do know what they are doing!) is to provide a format for spatial data. An overview of the evolution of spatial data in R can be found at https://edzer.github.io/UseR2017/.

Figure 3.2 The New Haven census blocks and road data



The idea is that a *feature* is a thing, or an object in the real world, such as a building or a tree. As is the case with objects, they often consist of other objects such that a set of features can form a single feature. Features have a

geometry describing where on Earth they are located, and they have attributes, which describe other properties. There are many sf object types, but the key ones (which are similar to *lines*, *points and areas*) are listed in Table 3.2 (taken from the sf vignette). This has a much stronger theoretical structure, with for example multipoint features being composed of point features etc. Only the more common types of geometries defined within sf are described in Table 3.2; other geometries exist but are much rarer.

Table 3.2https://r-spatial.github.io/sf/articles/sf1.html

Table 3.2 Spatial data formats in R from https://r-spatial.github.io/sf/articles/sf1.html

Feature type	Description	ArcGIS equivalent
POINT	Zero-dimensional geometry containing a single point	Point shapefiles
LINESTRING	Sequence of points connected by straight, non-self-intersecting line pieces; one-dimensional geometry	Line shapefiles
POLYGON	Geometry with a positive area (two-dimensional); sequence of points form a closed, non-self-intersecting ring; the first ring denotes the exterior ring, zero or more subsequent rings denote holes in this exterior ring	Polygon shapefiles
MULTIPOINT	Set of points; a MULTIPOINT is simple if no two points in the MULTIPOINT are equal	Point shapefiles
MULTILINESTRING	Set of linestrings	Line shapefiles
MULTIPOLYGON	Geometry with a positive area (two-dimensional); sequence of points form a closed, non-self-intersecting ring; the first ring denotes the exterior ring, zero or more subsequent rings denote holes in this exterior ring	Polygon shapefiles

Ultimately, sf formats will completely replace sp, and packages that use sp (such as GWmodel for geographically weighted regression) will all have to be updated to use sf at some point, but that is a few years away.

The sf package has a number of vignettes or tutorials that you could explore. These include an overview of the format, reading and writing from and to sf formats including conversions to and from sp and sf, and some illustrations of how sf objects can be manipulated. The code below will create a new window with a list of sf vignettes:

```
library(sf)
```

```
vignette(package = "sf")
```

And then to display a specific vignette topic, this can be called using the vignette function:

```
vignette("sf1", package = "sf")
```

```
I

Vignettes are an important part of R packages. They provide explanations of the package functionality additional to those found in the example code at the end of a help page. They can be accessed using the vignette function or through the R help. The sf1 vignette could also be accessed via the package help index: enter help(sf), navigate to the index through the link at the bottom of the overview page and then click on the User guides, package vignettes and other documentation link.
```

3.2.2.1 sf spatial data

The sp objects loaded by the GISTools data packages georgia and newhaven can be converted to sf. The fundamental function for converting to sf is st_as_sf(). In the code below it is used to convert the georgia sp object to sf:

```
# load the georgia data

data(georgia)

# conversion to sf

georgia_sf = st_as_sf(georgia)

class(georgia_sf)

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

You can examine the contents of georgia_sf by entering the following at the console:

```
georgia_sf
```

Notice how when georgia_sf is called the spatial information and the first 10

records of the attribute table are printed to the screen, rather than the entire object as with sp. For comparison you could enter:

```
georgia
```

The plot function is also different: it will create maps of sf objects, and if the sf object has attributes it will shade the first few of these:

```
# all attributes
plot(georgia_sf)
# selected attribute
plot(georgia_sf[, 6])
# selected attributes
plot(georgia_sf[,c(4,5)])
```

Finally, note that sf objects have a data frame. You could compare the data frames of sp and sf objects:

```
## sp SpatialPolygonDataFrame object
head(data.frame(georgia))
## sf polygon object
head(data.frame(georgia_sf))
```

Note that the data frames of the sf objects have geometry attributes.

We can also convert to sp by using the as function:

```
g2 <- as(georgia_sf, "Spatial")
class(g2)
[1] "SpatialPolygonsDataFrame"
attr(,"package")
```

```
[1] "sp"
```

This automatically recognises the georgia_sf is a multipolygon object in sf and converts it to a SpatialPolygonsDataFrame object in sp. You could try a similar set of operations with the roads layer loaded earlier to demonstrate this:

```
roads_sf <- st_as_sf(roads)

class(roads_sf)

r2 <- as(roads_sf, "Spatial")

class(r2)
```

3.3 Reading and Writing Spatial Data

Very often we have data that are in a particular format such as *shapefile* format. R has the ability to read and write data from and to many different spatial data formats using functions in the rgdal and sf packages – we will consider them both here.

3.3.1 Reading to and Writing from sp Format

As was briefly described in Chapter 2, the rgdal package includes two generic functions for reading and writing all kinds of spatial data: readOGR() and writeOGR(). Load the rgdal package:

```
library(rgdal)
```

As a reminder, the georgia object in sp format can be written to a shapefile using the writeOGR() function as follows:

```
writeOGR(obj=georgia, dsn=".", layer="georgia", driver="ESRI Shapefile",
overwrite_layer=T)
```

You will see that a shapefile has been written into your current working

directory, overwriting any previous instance of georgia.shp, with its associated supporting files (.dbf etc.) that can be recognised by other applications (QGIS etc.). Similarly, this can be read into R and assigned to a variable using the readOGR function:

```
new.georgia <- readOGR("georgia.shp")
```

If you enter:

```
class(new.georgia)
```

you will see that the class of the new.georgia object is sp. You should examine the writeOGR and readOGR functions in the rgdal package.

R is also able to read and write other proprietary spatial data formats using a number of packages, which you should be able to find through a search of the R help system or via an internet search engine. The rgdal package is the R version of the *Geospatial Data Abstraction Library*. It includes a number of methods for reading and writing spatial objects, including to and from SpatialXDataFrame objects. The full syntax can be important – the code below overwrites any existing similarly named file:

```
writeOGR(new.georgia, dsn = ".", layer = "georgia", driver="ESRI Shapefile",
overwrite_layer = T)
```

The dsn parameter is important here: for shapefiles it determines the folder the files are written to. In the above example it was set to "." which places the files in the current working directory.

You could specify a file path here. For a PC it might be something like D:\MyDocuments\MyProject\DataFiles; for a Mac, /Users/lex/my_docs/project.

The setwd() and getwd() functions can be used in determining and setting the file path. You may want to set the file path and then use the dsn setting as above:

```
setwd("/Users/lex/my_docs/project")

writeOGR(new.georgia, dsn = ".", layer = "georgia", driver="ESRI Shapefile",
overwrite_layer = T)
```

Or you could use the getwd() function, save the results to a variable and pass this to writeOGR:

```
td <- getwd()

writeOGR(new.georgia, dsn = td, layer = "georgia", driver="ESRI Shapefile",
overwrite_layer = T)
```

You should also examine the functions for reading and writing raster layers in rgdal, which are readGDAL and writeGDAL. These read and write functions in rgdal are incredibly powerful and can read/write almost any spatial data format.

3.3.2 Reading to and Writing from sf Format

Spatial data can be also be read in and written out using the sf functions **st_read**() and **st_write**(). For example, to read in the georgia.shp shapefile that was created above (and to overwrite g2) the following code can be used:

```
setwd("/MyPath/MyFolder")

g2 <- st_read("georgia.shp")
```

The working directory needs to be set to ensure that st_read looks in the right place to read the file from. Here a single argument is used to find both the data source and the layer. This works when the data source contains a single layer.

To write a simple features object to a file needs at least two arguments, the object and a filename. As before, this will not work if the georgia.shp file exists in the working directory, so the delete_layer = T parameter needs to be specified.

```
st_write(g2, "georgia.shp", delete_layer = T)
```

The filename is taken as the data source name. The default for the layer name is the basename (filename without path) of the data source name. For this, st_write needs to guess the driver. The above command, for instance, is equivalent to:

```
st_write(g2, dsn = "georgia.shp", layer = "georgia.shp", driver = "ESRI Shapefile",
delete_layer = T)
```

Typical users will use a filename with path for filename, or first set R's working directory with setwd() and use filename without path.

Note that the output driver is guessed from the data source name, from either its extension (.shp: ESRI Shapefile), or its prefix (PG:: PostgreSQL).

The list of extensions with corresponding driver (short driver name) can be found in the sf2 vignette. You will also note that there are a number of functions that can be used to read, write and convert. You can examine this:

```
vignette("sf2", package = "sf")
```

3.4 Mapping: An Introduction To tmap

3.4.1 Introduction

The first parts of this chapter have outlined basic commands for plotting data and for producing maps and graphics using R. These were based on the plot functions associated with sp objects. This section will now concentrate on developing and expanding these basic techniques using the functions in the tmap package. It will introduce some new plot parameters and will show how to extract and download Google Maps and to use OpenStreetMap data as background context and to create interactive (at least zoomable) maps in tmap. 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.

The tmap mapping package (Tennekes, 2015) focuses on mapping the spatial distribution of *thematic* data attributes. It can take sp and sf objects. It has a similar grammar to plotting with ggplot in that it seeks to handle each element of the map separately in a series of layers, and in so doing seeks to exercise control over each element. This is different from the basic plot functions used above to map sp and sf data.

In this section the workings of tmap will be introduced, and then in later sections on mapping attributes this will be expanded and refined to impose different mapping styles and embellishments. To begin with, you will need some predetermined data, and the code in this section will use the georgia and georgia_sf objects that were created earlier. As ever, you may wish to think about creating a script and a workspace folder in which you can store any results you generate. As a reminder, you can clear your workspace to remove all the variables and datasets you have created and opened using the previous code and commands. This can be done via the menu in RStudio via **Session > Clear Workspace**, or via the console by entering:

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

3.4.2 A quick tmap

The qtm() function can be used to compose a *quick* map. The code below loads the georgia data, recreates georgia_sf and generates a quick tmap using qtm. First load the data:

```
data(georgia)
```

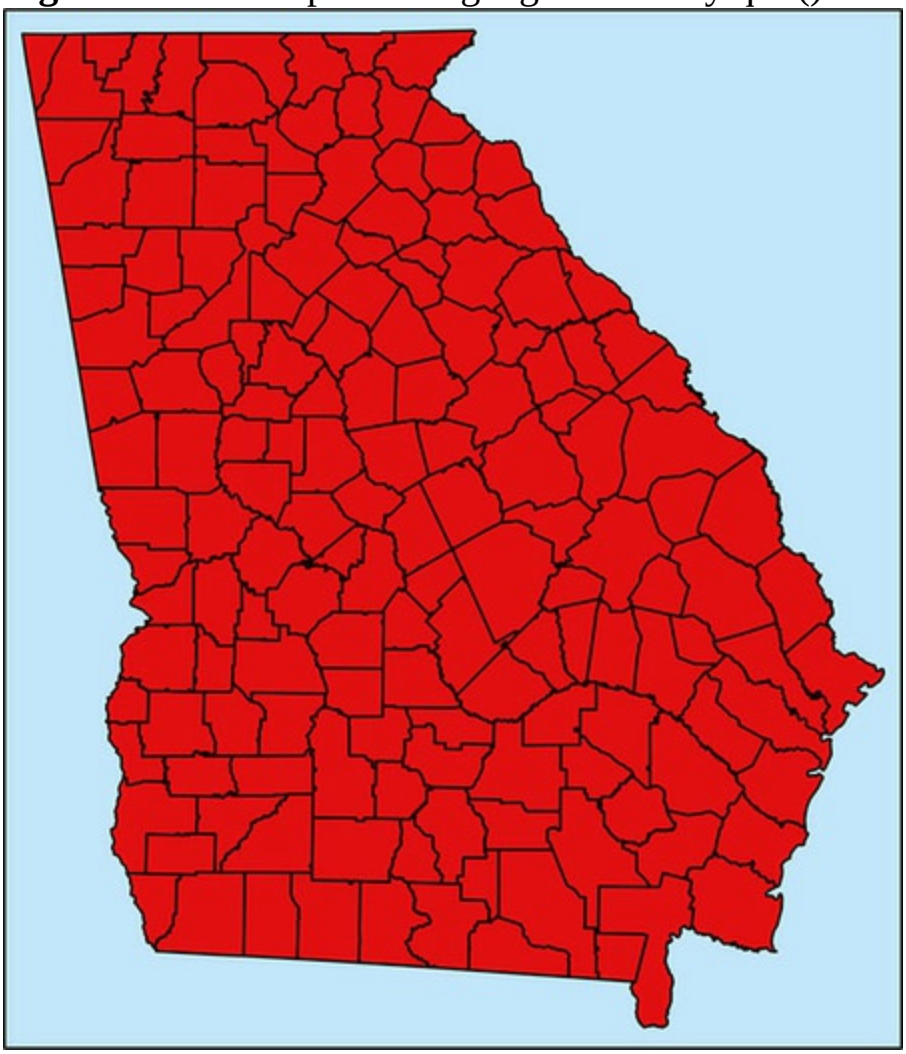
Check that the data have loaded correctly using ls(). There should be three Georgia datasets: georgia, georgia2 and georgia.polys. Then create the sf object georgia_sf as before:

```
georgia_sf <- st_as_sf(georgia)
```

Finally load tmap and create a quick map as in Figure 3.3:

```
library(tmap)

qtm(georgia, fill = "red", style = "natural")
```

Figure 3.3 The map of Georgia generated by qtm()

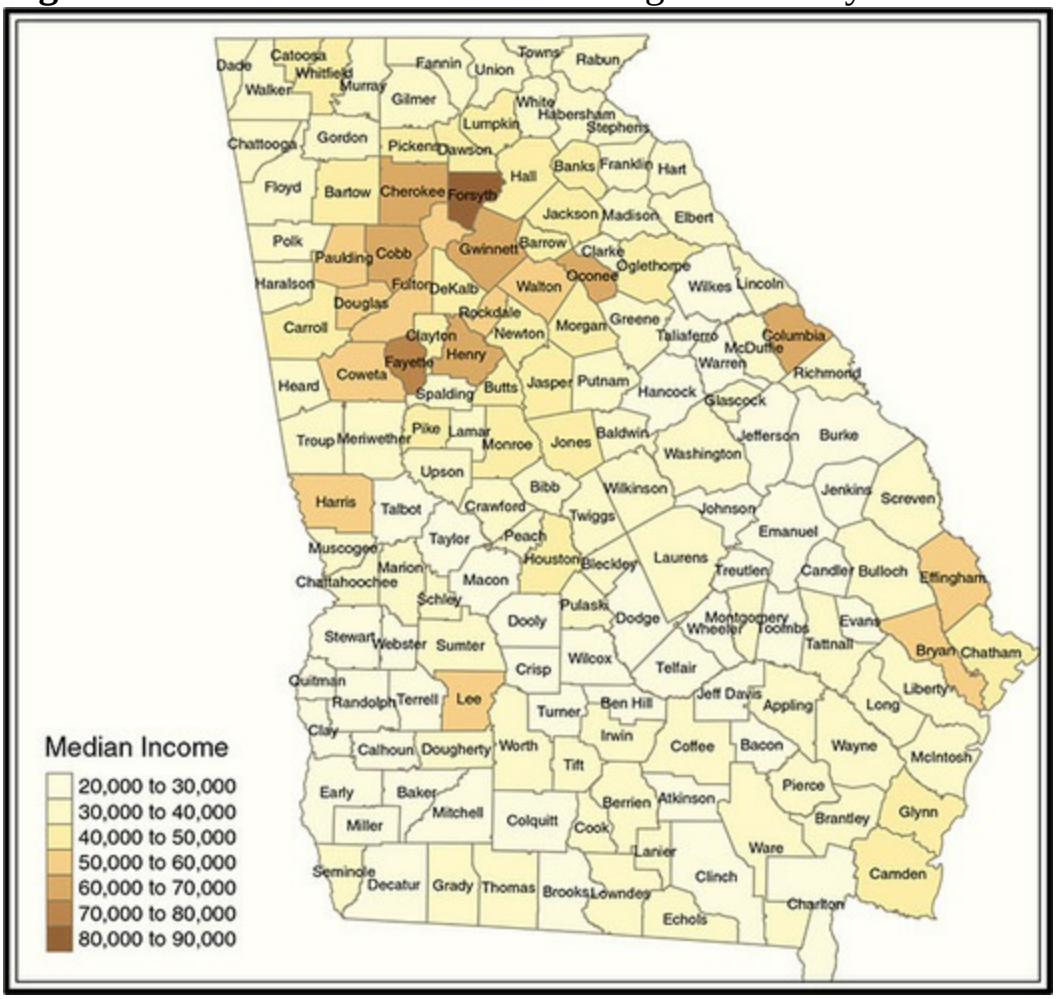


Note the use of the style parameter. This is a shortcut to a predefined style within the tmap package, in this case named tm_style. These styles can be called in abbreviated form using qtm. You should explore the qtm function through the help.

The fill parameter can be used to specify a colour as above, or a variable to be mapped. The code below generates Figure 3.4, which shows the distribution of the MedInc variable:

```
qtm(georgia_sf, fill="MedInc", text="Name", text.size=0.5, format="World_wide",
style="classic",

text.root=5, fill.title="Median Income")
```

Figure 3.4 Counties in the state of Georgia shaded by median income



3.4.3 Full tmap

The process of making maps using tmap is one in which a series of layers are added to the map. First the tm_shape() is specified, followed by a tmap aesthetic function that specifies what is to be plotted. This can be illustrated

by running the code snippets below and inspecting the results. You should see how the tmap functions are added as a series of layers to the map in a similar way to ggplot. Before this an outline of Georgia is created using the st_union() function in sf. An alternative for sp is the gUnaryUnion() function in the rgeos package loaded with GISTools. The manipulation of spatial data using overlay, union and intersection functions is covered in more depth in Chapter 5.

```
# do a merge
g <- st_union(georgia_sf)
# for sp
# g <- gUnaryUnion(georgia, id = NULL)
# plot the spatial layers
tm_shape(georgia_sf) + tm_fill("tomato")
```

Add the county borders:

```
tm_shape(georgia_sf) + tm_fill("tomato") +
tm_borders(lty = "dashed", col = "gold")
```

Add some styling:

```
tm_shape(georgia_sf) + tm_fill("tomato") +
tm_borders(lty = "dashed", col = "gold") +
tm_style("natural", bg.color = "grey90")
```

Include the outline, noting the second call to tm_shape to plot the second spatial object g:

```
tm_shape(georgia_sf) +
tm_fill("tomato") +
tm_borders(lty = "dashed", col = "gold") +
```

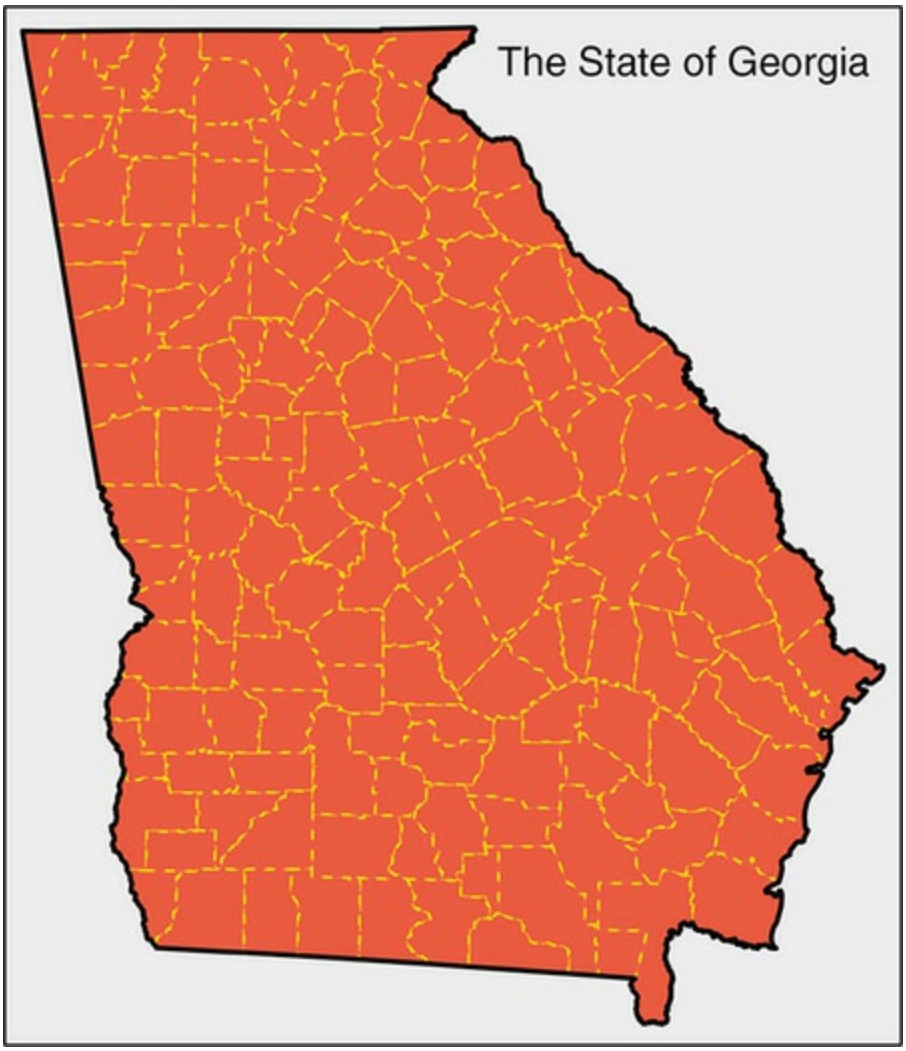
```
tm_style("natural", bg.color = "grey90") +
# now add the outline
        tm_shape(g) +
tm_borders(lwd = 2)
```

And finally putting it all together to create Figure 3.5:

```
tm_shape(georgia_sf) + tm_fill("tomato") +
tm_borders(lty = "dashed", col = "gold") +
tm_style("natural", bg.color = "grey90") +
# now add the outline
        tm_shape(g) + tm_borders(lwd = 2) +
        tm_layout(title = "The State of Georgia", title.size = 1, title.position = c(0.55,
"top"))
```

So what you can see in the above code are two sets of tmap plot commands: the first set plots the georgia_sf dataset, specifying a dashed gold line to show the county boundaries, a tomato (red) fill colour for the state and a map background colour of light grey. The second set adds the outline created by the union operation with a thicker line width before the title is added.

Figure 3.5 Counties in the state of Georgia



It is also possible to plot multiple different maps from different datasets together, but this requires a bit more control over the tmap parameters. The code below assigns each map to variables t1 and t2, and then a second set of functions is used to manipulate these in a plot window. Note that georgia2 is in sp format and has a different map projection than georgia. For this reason, the aspect of the second plot is specified for the second plot in the code below. The value was determined through trial and error. You will need to install and load the grid package.

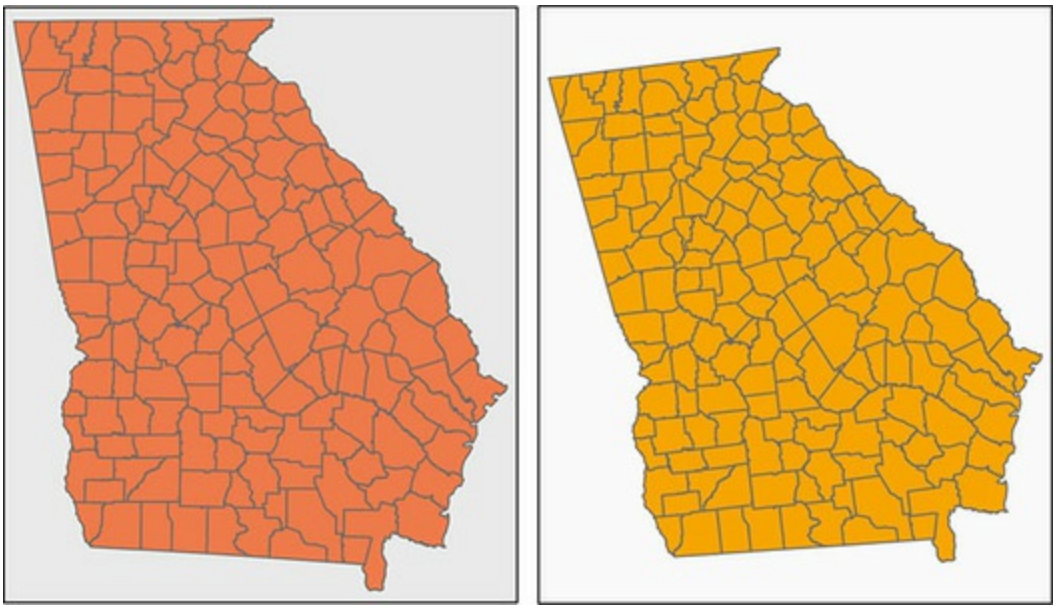
```
# 1st plot of georgia

t1 <- tm_shape(georgia_sf) +

                    tm_fill("coral") + tm_borders() +
```

```
tm_layout(bg.color = "grey85")

# 2nd plot of georgia2

t2 <- tm_shape(georgia2) +

tm_fill("orange") +

tm_borders() +

# the asp parameter controls aspect

# this makes the 2nd plot align

tm_layout(asp = 0.86,bg.color = "grey95")
```

Now you can specify the layout of the combined map plot as in Figure 3.6:

```
library(grid)

# open a new plot page

grid.newpage()

# set up the layout

pushViewport(viewport(layout=grid.layout(1,2)))

# plot using the print command

print(t1, vp=viewport(layout.pos.col = 1, height = 5))

print(t2, vp=viewport(layout.pos.col = 2, height = 5))
```

Figure 3.6 Examples of the use of tmap to generate multiple maps in the same plot window



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

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

It would be useful to display these on the map, and this can be done using the tm_text function in the maptools package that is loaded with tmap. The result is shown in Figure 3.7.

```
tm_shape(georgia_sf) +
        tm_fill("white") +
        tm_borders() +
        tm_text("Name", size = 0.3) +
        tm_layout(frame = FALSE)
```

And we can subset the data as with the sp format. The code below subsets the counties of Jefferson, Jenkins, Johnson, Washington, Glascock, Emanuel,

Candler, Bulloch, Screven, Richmond and Burke:

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

The notation for subsetting is the same as for sp objects, and enables individual areas or polygons to be selected from spatial datasets using the bracket notation as used in matrices, data frames and vectors. The subset can be plotted to generate Figure 3.8 using the code below.

```
tm_shape(georgia_sf.sub) +
        tm_fill("gold1") +
tm_borders("grey") +
tm_text("Name", size = 1) +
        # add the outline
tm_shape(g) +
tm_borders(lwd = 2) +
# specify some layout parameters
tm_layout(frame = FALSE, title = "A subset of Georgia",
title.size = 1.5, title.position = c(0., "bottom"))
```

Finally, we can bring together the different spatial data that have been created in a single map as in Figure 3.9 using the code below. You should note how the different tm_shape, tm_fill etc. functions are used to set up each layer of the map and that tmap determines the map extent from the layers:

```
# the 1st layer
tm_shape(georgia_sf) +
tm_fill("white") +
tm_borders("grey", lwd = 0.5) +
```

```
                # the 2nd layer

tm_shape(g) +

tm_borders(lwd = 2) +

# the 3rd layer

tm_shape(georgia_sf.sub) +

                tm_fill("lightblue") + tm_borders() +

# specify some layout parameters

                tm_layout(frame = T, title = "Georgia with a subset of counties", title.size = 1,
title.position = c(0.02, "bottom"))
```

Figure 3.7 Adding text to map objects with tmap

Figure 3.8 A subset of the counties in the state of Georgia

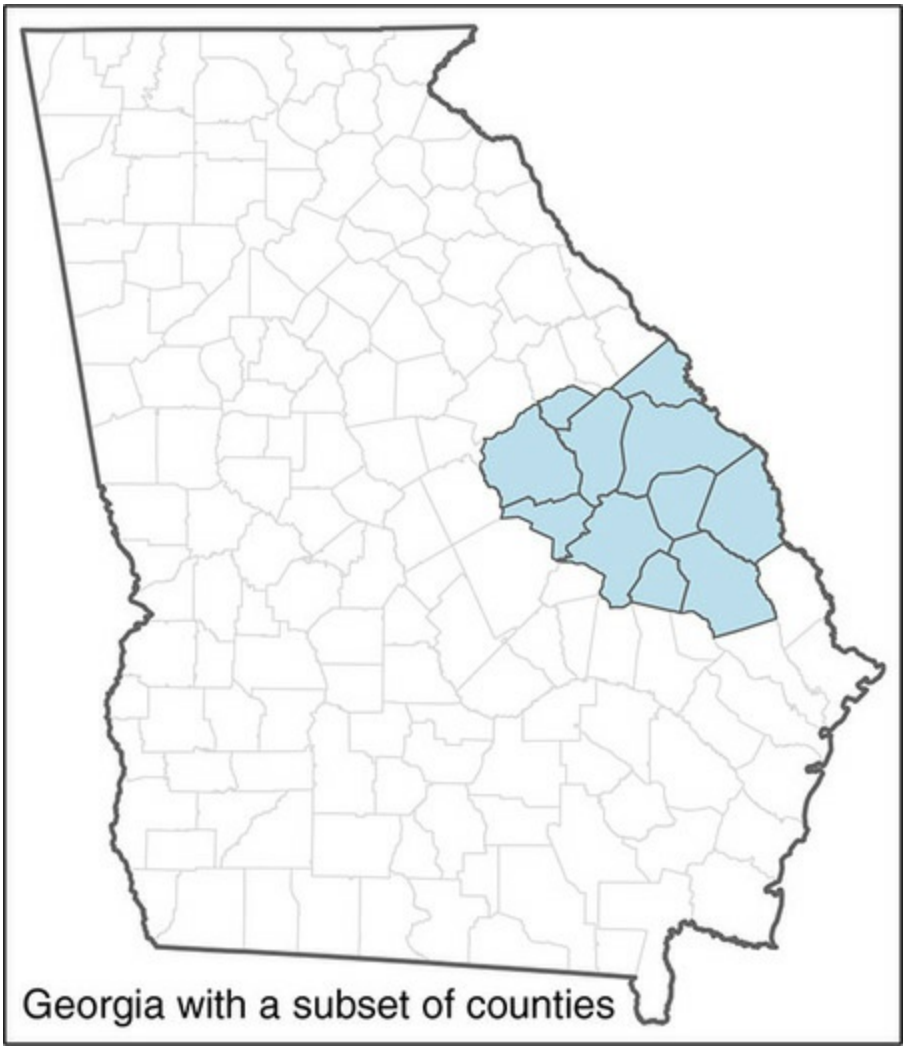
3.4.4 Adding Context

In some situations a map with background context may be more informative. There are a number of options for doing this, including OpenStreetMap,[1] Google Maps and Leaflet. This requires some additional packages to be downloaded and installed in R. If you have not done so already, install the OpenStreetMap package and load it into R:

[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 programs, 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.

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

library(OpenStreetMap)
```

Figure 3.9 The result of the code for plotting a spatial object and a spatial subset



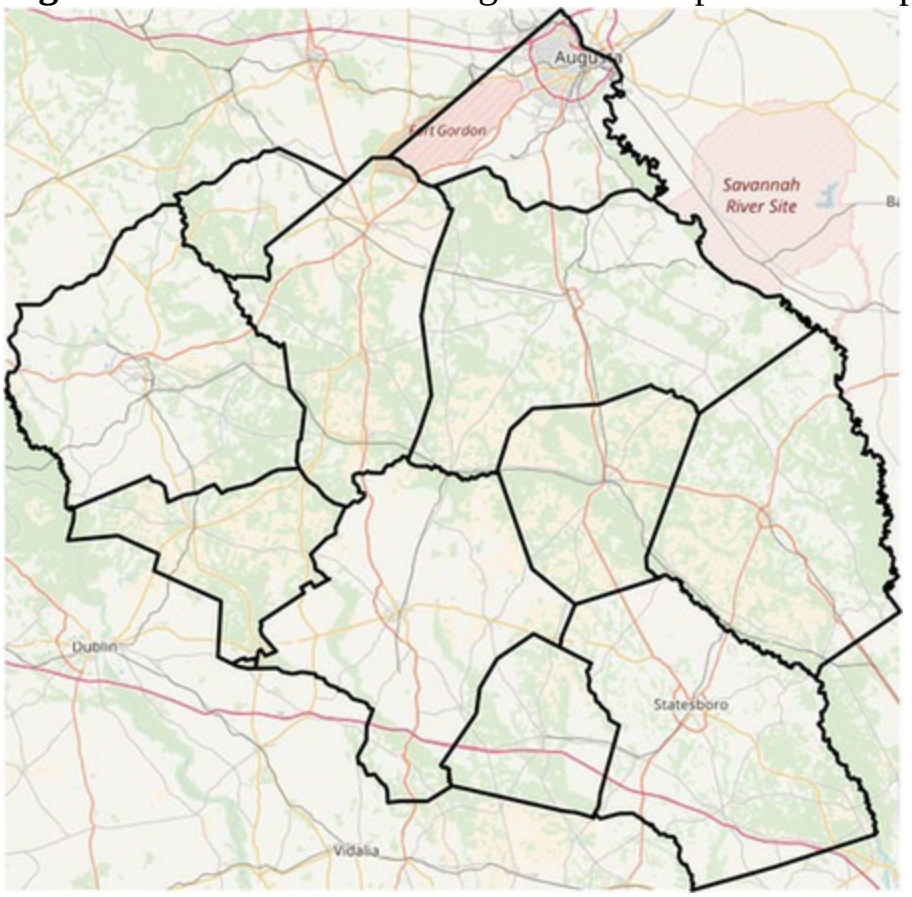
If using OpenStreetMap, the approach is to define the area of interest, to download and plot the map tile from OpenStreetMap and then to plot your data over the tiles. In this case the background map area is defined by the spatial extent of the Georgia subset created above which is used determine the tiles to download from OpenStreetMap. The results of the code below are shown in Figure 3.10. Note the use of the spTransform function in the rgdal package in the last line of the code. This transforms the geographical projection of the georgia.sub data to the same projection as the OpenStreetMap data layer. Here it is easier to work with sp objects.

```
# define upper left, lower right corners
```

```
georgia.sub <- georgia[index,]

ul <- as.vector(cbind(bbox(georgia.sub)[2,2], bbox(georgia.sub)[1,1]))

lr <- as.vector(cbind(bbox(georgia.sub)[2,1], bbox(georgia.sub)[1,2]))

# download the map tile

MyMap <- openmap(ul,lr)

# 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)
```

Google Maps can also be downloaded and used as context. Again, this package should be installed if you have not done so already.

Figure 3.10 A subset of Georgia with an OpenStreetMap backdrop



```
install.packages(c("RgoogleMaps"),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 using the SpatialPolygons2PolySet function in maptools (loaded with GISTools) and the last line that defines a polygon plot over Google Maps:

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

It is also possible to use the tmap package for context using Leaflet. Leaflet is an open source JavaScript library used to build interactive web mapping applications (see https://rstudio.github.io/leaflet/) and is embedded within the tmap package. It is useful if you want to embed interactive maps in an html file (e.g. by using RMarkdown). The code below maps georgia.sub with an interactive Leaflet backdrop as in Figure 3.11. Note that the interactive mode is set through the tmap_mode function, which in this case has been set to ’view’, which requires an internet connection, with the alternative being ’plot’.

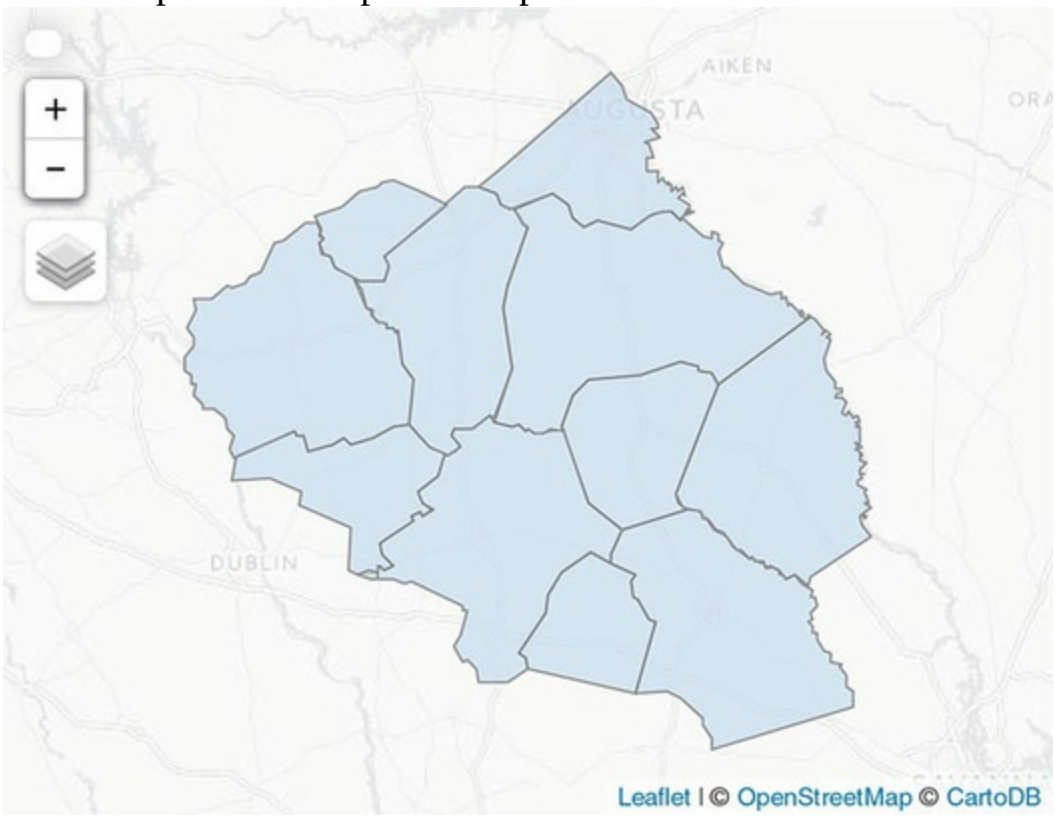
```
tmap_mode(’view’)
```

```
tmap mode set to interactive viewing

tm_shape(georgia_sf.sub) + tm_polygons(col = "#C6DBEF80" )
```

Finally, remember to reset the tmap_mode to plot:

```
tmap_mode("plot")
```

Figure 3.11 An interactive map of the Georgia subset with Leaflet/OpenStreetMap backdrop



3.4.5 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 in RStudio is to click on the Export icon in the plot pane for saving options (in R, right-click with the mouse on the map window), select **Copy to Clipboard**, and then paste it into a word-processing document (e.g. one being created in either OpenOffice or

MS Windows). Another is to use **Save as Image** 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 edits and 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 to re-create the image file. There are a number of formats for saving maps, such as PDF, PNG and TIFF.

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 RStudio, open a new document by selecting **File > New File > R script**. Then type in the following:

```
# load package and data
library(GISTools)
data(newhaven)
proj4string(roads) <- proj4string(blocks)
# plot spatial data
tm_shape(blocks) + tm_borders() +
        tm_shape(roads) +     tm_lines(col = "red") +
        # embellish the map
tm_scale_bar(width = 0.22) + tm_compass(position = c(0.8, 0.07)) +
tm_layout(frame = F, title = "New Haven, CT", title.size = 1.5,
title.position = c(0.55, "top"), legend.outside = T)
```

Save the file as newhavenmap.R in your working directory.

I

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 RStudio this is **Session > Set Working Directory > ...**. In R in Windows it 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 tm_lines command, and edit the line to become:

```
tm_lines(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 tm_borders command in the file to become:

```
tm_borders(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:

```
tm_lines(col = "blue", lwd = 2) +
```

and source the file again. This time the roads are thicker and drawn in blue.

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 tm_shape command in the newhavenmap.R file. This tells R that after this command, any graphics will

not be drawn on the screen, but instead are 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 insert the following at the end of the tmap commands:

```
dev.off()
```

This 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 should be 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.

Of course you do not need to load a .R file to do this! You can place the opening and closing commands around the mapping code.

There are a number of commonly used functions for writing maps out to PDF, PNG, TIFF, etc., files:

```
pdf()

png()
```

```
tiff()
```

Examine the help for these.

The key thing you need to know is that these functions all *open* a file. The open file needs to be closed using dev.off() after the map has been written to it. So the syntax is:

```
pdf(file = "MyPlot.pdf", other setting)

<tmap code>

dev.off()
```

You can write a .png file for the map using the code below. Note that you may want to set the working directory that you write to using the setwd() function. To illustrate this the code below creates some points for the georgia_sf polygon centroids, sets the working directory and then creates a map:

```
pts_sf <- st_centroid(georgia_sf)

setwd('~/Desktop/')

# open the file

png(filename = "Figure1.png", w = 5, h = 7, units = "in", res = 150)

# make the map

tm_shape(georgia_sf) + tm_fill("olivedrab4") + tm_borders("grey", lwd = 1) +

# the points layer

tm_shape(pts_sf) +

tm_bubbles("PctBlack", title.size = "% Black", col = "gold")+ tm_format_NLD()

# close the png file

dev.off()
```

3.5 Mapping Spatial Data Attributes

3.5.1 Introduction

This section describes some approaches for displaying and mapping spatial data attributes. Some of these ideas and commands have already been used in the preceding 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 (e.g. 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 one these are the census reporting areas (census blocks or tracts), and in Georgia the counties within the state. These areas have attributes from the population census 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.

3.5.2 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 geographical 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, convert to sf format and then examine the blocks, breach and tracts data:

```
# clear workspace

rm(list = ls())
```

```
# load & list the data
data(newhaven)
ls()
# convert to sf
blocks_sf <- st_as_sf(blocks)
breach_sf <- st_as_sf(breach)
tracts_sf <- st_as_sf(tracts)
# have a look at the attributes and object class
summary(blocks_sf)
class(blocks_sf)
summary(breach_sf)
class(breach_sf)
summary(tracts_sf)
class(tracts_sf)
```

You should notice a number of things from these summaries:

- Each of the datasets is spatial: blocks_sf and tracts_sf are *POLYGON* sf objects and breach is a *POINT* object.
- They all have *data frames* attached to them that contain attributes whose values are summarised by the summary function.
- breach_sf only has geometry attributes – it has no thematic attributes, it just records locations.

The data frame of these spatial objects can be 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, the individual polygons for example in blocks_sf, 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_sf)
```

would print all of the attribute information for each census block in New Haven to the R console window, until the print limit was reached, while:

```
head(data.frame(blocks_sf))
```

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_sf))

# or

names(blocks_sf)
```

Note that for sp objects, an alternative is to use @data to access the data frame of the *SpatialPolygonsDataFrame* objects, as well as the above code:

```
colnames(blocks@data)

head(blocks@data)
```

One of the data attributes or variables 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_sf$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_sf))
```

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, but this can lead to problems if you are not careful. To detach the data frame you attached earlier, enter:

```
detach(data.frame(blocks_sf))
```

You can *try* a similar set of commands with the tracts data, but the breaches data has no attributes: it simply records the locations of breaches of the peace. As with any point data, the breaches of the peace data can be used to create a *heat map* raster dataset.

```
# use kde.points to create a kernel density surface

breach.dens = st_as_sf(kde.points(breach,lims=tracts))

summary(breach.dens)
```

breach.dens is a raster/pixels dataset, and its attributes are held in a *data frame* which can be examined:

```
breach.dens
```

Notice that this has the kernel density estimation and geometry attributes that describe the X and Y locations, and you can plot the breach.dens object:

```
plot(breach.dens)
```

Also note that you can remove the st_as_sf function from the kde.points command to generate a SpatialPixelsDataFrame object, part of the sp family of spatial objects. This can be plotted with the image function.

A final key point about attributes is that you can create and assign new attributes to the spatial object, for both sf and sp. For example, the code below creates a normally distributed random value for each of the 129 areas in the blocks_sf object. Note the use of the $ to do this:

```
blocks_sf$RandVar <- rnorm(nrow(blocks_sf))
```

Of course it is more than likely that you will want to assign a new value to a spatial object that arises from the result of some kind of analysis, data join, etc. It is very easy to link new data attributes to spatial objects in this way.

3.5.3 Mapping Polygons and Attributes

A *choropleth* is a thematic map in which areas are shaded in proportion to their attributes. The tmap package includes a number of ways of generating choropleth maps. Enter:

```
tmap_mode('plot')

tm_shape(blocks_sf) +

            tm_polygons("P_OWNEROCC")
```

This produces a map of the census block in New Haven, shaded by the proportions of vacant properties. The tm_polygons element automatically includes a legend to allow the map to be interpreted, in this case the levels of vacancy associated with each of the different shade colours.

There are a couple of things to note about the use of tmap. First, tmap_mode was set to plot to generate a standard choropleth suitable for including in a report rather than an interactive map for use in a webpage, for example. Recall that the Leaflet mapping above used the interactive view (i.e. tmap_mode was set to 'view'). Second, in a similar way to the ggplot operations in Chapter 2, the tmap package constructs maps by combining different map elements. In this case blocks_sf was passed to the tm_shape function and then the tm_polygons function was used to specify the variable to be mapped, in this case P_OWNEROCC.

You should note that it is also possible to pass sp format spatial objects to tmap. Try replacing tm_shape(blocks_sf) with tm_shape(blocks) in the code above and below. Also note that in this case the variable P_OWNEROCC was mapped using five classes of equal interval. Try repeating the tmap code above using a different variable such as P_VACANT. What happens? You will see that tmap automatically determines the number of classes to be included and the class intervals or breaks. Finally, a colour shading scheme is automatically allocated to the map and the legend is included in the map. All of these, and many of the other default mapping settings that tmap uses, can be controlled and modified.

For example, to control the class intervals, the breaks parameter can be specified:

```
tm_shape(blocks_sf) + tm_polygons("P_OWNEROCC", breaks=seq(0, 100, by=25))
```

This can be done in many different ways:

```
tm_shape(blocks_sf) + tm_polygons("P_OWNEROCC", breaks=c(10, 40, 60, 90))
```

The legend placement and title can be modified. The tm_layout function is very useful here:

```
tm_shape(blocks_sf) + tm_polygons("P_OWNEROCC", title = "Owner Occ") +
tm_layout(legend.title.size = 1,

legend.text.size = 1,

legend.position = c(0.1, 0.1))
```

You could also try legend.position = c("centre","bottom")). Further documentation on tm_layout can be found at
https://www.rdocumentation.org/packages/tmap/versions/1.11/topics/tm_layo

It is also possible to alter the colours used in a shading scheme. The default colour scheme uses increasing intensities of yellow to red. Graduated lists of colours like this are generated using the RColorBrewer package, which is automatically loaded with both tmap and GISTools. This package makes use

of a set of colour palettes designed by Cynthia Brewer, intended to optimise the perceptual difference between each shade in the palette, so that visually each shading colour is distinct. 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"
```

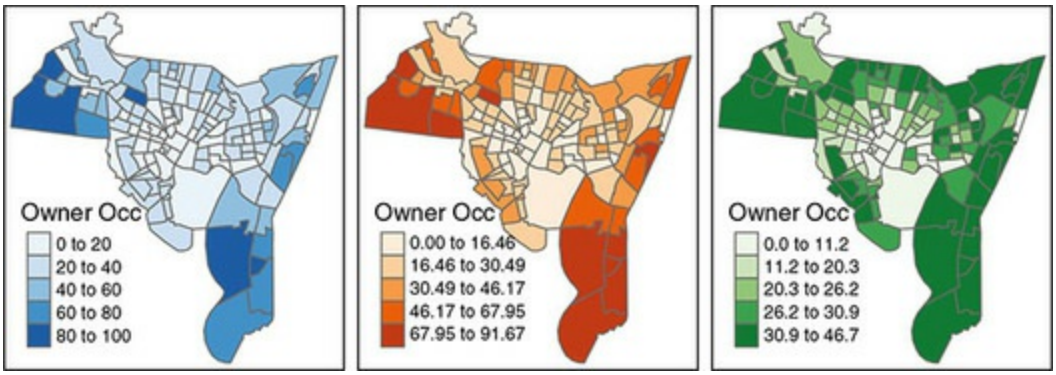
This is a list of colour codes used by R to specify the palette. The brewer.pal arguments specify that a five-stage palette based on shades of blue is required. The output of brewer.pal can be fed into tmap to give alternative colours in shading schemes. For example, enter the code below and a choropleth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

```
tm_shape(blocks_sf) +

tm_polygons("P_OWNEROCC", title = "Owner Occ", palette = "Reds") +
tm_layout(legend.title.size = 1)
```

Note that the same map would be produced if the tm_fill function were used instead of tm_polygons; however, without a tm_borders function, the census block outlines are not plotted. Try entering:

```
tm_shape(blocks_sf) +

tm_fill("P_OWNEROCC", title = "Owner Occ", palette = "Blues") +
tm_layout(legend.title.size = 1)
```

Figure 3.12 Different choropleth maps of owner-occupied properties in New Haven using different shades and class intervals



A final adjustment is to change the way the class interval boundaries are computed. As a default, they are based on equal-sized intervals of the attribute being mapped, but different palette *styles* are available. Have a look at the help for tm_polygons and you will see that a number of different plotting styles are available. You should explore these. The class intervals can be changed to quantiles or any other range of intervals using the breaks parameter. For example, the code below produces three maps in Figure 3.12 with equal intervals (left), with intervals based on *k*-means (middle) and with quantiles (right), using the quantileCuts function in GISTools, and using the pushViewport function in the grid package as before to plot multiple maps together.

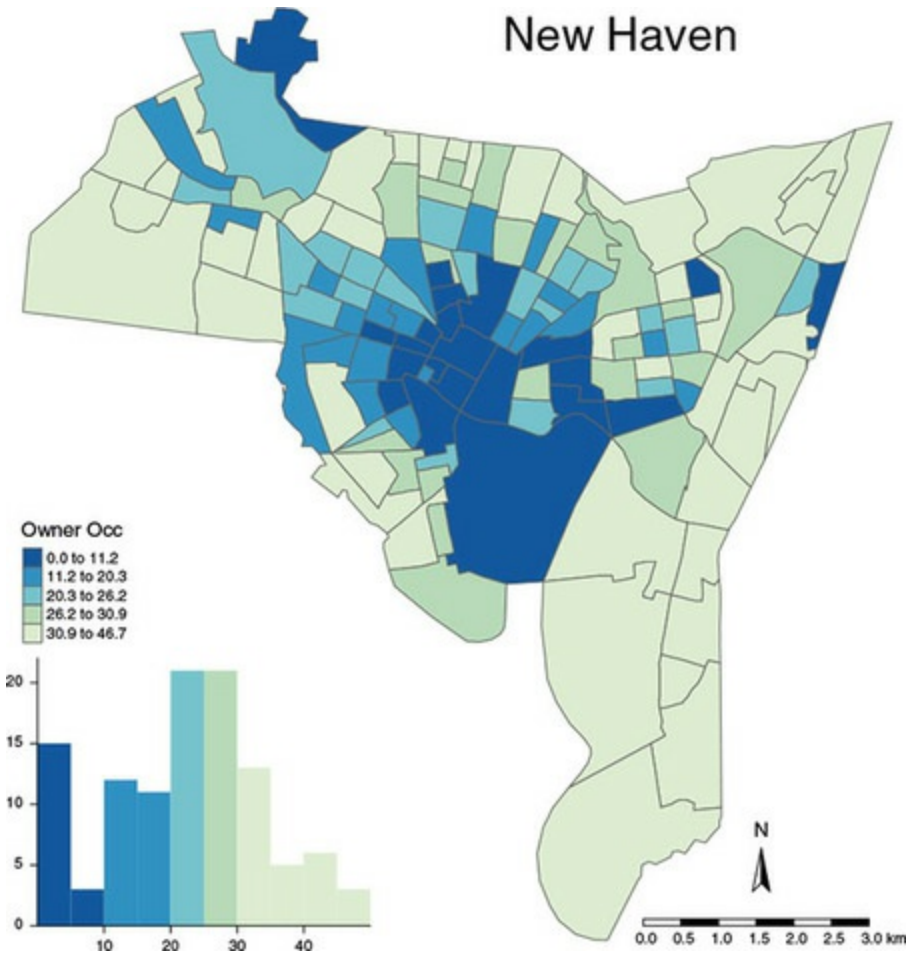
```
# with equal intervals: the tmap default

p1 <- tm_shape(blocks_sf) +

tm_polygons("P_OWNEROCC", title = "Owner Occ", palette = "Blues") +
tm_layout(legend.title.size = 0.7)

# with style = kmeans

p2 <- tm_shape(blocks_sf) +

        tm_polygons("P_OWNEROCC", title = "Owner Occ", palette = "Oranges", style =
"kmeans") +

tm_layout(legend.title.size = 0.7)

# with quantiles

p3 <- tm_shape(blocks_sf) +

            tm_polygons("P_OWNEROCC", title = "Owner Occ", palette = "Greens", breaks
= c(0, round(quantileCuts(blocks$P_OWNEROCC, 6), 1))) + tm_layout(legend.title.size
= 0.7)
```

```
# Multiple plots using the grid package

library(grid)

grid.newpage()

# set up the layout

pushViewport(viewport(layout=grid.layout(1,3)))

# plot using the print command

print(p1, vp=viewport(layout.pos.col = 1, height = 5))

print(p2, vp=viewport(layout.pos.col = 2, height = 5))

print(p3, vp=viewport(layout.pos.col = 3, height = 5))
```

It is also possible to display a histogram of the distribution of the variable or attribute being mapped using the legend.hist parameter. This is very useful for choropleth mapping as it gives a distribution of the attributes being examined. Bringing this all together allows you to create a map with a number of refinements as in Figure 3.13. Note, for example, the minus sign before the palette parameter to reverse the palette order and the various parameters passed to the tm_layout function.

```
tm_shape(blocks_sf) +

tm_polygons("P_OWNEROCC", title = "Owner Occ", palette = "-GnBu",
breaks = c(0, round(quantileCuts(blocks$P_OWNEROCC, 6), 1)), legend.hist = T) +

tm_scale_bar(width = 0.22) +                tm_compass(position = c(0.8, 0.07)) +
tm_layout(frame = F, title = "New Haven",

title.size = 2, title.position = c(0.55, "top"), legend.hist.size = 0.5)
```

Figure 3.13 An illustration of the various options for mapping with tmap



It is possible to compute certain derived attribute values on the fly in tmap. The code below first assigns a projection to the tracts_sf layer from the blocks_sf layer, then plots population density using the convert2density parameter applied to the POP1990 attribute.

```
# add a projection to tracts data and convert tracts data to sf

proj4string(tracts) <- proj4string(blocks)

tracts_sf <- st_as_sf(tracts)

tracts_sf <- st_transform(tracts_sf, "+proj=longlat +ellps=WGS84")

# plot

tm_shape(blocks_sf) + tm_fill(col="POP1990", convert2density=TRUE,

                    style="kmeans", title=expression("Population (per " * km^2 * ")"),
legend.hist=F, id="name") + tm_borders("grey25", alpha=.5) +
```

```
# add tracts context

tm_shape(tracts_sf) + tm_borders("grey40", lwd=2) +

tm_format_NLD(bg.color="white", frame = FALSE, legend.hist.bg.color="grey90")
```

The convert2density function automatically converts the projection units (in this case degrees of latitude and longitude) to a projection in metres and then determines areal density in square kilometres. You can check this by creating your own population density values, and examining the explanations of how the functions operate in the help pages for the functions used, such as st_area.

Compare the population density summary with the legend of the figure created using the code above:

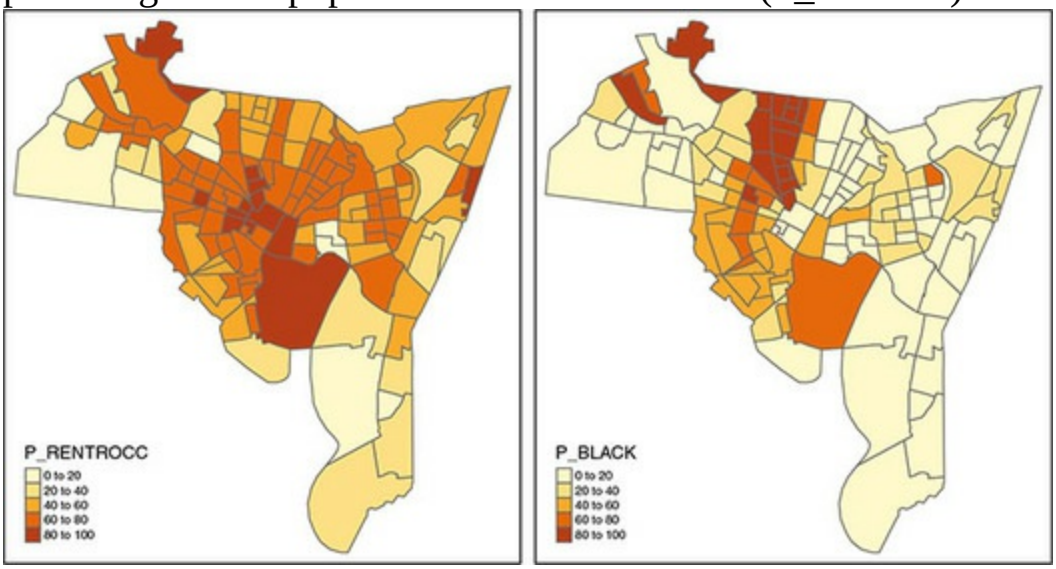
```
# add an area in km^2 to blocks

blocks_sf$area = st_area(blocks_sf) / (1000*1000)

# calculate population density manually

summary(blocks_sf$POP1990/blocks_sf$area)
```

A final consideration is the ability of tmap to map multiple attributes in the same operation. The code below plots two attributes in the same call (Figure 3.14):

```
tm_shape(blocks_sf) + tm_fill(c("P_RENTROCC", "P_BLACK")) +

tm_borders() + tm_layout( legend.format = list(digits = 0), legend.position = c("left",
"bottom"), legend.text.size = 0.5,

            legend.title.size = 0.8)
```

In summary, the tm_fill and tm_polygons functions in the tmap package generate choropleth maps of attributes held in spatial polygons data frame (sp) or simple feature (sf) data objects. They automatically shade the variables using equal intervals. The intervals and the palettes can both be adjusted. It is instructive to examine the plotting functions and the way they operate. Enter:

Figure 3.14 tmap choropleth maps of census blocks in New Haven showing the percentage of houses rented and occupied (P_RENTROCC) and the percentage of the population recorded as black (P_BLACK)




```
tm_polygons
```

The function code detail is displayed in the R console window. You will see that it takes a number of arguments and a number of default parameters. 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.

3.5.4 Mapping Points and Attributes

Point data can be mapped in R, as well as polygons and lines. The newhaven data include locations of reports of ‘breaches of the peace’. These events are essentially public disorder incidents, on many occasions requiring police intervention. The data are stored in a variable called breach, which was converted to sf format above. Plotting this variable works in the same way as plotting polygons or lines, using the tm_shape function:

```
tm_shape(blocks_sf) +
            tm_polygons("white") +
        tm_shape(breach_sf) +
```

```
tm_dots(size = 0.5, shape = 19, col = "red", alpha = 1)
```

This plots the locations of each of the breach of peace incidents with a symbol above the blocks_sf layer using the tm_dots function. This can take a number of parameters, including those to control the point size, colour and shape. The shape is drawn from the core R pch (plot character) argument. You should examine the help for pch and for points to see the different symbols (or *shapes* in the language of tmap) that can be used.

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 alpha parameter can be used to add a transparency term to the colour. Try adjusting the code above to change the transparency and the plot character. For example:

```
tm_shape(breach_sf) +

tm_dots(size = 0.5, shape = 19, col = "red", alpha = 0.5)
```

I

Transparency can also be added to shading colours manually. Remember that the full set of predefined and named colours available in R can be listed by entering colours(). Also you can list the colour in the RColorBrewer palettes. To see the palettes enter display.brewer.all() and to list colours in an individual palette enter brewer.pal(5, "Reds"). Any of these can be used in the call above. Additionally, a transparency term can be added to colour and palettes using the add.alpha function in the GISTools package. For 50% transparency enter add.alpha(brewer.pal(5, "Reds"), 0.5).

Commonly, point data come in a tabular format rather than as an R spatial object (i.e. of class sp or sf format), with attributes that include the latitude and longitude or easting and northing of the individual data points. One such dataset is the quakes dataset 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)
```

You will see that the data come 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 in sf format with the attributes included. Creating spatial data from scratch in sf is a bit convoluted, so perhaps the easiest way is to create an sp object and convert it. This is done in the code below:

```
# define the coordinates

coords.tmp <- cbind(quakes$long, quakes$lat)

# create the SpatialPointsDataFrame

quakes.sp <- SpatialPointsDataFrame(coords.tmp, data = data.frame(quakes),

proj4string = CRS("+proj=longlat "))

# convert to sf

quakes_sf <- st_as_sf(quakes.sp)
```

The result can be mapped as shown in Figure 3.15, which shows the spatial context of the data in the Pacific Ocean, to the north of New Zealand.

Figure 3.15 A plot of the Fiji earthquake data

```
# map the quakes
tm_shape(quakes_sf) +
tm_dots(size = 0.5, alpha = 0.3)
```

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

- define the coordinates for the spatial object

- assign these to an sp class of object as in Table 3.1
- then, if required, convert the sp object to sf

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

```
help("SpatialPoints-class")

help("sf")
```

You will have noticed that the quakes dataset has an attribute describing the depth of each earthquake. We can visualise the depths in a number of ways – for example, by plotting all the data points, but specifying the size of each data point to be proportional to the depth attribute, or by using choropleth mapping as above with tmap. These are shown in the code blocks below and in the results are in Figure 3.16. 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. The code below uses different plot character sizes and colours to indicate the magnitude of the variable being considered:

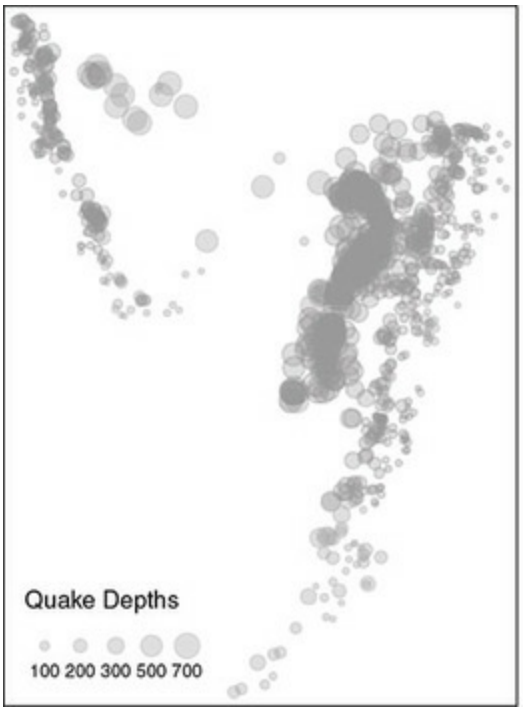
```
library(grid)

# by size

p1 <- tm_shape(quakes_sf)+

tm_bubbles("depth", scale = 1, shape = 19, alpha = 0.3, title.size="Quake Depths")

# by colour

p2 <- tm_shape(quakes_sf)+

        tm_dots("depth", shape = 19, alpha = 0.5, size = 0.6, palette = "PuBuGn",

title="Quake Depths")

# multiple plots using the grid package

grid.newpage()

# set up the layout
```

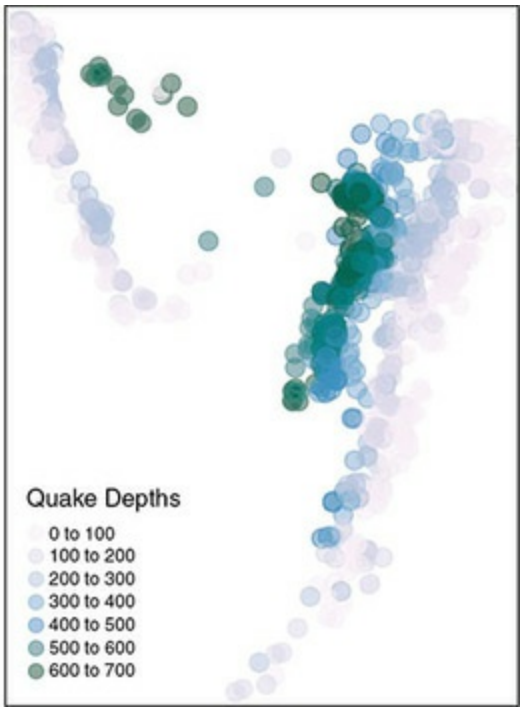
```
pushViewport(viewport(layout=grid.layout(1,2)))

# plot using the print command

print(p1, vp=viewport(layout.pos.col = 1, height = 5))

print(p2, vp=viewport(layout.pos.col = 2, height = 5))
```

It also possible to select specific data subsets to plot. The code below just maps earthquakes that have a magnitude greater than 5.5:

```
# create the index

index <- quakes_sf$mag > 5.5

summary(index)

# select the subset assign to tmp

tmp <- quakes_sf[index,]

# plot the subset

tm_shape(tmp) +

            tm_dots(col=brewer.pal(5, "Reds")[4], shape=19, alpha=0.5, size = 1) +

tm_layout(title="Quakes > 5.5", title.position = c("centre", "top"))
```

Figure 3.16 Plotting points with plot size (left) and plot colour (right) related to the attribute value.





I

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]
```

These operations are described in greater detail in Chapter 4.

Finally it is possible to use the PlotOnStaticMap function from the Rgoogle Maps package to plot the earthquake locations with some context from Google Maps. This is similar to Figure 3.10, which mapped a subset of Georgia counties against an OpenStreetMap backdrop. This time, points rather than polygons are being mapped and different Google Maps backdrops are being used as context: standard in Figure 3.17 and satellite imagery in Figure 3.18. The code for Figure 3.17 is as follows:

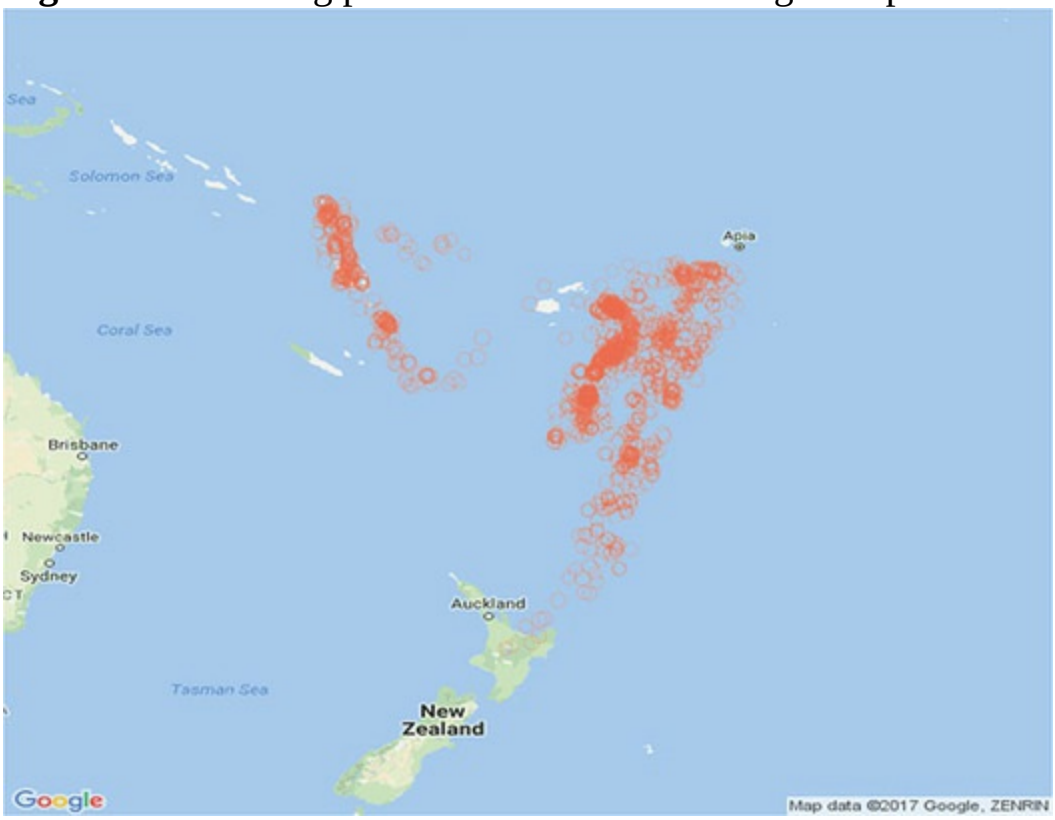
```
library(RgoogleMaps)

# define Lat and Lon
```

```
Lat <- as.vector(quakes$lat)

Long <- as.vector(quakes$long)

# get the map tiles

# you will need to be online

MyMap <- MapBackground(lat=Lat, lon=Long)

# define a size vector

tmp <- 1+(quakes$mag − min(quakes$mag))/max(quakes$mag)

PlotOnStaticMap(MyMap,Lat,Long,cex=tmp,pch=1,col='#FB6A4A30')
```

And here is the code for Figure 3.18:

```
MyMap <- MapBackground(lat=Lat, lon=Long, zoom = 10, maptype = "satellite")

PlotOnStaticMap(MyMap,Lat,Long,cex=tmp,pch=1, col='#FB6A4A50')
```

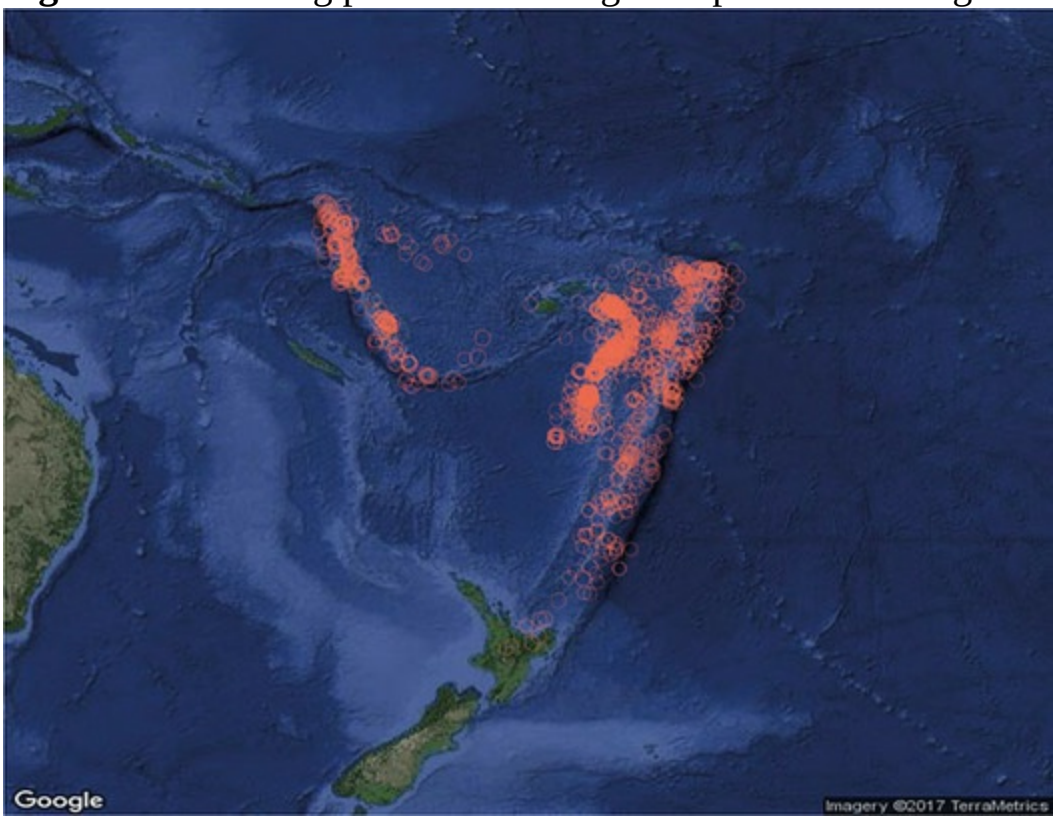
Figure 3.17 Plotting points with a standard Google Maps context



3.5.5 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. The first step in the code below assigns a coordinate system to roads and then selects a subset. This involves defining a polygon to clip the road data to, and converting and the datasets to sf objects.

Figure 3.18 Plotting points with Google Maps satellite image context




```
data(newhaven)

proj4string(roads) <- proj4string(blocks)

# 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)

proj4string(clip.bb) <- proj4string(blocks)

# 3. convert to sf

# convert the data to sf

clip_sf <- st_as_sf(clip.bb)

roads_sf <- st_as_sf(roads)

# 4. clip out the roads and the data frame

roads_tmp <- st_intersection(st_cast(clip_sf), roads_sf)
```

Note that the last line generates a warning. This is because the st_intersection function operates on geometries as well as geometry attributes under the assumption that they are the same. You can avoid this either by replacing the last line with:

```
st_intersection(st_geometry(st_cast(clip_sf)), st_geometry(roads_sf))
```

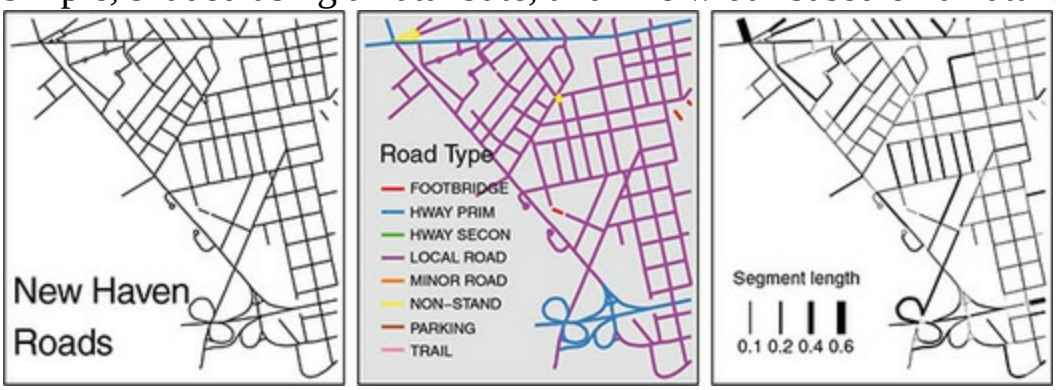
or by making the assumption that the attribute is constant throughout the geometry explicitly before the intersection as follows:

```
st_agr(x) = "constant"

st_agr(y) = "constant"
```

where x is assigned st_cast(clip_sf) and y assigned roads_sf.

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 3.19; note the different ways that the legend titles are specified.

Figure 3.19 A subset of the New Haven roads data, plotted in different ways: simple, shaded using an attribute, and line width based on an attribute



3.5.6 Mapping Raster Attributes

Earlier in this chapter a SpatialPixelsDataFrame object was created using a kernel density function. In this section the Meuse dataset, included as part of the sp package, will be used to illustrate how raster attributes can be mapped in sf.

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

```
# you may have to install the raster package

# install.packages("raster", dep = T)

library(raster)
```

```
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 as described in the help page for SpatialPixelsDataFrame and then to raster format. Note that, at the time of writing, the sf package does not have raster functionality. However, the raster package by Hijmans and van Etten (2014) handles gridded raster data excellently.

```
meuse.sp = SpatialPixelsDataFrame(points = meuse.grid[c("x", "y")], data = meuse.grid,
proj4string = CRS("+init=epsg:28992"))

meuse.r <- as(meuse.sp, "RasterStack")
```

To explore the data, you could try the simple plot and spplot functions as in the code below. For the sf object it plots all of the attributes, and for the sp object it plots the specified layer of the meuse grid:

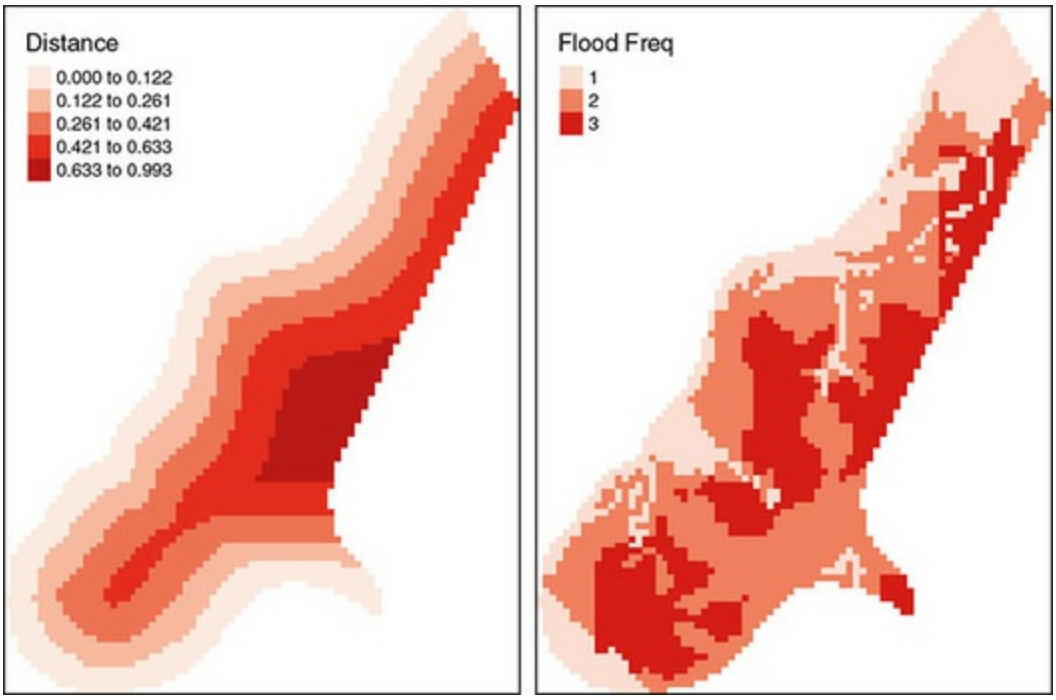
```
plot(meuse.r)

plot(meuse.sp[,5])

spplot(meuse.sp[, 3:4])

image(meuse.sp[, "dist"], col = rainbow(7))

spplot(meuse.sp, c("part.a", "part.b", "soil", "ffreq"),

col.regions=topo.colors(20))
```

However, it is possible to exercise more control over the mapping of the

attributes held in the data frame of the sf object using the functionality of tmap. Some examples of tmap mapping routines with tm_raster and different shading schemes are shown in Figures 3.20 and 3.21 with an interactive map context.

```
# set the tmap mode to plot

tmap_mode('plot')

# map dist and ffreq attributes

tm_shape(meuse.r) +

tm_raster(col = c("dist", "ffreq"), title = c("Distance","Flood Freq"), palette = "Reds",
style = c("kmeans", "cat"))

# set the tmap mode to view

tmap_mode('view')

# map the dist attribute

tm_shape(meuse.r) +

tm_raster(col = "dist", title = "Distance", breaks = c(seq(0,1,0.2))) +

tm_layout(legend.format = list(digits = 1))
```

Figure 3.20 Maps of the Meuse raster data



You could also experiment with some of the refinements as with the tm_polygons examples above. For example:

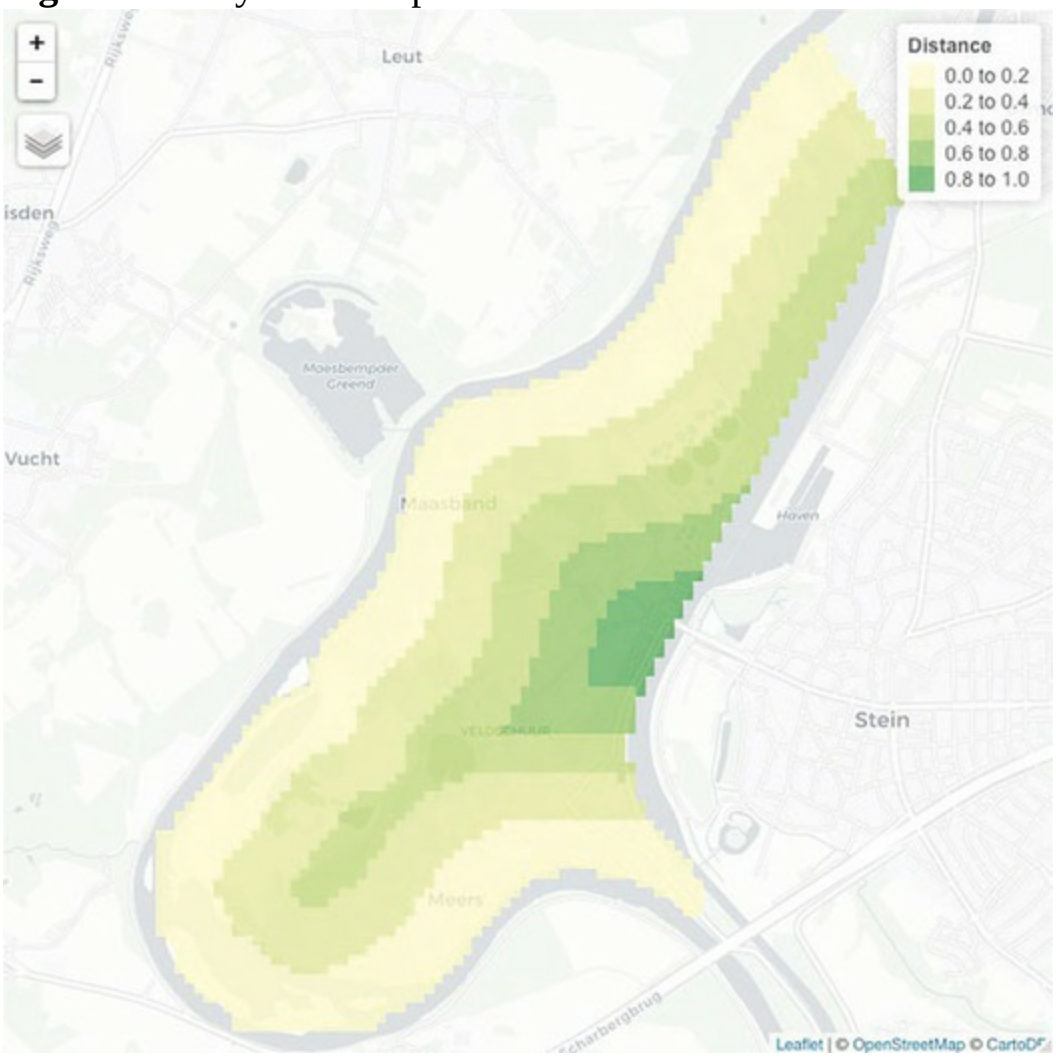
```
tm_shape(meuse.r) +

                tm_raster(col="soil", title="Soil", palette="Spectral", style="cat") +
                tm_scale_bar(width = 0.3) + tm_compass(position = c(0.74, 0.05)) +
tm_layout(frame = F, title = "Meuse flood plain", title.size = 2, title.position = c("0.2",
"top"), legend.hist.size = 0.5)
```

3.6 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. Importantly, a number of

standard plotting routines with their ggplot versions are introduced. You should load the tidyverse package which includes ggplot2, and the reshape2 package which includes some data manipulation functions:

Figure 3.21 Dynamic maps of the Meuse raster data with a Leaflet backdrop




```
install.packages("tidyverse", dep = T)

install.packages("reshape2", dep = T)
```

3.6.1 Histograms and Boxplots

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, summary and fivenum provide summaries of continuous variables, and histograms and boxplots can provide visual summaries. You should make sure the New Haven data are loaded from the GISTools package and then use these functions to explore the P_VACANT variables in blocks.

For example, typing summary(blocks$P_VACANT) or fivenum(blocks$P_VACANT) will produce other summaries of the distribution of the variable. R has some in-built functions for generating histograms and boxplots with the hist and boxplot functions. However, as described in Chapter 2, the ggplot2 package also includes functions for these visual data summaries. Code for both standard R and ggplot operations is provided in the snippets below; note the adjustment to the histogram bin sizes and the plot labels.

```
data(newhaven)

# the tidyverse package loads the ggplot2 package

library(tidyverse)

# standard approach with hist

hist(blocks$P_VACANT, breaks = 40, col = "cyan", border = "salmon",

main = "The distribution of vacant property percentages",

xlab = "percentage vacant", xlim = c(0,40))

# ggplot approach

ggplot(blocks@data, aes(P_VACANT)) +

geom_histogram(col = "salmon", fill = "cyan", bins = 40) + xlab("percentage vacant") +
labs(title = "The distribution of vacant property percentages")
```

A further way of providing visual descriptive summaries of variables is to use box-and-whisker plots via the boxplot function in R and the geom_boxplot function in ggplot2. These can summarise a single variable or multiple variables together. Here we will focus on the geom_boxplot function in the ggplot2 package. 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%. Setting the vac attribute as a factor is important for both approaches. and the melt function in the reshape2 package is critical

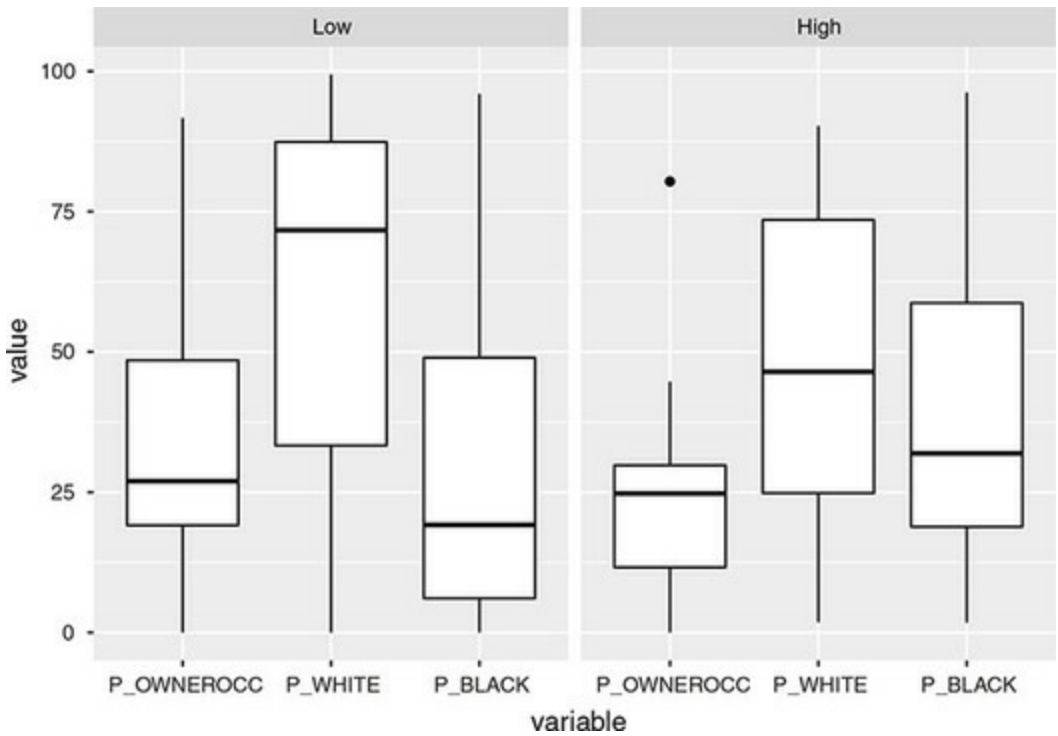
for many ggplot operations. You should examine the result of running melt(blocks@data). The geom_boxplot functions can be used to visualise the differences between these two subsets in terms of the distribution of owner occupancy and the proportion of different ethnic groups, as in Figure 3.22. First pre-process the data:

```
library(reshape2)

# a logical test

index <- blocks$P_VACANT > 10

# assigned to 2 high, 1 low

blocks$vac <- index + 1

blocks$vac <- factor(blocks$vac, labels = c("Low", "High"))
```

Then apply the geom_boxplot function:

```
library(ggplot2)

ggplot(melt(blocks@data[, c("P_OWNEROCC", "P_WHITE", "P_BLACK", "vac")]),

aes(variable, value)) + geom_boxplot() + facet_wrap(~vac)
```

Figure 3.22 Box-and-whisker plot examples



The boxplot can be enhanced in many ways in ggplot. Some parameters are used below. You may wish to search for examples of different themes and ways of manipulating boxplots.

```
ggplot(melt(blocks@data[, c("P_OWNEROCC", "P_WHITE", "P_BLACK", "vac")]),
aes(variable, value)) +
geom_boxplot(colour = "yellow", fill = "wheat", alpha = 0.7) + facet_wrap(~vac) +
xlab("") + ylab("Percentage") + theme_dark() +
ggtitle("Boxplot of High and Low property vacancies")
```

3.6.2 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 they can be examined more formally by using simple regression models 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

<- blocks$P_BLACK/100

# bind these together

df <- data.frame(p.vac, p.w, p.b)

# fit regressions

mod.1 <- lm(p.vac ~ p.w, data = df)

mod.2 <- lm(p.vac ~ p.b, data = df)
```

I

The function lm is used in R to fit regression models (lm stands for ‘linear model’). 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 basic regression model $y = ax + b$. The notation is sufficiently rich to allow the specification of a very broad set of linear models.

The two models above can be interpreted as follows: mod.1 describes the extent to which changes in p.vac are associated with changes in p.w; mod.2 describes the extent to which changes in p.vac are associated with changes in p.b. 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

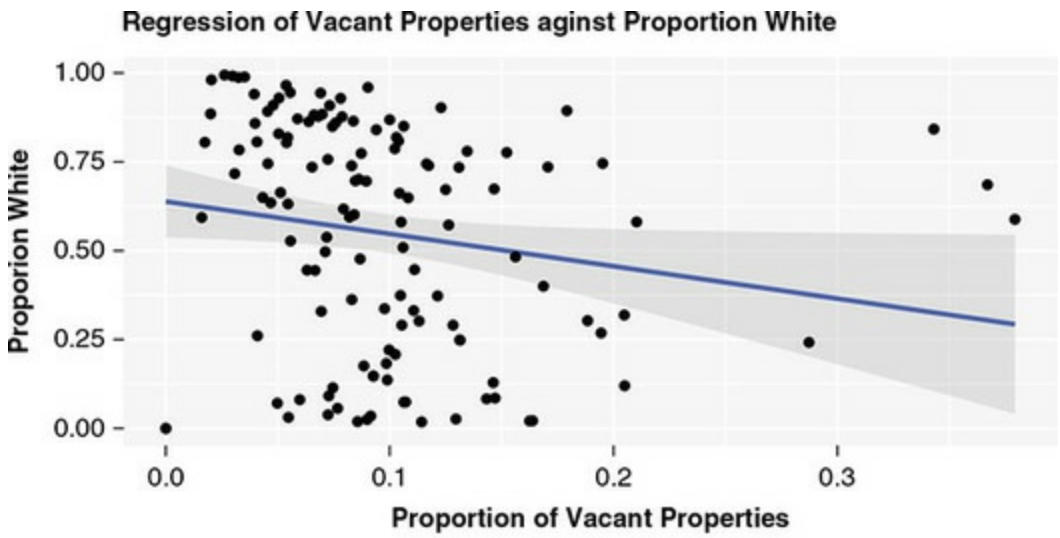
properties in a census block and that the proportion of black people is a weak positive predictor. Specifically, the model suggests relationships that indicate that the amount of vacant properties in a census block decreases by 1% for each 3.5% increase in the proportion of white people and that it increases by 1% for each 3.7% increase in the proportion of black people in the census block. However, the model fits are poor (examine the R-squared values), and when a multivariate analysis model is computed neither are found to be significant predictors of vacant properties. The models can be examined using the summary command:

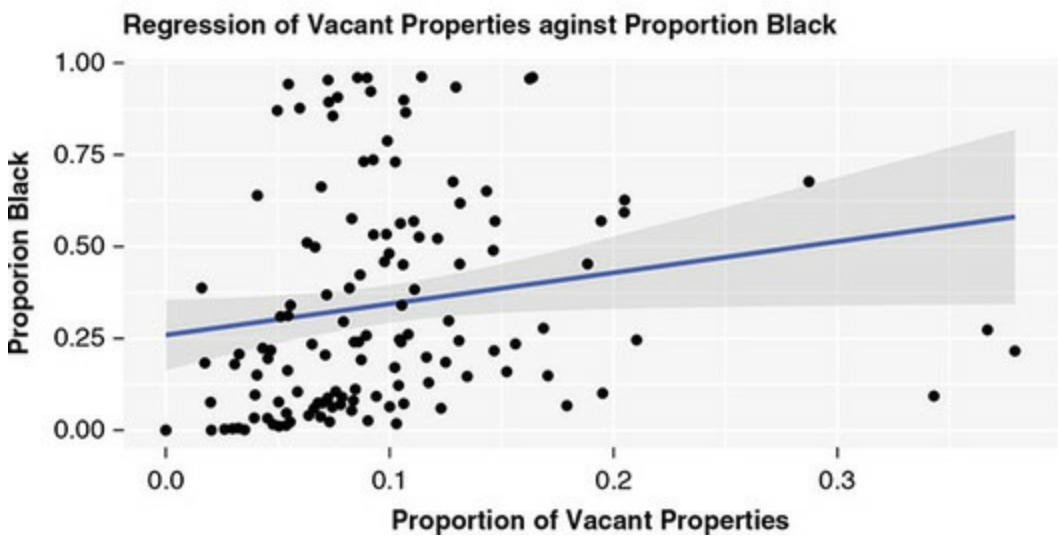
```
summary(mod.1)

Call:

lm(formula = p.vac ~ p.w, data = df)

15 Residuals:

        Min     1Q    Median     3Q     Max

−0.11747 −0.03729 −0.01199 0.01714 0.28271

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept)    0.11747    0.01092 10.755 <2e−16 ***

p.w        −0.03548 0.01723 −2.059 0.0415 *

---

Signif. codes:

0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.06195 on 127 degrees of freedom

Multiple R-squared: 0.03231, Adjusted R-squared: 0.02469

F-statistic:    4.24 on 1 and 127 DF, p-value: 0.04152

# not run below

# summary(mod.2)

# summary(lm(p.vac ~ p.w + p.b, data = df))
```

The trends can be plotted with the data as in Figure 3.23.

```
p1 <- ggplot(df,aes(p.vac, p.w))+
        #stat_summary(fun.data=mean_cl_normal) +
geom_smooth(method='lm') + geom_point() +
xlab("Proportion of Vacant Properties") + ylab("Proporion White") +
labs(title="Regression of Vacant Properties against Proportion White")
p2 <- ggplot(df,aes(p.vac, p.b))+
#stat_summary(fun.data=mean_cl_normal) +
geom_smooth(method='lm') + geom_point() +
xlab("Proportion of Vacant Properties") + ylab("Proporion Black") +
labs(title="Regression of Vacant Properties against Proportion Black")
grid.newpage()
# set up the layout
pushViewport(viewport(layout=grid.layout(2,1)))
# plot using the print command
print(p1, vp=viewport(layout.pos.row = 1, height = 5))
print(p2, vp=viewport(layout.pos.row = 2, height = 5))
```

Figure 3.23 Plotting regression coefficient slopes





3.6.3 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.

First install the ggmosaic package:

```
# install the package
```

```
install.packages("ggmosaic", dep = T)
```

Then prepare the data using the melt function from the reshape2 package:

```
# create the dataset

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)

# create a crosstabulation

mat.tab <- xtabs(pops ~vac.10)

# melt the data

df <- melt(mat.tab)
```

Finally, create the mosaic plot, as in Figure 3.24, using the stat_mosaic function in the ggmosaic extension to the ggplot2 package.

```
# load the packages

library(ggmosaic)

# call ggplot and stat_mosaic

ggplot(data = df) +

stat_mosaic(aes(weight = value, x = product(Var2), fill=factor(vac.10)), na.rm=TRUE)
+

theme(axis.text.x=element_text(angle=−90, hjust= .1)) + labs(y='Proportion of Vacant
Properties', x = 'Ethnic group',

title="Mosaic Plot of Vacant Properties with ethnicity") + guides(fill=guide_legend(title =
"> 10 percent", reverse = TRUE))
```

It has the usual ggplot feel. It shows that the census blocks with vacancy levels higher than 10% are *not* evenly distributed among different ethnic groups: the tiles in the mosaic plot have areas proportional to the counts (in

this case the number of people affected).

However, the stat_mosaic plot does not quite have information about residuals and whether differences are significant, as does the mosaicplot function in the graphics package. This can be used using the code below to create Figure 3.25:

```
# standard mosaic plot

ttext = sprintf("Mosaic Plot of Vacant Properties     with ethnicity")

mosaicplot(t(mat.tab),xlab=’’,

ylab=’Vacant Properties > 10 percent’, main=ttext,shade=TRUE,las=3,cex=0.8)
```

Figure 3.24 An example of a ggmosaic mosaic plot

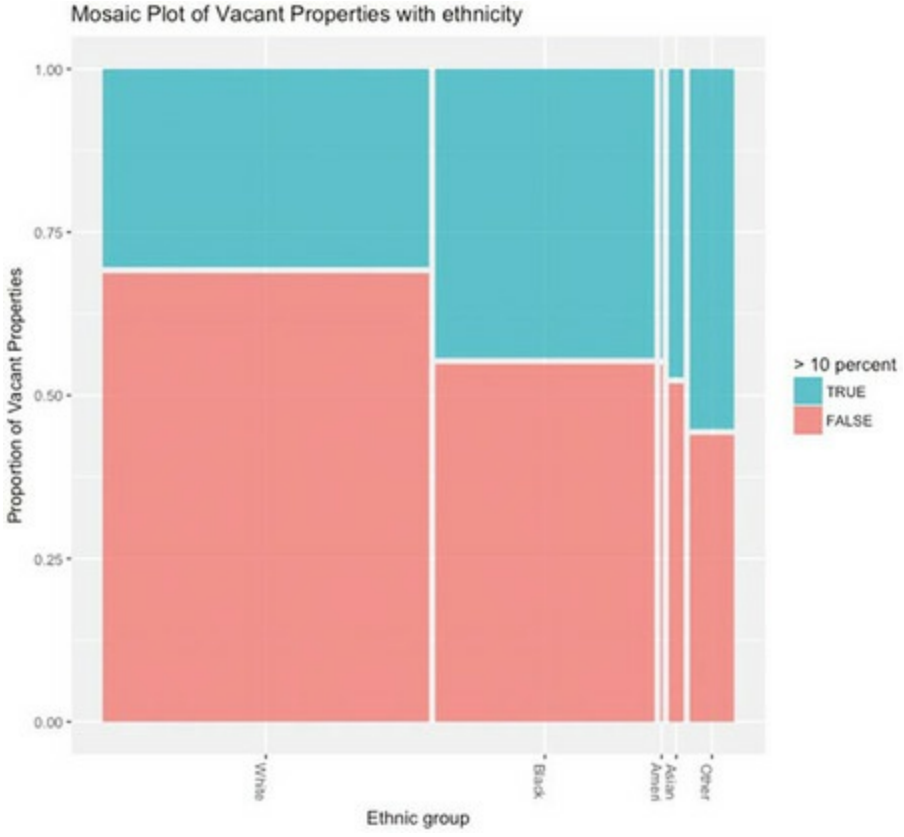


Figure 3.25 An example of a standard ‘graphics‘ mosaic plot with residuals

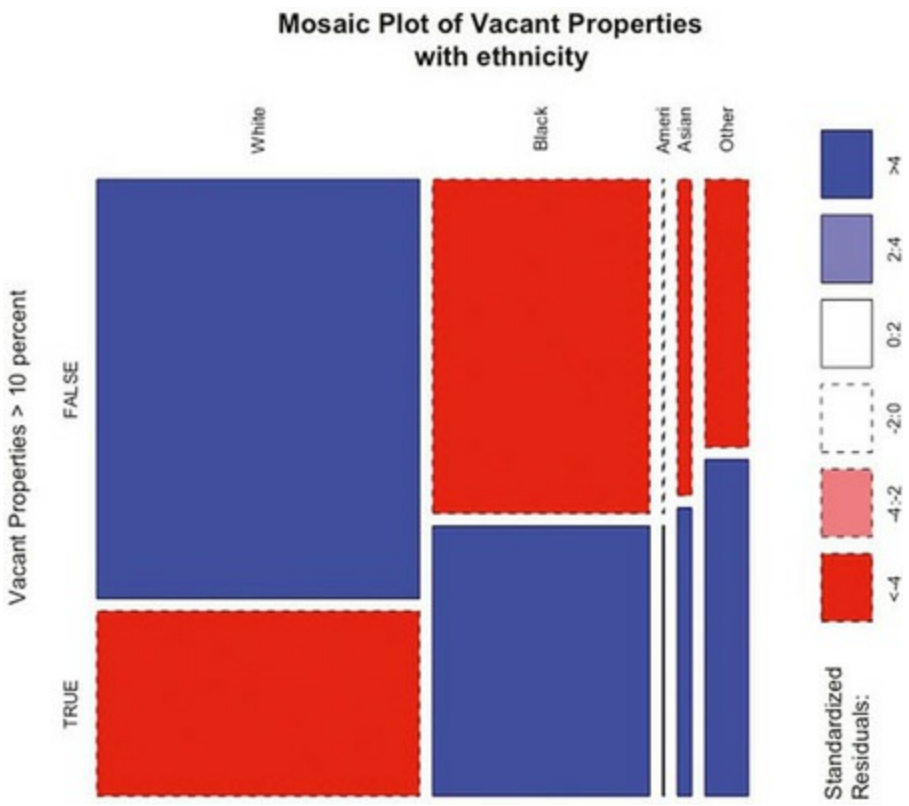


Figure 3.25 contains much more information. Its shading shows which groups are under- or overrepresented, when compared against a model of expected equality. The blue tiles show combinations of property vacancy and ethnicity that are higher than would be expected, with the tiles shaded deep blue corresponding 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 *more* than 10%, than would be expected in a model of equal distribution.

3.7 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 skills that you have acquired through this chapter (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 are 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 spTransform

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 10 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 scale bar and 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
breaks                    # to specify class breaks OR
style                     # in the tm_fill / tm_polygons help
# Tools
library(ggplot2)    # for the mapping tools
data(georgia)             # the Georgia data in the GISTools package
```

```
st_as_sf(georgia)    # to convert the data to sf format

tm_layout                    # takes many parameters, e.g. legend.position
```

Self-Test Question 2. Misrepresentation of continuous variables: using different breaks 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 aim in this exercise is to produce three maps that represent the same feature, and in so doing you will investigate the impact of different functions for grouping the continuous variable in the choropleth maps.

Write code that will create three maps, in the same window, of the numbers of houses in the New Haven census blocks. This is described by the HSE_UNITS variable. Apply different functions to divide the HSE_UNITS variable 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

p1 <- tm_shape(...) # assign the plots to a variable

pushViewport                                # from the grid package, used earlier...

viewport                        # ...to plot multiple tmaps

?quantileCuts             # quantiles, ranges std.dev...

?rangeCuts                        # ... from GISTools package

?sdCuts

breaks                          # to specify breaks in tm_polygon

tmap_mode('plot') # to specify a map view

# Tools

library(tmap)                          # for the mapping tools

library(grid)                          # for plotting the maps together
```

```
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 st_intersection in the st package to select roads within the extent of an st polygon object, and logical operators were used to select 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. The company has a model that says that they will sell one unit of their product for every 20 people in rural areas who are visited by one of their sales team, and they would like to know which counties have a rural population density of more than 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 some kind of *rural population density* score and map the counties in Georgia that have a score of greater than 20 rural people per square kilometre.

```
# Hints

library(GISTools)    # for the mapping tools

data(georgia)        # use georgia2 as it has a geographical projection

help("!")    # to examine logic operators

as.numeric    # use to coerce new attributes you create to numeric format

# e.g. georgia.sf$NewVariable <- as.numeric(1:159)

# Tools

st_area    # a function in the st package
```

Self-Test Question 4. Re-projections: transforming data using spTransform and st_transform

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 so easily done using models that use degrees and minutes. World geodetic systems such as WGS84 provide a standard reference system. For example, in the previous question you worked with the georgia2 dataset which is projected in metres, whereas georgia is projected in degrees in WGS84. And, when you plotted the Georgia subset with an OpenStreetMap backdrop, a transform operation was used to convert the data to the projection used in OpenStreetMap plotting. A range of different projections are described in formats for different packages and software on the Spatial Reference website (http://www.spatialreference.org). A typical re-projection would be something like:

```
# Using spTransform in sp

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

# Using st_transform in sf

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

You should note that the 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 which assigned a projection to the coordinates:

```
library(GISTools)

library(rgdal)

library(sf)

data(quakes)

coords.tmp <- cbind(quakes$long, quakes$lat)
```

```
# create the SpatialPointsDataFrame

quakes.sp <- SpatialPointsDataFrame(coords.tmp, data = data.frame(quakes),

proj4string = CRS("+proj=longlat "))
```

You can examine the projection properties of the SpatialPointsDataFrame and sf objects after the latter is created, by entering:

```
summary(quakes.spdf)

quakes_sf <- st_as_sf(quakes.sp)head(quakes.sf)
```

If the proj4string properties of sp and sf objects are empty, these can be populated if you know the spatial reference system and then the data can be transformed.

The objective of this exercise is to re-project the New Haven blocks and breach datasets from their original reference system to WGS84, using both the st_transform function in sf and the spTransform function in rgdal, and then to plot these transformed data on an OpenStreetMap backdrop. You may find it useful to use a transparency term in your colours.

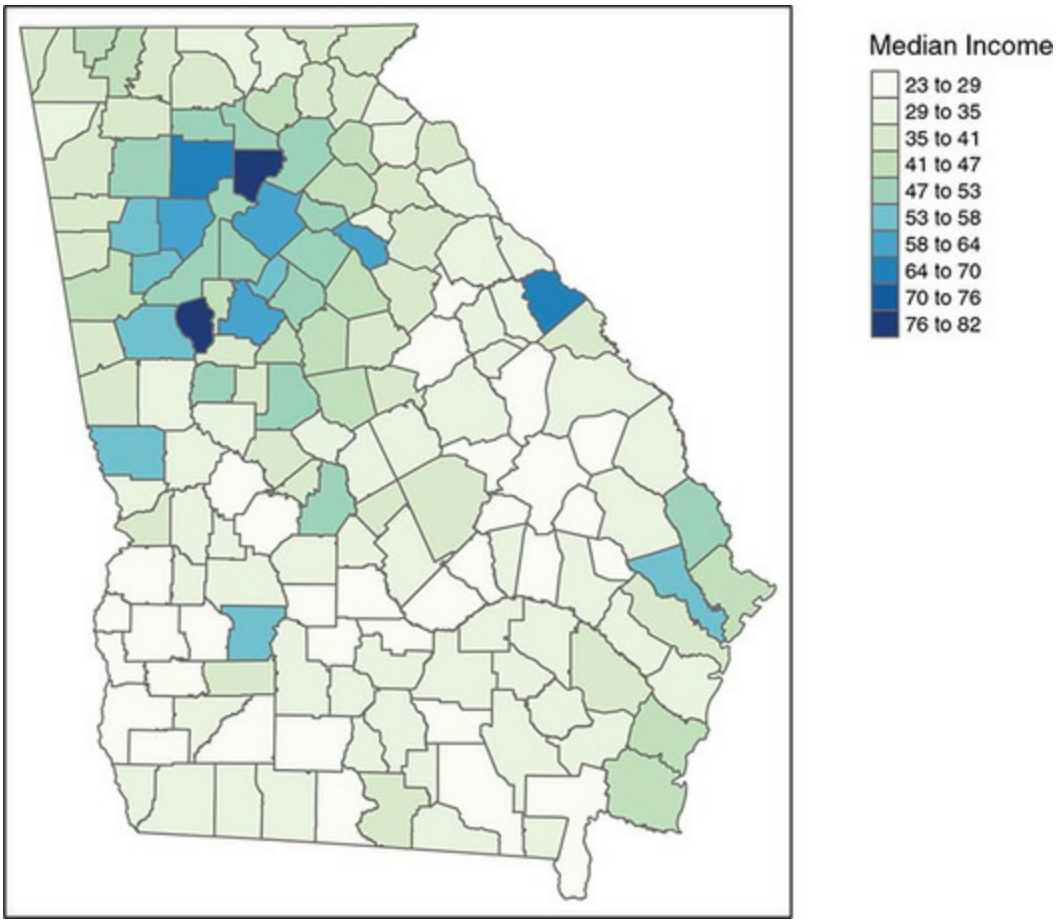
These datasets have a local projections system, using the State Plane Coordinate System for Connecticut, in US survey feet. You should transform the breaches of the peace and the census blocks data to latitude and longitude by assigning a projection using the CRS function in the sp package and st_crs function in the sf package. Then the spTransform and st_transform functions can be applied. Having transformed the datasets, you should map the locations of the breaches of peace and the census blocks with an OpenStreetMap backdrop. You could use the OpenStreetMap tools directly and/or the Leaflet embedded in the tmap tools when tmap_mode is set to ’view’.

3.8 Answers to Self-Test Questions

Q1: Plots and maps: working with map data. Your map should look something like Figure 3.26.

```
# load the data and the packages

library(GISTools)

library(sf)

library(tmap)

data(georgia)

# set the tmap plot type

tmap_mode('plot')

# convert to sf format

georgia_sf = st_as_sf(georgia)

# create the variable

georgia_sf$MedInc = georgia_sf$MedInc / 1000

# open the tiff file and give it a name

tiff("my_map.tiff")

# start the tmap commands

tm_shape(georgia_sf) + tm_polygons("MedInc", title = "Median Income", palette =
"GnBu",                                    style = "equal", n = 10) +

tm_layout(legend.title.size = 1, legend.format = list(digits = 0), legend.position = c(0.2,
"top")) +

tm_legend(legend.outside=TRUE)

# close the tiff file

dev.off()
```

Figure 3.26 The map produced by the code for Q1



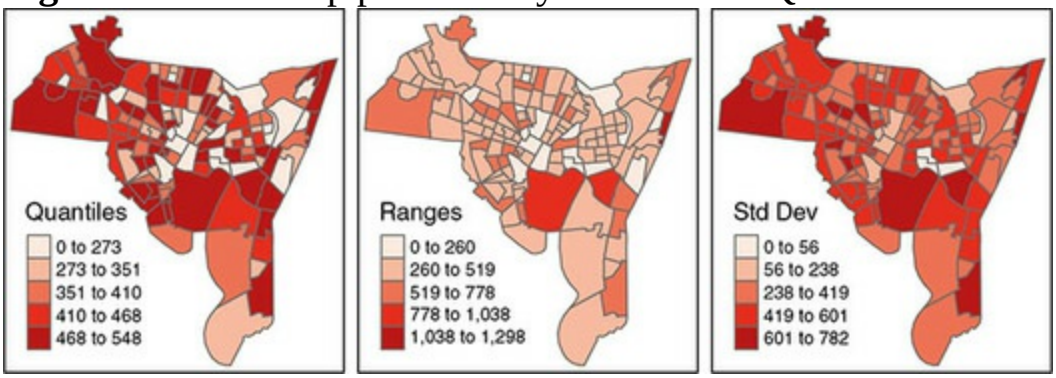
Q2: Misrepresentation of continuous variables – using different breaks for choropleth mapping. Your map should look something like Figure 3.27.

```
# load packages and data
library(tmap)
library(GISTools)
library(sf)
library(grid)
data(newhaven)
# convert data to sf format
blocks_sf = st_as_sf(blocks)
# 1. Initial Investigation
# You could start by having a look at the data
```

```
attach(data.frame(blocks_sf))

hist(HSE_UNITS, breaks = 20)

# You should notice that it has a normal distribution

# but with some large outliers

# Then examine different cut schemes

quantileCuts(HSE_UNITS, 6)

rangeCuts(HSE_UNITS, 6)

sdCuts(HSE_UNITS, 6)

# detach the data frame

detach(data.frame(blocks_sf))

# 2. Do the task

# a) mapping classes defined by quantiles

# define some breaks

br <- c(0, round(quantileCuts(blocks_sf$HSE_UNITS, 6),0))

# you could examine br

p1 <- tm_shape(blocks_sf) + tm_polygons("HSE_UNITS", title="Quantiles",

        palette="Reds", breaks=br)

# b) mapping classes defined by absolute ranges

# define some breaks

br <- c(0, round(rangeCuts(blocks$HSE_UNITS, 6),0))

# you could examine br

p2 <- tm_shape(blocks_sf) + tm_polygons("HSE_UNITS", title="Ranges",
palette="Reds", breaks=br)

# c) mapping classes defined by standard deviations

br <- c(0, round(sdCuts(blocks$HSE_UNITS, 6),0))

# you could examine br

p3 <- tm_shape(blocks_sf) + tm_polygons("HSE_UNITS", title="Std Dev",
```

```
palette="Reds", breaks=br)

# open a new plot page

grid.newpage()

# set up the layout

pushViewport(viewport(layout=grid.layout(1,3)))

# plot using the print command

print(p1, vp=viewport(layout.pos.col = 1, height = 5))

print(p2, vp=viewport(layout.pos.col = 2, height = 5))

print(p3, vp=viewport(layout.pos.col = 3, height = 5))
```

Figure 3.27 The map produced by the code for Q2

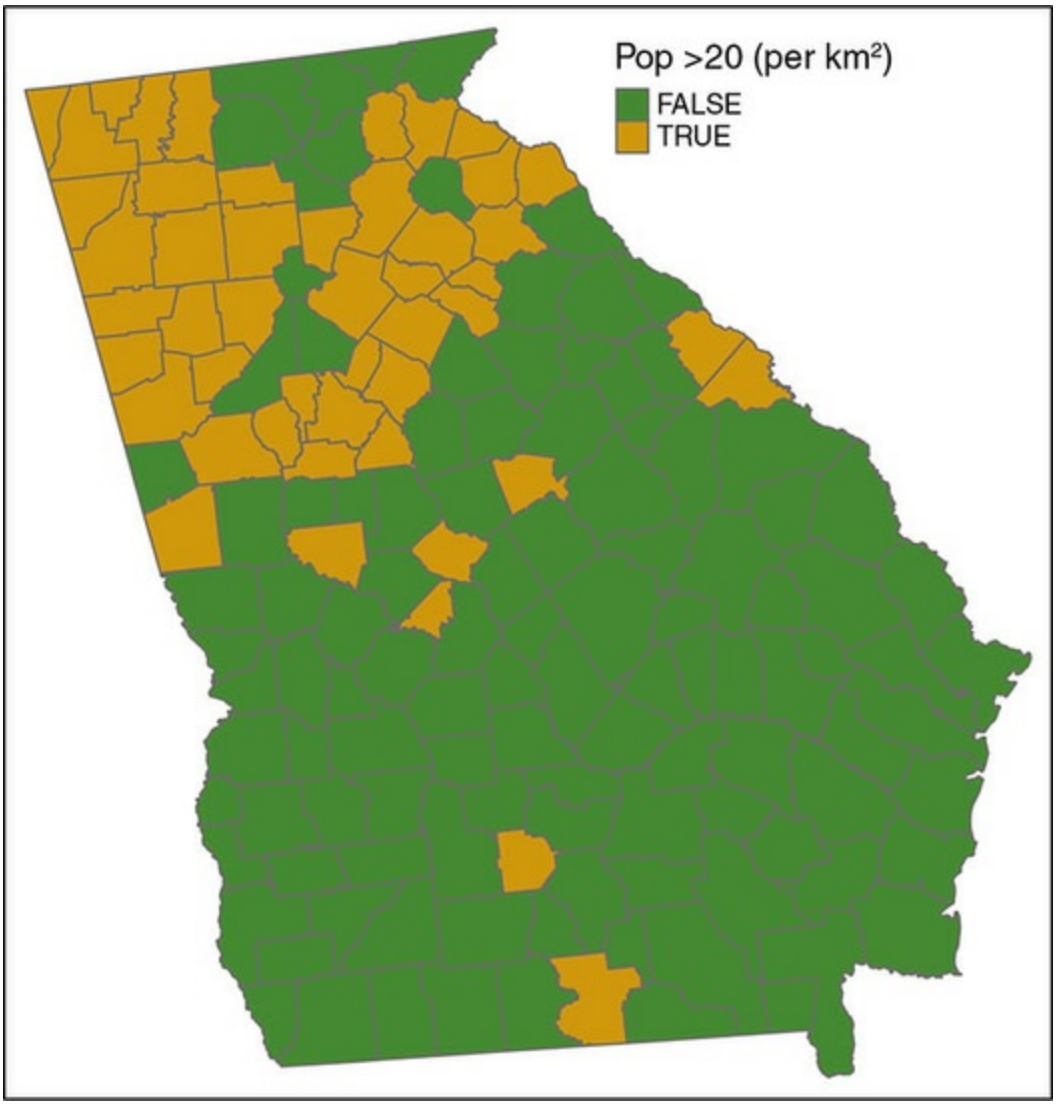


Q3: Selecting data: creating variables and subsetting data using logical statements. The code is below and your map should look something like Figure 3.28.

```
library(GISTools)

library(sf)

data(georgia)

# convert data to sf format

georgia_sf = st_as_sf(georgia2)

# calculate rural population

georgia_sf$rur.pop <- as.numeric(georgia_sf$PctRural
```

```
   * georgia_sf$TotPop90 / 100)
# calculate county areas in km^2
georgia_sf$areas <- as.numeric(st_area(georgia_sf)
/ (1000* 1000))
# calculate rural density
georgia_sf$rur.pop.den <- as.numeric(georgia_sf$rur.pop
/ georgia_sf$areas)
# select counties with density > 20
georgia_sf$rur.pop.den <- (georgia_sf$rur.pop.den > 20)
# map them
tm_shape(georgia_sf) + tm_polygons("rur.pop.den",
palette=c("chartreuse4","darkgoldenrod3"),
title=expression("Pop >20 (per " * km^2 * ")"), auto.palette.mapping = F)
```

Figure 3.28 The map produced by the code for Q3



Q4: Transforming data. Your map should look something like Figure 3.29 or Figure 3.30, depending on which way you did it! First you will need to transform the data:

```
library(GISTools)    # for the mapping tools
library(sf)    # for the mapping tools
library(rgdal)          # this has the spatial reference tools
library(tmap)
library(OpenStreetMap)
data(newhaven)
```

```
# Define a new projection
newProj <- CRS("+proj=longlat +ellps=WGS84")
# Transform blocks and breach
# 1. using spTransform
breach2 <- spTransform(breach, newProj)
blocks2 <- spTransform(blocks, newProj)
# 2. using st_transform
breach_sf <- st_as_sf(breach)
blocks_sf <- st_as_sf(blocks)
breach_sf <- st_transform(breach_sf, "+proj=longlat +ellps=WGS84")
blocks_sf <- st_transform(blocks_sf, "+proj=longlat +ellps=WGS84")
```

Then the transformed data can be mapped using Leaflet in the tmap package:

```
# set the mode
tmap_mode('view')
# plot the blocks
tm_shape(blocks_sf) + tm_borders() +
# and then plot the breaches
tm_shape(breach_sf) +
        tm_dots(shape=1, size=0.1, border.col = NULL, col = "red", alpha = 0.5)
```

It can also be mapped using the OpenStreetMap package. For this you need to extract the map tiles using the bounding box of the transformed data:

```
ul <- as.vector(cbind(bbox(blocks2)[2,2], bbox(blocks2)[1,1]))
lr <- as.vector(cbind(bbox(blocks2)[2,1], bbox(blocks2)[1,2]))
# download the map tile
MyMap <- openmap(ul,lr)
```

```
# now plot the layer and the backdrop
par(mar = c(0,0,0,0))
plot(MyMap, removeMargin=FALSE)
# notice how the data need to be transformed
# to the internal OpenStreetMap projection
plot(spTransform(blocks2, osm()), add = TRUE, lwd = 1)
plot(spTransform(breach2, osm()), add = T, pch = 19, col = "#DE2D2650")
```

Figure 3.29 The tmap map produced by the code for Q4

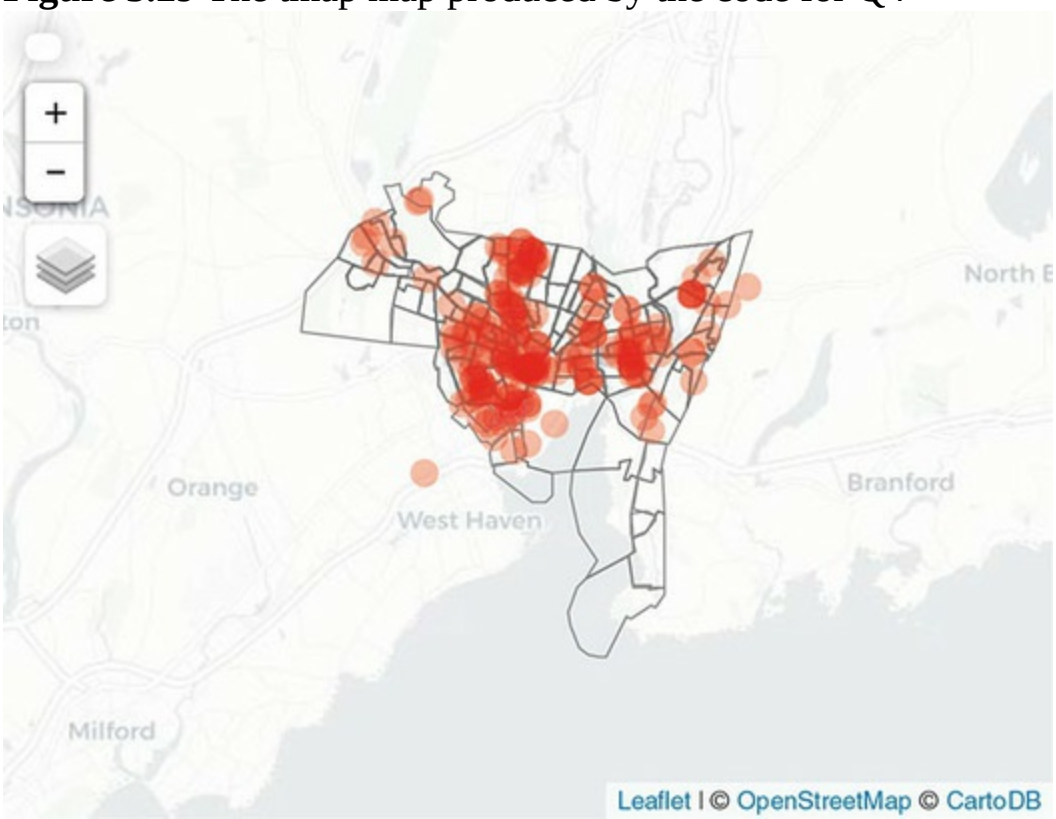
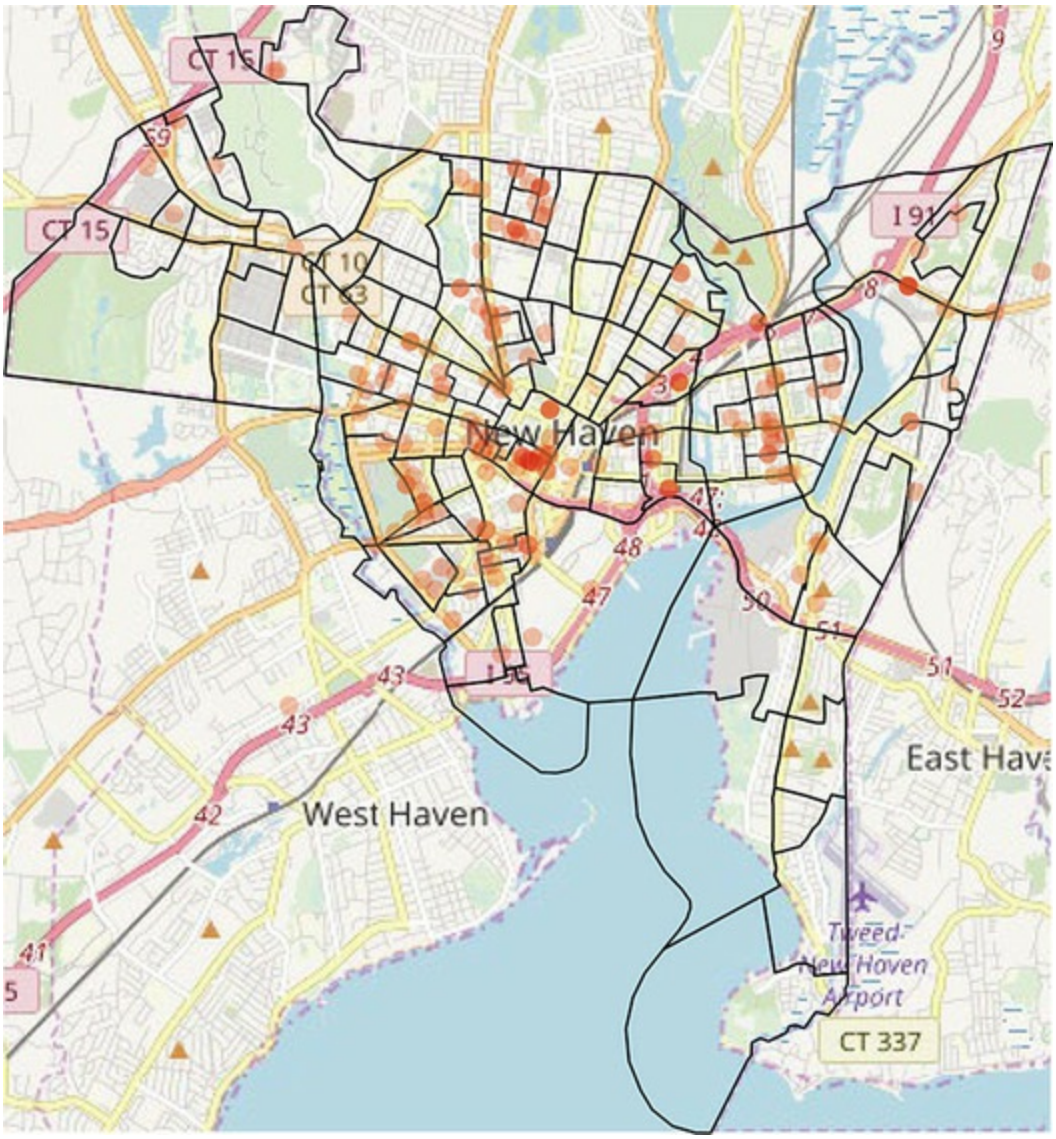


Figure 3.30 The OpenStreetMap map produced by the code for Q4



References

Anselin, L. (1995) Local indicators of spatial association – Lisa. Geographical Analysis, 27(2): 93–115.

Brunsdon, C. and Chen, H. (2014) GISTools: Some further GIS capabilities for R. R Package Version 0.7-4. http://cran.r-project.org/package=GISTools.

de Vries, A. and Meys, J. (2012) R for Dummies. Chichester: John Wiley &

Sons.

Hijmans, R.J. and van Etten, J. (2014) Raster: Geographic data analysis and modeling. R Package Version 2.6-7. http://cran.r-project.org/package=raster.

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

Ord, J.K. and Getis, A. (1995) Local spatial autocorrelation statistics: Distributional issues and an application. Geographical Analysis, 27(4): 286–306.

Pebesma, E., Bivand, R., Cook, I., Keitt, T., Sumner, M., Lovelace, R., Wickham, H., Ooms, J. and Racine, E. (2016) sf: Simple features for R. R Package Version 0.6-3. http://cran.r-project.org/package=sf.

Tennekes, M. (2015) tmap: Thematic maps. R Package Version 1. http://cran.r-project.org/package=tmap.

4 Scripting and Writing Functions in R

4.1 Overview

As you have been working through the code and exercises in this book you have applied a number of different tools and techniques for extracting, displaying and analysing data. In places you have used some quite advanced snippets of code. However, this has all been done in a step-by-step manner, with each line of code being run individually, and the occasional function has been applied individually to a specific dataset or attribute. Quite often in spatial analysis, we would like to do the same thing repeatedly, but adjusting some of the parameters on each iteration – for example, applying the same algorithm to different data, different attributes, or using different thresholds. The aim of this chapter is to introduce some basic programming principles and routines that will allow you to do many things repeatedly in a single block of code. This is the basics of writing computer programs. This chapter will:

- Describe how to combine commands into loops
- Describe how to control loops using if, else, repeat, etc.
- Describe logical operators to index and control
- Describe how to create functions, test them and to make them universal
- Explain how to automate short tasks in R
- Introduce the apply family of operations and how they can be used to apply functions to different data structures
- Introduce dplyr functions for data table manipulations and operations

4.2 Introduction

In spatial data analysis and mapping, we frequently want to apply the same set of commands over and over again, to cycle through data or lists of data and do things to data depending on whether some condition is met or not, and so on. These types of repeated actions are supported by *functions*, *loops* and *conditional statements*. A few simple examples serve to illustrate how R

programming combines these ideas through functions with conditional commands, loops and variables.

For example, consider the following variable tree.heights:

```
tree.heights <- c(4.3,7.1,6.3,5.2,3.2)
```

We may wish to print out the first element of this variable if it has a value less than 6: this is a *conditional command* as the operation (in this case to print something) is carried out conditionally (i.e. if the condition is met).

```
tree.heights

[1] 4.3 7.1 6.3 5.2 3.2

if (tree.heights[1] < 6) { cat('Tree is small\n') } else

{ cat('Tree is large\n')}

Tree is small
```

Alternatively, we may wish to examine all of the elements in the variable tree.heights and, depending on whether each individual value meets the condition, perform the same operation. We can carry out operations repeatedly using a *loop* structure as follows. Notice the construction of the for loop in the form:

```
for(variable in sequence) R expression
```

This is illustrated in the code below:

```
for (i in 1:3) {

if (tree.heights[i] < 6) { cat('Tree',i,' is small\n') }

else { cat('Tree',i,' is large\n')} }

Tree 1 is small

Tree 2 is large
```

```
Tree 3 is large
```

A third situation is where we wish to perform the same set of operations, group of conditional or looped commands over and over again, perhaps to different data. We can do this by grouping code and defining our own *functions*.

```
assess.tree.height <- function(tree.list, thresh)

{ for (i in 1:length(tree.list))

{ if(tree.list[i] < thresh) {cat('Tree',i, ' is small\n')}

else { cat('Tree',i,' is large\n')}

}

}

assess.tree.height(tree.heights, 6)

Tree 1 is small

Tree 2 is large

Tree 3 is large

Tree 4 is small

Tree 5 is small

tree.heights2 <- c(8,4.5,6.7,2,4)

assess.tree.height(tree.heights2, 4.5) Tree 1 is large

Tree 2 is large

Tree 3 is large

Tree 4 is small

Tree 5 is small
```

Notice how the code in the function assess.tree.height above modifies the original loop: rather than for(i in 1:3) it now uses the length of the variable 1:length(tree.list) to determine how many times to loop through the data. Also a variable thresh was used for whatever threshold the user wishes to

specify.

The sections in this chapter develop more detailed ideas around functions, loops and conditional statements and the testing and debugging of functions in order to support automated analyses in R.

4.3 Building Blocks for Programs

In the examples above, a number of programming concepts were introduced. Before we start to develop these more formally into functions it is important to explain these ingredients in a bit more detail.

4.3.1 Conditional Statements

Conditional statements test to see whether some *condition* is TRUE or FALSE, and if the answer is TRUE some specific actions are undertaken. Conditional statements are composed of if and else.

The if statement is followed by a *condition*, an expression that is evaluated, and then a *consequent* to be executed if the condition is TRUE. The format of an if statement is:

```
if – condition – consequent
```

Actually this could be read as ‘if the condition is true then the consequent is…’. The components of a conditional statement are:

- the condition, an R expression that is either TRUE or FALSE
- the consequent, any valid R statement which is only executed if the condition is TRUE

For example, consider the simple case below where the value of x is changed and the same condition is applied. The results are different because of the different values assigned to x: in the first case a statement is printed to the console, in the second it is not.

```
x <- −7

if (x < 0) cat("x is negative")

x is negative

x <- 8

if (x < 0) cat("x is negative")
```

Frequently if statements also have an *alternative* consequent that is executed when the condition is FALSE. Thus the format of the conditional statement is expanded to:

```
if – condition-– consequent– else – alternative
```

Again, this could be read as ‘if the condition is true then do the consequent; or, if the condition is not true then do the alternative’. The components of a conditional statement that includes an alternative are:

- the condition, an R expression that is either TRUE or FALSE
- the consequent and alternative, which can be any valid R statements
- the consequent is executed if the condition is TRUE
- the alternative is executed if the condition is FALSE

The example is expanded below to accommodate the alternative:

```
x <- −7

if (x < 0) cat("x is negative") else cat("x is positive")

x is negative

x <- 8

if (x < 0) cat("x is negative") else cat("x is positive")

x is positive
```

The condition statement is composed of one or more logical operators and in R these are defined in Table 4.1. In addition, R contains a number of *logical*

functions which can also be used to evaluate conditions. A sample of these is listed in Table 4.2 but many others exist.

Table 4.1

Table 4.1 Logical operators

Logical operator	Description
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
!	Not (goes in front of other expressions)
&	And (combines expressions)
|	Or (combines expressions)

Table 4.2

Table 4.2 Logical functions

Logical function	Description
`any(x)`	TRUE if any in a vector of conditions x is true
`all(x)`	TRUE if all of a vector of conditions x is true
`is.numeric(x)`	TRUE if x contains a numeric value
`is.logical(x)`	TRUE if x contains a true or false value
`is.character(x)`	TRUE if x contains a character value

There are quite a few more is-type functions (i.e. *logical* evaluation functions) that return TRUE or FALSE statements that can be used to develop conditional tests. To explore these enter:

```
??is.
```

The examples below illustrate how the logical tests all and any may be incorporated into conditional statements:

```
x <- c(1,3,6,8,9,5)

if (all(x > 0)) cat("All numbers are positive")

All numbers are positive
```

```
x <- c(1,3,6,−8,9,5)

if (any(x > 0)) cat("Some numbers are positive")

Some numbers are positive

any(x==0)

[1] FALSE
```

4.3.2 Code Blocks

Frequently we wish to execute a group of consequent statements together if, for example, some condition is TRUE. Groups of statements are called *code blocks* and in R are contained by { and }. The examples below show how code blocks can be used if a condition is TRUE to execute consequent statements and can be expanded to execute alternative statements if the condition is FALSE.

```
x <- c(1,3,6,8,9,5)

if (all(x > 0)) {

cat("All numbers are positive\n")

total <- sum(x)

cat("Their sum is ",total) }

All numbers are positive

Their sum is 32
```

The curly brackets are used to group the consequent statements: that is, they contain all of the actions to be performed if the condition is met (i.e. is TRUE) and all of the alternative actions if the condition is not met (i.e. is FALSE):

```
if condition { consequents } else { alternatives }
```

These are illustrated in the code below:

```
x <- c(1,3,6,8,9,−5)
if (all(x > 0)) {
cat("All numbers are positive\n")
total <- sum(x)
cat("Their sum is ",total) } else {
cat("Not all numbers are positive\n")
cat("This is probably an error as numbers are rainfall levels") }
Not all numbers are positive
This is probably an error as numbers are rainfall levels
```

4.3.3 Functions

The introductory section above included a function called assess.tree.height. The format of a function is:

```
function name <- function(argument list) { R expression }
```

The R expression is usually a code block and in R the code is contained by curly brackets or braces: { and }. Wrapping the code into a function allows it to be used without having to retype the code each time you wish to use it. Instead, once the function has been defined and compiled, it can be called repeatedly and with different arguments or parameters. Notice in the function below that there are a number of sets of containing brackets { } that are variously related to the condition, the consequent and the alternative.

```
mean.rainfall <- function(rf)
{ if (all(rf> 0))                    #open Function
{ mean.value <- mean(rf)             #open Consequent
      cat("The mean is ",mean.value)
} else                               #close Consequent
```

```
{ cat("Warning: Not all values are positive\n")    #open Alternative

}                                    #close Alternative

}                                       #close Function

mean.rainfall( c(8.5,9.3,6.5,9.3,9.4))

The mean is 8.6
```

More commonly, functions are defined that do something to the input specified in the *argument list* and return the result, either to a variable or to the console window, rather than just printing something out. This is done using return() within the function. Its format is return(R expression). Essentially what this does if it is used in a function is to make R expression the value of the function. In the following code the mean.rainfall2 function now returns the mean of the data passed to it, and this is assigned to another variable:

```
mean.rainfall2 <- function(rf) {

if ( all(rf > 0)) {

return( mean(rf))} else {

return(NA)}

}

mr <- mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))

mr

[1] 8.6
```

I

Notice that the code blocks used in the functions contained within the curly brackets or braces { and } are indented. There are a number of commonly accepted protocols for doing this but no unique one. The aim is to make the code and the nesting of sub-clauses indicated by { and } clear. In the code for mean.rainfall above, { is used before the first line of the code block, whereas for mean.rainfall.2 the { is positioned immediately after the function declaration.

It is possible to declare variables inside functions, and you should note that

these are distinct from external variables with the same name. Consider the internal variable rf in the mean.rainfall2 function above. Because this is a variable that is *internal* to the function, it only exists *within* the function and will not alter any *external* variable of the same name. This is illustrated in the code below.

```
rf <- "Tuesday"

mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))

[1] 8.6

rf

[1] "Tuesday"
```

4.3.4 Loops and Repetition

Very often, we would like to run a code block a certain number of times, for example for each record in a data frame or a spatial data frame. This is done using for loops. The format of a loop is:

```
for( 'loop variable' in 'list of values' ) do R expression
```

Again, typically code blocks are used, as in the following example of a for loop:

```
for (i in 1 :5) {

i.cubed <- i * i * i

cat("The cube of",i,"is ",i.cubed,"\n")}

The cube of 1 is 1

The cube of 2 is 8

The cube of 3 is 27

The cube of 4 is 64

The cube of 5 is 125
```

When working with a data frame and other tabular-like data structures, it is common to want to perform a series of R expressions on each row, on each column or on each data element. In a for loop the list of values can be a simple sequence of 1 to *n* (1:n), where *n* is related to the number of rows or columns in a dataset or the length of the input variable as in the assess.tree.height function above.

However, there are many other situations when a different list of values is required. The function seq is a very useful helper function that generates number sequences. It has the following formats:

```
seq(from, to, by = step value)
```

or

```
seq(from, to, length = sequence length)
```

In the example below, it is used to generate a sequence of 0 to 1 in steps of 0.25:

```
for (val in seq(0,1,by=0.25)) {
        val.squared <- val * val
cat("The square of",val,"is ",val.squared,"\n")}
The square of 0 is 0
The square of 0.25 is 0.0625
The square of 0.5 is 0.25
The square of 0.75 is 0.5625
The square of 1 is 1
```

Conditional loops are very useful when you wish to run a code block until a certain condition is met. In R these can be specified using the repeat and break functions. Here is an example:

```
i <- 1; n <- 654
repeat{
   i.squared <- i * i
if (i.squared > n) break
i <- i + 1}
cat("The first square number exceeding",n, "is ",i.squared,"\n")
The first square number exceeding 654 is 676
```

Finally, it is possible to include loops in functions as in the following example with a conditional loop:

```
first.bigger.square <- function(n) {
i <- 1
repeat{
i.squared <- i * i
if (i.squared > n) break
i <- i + 1 }
return(i.squared)}
first.bigger.square(76987)
[1] 77284
```

4.3.5 Debugging

As you develop your code and compile it into functions, especially initially, you will probably encounter a few teething problems: hardly any function of reasonable size works first time! There are two general kinds of problem:

- The function crashes (i.e. it throws up an error)
- The function does not crash, but returns the wrong answer

Usually the second kind of error is the worst. *Debugging* is the process of finding the problems in the function. A typical approach to debugging is to 'step' through the function line by line and in so doing find out where a crash occurs, if one does. You should then check the values of variables to see if they have the values they are supposed to. R has tools to help with this.

To debug a function:

- Enter debug(function name)
- Then call the function

For example, enter:

```
debug(mean.rainfall2)
```

Then just use the function you are trying to debug and R goes into 'debug mode':

```
mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))

[1] 8.6
```

You will notice that the prompt becomes Browse[2]> and the line of the function about to be executed is listed. You should note a number of features associated with debug:

- Entering a return executes it, and debug goes to next line
- Typing in a variable lists the value of that variable
- R can 'see' variables that are specific to the function
- Typing in any other command executes that command

When you enter c the return runs to the end of a loop/function/block. Typing in Q exits the function. To return to normal enter undebug(function name) and note that if there are no bugs, entering c has the same effect as undebug.

A final comment is that learning to write functions and programming is a bit like learning to drive: you may pass the test, but you will become a good driver by spending time behind the wheel. Similarly, the best way to learn to

write functions is to practise, and the more you practise the better you will get at programming. You should try to set yourself various function writing tasks and examine the functions that are introduced throughout this book. Most of the commands that you use in R are functions that can themselves be examined: entering them without any brackets afterwards will reveal the blocks of code they use. Have a look at the ifelse function by entering at the R prompt:

ifelse

This allows you to examine the code blocks, the control, etc., in existing functions.

4.4 Writing Functions

4.4.1 Introduction

In this section you will gain some initial experience in writing functions that can be used in R, using a number of coding illustrations. You should enter the code blocks for these, compile them and then run them with some data to build up your experience. Unless you already have experience in writing code, this will be your first experience of programming. This section contains a series of specific tasks for you to complete in the form of self-test questions. The answers to the questions are provided in the final section of the chapter.

In the preceding section, the basic idea of writing functions was described. You can write functions directly by entering them at the R command line:

```
cube.root <- function(x) {
    result <- x ^ (1/3)
    return(result)}
cube.root(27)
[1] 3
```

Note that ^ means ‘raise to the power’, and recall that a number to the power of one-third is its cube root. The cube root of 27 is 3, since 27 = 3 × 3 × 3, hence the answer printed out for cube.root(27). However, entering functions from the command line is not always very convenient:

- If you make a typing error in an early line of the definition, it is not possible to go back and correct it
- You would have to type in the definition every time you used R

A more sensible approach is to type the function definition into a text file. If you write this definition into a file – calling it, say, functions.R – then you can load this file when you run R, without having to type in the whole definition. Assuming you have set R to work in the directory where you have saved this file, just enter:

source("functions.R")

This has the same effect of entering the entire function at the command line. In fact any R commands in a file (not just function definitions) will be executed when the source function is used. Also, because the function definition is edited in a file, it is always possible to return to any typing errors and correct them – and if a function contains an error, it is easy to correct this and just redefine the function by re-entering the command above. Using an editor for writing and saving R code was introduced in previous chapters.

Open a new R script or editing window. In it, enter in the code for the program:

```
cube.root <- function(x) {

result <- x ^ (1/3)

return(result)}
```

Then use **Save As** to save the file as functions.R in the directory you are working in. In R you can now use source as described:

```
source(’functions.R’)
```

```
cube.root(343)

cube.root(99)
```

Note that you can type in several function definitions in the same file. For example, underneath the code for the cube.root function, you should define a function to compute the area of a circle. Enter:

```
circle.area <- function(r) {

result <- pi * r ^ 2

return(result)}
```

If you save the file and enter source(’functions.R’) again then the function circle.area will be defined as well as cube.root. Enter:

```
source(’functions.R’)

cube.root(343)

circle.area(10)
```

4.4.2 Data Checking

One issue when writing functions is making sure that the data that have been given to the function are the right kind. For example, what happens when you try to compute the cube root of a negative number?

```
cube.root(−343)

[1] NaN
```

That probably was not the answer you wanted. NaN stands for ‘not a number’, and is the value returned when a mathematical expression is numerically indeterminate. In this case, this is actually due to a shortcoming with the ^ operator in R, which only works for positive base values. In fact −7 is a perfectly valid cube root of −343, since (−7) × (−7) × (−7) = −343. In fact we can state a conditional rule:

- If $x \geq 0$: calculate the cube root of x normally
- Otherwise: use cube.root(-x)

That is, for cube roots of negative numbers, work out the cube root of the positive number, then change it to negative. This can be dealt with in an R function by using an if statement:

```
cube.root <- function(x) {
    if (x >= 0) {
result <- x ^ (1/3) } else {
result <- -(-x) ^ (1/3) }
return(result)}
```

Now you should go back to the text editor and modify the code in functions.R to reflect this. You can do this by modifying the original cube.root function. You can now save this edited file, and use source to reload the updated function definition. The function should work with both positive and negative values.

```
cube.root(3)
[1] 1.44225
cube.root(−3)
[1] −1.44225
```

Next, try debugging the function – since it is working properly, you will not (hopefully!) find any errors, but this will demonstrate the debug facility. Enter:

```
debug(cube.root)
```

at the R command line (not in the file editor!). This tells R that you want to run cube.root in debug mode. Next, enter:

```
cube.root(−50)
```

at the R command line and see how repeatedly pressing the return key steps you through the function. Note particularly what happens at the if statement.

At any stage in the process you can type an R expression to check its value. When you get to the if statement enter:

```
x > 0
```

at the command line and press Return to see whether it is true or false. Checking the value of expressions at various points when stepping through the code is a good way of identifying potential bugs or glitches in your code. Try running through the code for a few other cube root calculations, by replacing −50 above with different numbers, to get used to using the debugging facility. When you are finished, enter:

```
undebug(cube.root)
```

at the R command line. This tells R that you are ready to return cube.root to running in normal mode. For further details about the debugger, at the command line enter:

```
help(debug)
```

4.4.3 More Data Checking

In the last section, you saw how it was possible to check for negative values in the cube.root function. However, other things can go wrong. For example, try entering:

```
cube.root('Leeds')
```

This will cause an error to occur and to be printed out by R. This is not surprising because cube roots only make sense for numbers, not character

variables. However, it might be helpful if the cube root function could spot this and print a warning explaining the problem, rather than just crashing with a fairly obscure error message such as the one above, as it does at the moment. Again, this can be dealt with using an if statement. The strategy to handle this is:

- If *x* is numerical: compute its cube root
- If *x* is not numerical: print a warning message explaining the problem

Checking whether a variable is numerical can be done using the is.numeric function:

```
is.numeric(77)

is.numeric("Lex")

is.numeric("77")

v <- "Two Sevens Clash"

is.numeric(v)
```

The function could be rewritten to make use of is.numeric in the following way:

```
cube.root <- function(x) {

if (is.numeric(x)) {

if (x >= 0) { result <- x^(1/3) }

        else { result <- -(-x)^(1/3) }

        return(result) }

else {

        cat("WARNING: Input must be numerical, not character\n")

return(NA)}

}
```

Note that here there is an if statement inside another if statement – this is an

example of a *nested* code block. Note also that when no proper result is defined, it is possible to return the value NA instead of a number (NA stands for 'not available'). Finally, recall that the \n in the cat statement tells R to add a carriage return (new line) when printing out the warning. Try updating your cube root function in the editor with this latest definition, and then try using it (in particular with character variables) and stepping through it using debug.

An alternative way of dealing with cube roots of negative numbers is to use the R functions sign and abs. The function sign(x) returns a value of 1 if x is positive, −1 if it is negative, and 0 if it is zero. The function abs(x) returns the absolute value of x without the sign, so for example abs(−7) is 7, and abs(5) is 5. This means that you can specify the core statement in the cube root function without using an if statement to test for negative values, as:

```
result <- sign(x)*abs(x)^(1/3)
```

This will work for both positive and negative values of x.

> **Self-Test Question 1.** Define a new function cube.root.2 that uses this way of computing cube roots – and also include a test to make sure x is a numerical variable, and print out a warning message if it is not.

4.4.4 Loops Revisited

In this section you will revisit the idea of looping in function definitions. There are two main kinds of loops in R: *deterministic* and *conditional* loops. The former are executed a fixed number of times, specified at the beginning of the loop. The latter are executed until a specific condition is met.

4.4.4.1 Conditional Loops

A very old example of a conditional loop is *Euclid's algorithm*. This is a method for finding the *greatest common divisor* (GCD) of a pair of numbers. The GCD of a pair of numbers is the largest number that divides exactly (i.e. with remainder zero) into each number in the pair. The algorithm is set out

below:

1. Take a pair of numbers *a* and *b* – let the *dividend* be max(*a*, *b*), and the *divisor* be min(*a*, *b*).
2. Let the *remainder* be the arithmetic remainder when the dividend is divided by the divisor.
3. Replace the dividend with the divisor.
4. Replace the divisor with the remainder.
5. If the remainder is not equal to zero, repeat from step 2 to here.
6. Once the remainder is zero, the GCD is the dividend.

Without considering in depth the reasons *why* this algorithm works, it should be clear that it makes use of a conditional loop. The test to see whether further looping is required in step 5 above. It should also be clear that the divisor, dividend and remainder are all variables. Given these observations, we can turn Euclid's algorithm into an R function:

```
gcd <- function(a,b)
{
    divisor <- min(a,b)
    dividend <- max(a,b)
    repeat
{ remainder <- dividend %% divisor
        dividend <- divisor
divisor <- remainder
if (remainder == 0) break
}
return(dividend)
}
```

The one unfamiliar thing here is the %% symbol. This is just the remainder operator – the value of x %% y is the remainder when x is divided by y.

Using the editor, create a definition of this function, and read it into R. You can put the definition into functions.R. Once the function is defined, it may be tested:

```
gcd(6,15)

gcd(25,75)

gcd(31,33)
```

Self-Test Question 2. Try to match up the lines in the function definition with the lines in the description of Euclid's algorithm. You may also find it useful to step through an example of gcd in debug mode.

4.4.4.2 Deterministic Loops

As described in earlier sections, the form of a deterministic loop is:

```
for (<VAR > in <Item1 > :<Item2 >)

{

... code in loop ...

        }
```

where <VAR> refers to the looping variable. It is common practice to refer to <VAR> in the code in the loop. <Item1> and <Item2> refer to the range of values over which <VAR> loops. For example, a function to print the cube roots of numbers from 1 to n takes the form:

```
cube.root.table <- function(n)

{

for (x in 1 :n)

{

cat("The cube root of ",x," is", cube.root(x),"\n")
```

```
}

}
```

Self-Test Question 3. Write a function to compute and print out GCD(x,60) for x in the range 1 to n. When this is done, write another function to compute and print out GCD(x,y) for x in the range 1 to n1 and y in the range 1 to n2. In this exercise you will need to nest one deterministic loop inside another one.

Self-Test Question 4. Modify the cube.root.table function so that the loop variable runs from 0.5 in steps of 0.5 to n. The key to this is provided in the descriptions of loops in the sections above.

4.4.5 Further Activity

You will notice that in the previous example the output is rather messy, with the cube roots printing to several decimal places – it might look neater if you could print to fixed number of decimal places. In the function cube.root.table replace the cat line with:

```
cat(sprintf("The cube root of %4.0f is %8.4f \n",x, cube.root(x)))
```

Then enter help(sprintf) and try to work out what is happening in the code above.

Self-Test Question 5. Create a for loop that cycles through each county / row in the data frame of the georgia2 dataset in the GISTools package and creates a list of the adjacent counties. The code to do this for a single county, Appling, is as follows:

```
library(GISTools)

library(sf)

data(georgia)

# create an empty list for the results

adj.list <- list()
```

```
# convert georgia to sf
georgia_sf <- st_as_sf(georgia2)
# extract a single county
county.i <- georgia_sf[1,]
# determine the adjacent counties
# the [−1] removes Appling form its own list
adj.i <- unlist(st_intersects(county.i, georgia_sf))[−1]
# extract their names
adj.names.i <- georgia2$Name[adj.i]
# add to the list
adj.list[[1]] <- adj.i
# name the list elements
names(adj.list[[1]]) <- adj.names.i
```

This creates a list with a single element, with the names of the counties adjacent to Appling and an index or reference to their location within the georgia2 dataset.

```
adj.list
[[1]]
      Bacon Jeff Davis Pierce Tattnall Toombs
          3              80     113     132    138
Wayne
151
```

Note that once lists are defined as in adj.list in the code above, elements can be added:

```
# in sequence
adj.list[[2]] <- sample(1:100, 3)
```

```
# or not!

i = 4

adj.list[[i]] <- c("Chris", "and", "Lex")

# have a look!

adj.list
```

Self-Test Question 6. Take the loop you created in Question 5 and create a function that returns a list of the indices of adjacent polygons for each polygon in *any* polygon dataset in sf or sp format. *Hint*: you will need to do any conversions to sf and define the list to be returned inside the function.

4.5 Spatial Data Structures

This section unpicks some of the detail of spatial data structures in R as a precursor to manipulating and interrogating spatial data with functions. It examines their coordinate encoding and briefly revisits their attribute/variable structures.

To begin with, you will load the GISTools package and the georgia data. However, before doing this and running the code below, you need to check that you are in the correct working directory. You should already be in the habit of doing this at the start of every R session. Also, if this is not a fresh R session then you should clear the workspace of any variables and functions you have created. This can be done by entering:

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

Then load the GISTools package and the Georgia datasets:

```
library(GISTools)

data(georgia)
```

One of the variables is called georgia.polys. There are two ways to confirm

this. One way is to type ls() into R. This function tells R to list out all currently defined variables:

```
ls()
```

The other way of checking that georgia.polys now exists is just to type it into R and see it printed out.

```
georgia.polys
```

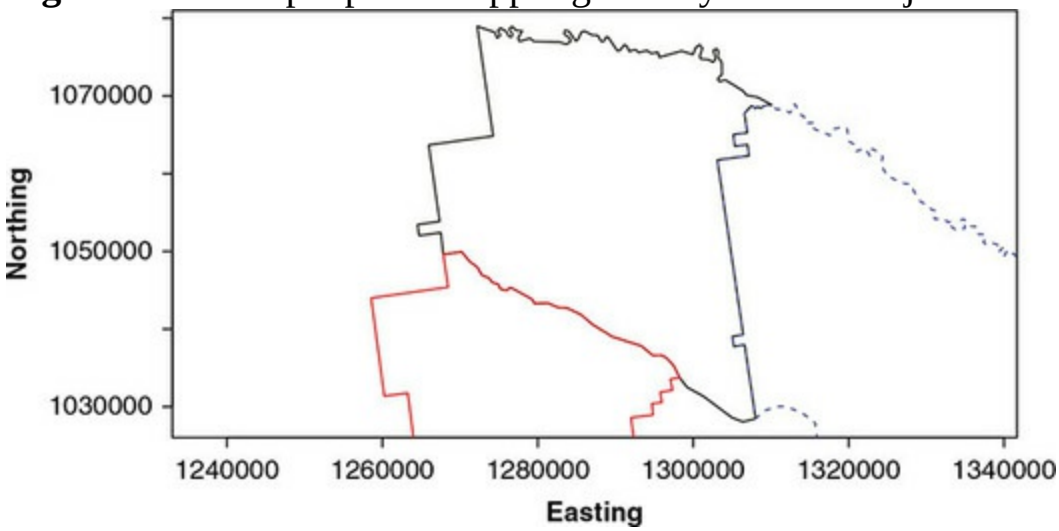
What is actually printed out has been excluded here, as it would go on for pages and pages. However, the content of the variable will now be explained. georgia.polys is a variable of type list, with 159 items in the list. Each item is a matrix of *k* rows and 2 columns. The two columns correspond to *x* and *y* coordinates describing a polygon made from *k* points. Each polygon corresponds to one of the 159 counties that make up the state of Georgia in the USA. To check this quickly, enter:

```
class(georgia.polys)
[1] "list"
head(georgia.polys[[1]])
```

```
         [,1]     [,2]
[1,]  1292287  1075896
[2,]  1292654  1075919
[3,]  1292949  1075590
[4,]  1294045  1075841
[5,]  1294603  1075472
[6,]  1295467  1075621
```

Each of the list elements, containing the bounding coordinates of each of the counties in Georgia, can be plotted. Enter the code below to produce Figure 4.1.

Figure 4.1 A simple plot of Appling County and two adjacent counties




```
# plot Appling
plot(georgia.polys[[1]],asp=1,type='l',        xlab = "Easting", ylab = "Northing")
# plot adjacent county outlines
points(georgia.polys[[3]],asp=1,type='l', col = "red")
points(georgia.polys[[151]],asp=1,type='l', col = "blue", lty = 2)
```

Notice the use of the plot and points functions as were introduced in Chapter 2.

Figure 4.1 will not win any prizes for cartography – but it should be recognisable as Appling County, as featured in earlier chapters. However, it highlights that spatial data objects in R have coordinates whether defined in the sp and sf packages. The code below extracts the coordinates for the first polygon in the georgia2 dataset, a SpatialPolygonsDataFrame object that has the same coordinates as georgia.polys. These are the same as the above.

```
head(georgia2@polygons[[1]]@Polygons[[1]]@coords)
head(georgia2@data[, 13:14])
```

If georgia2 is converted to sf format the coordinates are also evident:

```
g <- st_as_sf(georgia2)

head(g[,13:14])
```

So we can see that both sp and sf objects explicitly hold the spatial attributes and the thematic and variable attributes of spatial objects.

4.6 apply Functions

The final sections of this chapter describe a number of different functions that can make programming easier by offering a number of different ways of interrogating, manipulating and summarising spatial data, either by their variable attributes or by their spatial properties. This section examines the apply family of functions that come with the base installation of R.

Like other programming languages, R includes a group of functions which are generally termed apply functions. These can be used to apply the same set of operations over each element in a data object (row, column, list element). They take some input data and a function as inputs. Here we will briefly explore three of the most commonly used apply functions: apply, lapply and mapply.

Load the newhaven data and examine the blocks object. It contains a number of variables describing the percentage of different ethnicities living in each census block:

```
library(GISTools)

data(newhaven)

## the @data route

head(blocks@data[, 14:17])

## the data frame route

head(data.frame(blocks[, 14:17]))
```

A basic illustration of apply that returns the percentage value of the largest group in each block is as follows:

```
apply(blocks@data[,14:17], 1, max)
```

Have a look at the help for apply. The code above passes the 14th to 17th columns of the blocks data frame to apply, the 1 is passed to the MARGIN parameter to indicate that apply will operate over each row, and the function that is applied is max. Compare the result when the MARGIN parameter is set to be columns:

```
apply(blocks@data[,14:17], 2, max)
```

The code above returns the largest percentage of each ethnic group in any census block.

Now suppose we wanted to determine which ethnicity formed the largest group in each block. One way would be to create a for loop. Another would be to define a function and use apply.

```
# set up vector to hold result

result.vector <- vector()

for (i in 1:nrow(blocks@data)){

# for each row determine which column has the max value

result.i <- which.max(data.frame(blocks[i,14:17]))

# put into the result vector

result.vector <- append(result.vector, result.i)

}
```

This can also be determined using apply as in the code below and the two results compared:

```
res.vec <-apply(data.frame(blocks[,14:17]), 1, which.max)
```

```
# compare the two results

identical(as.vector(res.vec), as.vector(result.vector))
```

Why use apply? Loops are tractable but slow! Typically apply functions are much quicker than loops, as is clear if the timings are compared. In many cases this will not matter, but it will when you have large data or heavy computations and processing. You may have to define your own functions and in some cases manipulate the data that are passed to apply, but they are a very useful family of functions.

```
# Loop

t1 <- Sys.time()

result.vector <- vector()

for (i in 1:nrow(blocks@data)){

result.i <- which.max(data.frame(blocks[i,14:17]))

result.vector <- append(result.vector, result.i)

}

Sys.time() - t1

# Apply

t1 <- Sys.time()

res.vec <-apply(data.frame(blocks[,14:17]), 1, which.max)

Sys.time() - t1
```

The second example uses mapply to plot the coordinates of each element of the georgia.polys list. Here a plot extent has to be defined, and then each polygon is plotted in turn (actually this is what plotting routines for sf and sp objects do). One way to do this is as follows:

```
plot(bbox(georgia2)[1,], bbox(georgia2)[2,], asp = 1,

type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')

for (i in 1:length(georgia.polys)){
```

```
    points(georgia.polys[[i]], type='l')
# small delay so that you can see the plotting
Sys.sleep(0.05)
}
```

Another would be use to mapply:

```
plot(bbox(georgia2) [1,], bbox(georgia2) [2,], asp = 1,
type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
invisible(mapply(polygon,georgia.polys))
```

The for loop below returns two objects: count.vec, a vector of the number of counties within 50 km of each of the 159 counties in the georgia2 dataset; and a list object with 159 elements of the names of these.

```
# convert Georgia2 to sf
georgia_sf <- st_as_sf(georgia2)
# create a distance matrix
dMat <- as.matrix(dist(coordinates(georgia2)))
dim(dMat)
# create an empty vector
count.vec <- vector()
# create an empty list
names.list <- list()
# for each county...
for( i in 1:nrow(georgia_sf)) {
    # which counties are within 50km
vec.i <- which(dMat[i,] <= 50000)
# add to the vector
```

```
count.vec <- append(count.vec, length(vec.i))

    # find their names

names.i <- georgia_sf$Name[vec.i]

# add to the list

names.list[[i]] <- names.i

}

# have a look!

count.vec

names.list
```

You could of course use lapply to investigate the list you have just created. Notice how this does not require a MARGIN to be specified as does apply. Rather it just requires a function to be applied to each element in a list:

```
lapply(names.list, length)
```

Self-Test Question 7. Recode the for loop above into two functions to be applied to the distance matrix, dMat, and called in a similar way to the following:

```
count.vec <- apply(dMat,1.my.func1)

names.list <- apply(dMat,1.my.func2)
```

4.7 Manipulating Data with dplyr

A second set of very useful tools in the context of programming is provided by the data table operations within the dplyr package, included within the tidyverse. These can be used with tabular data, including the data frames containing the attributes of spatial data. To start you should clear your R workspace and install and load the tidyverse package and explore the introduction vignette. Recall that vignettes were introduced in Chapter 3.

```
vignette("dplyr", package = "dplyr")
```

For the dplyr vignettes you will also have to install the nycflights13 package that contains some example data describing flights and airlines, and note that the default data table format for the tidyverse is tibble.

```
install.packages("nycflights13")

library("nycflights13")

class(flights)

flights
```

You can examine the other datasets included in this package as well:

```
data(package = "nycflights13")
```

You should explore the different functions for summarising and filtering individual data tables. The important ones are summarised in Table 4.3.

Table 4.3dplyr

Table 4.3 Functions in the `dplyr` package for manipulating data tables

Function	Description
`filter()`	Selects a subset of rows in a data frame, according to user-defined conditional statements
`slice()`	Selects a subset of rows in a data frame by their position (row number)
`arrange()`	Changes the row order according to the columns specified (by 1st, 2nd and then 3rd column, etc.)
`desc()`	Orders a column in descending order
`select()`	Selects the subset of specified columns and reorders them vertically
`distinct()`	Finds unique values in a table
`mutate()`	Creates and adds new columns based on operations applied to existing columns, e.g. `NewCol = Col1 + Col2`
`transmute`	As `select` but only retains the new variables
`summarise`	Summarises values with functions that are passed to it
`sample_n`	Takes a random sample of table rows
`sample_frac`	Selects a fixed fraction of rows

Then you should explore the two-table vignette.

```
vignette("two-table", package = "dplyr")
```

Again, you should work through the various join and summary operations in the two-table vignette. The first command is to select variables from flights to create flight2.

```
flights2 <- flights %>% select(year:day,hour,origin,dest,tailnum,carrier)
```

You will note that the vignette uses the piping syntax. The %>% command *pipes* the flights dataset to the select function, specifying the columns of data to be selected. The result is assigned to flights2. A non-piped version would be:

flights2 <- select(flights, year:day,hour,origin,dest,tailnum,carrier)

The dplyr package contains a number of methods for summarising and joining tables, including different _join functions: inner_join, left_join, right_join, full_join, semi_join and anti_join. You should familiarise yourself with how these different join functions operate and how they relate to the two data table inputs they take.

Self-Test Question 8. The code below creates flights2, a tibble data table in dplyr with variables of the destination (dest), the number of flights in 2013 (count) and the latitude and longitude of the origin (OrLat and OrLon) in the New York area.

```
library(nycflights13)

library(tidyverse)

# select the variables

flights2 <- flights %>% select(origin, dest)

# remove Alaska and Hawaii

flights2 <- flights2[-grep("ANC", flights2$dest),]

flights2 <- flights2[-grep("HNL", flights2$dest),]
```

```
# group by destination
flights2 <- group_by(flights2, dest)
flights2 <- summarize(flights2, count = n())
# assign Lat and Lon for Origin
flights2$OrLat <- 40.6925
flights2$OrLon <- −74.16867
# have a look!
flights2
# A tibble: 103 x 4
```

	dest	count	OrLat	OrLon
	<chr>	<int>	<dbl>	<dbl>
1	ABQ	254	40.7	−74.2
2	ACK	265	40.7	−74.2
3	ALB	439	40.7	−74.2
4	ATL	17215	40.7	−74.2
5	AUS	2439	40.7	−74.2
6	AVL	275	40.7	−74.2
7	BDL	443	40.7	−74.2
8	BGR	375	40.7	−74.2
9	BHM	297	40.7	−74.2
10	BNA	6333	40.7	−74.2

```
# ... with 93 more rows
```

Your task is to join the flights2 data table to the airports dataset and determine the latitude and longitude of the destinations. A secondary task, if

you wish, is to then map the flights using the gcIntermediate function in the geosphere package and the datasets in the maps package (both of which you may need to install).

Some hints about the mapping are provided in the code below. This example plots two locations and then uses the gcIntermediate function in geosphere to plot a path between them.

```
library(maps)

library(geosphere)

# origin and destination examples

dest.eg <- matrix(c(77.1025, 28.7041), ncol = 2)

origin.eg <- matrix(c(−74.16867, 40.6925), ncol = 2)

# map the world from the maps package data

map("world", fill=TRUE, col="white", bg="lightblue")

# plot the points

points(dest.eg, col="red", pch=16, cex = 2)

points(origin.eg, col = "cyan", pch = 16, cex = 2)

# add the route

for (i in 1:nrow(dest.eg)) {

lines(gcIntermediate(dest.eg[i,], origin.eg[i,], n=50,

breakAtDateLine=FALSE, addStartEnd=FALSE,

                sp=FALSE, sepNA), lwd = 2, lty = 2)

}
```

You may wish to explore the use of other basemaps from the maps package:

```
map("usa", fill=TRUE, col="white", bg="lightblue")
```

4.8 Answers to Self-Test Questions

Q1: A new cube root function:

```
cube.root.2 <- function(x)
{ if (is.numeric(x))
{ result <- sign(x)*abs(x)^(1/3)
return(result)
} else
{ cat("WARNING: Input must be numerical, not character\n")
return(NA) }
}
```

Q2: Match up the lines in the gcd function to the lines in the description of Euclid's algorithm:

```
gcd <- function(a,b)
{
        divisor <- min(a,b) # line 1
dividend <- max(a,b) # line 1
repeat #line 5
            { remainder <- dividend %% divisor #line 2
            dividend <- divisor # line 3
divisor <- remainder # line 4
if (remainder == 0) break #line 6
        }
      return(dividend)
}
```

Q3: (i) Write a function to compute and print out gcd(x,60):

```
gcd.60 <- function(a)
   {
      for(i in 1:a)
{ divisor <- min(i,60)
          dividend <- max(i,60)
repeat
{ remainder <- dividend %% divisor
                    dividend <- divisor
divisor <- remainder
if (remainder == 0) break
}
cat(dividend, "\n")
}
   }
```

Alternatively you could nest the predefined gcd function inside the modified one:

```
gcd.60 <- function(a)
{ for(i in 1:a)
      { dividend <- gcd(i,60)
cat(i,":", dividend, "\n")
}
}
```

(ii) Write a function to compute and print out gcd(x,y):

```
gcd.all <- function(x,y)
   { for(n1 in 1:
```

```
x) { for (n2 in 1:y)
{ dividend <- gcd(n1, n2)
cat("when x is",n1,"& y is",n2,"dividend =",dividend,"\n")
}
}
}
```

Q4: Modify cube.root.table to run from 0.5 to n in steps of 0.5. The obvious solution to this is:

```
cube.root.table <- function(n)
   { for (x in seq(0.5, n, by = 0.5))
{ cat("The cube root of ",x," is",
                sign(x)*abs(x)^(1/3),"\n")}
}
```

However, this will not work when negative values are passed to it: seq cannot create the array. The function can be modified to accommodate sequences running from 0.5 to both negative and positive values of n:

```
cube.root.table <- function(n)
   { if (n < 0 ) by.val = 0.5
if (n < 0 ) by.val =−0.5
for (x in seq(0.5, n, by = by.val))
{ cat("The cube root of ",x," is",
sign(x)*abs(x)^(1/3),"\n") }
}
```

Q5: Create a for loop that cycles through each county/row in the data frame of the georgia2 dataset and creates a list of the adjacent counties. You were given the code for a single county – this needs to be put into a

loop, replacing the 1 with i or similar.

```
# create an empty list for the results
adj.list <- list()
# convert georgia to sf
georgia_sf <- st_as_sf(georgia2)
for (i in 1:nrow(georgia_sf)) {
    # extract a single county
county.i <- georgia_sf[i,]
# determine the adjacent counties
# the [−1] removes Appling form its own list
adj.i <- unlist(st_intersects(county.i, georgia_sf))[−1]
# extract their names
adj.names.i <- georgia2$Name[adj.i]
# add to the list
adj.list[[i]] <- adj.i
# name the list elements
names(adj.list[[i]]) <- adj.names.i
}
```

Q6: Create a function that returns a list of the indices of adjacent polygons for each polygon in any polygon dataset in sf or sp format.

```
return.adj <- function(sf.data){
# convert to sf regardless!
    sf.data <- st_as_sf(sf.data)
adj.list <- list()
for (i in 1:nrow(sf.data)) {
```

```
# extract a single county
poly.i <- sf.data[i,]
# determine the adjacent counties
        adj.i <- unlist(st_intersects(poly.i, sf.data))[−1]
# add to the list
adj.list[[i]] <- adj.i
        }
return(adj.list)
}
# test it!
return.adj(georgia_sf)
return.adj(blocks)
```

Q7: Recode the for loop into two functions replicating the functionality of the loop:

```
# number of counties within 50km
my.func1 <- function(x){
vec.i <- which(x <= 50000)[−i]
return(length(vec.i))
}
# their names
my.func2 <- function(x){
    vec.i <- which(x <= 50000)
    names.i <- georgia_sf$Name[vec.i]
    return(names.i)
}
count.vec <- apply(dMat,1, my.func1)
```

```
names.list <- apply(dMat,1, my.func2)
```

Q8: Join the flights2 data table to the airports dataset and determine the latitude and longitude of the destinations. Then map the flights using the gcIntermediate function in the geosphere package and the datasets in the maps package.

```
# Part 1: the join

flights2 <- flights2 %>% left_join(airports, c("dest" = "faa"))

flights2 <- flights2 %>% select(count,dest,OrLat,OrLon, DestLat=lat,DestLon=lon)

# get rid of any NAs

flights2 <- flights2[!is.na(flights2$DestLat),] flights2

# Part 2: the plot

# Using standard plots

dest.eg <- matrix(c(flights2$DestLon, flights2$DestLat), ncol = 2)

origin.eg <- matrix(c(flights2$OrLon, flights2$OrLat), ncol = 2)

map("usa", fill=TRUE, col="white", bg="lightblue")

points(dest.eg, col="red", pch=16, cex = 1)

points(origin.eg, col = "cyan", pch = 16, cex = 1) for (i in 1:nrow(dest.eg)) {

lines(gcIntermediate(dest.eg[i,], origin.eg[i,], n=50,

breakAtDateLine=FALSE,

addStartEnd=FALSE, sp=FALSE, sepNA))

}

# using ggplot

all_states <- map_data("state")

dest.eg <- data.frame(DestLon = flights2$DestLon, DestLat = flights2$DestLat)

origin.eg <- data.frame(OrLon = flights2$OrLon, OrLat = flights2$OrLat)

library(GISTools)

# Figure 2 using ggplot
```

```
# create the main plot
mp <- ggplot() +
        geom_polygon( data=all_states,
aes(x=long, y=lat, group = group), colour="white", fill="grey20") + coord_fixed() +
    geom_point(aes(x = dest.eg$DestLon, y = dest.eg$DestLat), color="#FB6A4A", size=2)
+
        theme(axis.title.x=element_blank(), axis.text.x=element_blank(),
axis.ticks.x=element_blank(), axis.title.y=element_blank(),
                axis.text.y=element_blank(), axis.ticks.y=element_blank())
# create some transparent shading
cols=add.alpha(colorRampPalette(brewer.pal(9,"Reds"))(nrow(flights2)), 0.7)
# loop through the destinations
for (i in 1:nrow(flights2)) {
# line thickness related flights
    lwd.i = 1+ (flights2$count[i]/max(flights2$count))
# a sequence of colours
cols.i = cols[i]
# create a dataset
link <- as.data.frame(gcIntermediate(dest.eg[i,], origin.eg[i,],
n=50, breakAtDateLine=FALSE, addStartEnd=FALSE, sp=FALSE, sepNA))
names(link) <- c("lon", "lat")
mp <- mp + geom_line(data=link, aes(x=lon, y=lat),
                        color= cols.i, size = lwd.i)
}
# plot!
mp
```

5 Using R as a Gis

5.1 Introduction

In GIS and spatial analysis, we are often interested in finding out how the information contained in one spatial dataset relates to that contained in another. The kinds of questions we may be interested in include:

- How does X interact with Y?
- How many X are there in different locations of Y?
- How does the incidence of X relate to the rate of Y?
- How many of X are found within a certain distance of Y?
- How does process X vary with Y spatially?

X and Y may be diseases, crimes, pollution events, attributed census areas, environmental factors, deprivation indices or any other geographical process or phenomenon that you are interested in understanding. Answering such questions using a spatial analysis frequently requires some initial data pre-processing and manipulation. This might be to ensure that different data have the same spatial extent, describe processes in a consistent way (e.g. to compare land cover types from different classifications), are summarised over the same spatial framework (e.g. census reporting areas), are of the same format (raster, vector, etc.) and are projected in the same way (the latter was introduced in Chapter 3).

This chapter uses worked examples to illustrate a number of fundamental and commonly applied spatial operations on spatial datasets. Many of these form the basis of most GIS software. The datasets may be ones you have read into R from shapefiles or ones that you have created in the course of your analysis. Essentially, the operations illustrate different methods for extracting information from one spatial dataset based on the spatial extent of another. Many of these are what are frequently referred to as *overlay* operations in GIS software such as ArcGIS or QGIS, but here are extended to include a number of other types of data manipulation. The sections below describe the following operations:

- Intersections and clipping one dataset to the extent of another
- Creating buffers around features
- Merging the features in a spatial dataset
- Point-in-polygon and area calculations
- Creating distance attributes
- Combining spatial data and attributes
- Converting between raster and vector

As you work through the example code in this chapter a number of self-test questions are introduced. Some of these go into much greater detail and complexity than in earlier chapters and come with extensive direction for you to work through and follow.

The chapter draws on functionality from a number of packages that have been introduced in earlier chapters (sf, sp, maptools, GISTools, tidyverse, rgeos, etc.) for performing overlay and other spatial operations on spatial datasets which create new data, information or attributes. In many cases, it is up to the analyst (you!) to decide which operations to undertake and in what order for a particular analysis and, depending on your objectives, a given operation may be considered as a pre-processing step or as an analytical one. For example, calculating distances, areas, or point-in-polygon counts prior to a statistical test may be pre-processing steps prior to the actual data analysis or used as the actual analysis itself. The key feature of these operations is that they create new data or information. Similarly, this chapter will use both sf and sp data formats as needed, both of which have their own set of functions linking to rgeos. As a reminder, sf data formats are relatively new and have strong links to dplyr (part of the tidyverse package). This chapter will highlight operations in both, and where we think there is a distinct advantage to one approach this will be presented.

It is important to recall that there are conversion functions for moving between sf and sp formats:

```
library(sf)

library(GISTools) # a wrapper for sp, rgeos, etc.

# load some data
```

```
data(georgia)

class(georgia)

# convert to sf

georgia_sf <- st_as_sf(georgia)

class(georgia_sf)

# convert back to sp

georgia_v2 <- as(georgia_sf, "Spatial")

class(georgia_v2)
```

5.2 Spatial Intersection and Clip Operations

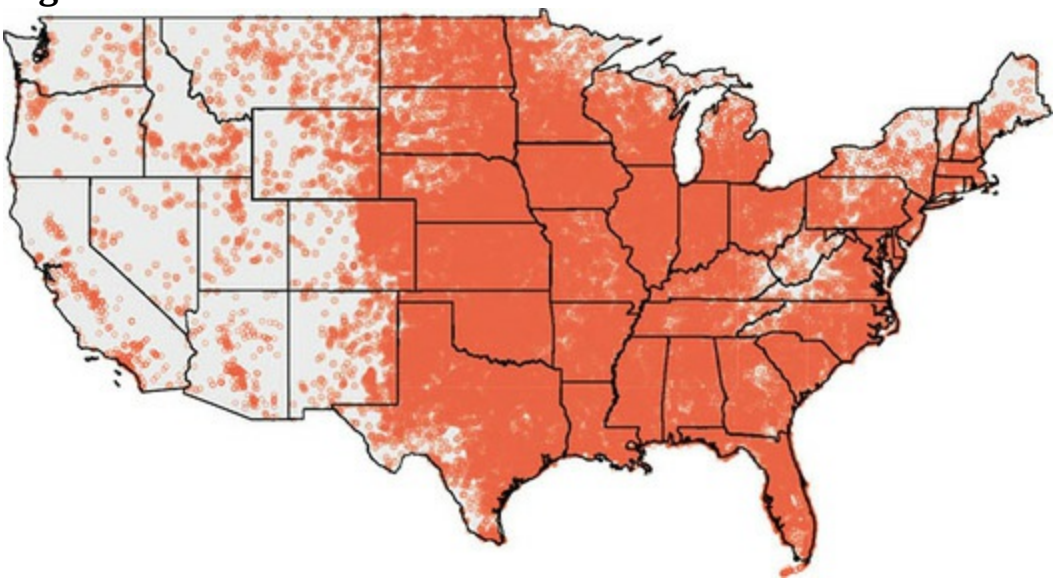
The GISTools package comes with datasets describing tornadoes in the USA. Load the package and these data into a new R session.

```
library(GISTools)

data(tornados)
```

You will see that four sp datasets are now loaded: torn, torn2, us_states and us_states2. The torn and torn2 data describe the locations of tornadoes recorded between 1950 and 2004, and the us_states and us_states2 datasets are spatial data describing the states of the USA. Two of these are in WGS84 projections (torn and us_states) and two are projected in a GRS80 datum (torn2 and us_states2). We can plot these and examine the data as in Figure 5.1.

```
library(tmap)

library(sf)

# convert to sf objects

torn_sf <- st_as_sf(torn)

us_states_sf <- st_as_sf(us_states)

# plot extent and grey background

tm_shape(us_states_sf) +tm_polygons("grey90") +

# add the torn points

tm_shape(torn_sf) +

tm_dots(col = "#FB6A4A", size = 0.04, shape = 1, alpha = 0.5) +

# map the state borders

tm_shape(us_states_sf) +tm_borders(col = "black") +

tm_layout(frame = F)
```

Figure 5.1 The tornado data



Note that the sp plotting code takes a very similar form:

```
plot(us_states, col = "grey90")

plot(torn, add = T, pch = 1, col = "#FB6A4A4C", cex = 0.4)

plot(us_states, add = T)
```

Remember that you can examine the attributes of a variable using the summary() function. For sp objects this also includes a summary of the object projection. This can be seen using the st_geometry function in sf:

```
summary(torn)

summary(torn_sf)

st_geometry(torn_sf)
```

Now, consider the situation where the aim was to analyse the incidence of tornadoes in a particular area: we do not want to analyse *all* of the tornado data but only those records that describe events in our study area – the area we are interested in. The code below selects a group of US states, in this case Texas, New Mexico, Oklahoma and Arkansas – note the use of the OR logical operator | to make the selection.

```
index <- us_states$STATE_NAME == "Texas" |
us_states$STATE_NAME == "New Mexico" |
us_states$STATE_NAME == "Oklahoma" |
us_states$STATE_NAME == "Arkansas"
AoI <- us_states[index,]
# OR....
AoI_sf <- us_states_sf[index,]
```

This can be plotted using the usual commands as in the code below. You can see that the plot extent is defined by the spatial extent of area of interest (called AoI_sf) and that all of the tornadoes within that extent are displayed.

```
tm_shape(AoI_sf) +
tm_borders(col = "black") +
tm_layout(frame = F) +
# add the torn points
tm_shape(torn_sf) +tm_dots(col = "#FB6A4A", size = 0.2, shape = 1, alpha = 0.5)
# OR in sp
plot(AoI)
plot(torn, add = T, pch = 1, col = "#FB6A4A4C")
```

There are a number of ways of clipping spatial data in R. The simplest of these is to use the spatial extent of one as an index to subset another. (Note that this can be done using sp objects as well.)

```
torn_clip_sf <- torn_sf[AoI_sf,]
```

This simply *clips out* the data from torn_sf that is within the spatial extent of AoI_sf. You can check this:

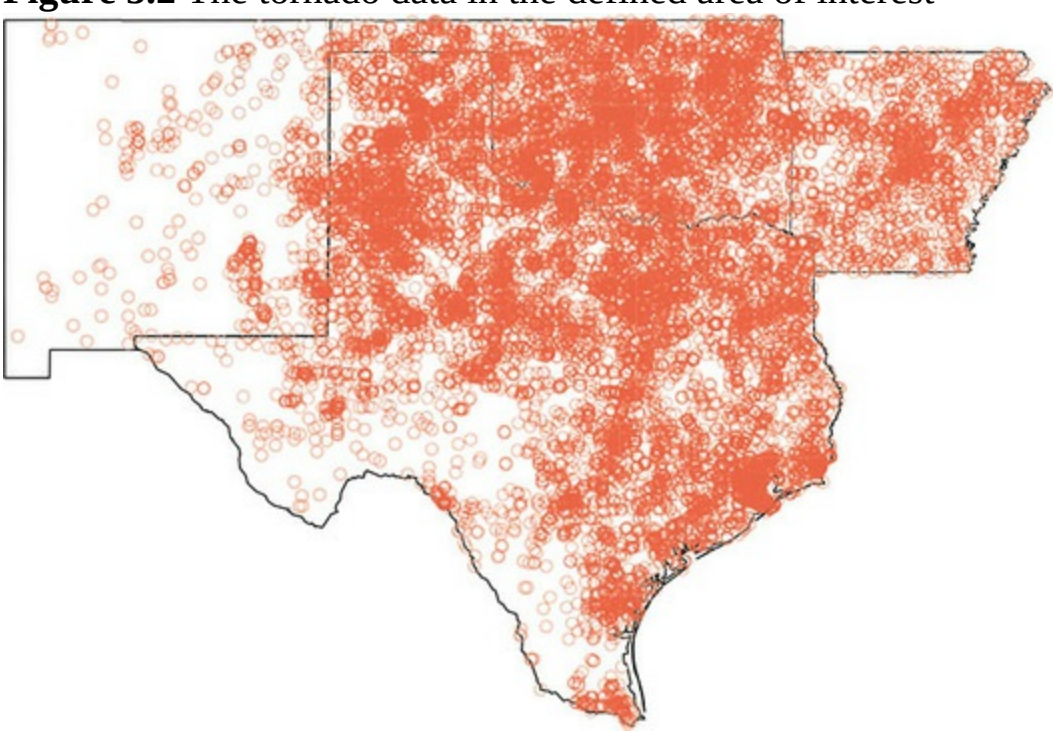
```
tm_shape(torn_clip_sf) +
```

```
tm_dots(col = "#FB6A4A", size = 0.2, shape = 1, alpha = 0.5) +
  tm_shape(AoI_sf) +
tm_borders()
```

However, such clip (or crop) operations simply subset data based on their spatial extents. There may be occasions when you wish to combine the *attributes* of difference datasets based on the *spatial intersection*. The gIntersection function in rgeos or the st_intersection in sf allows us to do this as shown in the code below. The results are mapped in Figure 5.2.

```
AoI_torn_sf <- st_intersection(AoI_sf, torn_sf)
tm_shape(AoI_sf) + tm_borders(col = "black") + tm_layout(frame = F) +
# add the torn points
tm_shape(AoI_torn_sf) +
tm_dots(col = "#FB6A4A", size = 0.2, shape = 1, alpha = 0.5)
```

Figure 5.2 The tornado data in the defined area of interest



The st_intersection operation creates an sf dataset of the locations of the

tornadoes within the area of interest. The gIntersection function does the same thing:

```
AoI.torn <- gIntersection(AoI, torn, byid = TRUE)

plot(AoI)

plot(AoI.torn, add = T, pch = 1, col = "#FB6A4A4C")
```

If you examine the data created by the intersection, you will notice that each of the intersecting points has the full attribution from input datasets. You can examine the attributes of the AoI_torn_sf data and the AoI.torn data by entering:

```
head(data.frame(AoI_torn_sf))

head(data.frame(AoI.torn))
```

Once extracted, the subset can be written out for use elsewhere as described in Chapters 2 and 3. You should examine the help for both st_intersection and gIntersection to see how they work. You should particularly note that both functions operate on any pair of spatial objects provided they are projected using the same datum (in this case WGS84). In order to perform spatial operations you may need to re-project your data to the same datum using spTransform or st_transform as described in Chapter 3.

5.3 Buffers

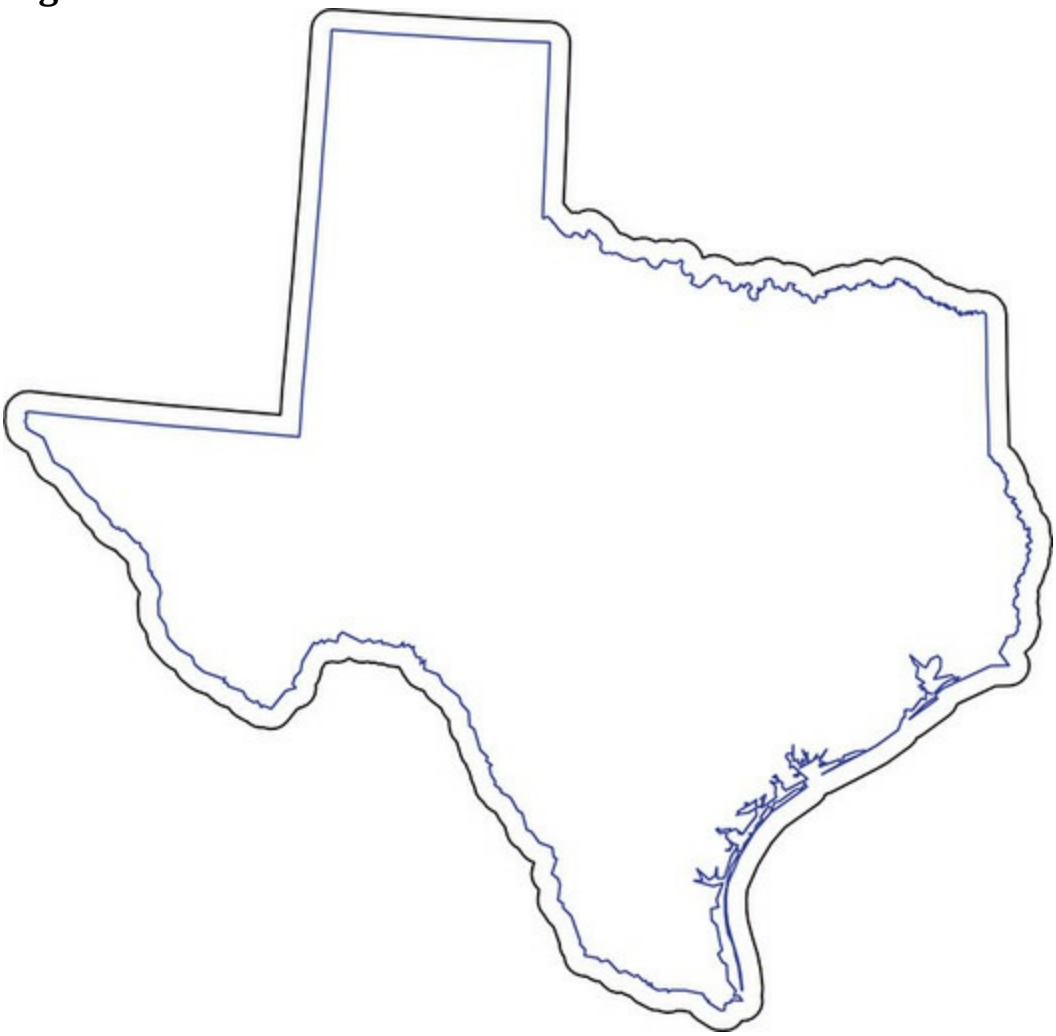
In many situations, we are interested in events or features that occur near to our area of interest as well as those within it. Environmental events such as tornadoes, for example, do not stop at state lines or other administrative boundaries. Similarly, if we were studying crime locations or spatial access to facilities such as shops or health services, we would want to know about locations near to the study area border. *Buffer* operations provide a convenient way of doing this, and buffers can be created in R using the gBuffer function in rgeos or the st_buffer function in sf.

Continuing with the example above, we might be interested in extracting the

tornadoes occurring in Texas and those within 25 km of the state border. Thus the objective is to create a 25 km buffer around the state of Texas and to use that to select from the tornado dataset. Both buffer functions allow us to do that, and require a distance for the buffer to be specified in terms of the units used in the projection. However, in order to do this, a different projection is required as distances are difficult to determine directly from projections in degrees (essentially, the relationship between planar distance measures such as metres and kilometres to degrees varies with latitude). And the buffer will return an error message if you try to buffer a non-projected spatial dataset. Therefore, the code below uses the projected US data, us_states2, and the resultant buffer is shown in Figure 5.3.

```
# select an area of interest and apply a buffer

# in rgeos

AoI <- us_states2[us_states2$STATE_NAME == "Texas",]

AoI.buf <- gBuffer(AoI, width = 25000)

# in sf

us_states2_sf <- st_as_sf(us_states2)

AoI_sf <- st_as_sf(us_states2_sf[us_states2_sf$STATE_NAME == "Texas",])

AoI_buf_sf <- st_buffer(AoI_sf, dist = 25000)

# map the buffer and the original area

# sp format

par(mar=c(0,0,0,0))

plot(AoI.buf)

plot(AoI, add = T, border = "blue")

# tmap: commented out!

# tm_shape(AoI_buf_sf) + tm_borders("black") +

# tm_shape(AoI_sf) + tm_borders("blue") +

# tm_layout(frame = F)
```

Figure 5.3 Texas with a 25 km buffer



The buffered object, shown in Figure 5.3, or objects can be used as input to clip or intersection operations as above, for example to extract data within a certain distance of an object. You should also examine the impact on the output of other parameters in both buffer functions that control how line segments are created, the geometry of the buffer, join styles, etc. Note that any sp or sf objects can be used as an input to gBuffer and st_intersection functions, respectively: try applying them to the breach dataset that is put into working memory when the newhaven data are loaded.

There are number of options for defining how the buffer is created. If you enter the code below, using IDs, then buffers are created around each of the counties within the georgia2 dataset:

```
data(georgia)

georgia2_sf <- st_as_sf(georgia2)

# apply a buffer to each object

# sf

buf_t_sf <- st_buffer(georgia2_sf, 5000)

# rgeos

buf.t <- gBuffer(georgia2, width = 5000, byid = T, id = georgia2$Name)

# now plot the data

# sf

tm_shape(buf_t_sf) +tm_borders() +tm_shape(georgia2) +tm_borders(col = "blue")
+tm_layout(frame = F)

# rgeos

plot(buf.t)

plot(georgia2, add = T, border = "blue")
```

The IDs of the resulting buffer datasets relate to each of the input features, which in the above code has been specified to be the county names. This can be checked by examining how the buffer object has been named using names(buf.t). If you are not convinced that the indexing has been preserved then you can compare the output with a familiar subset, Appling County:

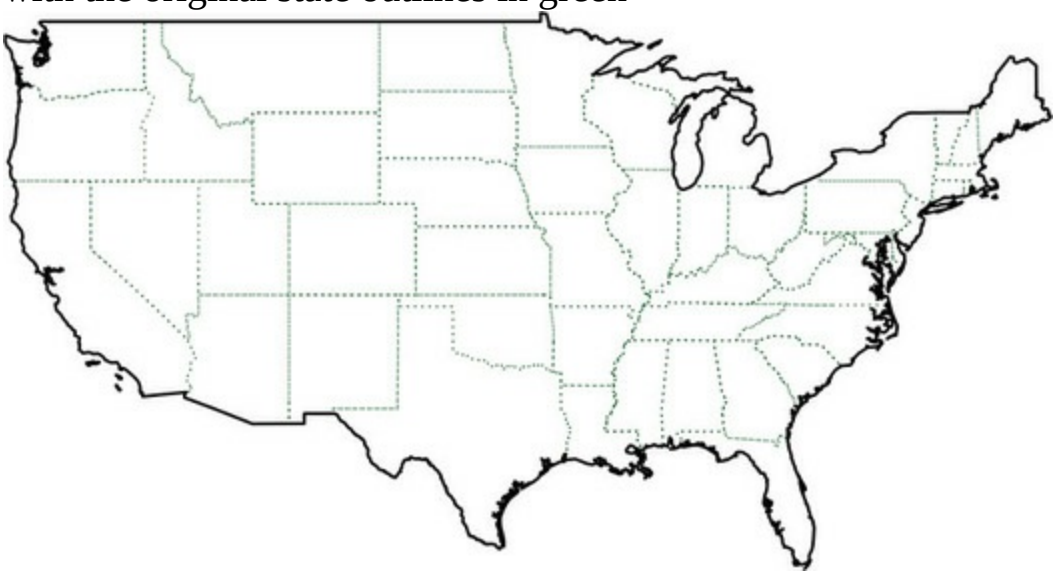
```
plot(buf.t[1,])

plot(georgia2[1,], add = T, col = "blue")
```

5.4 Merging Spatial Features

In the intersection example above, four US states were selected and used to define the area of interest over which the tornado data were extracted. An attribute describing in which state each tornado occurred was added to the data frame of the intersected object. In other instances we may wish to consider the area as a single object and to merge the features within it. This

can be done using the gUnaryUnion function in the rgeos package which was used in Chapter 3, and also the st_union and st_combine functions in the sf package, to create an outline of the state of Georgia from its constituent counties. In the code below the US states are merged into a single object and then plotted over the original data as shown in Figure 5.4. Note the use of the st_sf function to convert the sfc output of the st_union function to sf class before passing to the tmap functions.

Figure 5.4 The outline of the merged US states created by gUnaryUnion, with the original state outlines in green



```
library(tmap)

### with rgeos and sp commented out

# AoI.merge <- gUnaryUnion(us_states)

# plot(us_states, border = "darkgreen", lty = 3)

# plot(AoI.merge, add = T, lwd = 1.5)

### with sf and tmap

us_states_sf <- st_as_sf(us_states)

AoI.merge_sf <- st_sf(st_union(us_states_sf))

tm_shape(us_states_sf) + tm_borders(col = "darkgreen", lty = 3)
+tm_shape(AoI.merge_sf) + tm_borders(lwd = 1.5, col = "black") +tm_layout(frame = F)
```

The union operations merge spatial object sub-geometries. Once the merged objects have been created they can be used as inputs into the intersection and buffering procedures above in order to select data for analysis, as well as the analysis operations described below. The merged objects can also be used in a cartographic context to provide a border to the study area being considered.

5.5 Point-in-Polygon and Area Calculations

5.5.1 Point-in-Polygon

It is often useful to count the number of points falling within different zones in a polygon dataset. This can be done using the poly.counts function in the GISTools package, which extends the gContains function in rgeos, or using a similar method with the st_contains function in sf.

> I
>
> Remember that you can examine how a function works by entering it into the console without the brackets – try entering poly.counts at the console.

The code below assigns a list of counts of the number of tornadoes that occur inside each US state to the variable torn.count and prints the first six of these to the console using the head function:

```
torn.count <- poly.counts(torn, us_states)

head(torn.count)

1    2    3    4    5    6

79    341    87    1121    1445    549
```

The numbers along the top are the ‘names’ of the elements in the variable tmp, which in this case are the polygon ID numbers of the us_states variable. The values are the counts of the points in the corresponding polygons. You can check this by entering:

```
names(torn.count)
```

5.5.2 Area Calculations

Another useful operation is to be able calculate polygon areas. The gArea and st_area functions in rgeos and sf do this. To check the projection, and therefore the map units, of an sp class object (including SpatialPolygons, SpatialPoints, etc.), use the proj4string function, and for sf objects use the st_crs function:

```
proj4string(us_states2)

st_crs(us_states2_sf)
```

This declares the projection to be in metres. To see the areas in square metres of each US state, enter:

```
poly.areas(us_states2)

st_area(us_states2_sf)
```

These are not particularly useful, and more realistic measures are to report areas in hectares or square kilometres:

```
# hectares

poly.areas(us_states2) / (100 * 100)

st_area(us_states2_sf) / (100 * 100)

# square kilometres

poly.areas(us_states2) / (1000 * 1000)

st_area(us_states2_sf) / (1000 * 1000)
```

Self-Test Question 1. Create the code to produce maps of the densities of breaches of the peace in each census block in New Haven in *breaches per square kilometre*. For the analysis you will need to use the breach point data and the census blocks in the newhaven dataset and undertake a point-in-polygon operation, apply an area function and undertake a conversion to

square kilometres. The maps should be produced using the tm_shape and tm_fill functions in the tmap package. The New Haven data are included in the GISTools package:

```
data(newhaven)
```

Reminder: As with all self-test questions, worked answers are provided in the final section of the chapter.

You should note that the New Haven dataset is projected in feet. One way is to leave the data in feet, calculate densities in squares miles and convert to square kilometres, apply the ft2miles function to the results of the area calculation, and as areas are in squared units, you will need to apply it twice, noting that there are approximately 2.58999 square kilometres in each square mile. The code below calculates the area in square kilometres of each block:

```
ft2miles(ft2miles(gArea(blocks, byid = T))) ∗ 2.58999
```

5.5.3 Point and Areas Analysis Exercise

An important advantage of using R to handle spatial data is that it is very easy to incorporate your data into statistical analysis and graphics routines. For example, in the New Haven blocks data frame, there is a variable called P_OWNEROCC which states the percentage of owner-occupied housing in each census block. It may be of interest to see how this relates to the breach of peace densities calculated in Self-Test Question 1. A useful statistic is the correlation coefficient generated by the cor function which causes the correlation to be printed out:

```
data(newhaven)

blocks$densities=poly.counts(breach,blocks)/ ft2miles(ft2miles(poly.areas(blocks)))

cor(blocks$P_OWNEROCC,blocks$densities)

[1] −0.2038463
```

In this case the two variables have a correlation of around −0.2, a weak negative relationship, suggesting that, in general, places with a higher proportion of owner-occupied homes tend to see fewer breaches of peace. It is also possible to plot the relationship between the quantities:

```
ggplot(blocks@data, aes(P_OWNEROCC,densities))+ geom_point()
+geom_smooth(method = "lm")
```

A more detailed approach might be to model the number of breaches of peace. Typically, these are relatively rare, and a Poisson distribution might be an appropriate model. A possible model might then be:

```
breaches ~ Poisson(AREA ∗ exp(a + b ∗ P_OWNEROCC))
```

where AREA is the area of a block, P_OWNEROCC is the percentage of owner occupiers in the block, and *a* and *b* are coefficients to be estimated, *a* being the intercept term. The AREA variable plays the role of an *offset* – a variable that always has a coefficient of 1. The idea here is that even if breaches of peace were uniformly distributed, the number of incidents in a given census block would be proportional to the AREA of that block. In fact, we can rewrite the model such that the offset term is the log of the area:

```
breaches ~ Poisson(exp(a + b ∗ P_OWNEROCC+log(AREA)))
```

Seeing the model written this way makes it clear that the offset term has a coefficient that must always be equal to 1. The model can be fitted in R using the following code:

```
# load and attach the data

data(newhaven)

attach(data.frame(blocks))

# calculate the breaches of the peace in each block

n.breaches = poly.counts(breach,blocks)

area = ft2miles(ft2miles(poly.areas(blocks)))
```

```
# fit the model

model1=glm(n.breaches~P_OWNEROCC,offset=log(area),family=poisson)

# detach the data

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

The first two lines compute the counts, storing them in n.breaches, and the areas, storing them in area. The next line fits the Poisson model. glm stands for 'generalised linear model', and extends the standard lm routine to fit models such as Poisson regression. As a reminder, further information about linear models and the R modelling language was provided in one of the *information boxes* in Chapter 3 and an example of its use was given. The family=poisson option specifies that a Poisson model is to be fitted here. The offset option specifies the offset term, and the first argument specifies the actual model to be fitted. The model- fitting results are stored in the variable model1. Having created the model in this way, entering:

```
model1
```

returns a brief summary of the fitted model. In particular, it can be seen that the estimated coefficients are $a = 3.02$ and $b = -0.0310$.

A more detailed view can be obtained using:

```
summary(model1)
```

Now, among other things, the standard errors and Wald statistics for a and b are now shown. The Wald Z-statistics are similar to *t-statistics* in ordinary least squares regression, and may be tested against the normal distribution. The results in Table 5.1 summarise the information, showing that both a and b are significant, and that therefore there is a statistically significant relationship between owner occupation and breach of peace incidents.

Table 5.1

Table 5.1 Summary of the Poisson model of the breaches of the peace over census blocks

	Estimate	Std. error	Wald's Z	*p*-value
Intercept	3.02	0.11	27.4	<0.01
Owner Occ. %	−0.031	0.00364	−8.5	<0.01

It is also possible to extract diagnostic information from fitted models. For example, the rstandard function extracts the standardised residuals from a model. Whereas residuals are the difference between the observed value (i.e. in the data) and the value when estimated using the model, standardised residuals are rescaled to have a variance of 1. If the model being fitted is correct, then these residuals should be independent, have a mean of 0, a variance of 1 and an approximately normal distribution. One useful diagnostic is to map these values. The code below computes them and stores them in a variable called s.resids:

```
s.resids = rstandard(model1)
```

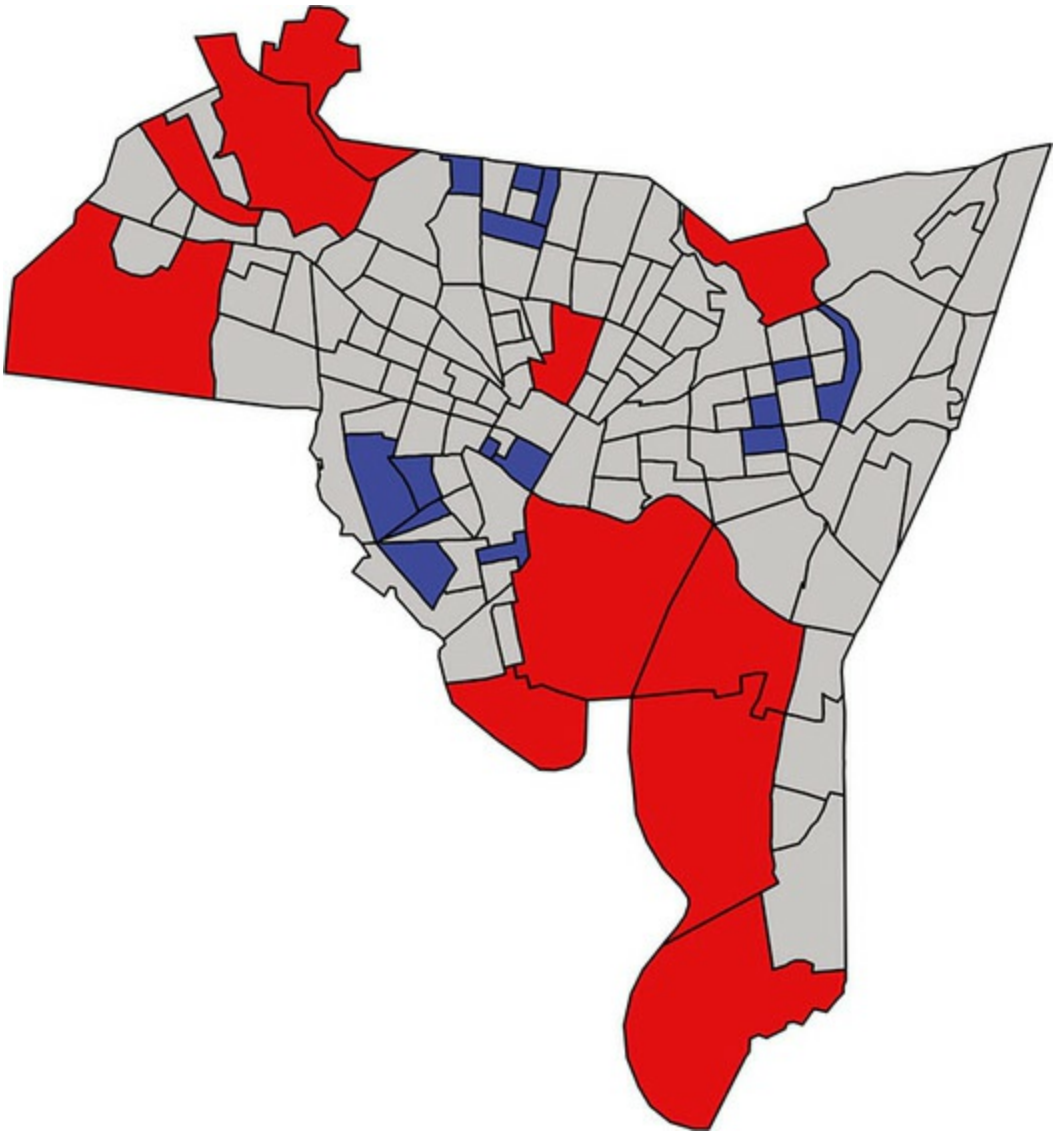
Now to plot the map it will be more useful to specify a shading scheme directly using the shading command:

```
resid.shades = shading(c(−2,2),c("red","grey","blue"))
```

This specifies that the map will have three class intervals: below −2, between −2 and 2, and above 2. These are useful intervals, given that the residuals should be normally distributed, and these values are the approximate two-tailed 5% points of this distribution. Residuals within these points will be shaded grey, large negative residuals will be red, and large positive ones will be blue:

```
par(mar=c(0,0,0,0))

choropleth(blocks,s.resids,resid.shades)
```

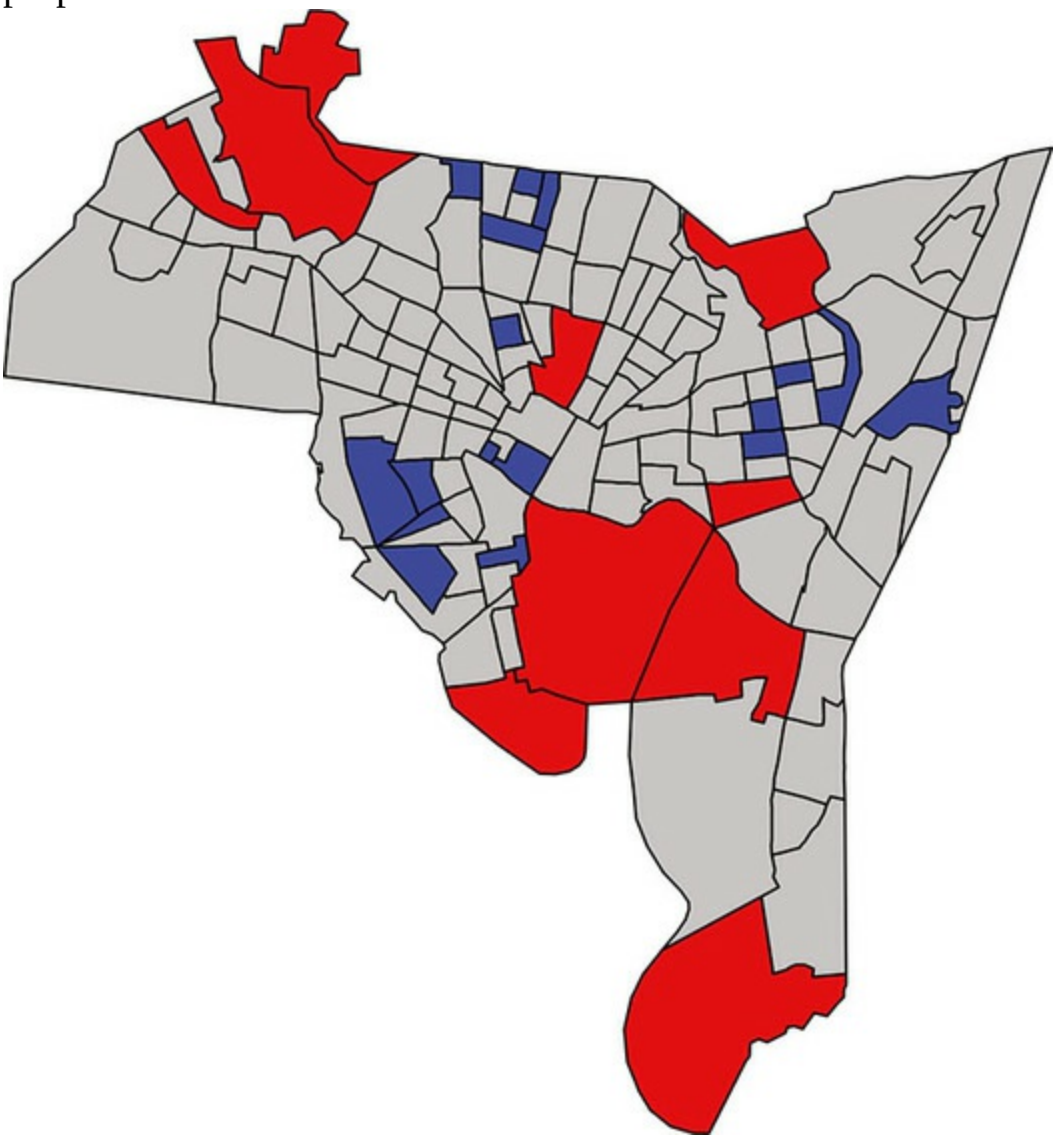
Figure 5.5 The distribution of the model1 residuals, describing the relationship between breaches of the peace and owner occupancy



From Figure 5.5 it can be seen that in fact there is notably more variation than one might expect (there are 21 blocks shaded blue or red, about 16% of the total, when around 5% would appear based on the model's assumptions), and also that the shaded blocks seem to cluster together. This last observation casts doubt on the assumption of independence, suggesting instead that some degree of spatial correlation is present. One possible reason for this is that further variables may need to be added to the model, to explain this extra variability and spatial clustering among the residuals.

It is possible to extend this analysis by considering P_VACANT, the percentage of vacant properties in each census block, as well as P_OWNEROCC. This is done by extending model1 and entering:

Figure 5.6 The distribution of the model2 residuals, describing the relationship between breaches of the peace with owner occupancy and vacant properties



```
attach(data.frame(blocks))
n.breaches = poly.counts(breach,blocks)
area = ft2miles(ft2miles(poly.areas(blocks)))
model2=glm(n.breaches~P_OWNEROCC+P_VACANT,
offset=log(area),family=poisson)
s.resids.2 = rstandard(model2)
```

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

This sets up a new model, with a further term for the percentage of vacant housing in each block, and stores it in model2. Entering summary(model2) shows that the new predictor variable is significantly related to breaches of the peace, with a positive relationship. Finally, it is possible to map the standardised residuals for the new model reusing the shading scheme defined above:

```
s.resids.2 = rstandard(model2)

par(mar=c(0,0,0,0))

choropleth(blocks,s.resids.2,resid.shades)
```

This time, Figure 5.6 shows that there are fewer red- and blue-shaded census blocks, although perhaps still more than we might expect, and there is still some evidence of spatial clustering. Adding the extra variable has improved things to some extent, but perhaps there is more investigative research to be done. A more comprehensive treatment of spatial analysis of spatial data attributes is given in Chapter 7.

Self-Test Question 2. The above code uses the choropleth function in GISTools to produce a map of outlying residuals. Create a similar-looking map but using the tm_shape function of the tmap package. You may find it useful to unpick the choropleth function, to think about passing a user-defined palette to tm_polygons, to assign s.resids.2 as a blocks variable, and/or to pass a set of break values.

5.6 Creating Distance Attributes

Distance is fundamental to spatial analysis. For example, we may wish to analyse the number of locations (health facilities, schools, etc.) within a certain distance of the features we are considering. In the exercise below, distance measures are used to evaluate differences in accessibility for different social groups, as recorded in census areas. Such approaches form the basis of supply and demand modelling and provide inputs into location–allocation models.

Distance could be approximated using a series of buffers created at specific distance intervals around our features (whether point or polygons). These could be used to determine the number of features or locations that are within different distance ranges, as specified by the buffers using the poly.counts function above. However, distances can also be measured directly and there a number of functions available in R to do this.

First, the most commonly used function is dist. This calculates the Euclidean distance between points in *n*-dimensional feature space. The example below, developed from the help for dist, shows how it is used to calculate the distances between five records (rows) in a feature space of 20 hypothetical variables.

```
x <- matrix(rnorm(100), nrow = 5)

colnames(x) <- paste0("Var", 1:20)

dist(x)

as.matrix(dist(x))
```

If your data are projected (in metres, feet, etc.) then dist can also be used to calculate the Euclidean distance between pairs of coordinates.

```
as.matrix(dist(coordinates(blocks))) # in feet

as.matrix(dist(coordinates(georgia2))) # in metres
```

When determining geographical distances, it is important that you consider the projection properties of your data: if the data are projected using degrees (i.e. in latitude and longitude) then this needs to be considered in any calculation of distance. The gDistance function in rgeos calculates the Cartesian minimum (straight-line) distance between two spatial datasets of class sp projected in planar coordinates. Try entering:

```
# this will not work

gDistance(georgia[1,], georgia[2,])

# this will!
```

```
gDistance(georgia2[1,], georgia2[2,])
```

The st_distance function in sf is similar but is also able to calculate great circle distances for projected points.

```
# convert to sf

georgia2_sf <- st_as_sf(georgia2)

georgia_sf <- st_as_sf(georgia)

st_distance(georgia2_sf[1,], georgia2_sf[2,])

st_distance(georgia_sf[1,], georgia_sf[2,])

# with points

sp <- st_as_sf(SpatialPoints(coordinates(georgia)))st_distance(sp[1,], sp[1:3,])
```

The distance functions return a to–from matrix of the distances between each pair of locations. These could describe distances between *any* objects, and such approaches underpin supply and demand modelling and accessibility analyses.

For example, the code below uses gDistance to calculate the distances between the centroids of the newhaven blocks data and the places locations. The latter are simply random locations, but could represent any kind of facility or *supply* feature, and the centroids of the census blocks in New Haven represent *demand* locations. In the first few lines of code, the projections of the two variables are set to be the same, before SpatialPoints is used to extract the geometric centroids of the census block areas and the distance between places and cents are calculated:

```
data(newhaven)

proj4string(places) <- CRS(proj4string(blocks))

cents <- SpatialPoints(coordinates(blocks),

proj4string = CRS(proj4string(blocks)))

# note the use of the ft2miles function to convert to miles

distances <- ft2miles(gDistance(places, cents, byid = T))
```

You can examine the result in relation to the inputs to gDistance and you will see that the distances variable is a matrix of distances (in miles) from each of the 129 census block centroids to each of the nine locations described in the places variable.

```
head(round(distances, 3))
```

It is possible to use the census block polygons in the above gDistance calculation, and the distances returned will be to the nearest point of the census area. Using the census area centroid provides a more representative measure of the average distance experienced by people living in that area.

A related function is the gWithinDistance function, which tests whether each to–from distance pair is less than a specified threshold. It returns a matrix of TRUE and FALSE describing whether the distances between the elements of the two sp dataset elements are less than or equal to the specified distance or not. In the example below the distance specified is 1.2 miles.

```
distances <- gWithinDistance(places, cents,     byid = T, dist = miles2ft(1.2))
```

You should note that the distance functions work with whatever distance units are specified in the projections of the spatial features. This means the inputs need to have the same units. Also remember that the newhaven data are projected in feet, hence the use of the miles2ft and ft2miles functions.

5.6.1 Distance Analysis/Accessibility Exercise

The use of distance measures in conjunction with census data is particularly useful for analysing access to the supply of some facility or service for different social groups. The code below replicates the analysis developed by Comber et al. (2008), examining access to green spaces for different social groups. In this exercise a hypothetical example is used: we wish to examine the equity of access to the locations recorded in the places variable (supply) for different ethnic groups as recorded in the blocks dataset (demand), on the basis that we expect everyone to be within 1 mile of a facility. We will use

the census data to approximate the number of people with and without access of less than 1 mile to the set of hypothetical facilities.

First, the distances variable is recalculated in case it was overwritten in the gWithinDistance example above. Then the minimum distance to a supply facility is determined for each census area using the apply function. Finally, a logical statement is used to generate a TRUE or FALSE statement for each block:

```
distances <- ft2miles(gDistance(places, cents, byid = T))

min.dist <- as.vector(apply(distances,1, min))

blocks$access <- min.dist < 1

# and this can be mapped

#qtm(blocks, "access")
```

The populations of each ethnic group in each census block can be extracted from the blocks dataset:

```
# extract the ethnicity data from the blocks variable

ethnicity <- as.matrix(data.frame(blocks[,14:18])/100)

ethnicity <- apply(ethnicity, 2, function(x) (x * blocks$POP1990))

ethnicity <- matrix(as.integer(ethnicity), ncol = 5)

colnames(ethnicity) <- c("White", "Black", "Native American", "Asian", "Other")
```

And then a crosstabulation is used to bring together the access data and the populations:

```
# use xtabs to generate a crosstabulation

mat.access.tab = xtabs(ethnicity~blocks$access)

# then transposes the data

data.set = as.data.frame(mat.access.tab)

#sets the column names
```

```
colnames(data.set) = c("Access","Ethnicity", "Freq")
```

You should examine the data.set variable. This summarises all of the factors being considered: access, ethnicity and the counts associated with all factor combinations. If we make an assumption that there is an interaction between ethnicity and access, then this can be tested for using a generalised regression model with a Poisson distribution using the glm function:

```
modelethnic = glm(Freq~Access*Ethnicity, data=data.set,family=poisson)

# the full model can be printed to the console

# summary(modelethnic)
```

The model coefficient estimates show that there is significantly less access for some groups than would be expected under a model of equal access when compared to the largest ethnic group, White, which was listed first in the data.set variable, and significantly greater access for the Other ethnic group. Examine the model coefficient estimates, paying particular attention to the AccessTRUE: coefficients:

```
summary(modelethnic)$coef
```

Then assign these to the a variable:

```
mod.coefs = summary(modelethnic)$coef
```

By subtracting 1 from the coefficients and converting them to percentages, it is possible to attach some likelihoods to the access for different groups when compared to the White ethnic group. Again, you should examine the terms in the model outputs prefixed by AccessTRUE:, as below:

```
tab <- 100*(exp(mod.coefs[,1]) − 1)

tab <- tab[7:10]

names(tab) <- colnames(ethnicity)[2:5]

round(tab, 1)
```

```
                Black Native American    Asian

−35.1          −11.7    −29.8

Other

256.3
```

The results in tab tell us that some ethnic groups have significantly less access to the hypothetical supply facilities than the White ethnic group (as recorded in the census): Black 35% less, Native American 12% less (although this is not significant), and Asian 30% less. The Other ethnic group has 256% more access than the White ethnic group.

It is possible to visualise the variations in access for different groups using a mosaic plot. Mosaic plots show the counts (i.e. population) as well as the residuals associated with the interaction between groups and their access, the full details of which were given in Chapter 3.

```
mosaicplot(t(mat.access.tab),xlab='',ylab='Access to Supply', main="Mosaic Plot of Access",shade=TRUE,las=3,cex=0.8)
```

Self-Test Question 3. In working through the exercise above you have developed a number of statistical techniques. In answering this self-test question you will explore the impact of using census data summarised over different areal units in your analysis. Specifically, you will develop and compare the results of two statistical models using different census areas in the newhaven datasets: blocks and tracts. You will analyse the relationship between residential property occupation and burglaries. You will need to work through the code below before the tasks associated with this questions are posited. To see the relationship between the census tracts and the census blocks, enter:

```
plot(blocks,border='red')

plot(tracts,lwd=2,add=TRUE)
```

You can see that the census blocks are nested within the tracts.

The analysis described below develops a statistical model to describe the

relationship between residential property occupation and burglary using two of the New Haven crime variables related to residential burglaries. These are both point objects, called burgres.f and burgres.n: the former is a list of burglaries where entry was forced into the property, and the latter is a list of burglaries where entry was not forced, suggesting that the property was left insecure, perhaps by leaving a door or window open. The burglaries data cover the six-month period between 1 August 2007 and 31 January 2008.

The questions you will consider are:

- Do both kinds of residential burglary occur in the same places – that is, if a place is a high-risk area for non-forced entry, does it imply that it is also a high-risk for forced entry?
- How does this relationship vary over different census units?

To investigate these, you should use a bivariate regression model that attempts to predict the density of forced burglaries from the density of non-forced ones. The indicators needed for this are the rates of burglary given the number of properties at risk. You should use the variable OCCUPIED, present in both the census blocks data frame and the census tracts data frame, to estimate the number of properties at risk. If we were to compute rates per 1000 households, this would be: 1000*(number of burglaries in block)/OCCUPIED, and since this is over a six-month period, doubling this quantity gives the number of burglaries per 1000 households per year. However, entering:

```
blocks$OCCUPIED
```

shows that some blocks have no occupied housing, so the above rate cannot be defined. To overcome this problem you should select the subset of the blocks with more than zero occupied dwellings. For polygon spatial objects, each individual polygon can be treated like a row in a data frame for the purposes of subset selection. Thus, to select only the blocks where the variable OCCUPIED is greater than zero, enter:

```
blocks2 = blocks[blocks$OCCUPIED > 0,]
```

We can now compute the burglary rates for forced and non-forced entries by first counting the burglaries in each block in blocks2 using the poly.counts function, dividing these numbers by the OCCUPIED counts and then multiplying by 2000 to get yearly rates per 1000 households. However, before we do this, you should remember that you need the OCCUPIED attribute from blocks2 and not blocks. Attach the blocks2 data and then calculate the two rate variables:

```
attach(data.frame(blocks2))

forced.rate = 2000*poly.counts(burgres.f,blocks2)/OCCUPIED

notforced.rate = 2000*poly.counts(burgres.n,blocks2)/OCCUPIED

detach(data.frame(blocks2))
```

You should have two rates stored in forced.rate and notforced.rate. A first attempt at modelling the relationship between the two rates could be via simple bivariate regression, ignoring any spatial dependencies in the error term. This is done using the lm function, which creates a simple regression model, model1:

```
model1 = lm(forced.rate~notforced.rate)
```

To examine the regression coefficients, enter:

```
summary(model1)

coef(model1)
```

The key things to note here are that forced.rate is related to notforced.rate by the formula:

```
expected(forced.rate) = a + b × (notforced.rate)
```

where *a* is the *intercept* term and *b* is the slope or coefficient for the predictor variable. If the coefficient for notforced.rate is statistically different from zero, indicated in the summary of the model, then there is evidence that the

two rates are related. One possible explanation is that if burglars are active in an area, they will only use force to enter dwellings when it is necessary, making use of an insecure window or door if they spot the opportunity. Thus in areas where burglars are active, both kinds of burglary could potentially occur. However, in areas where burglars are less active it is less likely for either kind of burglary to occur.

Having outlined the approach, your specific tasks in this question are:

- To determine the coefficients *a* and *b* in the formula above for two different analyses using the blocks and tracts datasets
- To comment on the difference between the analyses using different areal units

5.7 Combining Spatial Datasets and their Attributes

The point-in-polygon calculation using poly.counts generates counts of the points falling in each polygon. A common situation in spatial analysis is the need to combine (overlay) different polygon features that describe the spatial distribution of different variables, attributes or processes that are of interest. The problem is that the data may have different underlying area geographies. In fact, it is commonly the case that different agencies, institutions and government departments use different geographical areas, and even where they do not, geographical areas frequently change over time. In these situations, we can use the intersection functions (gIntersection in rgeos or st_intersection in sf) to identify the area of intersection between different spatial datasets. With some manipulation it is possible to determine the proportions of the objects in dataset X that fall into each of the polygons of dataset Y. This section uses a worked example to illustrate how this can be done in R. In the subsequent self-test question you will develop a function to do this.

The key thing to note with all spatial operations, whether using sp and sf datasets, is that the input data need to have the same projections. You can examine their projection attributes with proj4string in sp and st_crs in sf to check whether they need to be transformed, using spTransform (sp) or st_transform (sf) functions to put the data into the same projection.

The stages in this analysis are as follows:

1. Create a zone dataset for which the number of houses in each zone will be calculated. The New Haven tracts data include the variable HSE_UNITS, describing the number of residential properties in each census tract. In this case the zones are hypothetical, but could perhaps be zones used by the emergency services for planning purposes and resource allocation.
2. Do an overlay of the new zones and the original areas. The key here is to make sure that both the layers have an identifier that allows the proportions of each original area in each zone to be calculated. This will then be used to allocate houses based on the proportion of each intersecting area in each zone.

First, you should make sure you have the tmap and sf packages loaded. Then create the zones, number them with an ID and plot these on a map with the tracts data. This is easily done by defining a grid and then converting this to a SpatialPolygonsDataFrame object. Enter:

```
library(GISTools)

library(sf)

## linking to GEOS 3.6.1, GDAL 2.1.3, proj.4.4.9.3

library(tmap)

data(newhaven)

## define sample grid in polygons

bb <- bbox(tracts)

grd <- GridTopology(cellcentre.offset=c(bb[1,1]−200,bb[2,1]−200),

cellsize=c(10000,10000), cells.dim = c(5,5))

int.layer <- SpatialPolygonsDataFrame( as.SpatialPolygons.GridTopology(grd), data =
data.frame(c(1:25)), match.ID = FALSE)

ct <- proj4string(blocks)

proj4string(int.layer) <- ct

proj4string(tracts) <- ct
```

```
names(int.layer) <- "ID"
```

You can examine the intersection layer:

```
plot(int.layer)
```

Next, you should undertake an intersection of the zone and area layers. Projections can be checked using proj4string(int.layer) and proj4string (tracts). These have the same projections, so they can be intersected. The code below converts them to sf format and then uses st_intersection:

```
int.layer_sf <- st_as_sf(int.layer)

tracts_sf <- st_as_sf(tracts)

int.res_sf <- st_intersection(int.layer_sf, tracts_sf)
```

You can examine the intersected data, the original data and the zones in the same plot window, as in Figure 5.7. Remember that the grid.arrange function in the gridExtra package allows multiple graphics to be included in the plot.

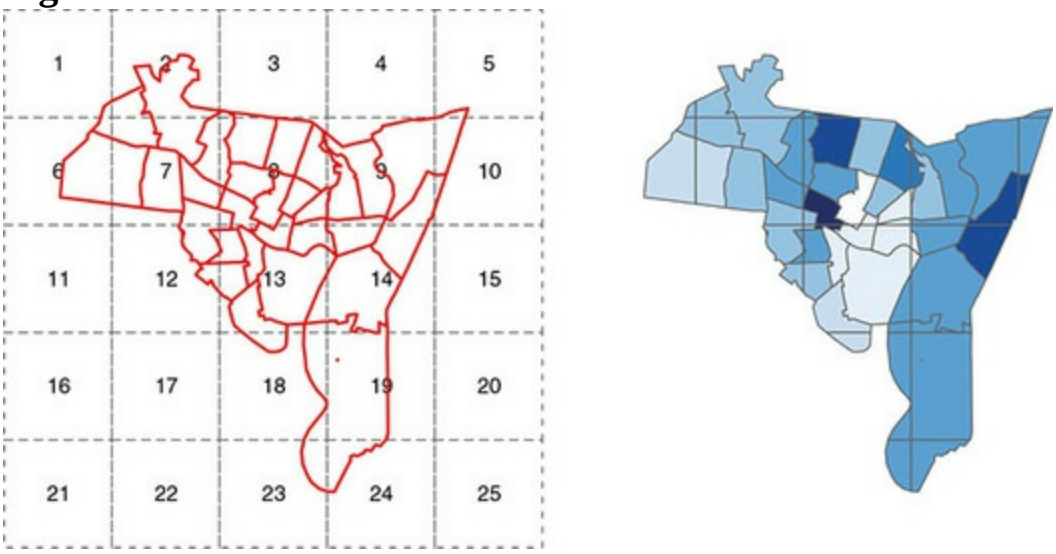
```
# plot and label the zones

p1 <- tm_shape(int.layer_sf) + tm_borders(lty = 2) +tm_layout(frame = F) +

tm_text("ID", size = 0.7) +

# plot the tracts

tm_shape(tracts_sf) + tm_borders(col = "red", lwd = 2)

# plot the intersection, scaled by int.later_sf

p2 <- tm_shape(int.layer_sf) + tm_borders(col="white") +tm_shape(int.res_sf) +
tm_polygons("HSE_UNITS", palette = blues9) +tm_layout(frame = F, legend.show = F)

library(grid)

grid.newpage()

pushViewport(viewport(layout=grid.layout(1,2)))

print(p1, vp=viewport(layout.pos.col = 1))

print(p2, vp=viewport(layout.pos.col = 2))
```

As in the gIntersection operation described in earlier sections, you can examine the result of the intersection:

```
head(int.res_sf)
```

You will see that the data frame of the intersected object contains composites of the inputs. These links can be used to create attributes for the intersection output data.

Figure 5.7 The zones and census tracts data before and after intersection



Recall the need to have an identifier for both the zone and area layers. The ID variable of the intersection output, int.res_sf, lists the ID variable of the two input layers, the ID variable of int.layer_sf and the T009075H_I variable of tracts.sf. In this case, we wish to summarise the HSE_UNITS of tracts_sf over the zones of int.layer_sf. Here the functionality of dplyr single-table operations that were introduced in Chapter 4 can be useful. However, first we need to work out what proportion of the original tracts areas intersect with each zone, and we can weight the HSE_UNITS variable appropriately to proportionally allocate the counts of houses to the zones. Knowing the unique identifiers of each polygon in both of the intersected layers is critical for working out proportions.

```
# generate area and proportions
```

```
int.areas <- st_area(int.res_sf)

tract.areas <- st_area(tracts_sf)

# match tract area to the new layer

index <- match(int.res_sf$T009075H_I, tracts$T009075H_I)

tract.areas <- tract.areas[index]

tract.prop <- as.vector(int.areas)/as.vector(tract.areas)
```

The tract.prop object can be used to create a variable in the data frame of the new layer, using the index variable which indicates in which of the original tract areas each intersected area belongs. (Note that you could examine index to see this.)

```
int.res_sf$houses <- tracts$HSE_UNITS[index] * tract.prop
```

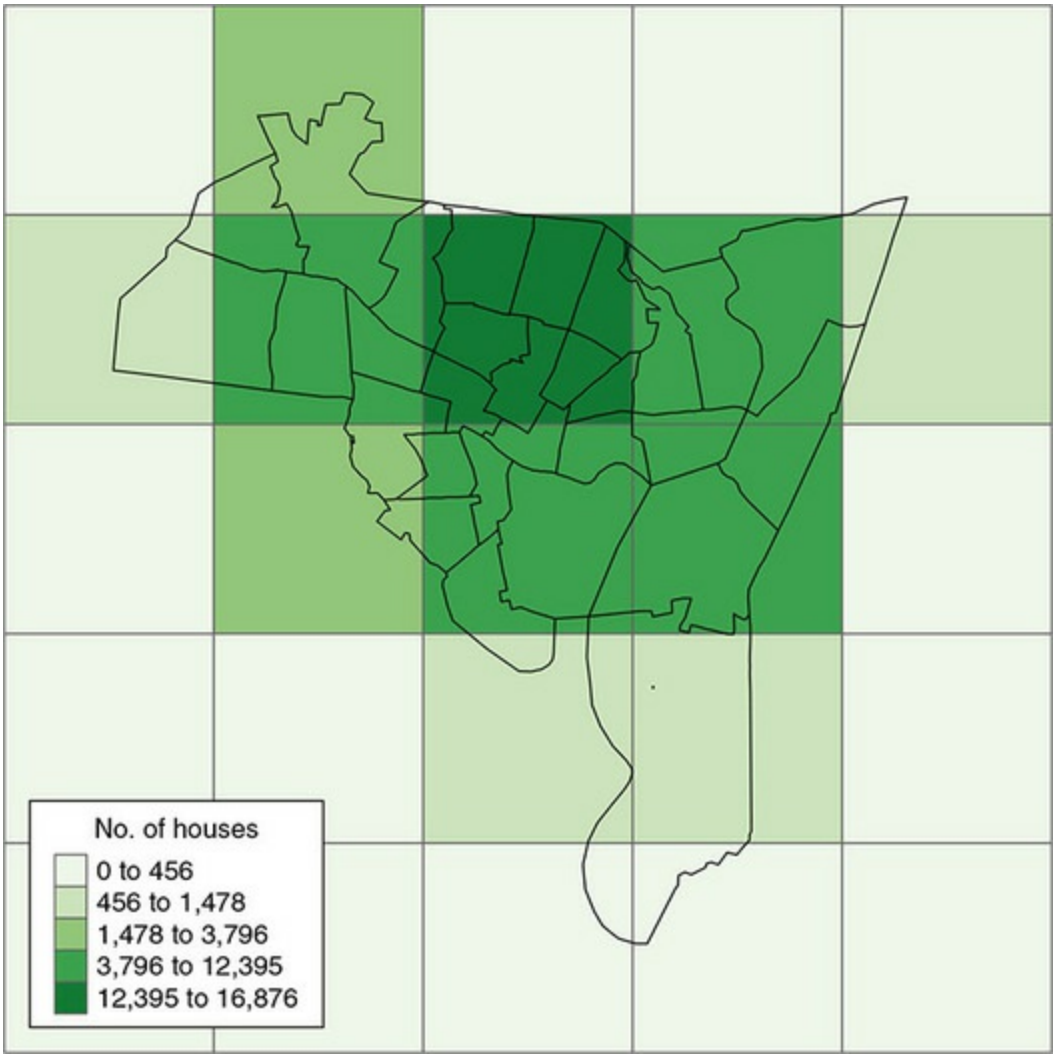
And this can be summarised using the functionality in dplyr and linked back to the original int.layer_sf:

```
library(tidyverse)

houses <- summarise(group_by(int.res_sf, ID), count = sum(houses))

# create an empty vector

int.layer_sf$houses <- 0

# and populate this using houses$ID as the index

int.layer_sf$houses[houses$ID] <- houses$count
```

The results can be plotted as in Figure 5.8 and checked against the original inputs in Figure 5.7.

```
tm_shape(int.layer_sf) +tm_polygons("houses", palette = "Greens", style = "kmeans", title
= "No. of houses") +

tm_layout(frame = F, legend.position = c(1,0.5)) +

tm_shape(tracts_sf) + tm_borders(col = "black")
```

Figure 5.8 The zones shaded by the number of households after intersection with the census tracts



Self-Test Question 4. Write a function that will return an intersected dataset, with an attribute of counts of some variable (houses, population, etc.) as held in another sf format dataset. Base your function on the code used in the illustrated example above. Compile it such that the function returns the portion of the variable (typically this should be a count) covered by each zone. For example, it should be able to intersect the int.layer_sf layer with the blocks_sf layer and return an sf dataset with an attribute of the number of people, as described in the POP1990 variable of blocks, covered by each zone. You should remember that many spatial functions require their inputs to have the same projections. The int.layer_sf defined above and the tracts originally had no projections. You may find it useful to check and/or align

the input layers – for example, the int.layer defined above and the blocks data in the following way using the rgdal or sf packages:

```
## in rgdal

library(rgdal)

ct <- proj4string(blocks)

proj4string(int.layer) <- CRS(ct)

blocks <- spTransform(blocks, CRS(proj4string(int.layer)))

## in sf

library(sf)

ct <- st_crs(blocks_sf)

st_crs(int.layer_sf) <- (ct)

blocks_sf <- st_transform(blocks_sf, st_crs(int.layer_sf))
```

Your function will have to take identifier variables for the layer and the intersect layer as inputs, and you will find it useful in your code to assign these to new ID variables in each layer. For example, your function could require the following parameters when compiled, setting some default values:

```
# define the function

area_intersect_func <- function(int.sf = int.layer.sf, layer.sf = blocks.sf, int.ID <- "ID",

    layer.ID <- "T009075H_I", target <- "POP1990"){

    ...

    ...

    }
```

Also, extracting values from data in sf format can be tricky. A couple of possible ways are:

```
# directly from the data frame
```

```
as.vector(data.frame(int.res_sf[,"T009075H_I"])[,1])
as.vector(unlist(select(as.data.frame(int.res_sf), T009075H_I)))

# set the geometry to null and then extract

st_geometry(int.res_sf) <- NULL

int.res_sf[,"T009075H_I"]

# using select from dplyr

as.vector(unlist(select(as.data.frame(int.res_sf), T009075H_I)))
```

5.8 Converting Between Raster and Vector

Very often we would like to move or convert our data between vector and raster environments. In fact the very persistence of these dichotomous data structures, with separate raster and vector functions and analyses in many commercial GIS software programs, is one of the long-standing legacies in GIS.

This section briefly describes methods for converting data between raster and vector structures. There are three reasons for this brief treatment. First, many packages define their own data structures. For example, the functions in the PBSmapping package require a PolySet object to be passed to them. This means that conversion between one class of raster objects and, for example, the sp class of SpatialPolygons will require different code. Second, the separation between raster and vector analysis environments is no longer strictly needed, especially if you are developing your spatial analyses using R, with the easy ability for users to compile their own functions and to create their own analysis tools. Third, advanced raster mapping and analysis is extensively covered in other books (see, for example, Bivand et al., 2013).

The sections below describe methods for converting the sp class of objects (SpatialPoints, SpatialLines and SpatialPolygons, etc.) and the sf class of objects (see the first sf vignette) as well as to and from the RasterLayer class of objects as defined in the raster package, created by Hijmans and van Etten (2014). They also describe how to convert between sp classes, for example to and from SpatialPixels and SpatialGrid sp objects.

5.8.1 Vector to Raster

In this section simple approaches for converting are illustrated using datasets in the tornados package that you have already encountered. We shall examine techniques for converting the sp class of objects to the raster class, considering in turn points, lines and areas.

Unfortunately, at the time of writing there is no parallel operation for converting from sf formats to raster formats. If you have data in sf format, you could convert to an sp format before converting to raster format as described earlier:

```
# convert to sf sp

sp <- as(sf, "Spatial")

# do the conversions...as below
```

You will need to load the data and the packages – you may need to install the raster package using the install.packages function if this is the first time that you have used it.

5.8.1.1 Converting Points to Raster

First, convert from sp to raster formats. The torn2 is a Spatial PointsDataFrame object:

```
library(GISTools)

library(raster)

data(tornados)

class(torn2)
```

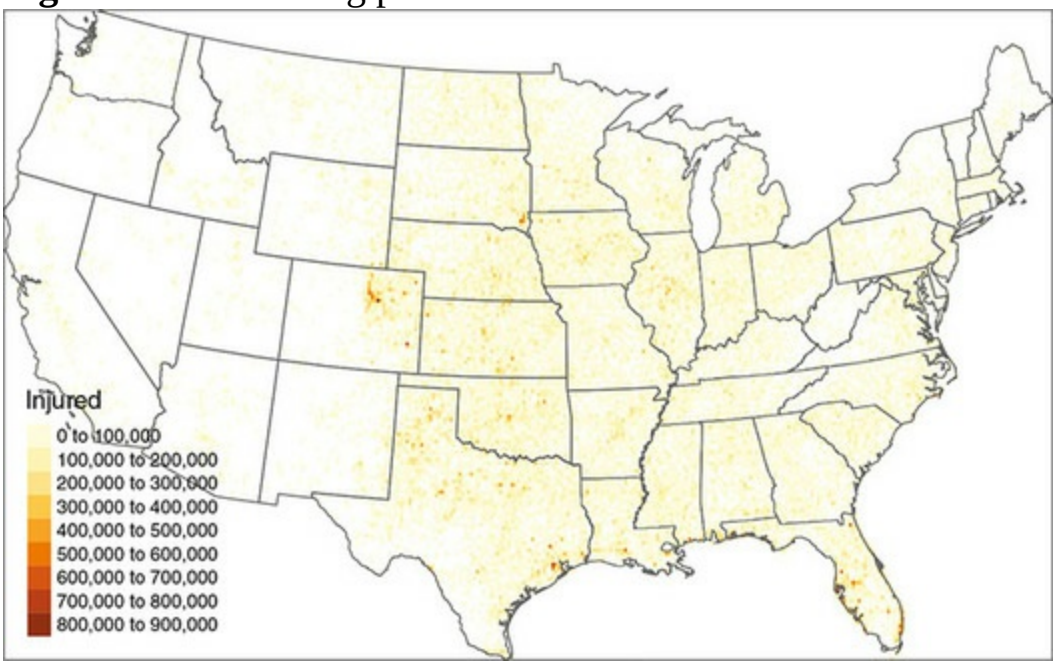
Then create a raster and use the rasterize function to convert the data. Note the need for a function to be specified to determine *how* the point dataset are summarised over the raster grid and, if the data have attributes, which attribute is to be summarised:

```
# rasterize a point attribute
r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))
r <- rasterize(torn2, r, field = "INJ", fun=sum)
# rasterize count of point dataset
r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))
r <- rasterize(as(torn2, "SpatialPoints"), r, fun=sum)
```

The resultant raster has cells describing different tornado densities that can be mapped as in Figure 5.9:

```
# set the plot extent by specify the plot colour 'white'
tm_shape(us_states2)+tm_borders("white")+
   tm_shape(r) +
tm_raster(title = "Injured", n= 7) +tm_shape(us_states2) +
tm_borders() +
tm_layout(legend.position = c("left", "bottom"))
```

Figure 5.9 Converting points to raster format



5.8.1.2 Converting Lines to Raster

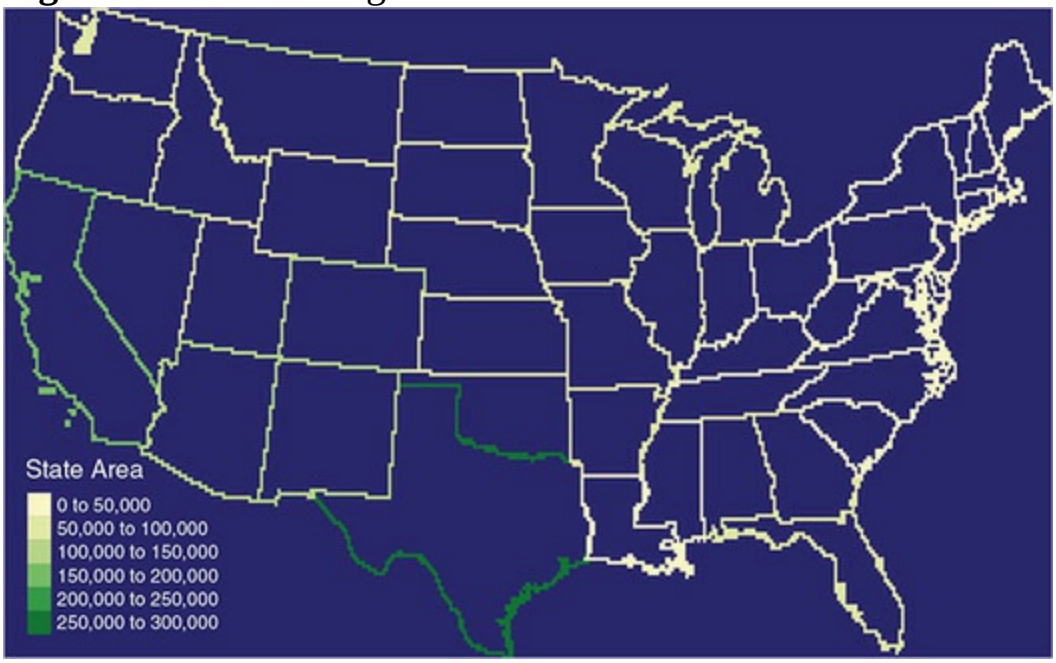
For illustrative purposes the code below creates a SpatialLinesDataFrame object of the outline of the polygons with an attribute based on the area of the state.

```
# Lines

us_outline <- as(us_states2 , "SpatialLinesDataFrame")

r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))

r <- rasterize(us_outline , r, "AREA")
```

This takes a bit longer to run but again the results can be mapped and this time with the shading indicating area (Figure 5.10):

```
tm_shape(r) +

tm_raster(title = "State Area", palette = "YlGn") +tm_style("albatross") +

tm_layout(legend.position = c("left", "bottom"))
```

Figure 5.10 Converting lines to raster format



5.8.1.3 Converting Polygons or Areas to Raster

Finally, polygons can easily be converted to a RasterLayer object using tools in the raster package and plotted as in Figure 5.11. In this case the 1997 population for each state is used to generate raster cell or pixel values.

```
# Polygons

r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))

r <- rasterize(us_states2, r, "POP1997")

tm_shape(r) +tm_raster(title = "Population", n=7, style="kmeans", palette="OrRd")
+tm_layout(legend.outside = T,

legend.outside.position = c("left"), frame = F)
```

It is instructive to examine the outputs of these processes. Enter:

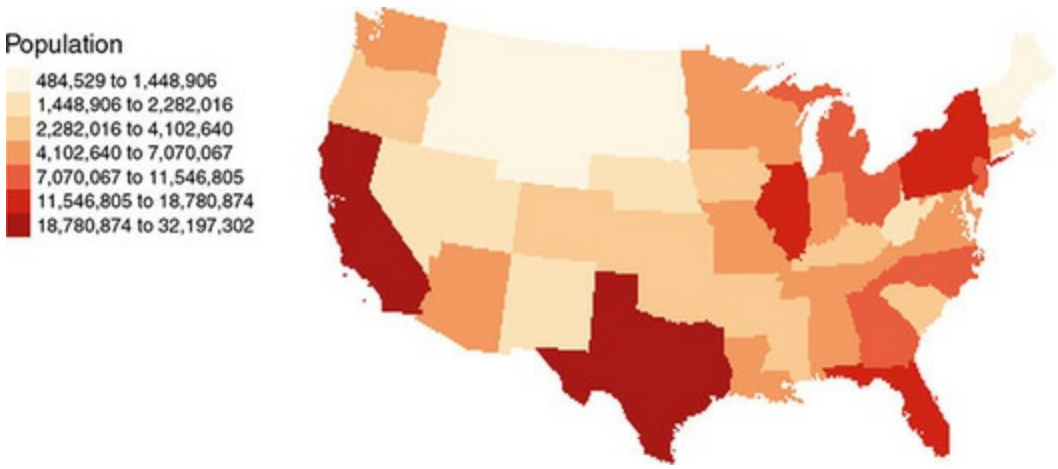
```
r
```

This summarises the characteristics of the raster object, including the resolution, dimensions and extent. The data values of r can be accessed using the getValues function:

unique(getValues(r))

It is possible to specify particular dimensions for the raster grid cells, rather than just dividing the dataset's extent by ncol and nrow in the raster function. The code below is a bit convoluted, but cleanly allocates the values to raster grid cells of a specified size, allocating cell values from a polygon variable to the raster cells.

Figure 5.11 Converting polygons to raster format




```
# specify a cell size in the projection units
d <- 50000
dim.x <- d
dim.y <- d
bb <- bbox(us_states2)
# work out the number of cells needed
cells.x <- (bb[1,2]−bb[1,1]) / dim.x
cells.y <- (bb[2,2]−bb[2,1]) / dim.y
round.vals <- function(x){
if(as.integer(x) < x) {
x <- as.integer(x) + 1
} else {x <- as.integer(x)
}}
# the cells cover the data completely
cells.x <- round.vals(cells.x)
cells.y <- round.vals(cells.y)
# specify the raster extent
ext <- extent(c(bb[1,1], bb[1,1]+(cells.x*d),
bb[2,1],bb[2,1]+(cells.y*d)))
```

```
# now run the raster conversion

r <- raster(ncol = cells.x,nrow =cells.y)

extent(r) <- ext

r <- rasterize(us_states2, r, "POP1997")

# and map

tm_shape(r) +

tm_raster(col = "layer", title = "Populations", palette = "Spectral", style = "kmeans") +

tm_layout(frame = F, legend.show = T, legend.position = c("left","bottom"))
```

5.8.2 Converting to sp raster classes

You may have noticed that the sp package also has two data classes that are able to represent raster data, or data located on a regular grid. These are SpatialPixelsDataFrame and SpatialGridDataFrame. It is possible to convert the rasters to these. First, create a spatially coarser raster layer of US states similar to the above.

```
r <- raster(nrow = 60 , ncols = 120, ext = extent(us_states2))

r <- rasterize(us_states2 , r, "BLACK")
```

Then the as function can be used to coerce this to SpatialPixelsDataFrame and SpatialGridDataFrame objects, which can also be mapped using the image, plot and tm_raster commands in the usual way:

```
g <- as(r, 'SpatialGridDataFrame')

p <- as(r, 'SpatialPixelsDataFrame')

# image(g, col = topo.colors(51))
```

You can examine the data values held in the data frame by entering:

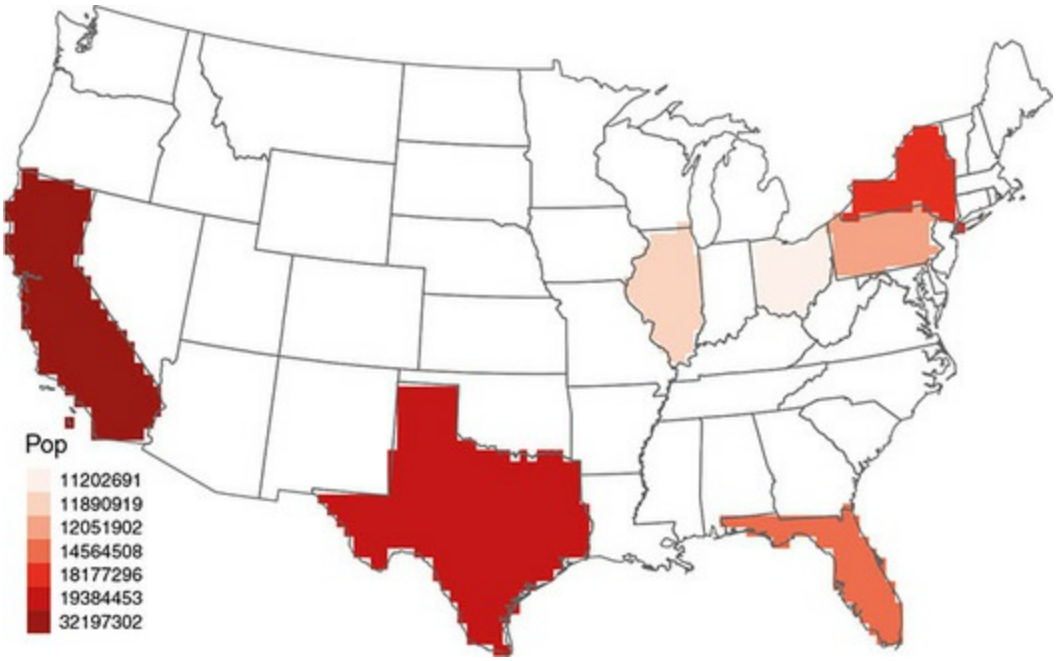
```
head(data.frame(g))
```

```
head(data.frame(p))
```

The data can also be manipulated to select certain features, in this case selecting the states with populations greater than 10 million people. The code below assigns NA values to the data points that fail this test and plots the data as in Figure 5.12.

```
# set up and create the raster

r <- raster(nrow = 60 , ncols = 120, ext = extent(us_states2))

r <- rasterize(us_states2 , r, "POP1997")

r2 <- r

# subset the data

r2[r < 10000000] <- NA

g <- as(r2, 'SpatialGridDataFrame')

p <- as(r2, 'SpatialPixelsDataFrame')

# not run

# image(g, bg = "grey90")

tm_shape(r2) +

tm_raster(col = "layer", title = "Pop", palette = "Reds", style = "cat") +

tm_layout(frame = F, legend.show = T, legend.position = c("left","bottom")) +

tm_shape(us_states2) + tm_borders()
```

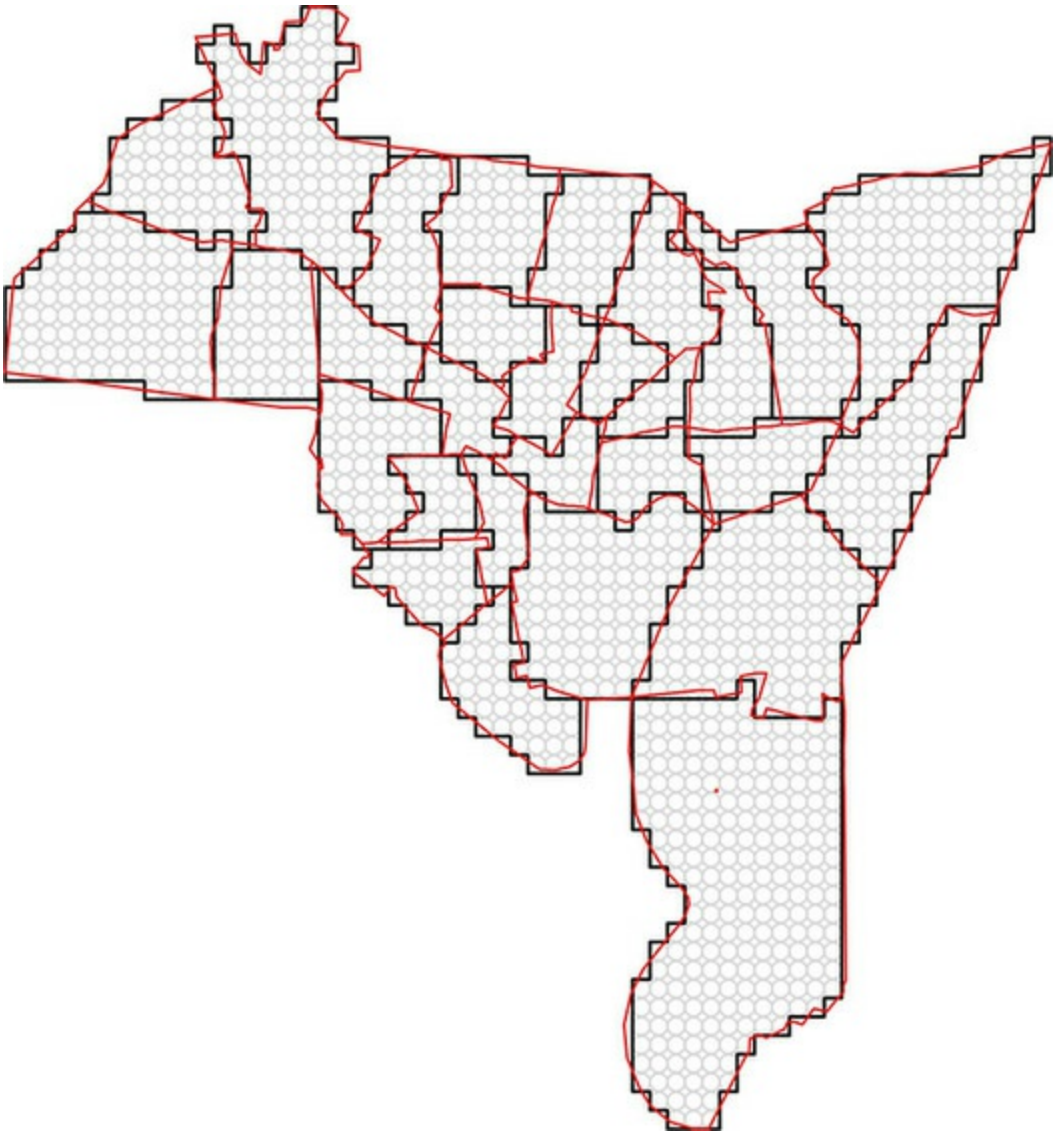
Figure 5.12 Selecting data in a raster object



5.8.2.1 Raster to Vector

The raster package contains a number of functions for converting from vector to raster formats. These include rasterToPolygons which converts to a SpatialPolygonsDataFrame object, and rasterToPoints which converts to a matrix object. Both are illustrated in the code below and the results shown in Figure 5.13. Notice how the original raster imposes a grid structure on the polygons that are created. In this case the default mapping options with plot are easier than using the options in the tmap or ggplot2 packages.

Figure 5.13 Converting from rasters to polygons and points, with the original polygon data in red



```
# load the data and convert to raster

data(newhaven)

# set up the raster, r

r <- raster(nrow = 60 , ncols = 60, ext = extent(tracts))

# convert polygons to raster

r <- rasterize(tracts , r, "VACANT")

poly1 <- rasterToPolygons(r, dissolve = T)

# convert to points
```

```
points1 <- rasterToPoints(r)

# plot the points, rasterised polygons & original polygons

par(mar=c(0,0,0,0))

plot(points1, col = "grey", axes = FALSE, xaxt='n', ann=FALSE, asp= 1)

plot(poly1, lwd = 1.5, add = T)

plot(tracts, border = "red", add = T)
```

However, regarding tmap … it can be done!

```
# first convert the point matrix to sp format

points1.sp <- SpatialPointsDataFrame(points1[,1:2], data = data.frame(points1[,3]))

# then pplot

tm_shape(poly1) + tm_borders(col = "black") +tm_shape(tracts) + tm_borders(col =
"red") +tm_shape(points1.sp) + tm_dots(col = "grey", shape = 1) +tm_layout(frame = F)
```

5.9 Introduction to Raster Analysis

This section provides the briefest of overviews of how raster data may be manipulated and overlaid in R in a similar way to a standard GUI GIS such as QGIS. This section will cover the reclassification of raster data as a precursor to some basic methods for performing what is sometimes referred to as *map algebra*, using a *raster calculator* or *raster overlay*. As a reminder, many packages include user guides in the form of a PDF document describing the package. This is listed at the top of the package index page. The raster package includes example code for the creation of raster data and different types of multi-layered raster composites. These will not be covered in this section. Rather, the coded examples illustrate some basic methods for manipulating and analysing raster layers in a similar way to what is often referred to as *sieve mapping*, *multi-criteria evaluation* or *multi-criteria analysis*. In these, different layers are combined to identify locations that have specific combinations of properties, such as height above sea level > 200 m AND soil_type is 'good'.

Raster analysis requires that the different input data have a number of

characteristics in common: typically they should cover the same spatial extent, have the same spatial resolution (grid or cell size), and, as with data for any spatial analysis, they should have the same projection or coordinate system. The data layers used in the example code in this section all have these properties. When you come to develop your own analyses, you may have to perform some manipulation of the data prior to analysis to ensure that your data also have these properties.

5.9.1 Raster Data Preparation

The Meuse data in the sp package will be used to illustrate the functions below. You could read in your raster data using the readGDAL function in the rgdal package, which provides an excellent R interface into the Geospatial Data Abstraction Library (GDAL). This has been described as the 'swiss army knife for spatial data' (https://cran.r-project.org/web/packages/sf/vignettes/sf2.html) as it is able to read or write vector and raster data of all file formats. You can inspect the properties and attributes of the Meuse data by examining the associated help files ? meuse.grid.

```
library(GISTools)

library(raster)

library(sp)

# load the meuse.grid data

data(meuse.grid)

# create a SpatialPixels DF object

coordinates(meuse.grid) <- ~x+y

proj4string(meuse.grid) <- CRS("+init=epsg:28992")

meuse.grid <- as(meuse.grid, "SpatialPixelsDataFrame")

# create 3 raster layers

r1 <- raster(meuse.grid, layer = 3) #dist

r2 <- raster(meuse.grid, layer = 4) #soil
```

```
r3 <- raster(meuse.grid, layer = 5) #ffreq
```

The code above loads the meuse.grid data, converts it to a SpatialPixelsDataFrame format and then creates three separate raster layers in the raster format. These three layers will form the basis of the analyses in this section. You could visually inspect their attributes by using some simple image commands:

```
# set the plot parameters for 3 rows

par(mfrow = c(1,3))

image(r1, asp = 1)

image(r2, asp = 1)

image(r3, asp = 1)

# reset par

par(mfrow = c(1,1))
```

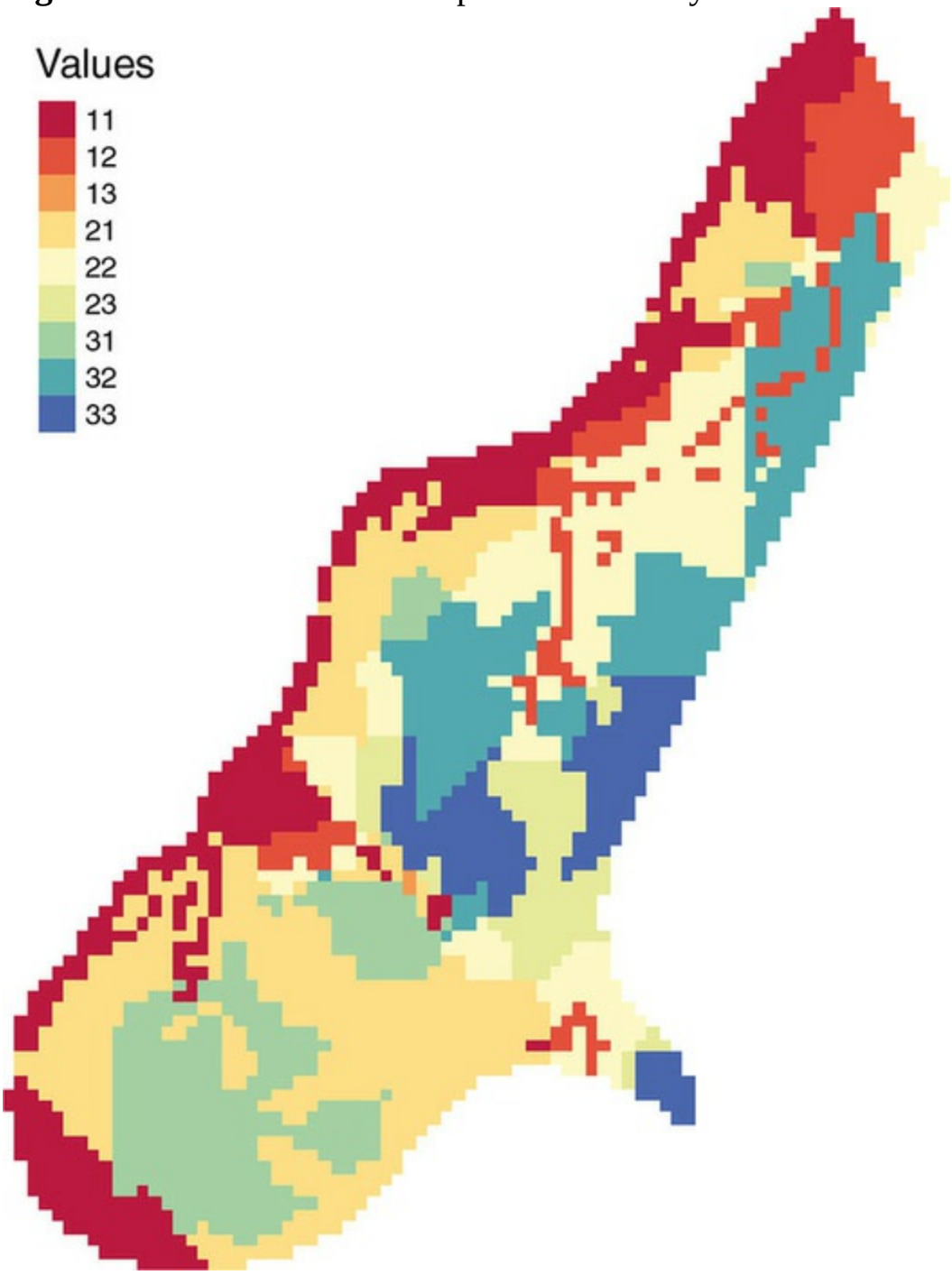
5.9.2 Raster Reclassification

Raster analyses frequently employ simple numerical and mathematical operations. In essence, they allow you to add, multiply, subtract, etc., raster data layers, and these operations are performed on a cell-by-cell basis. So for an addition this might be in the form:

```
Raster_Result <- Raster.Layer.1 + Raster.Layer.2
```

Remembering that raster data are numerical, if the Raster.Layer.1 and Raster.Layer.2 data both contained the values 1, 2 and 3, it would be difficult to know the origin, for example, of a value of 3 in the Raster_Result output. Specifically, if the r2 and r3 layers created above are considered, these both contain values in the range 1–3 describing soil types and flooding frequency, respectively (as described in the help for the meuse.grid data). Therefore we may wish to reclassify them in some way to understand the results of any combination or overlay operation.

Figure 5.14 The result of a simple raster overlay

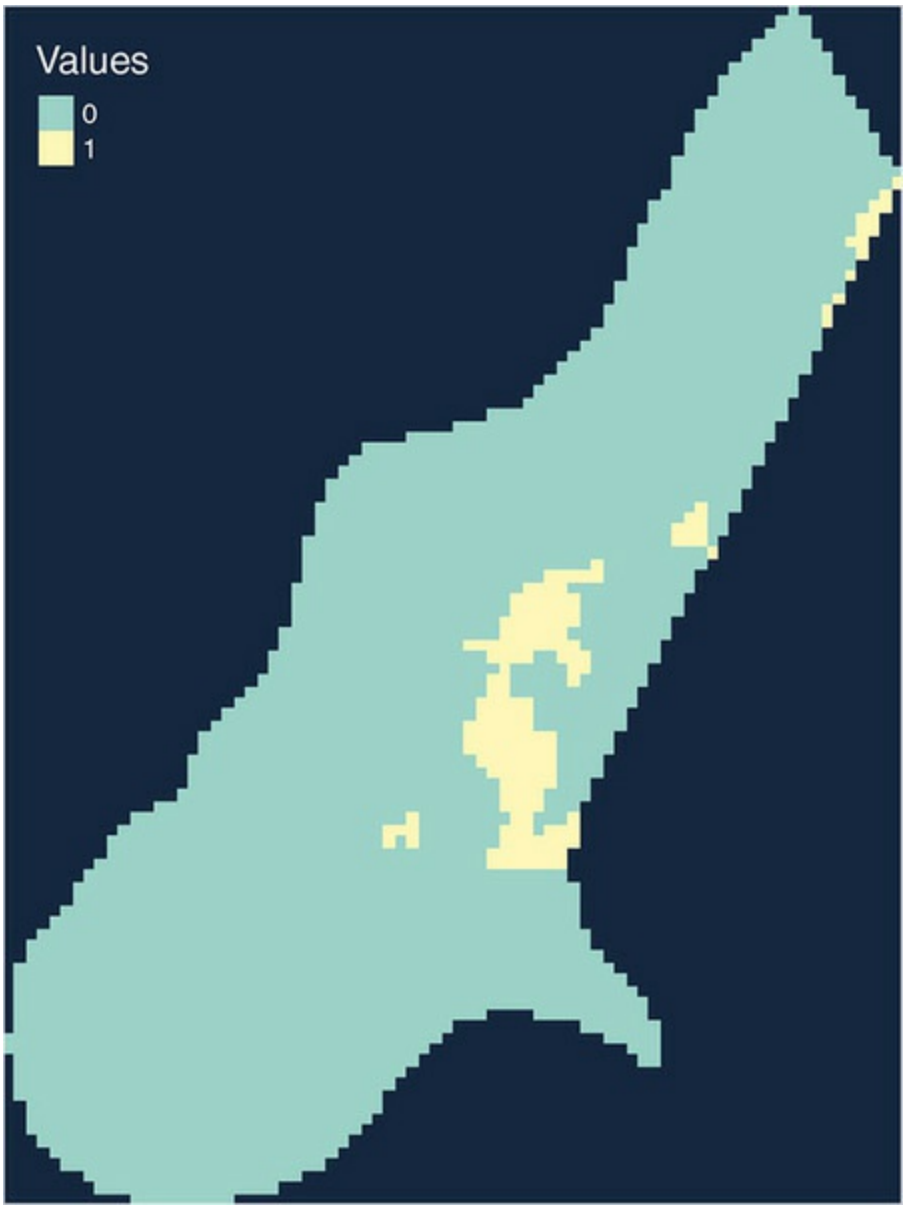


It is possible to reclassify raster data in a number of ways. First, the raster data values can be manipulated using simple mathematical operations. These produce raster outputs describing the mathematical combination of the input raster layers. The code below multiplies one of the layers by 10. This means that the result combining both raster data layers using the add (+) function

contains a fixed set of values (in this case 9) which are tractable to the combinations of inputs used. A value of 32 would indicate values of 3 in r3 (a flooding frequency of ‘one in 50 years’) and 2 in r2 (a soil type of ‘Rd90C/VII’, whatever that is). The results of this simple overlay are shown in Figure 5.14 and in the table of values printed.

```
Raster_Result <- r2 + (r3 * 10)

table(getValues(Raster_Result))

11     12 13     21     22     23     31     32     33

535    242     2    736    450    149    394    392    203

tm_shape(Raster_Result) + tm_raster(col = "layer", title = "Values", palette = "Spectral",
style = "cat") +

tm_layout(frame = F)
```

Figure 5.15 A raster overlay using a combinatorial AND



A second approach to reclassifying raster data is to employ logical operations on the data layers prior to combining them. These return TRUE or FALSE for each raster grid cell, depending on whether it satisfies the logical condition. The resultant layers can then be combined in mathematical operations as above. For example, consider the analysis that wanted to identify the locations in the Meuse data that satisfied the following conditions:

- Are greater than half of the rescaled distance away from the Meuse River

- Have a soil class of 1, that is calcareous weakly developed meadow soils, light sandy clay
- Have a flooding frequency class of 3, namely once in a 50-year period

The following logical operations can be used to do this:

```
r1a <- r1 > 0.5

r2a <- r2 >= 2

r3a <- r3 < 3
```

These can then be combined using specific mathematical operations, depending on the analysis. For example, a simple suitability multi-criteria evaluation, where all the conditions have to be true and where a crisp, Boolean output is required, would be coded using the multiplication function as follows, with the result shown in Figure 5.15:

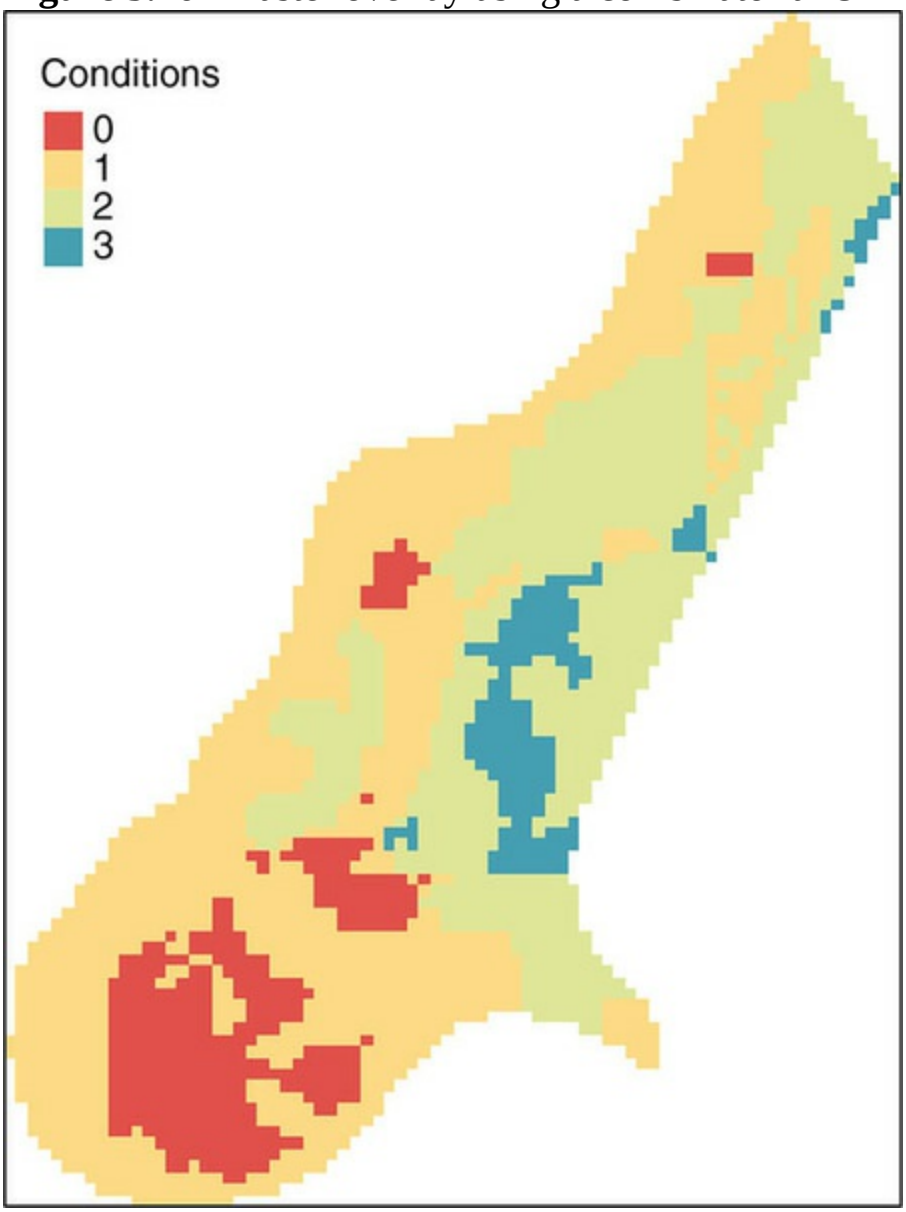
```
Raster_Result <- r1a * r2a * r3a

table(getValues(Raster_Result))

0       1

2924    179

tm_shape(Raster_Result) +

tm_raster(title = "Values", style = "cat") +tm_style("cobalt")
```

This is equivalent to a combinatorial AND operation, also known as an *intersection*. Alternatively, the analysis may be interested in identifying where any of the conditions are true, a combinatorial OR also known as a *union*, with a different result as shown in Figure 5.16:

```
Raster_Result <- r1a + r2a + r3a

table(getValues(Raster_Result))

0          1        2       3

386    1526    1012    179
```

```
# plot the result and add a legend

tm_shape(Raster_Result) + tm_raster(title ="Conditions", style = "cat"), palette =
"Spectral")

#tm_layout(frame = F, bg.color = "grey85")

tm_style_col_blind()
```

Figure 5.16 A raster overlay using a combinatorial OR



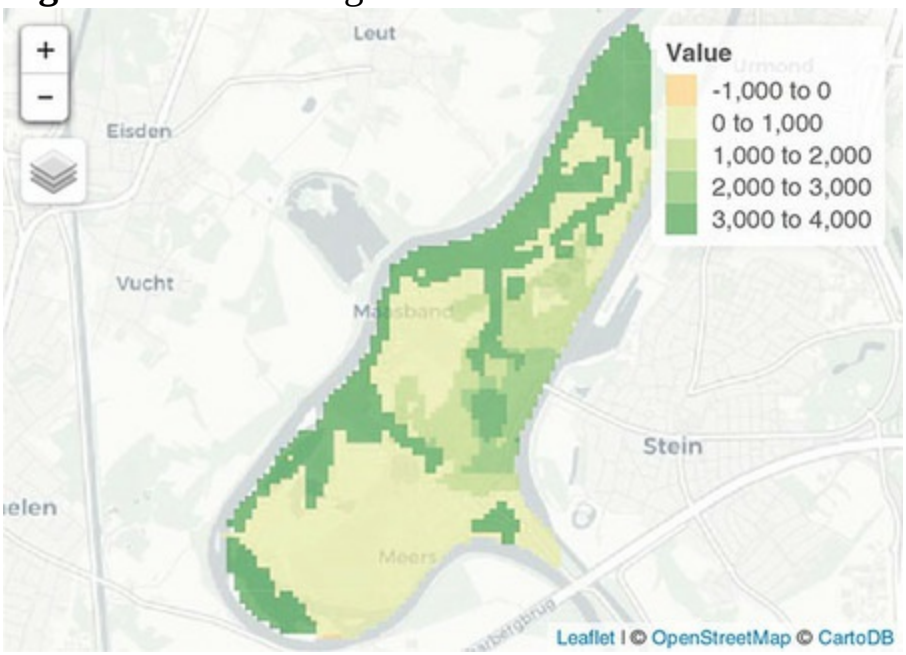
5.9.3 Other Raster Calculations

The above examples illustrated code to reclassify raster layers and then combined them using simple mathematical operations. You should note that it is possible to apply any kind of mathematical function to a raster layer. For example:

```
Raster_Result <- sin(r3) + sqrt(r1)

Raster_Result <- ((r1 * 1000 ) / log(r3) ) * r2

tmap_mode('view')

tm_shape(Raster_Result) + tm_raster(col = "layer", title = "Value")

tmap_mode("plot")
```

which produces Figure 5.17.

Figure 5.17 A raster generated from a number of mathematical operations



A number of other operations are possible using different functions included in the raster package. They are not given a full treatment here, but are introduced such that the interested reader can explore them in more detail.

The calc function performs a computation over a single raster layer, in a similar manner to the mathematical operations in the preceding text. The advantage of the calc function is that it should be faster when computing

more complex operations over large raster datasets.

```
my.func <- function(x) {log(x)}

Raster_Result <- calc(r3, my.func)

# this is equivalent to

Raster_Result <- calc(r3, log)
```

The overlay function provides an alternative to the mathematical operations illustrated in the reclassification examples above for combining multiple raster layers. The advantage of the overlay function, again, is that it is more efficient for performing computations over large raster objects.

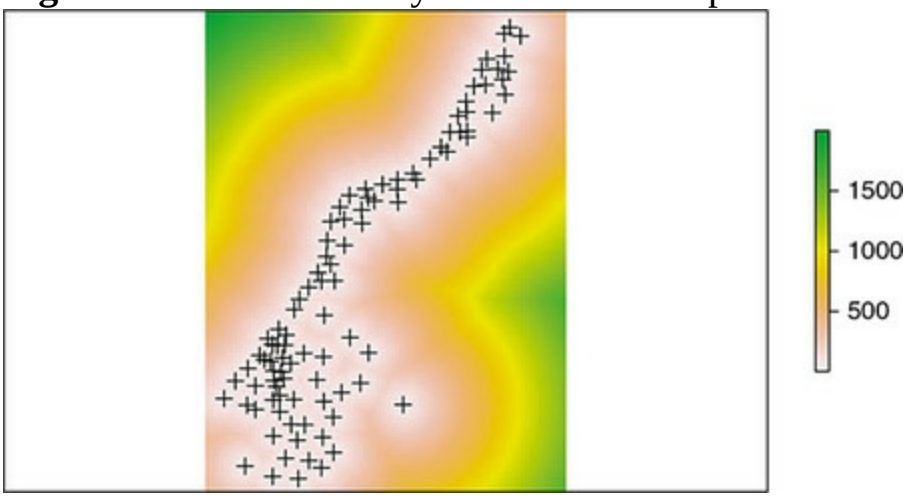
```
Raster_Result <- overlay(r2,r3,

fun = function(x, y) {return(x + (y * 10))} )

# alternatively using a stack

my.stack <- stack(r2, r3)

Raster_Result <- overlay(my.stack, fun = function(x, y) (x + (y * 10)) )
```

There are a number of distance functions for computing distances to specific features. The distanceFromPoints function calculates the distance between a set of points to all cells in a raster surface and produces a distance or cost surface as in Figure 5.18.

```
# load meuse and convert to points

data(meuse)

coordinates(meuse) <- ~x+y

# select a point layer

soil.1 <- meuse[meuse$soil == 1,]

# create an empty raster layer

# this is based on the extent of meuse

r <- raster(meuse.grid)
```

```
dist <- distanceFromPoints(r, soil.1)

plot(dist,asp = 1, xlab='',ylab='',xaxt='n',yaxt='n',bty='n', axes =F)

plot(soil.1, add = T)

# the tmap version but this is not as nice as plot

# tm_shape(dist) + tm_raster(palette = rev(terrain.colors(10)),

#        title = "Distance", style = "kmeans") +

#     tm_layout(frame = F, legend.outside = T)
```

Figure 5.18 A raster analysis of distance to points



You are encouraged to explore the raster package (and indeed the sp package) in more detail if you are specifically interested in raster-based analyses. There are a number of other distance functions, functions for computing over neighbourhoods (focal functions), accessing raster cell values and assessing spatial configurations of raster layers.

5.10 Answers to Self-Test Questions

Q1: Produce maps of the densities of breaches of the peace in each census block in New Haven in breaches per square kilometre. First, using sf formats:

```
# convert to sf

breach_sf <- st_as_sf(breach)
```

```
blocks_sf <- st_as_sf(blocks)

# point in polygon

b.count <- rowSums(st_contains(blocks_sf,breach_sf,sparse = F))

# area calculation

b.area <- ft2miles(ft2miles(st_area(blocks_sf))) * 2.58999

# combine and assign to the blocks data

blocks_sf$b.p.sqkm <- as.vector(b.count/b.area)

# map

tm_shape(blocks_sf) +

tm_polygons("b.p.sqkm", style = "kmeans", title ="")
```

Second, using sp formats:

```
# point in polygon

b.count <- poly.counts(breach, blocks)

# area calculation

b.area <- ft2miles(ft2miles(gArea(blocks, byid = T))) * 2.58999

# combine and assign to the blocks data

blocks$b.p.sqkm <- b.count/b.area

tm_shape(blocks) + tm_polygons("b.p.sqkm", style = "kmeans", title ="")
```

Q2: Produce a map of the outlying residuals using tm_shape functions etc. from the tmap package.

```
blocks$s.resids.2 <- s.resids.2

tm_shape(blocks) +

tm_polygons("s.resids.2", breaks = c(−8,−2,2,8), auto.palette.mapping = F,

palette = resid.shades$cols)
```

Q3: Determine the coefficients *a* and *b* for two different analyses using

blocks and tracts data and comment on the difference between the analyses using different areal units. First, calculate the coefficients for the analysis using census blocks:

```
# Analysis with blocks

blocks2 = blocks[blocks$OCCUPIED > 0,]

attach(data.frame(blocks2))

forced.rate = 2000*poly.counts(burgres.f,blocks2)/OCCUPIED

notforced.rate = 2000*poly.counts(burgres.n,blocks2)/OCCUPIED

model1 = lm(forced.rate~notforced.rate)

coef(model1)
```

(Intercept) notforced.rate

```
5.4667222          0.3789628

detach(data.frame(blocks2))
```

The results can be printed out:

```
# from the model

coef(model1)

# or in a formatted statement

cat("expected(forced rate)=",coef(model1)[1], "+",

coef(model1)[2], "* (not forced rate)")
```

Now calculate the coefficients using census tracts:

```
# analysis with tracts

tracts2 = tracts[tracts$OCCUPIED > 0,]

# align the projections
```

```
ct <- proj4string(burgres.f)

proj4string(tracts2) <- CRS(ct)

# now do the analysis

attach(data.frame(tracts2))

forced.rate = 2000*poly.counts(burgres.f,tracts2)/OCCUPIED

notforced.rate = 2000*poly.counts(burgres.n,tracts2)/OCCUPIED

model2=lm(forced.rate~notforced.rate)detach(data.frame(tracts2))
```

Again the results can be printed out:

```
# from the model

coef(model2)

# or in a formatted statement

cat("expected(forced rate) = ",coef(model2)[1], "+",

coef(model2)[2], "* (not forced rate)")
```

These two analyses show that, in this case, there are only small differences between the coefficients arising from analyses using different areal units. Print out both results:

```
cat("expected(forced rate) = ",coef(model1)[1], "+", coef(model1)[2], "* (not forced rate)")

cat("expected(forced rate) = ",coef(model2)[1], "+", coef(model2)[2], "* (not forced rate)")

expected(forced rate) = 5.466722 + 0.3789628 * (not forced rate)

expected(forced rate) = 5.243477 + 0.4132951 * (not forced rate)
```

This analysis tests what is referred to as the modifiable areal unit problem, first identified in the 1930s, and extensively research by Stan Openshaw in the 1970s and beyond – see Openshaw (1984) for a comprehensive review. Variability in analyses can arise when data are summarised over different spatial units and the importance of the modifiable areal unit problem cannot be overstated as a critical consideration in spatial analysis.

Q4: Write a function that will return an intersected dataset, with an attribute of counts of some variable (houses, population, etc.) as held in another sf format dataset.

```
int.count.function <- function(
int_sf, layer_sf, int.ID, layer.ID, target.var) {
# Use the IDs to assign ID variables to both inputs
# this makes the processing easier later on
int_sf$IntID <- as.vector(data.frame(int_sf[, int.ID])[,1])
layer_sf$LayerID <- as.vector(data.frame(layer_sf[, layer.ID])[,1])
# do the same for the target.var
layer_sf$target.var<-as.vector(data.frame(layer_sf[,target.var])[,1])
# check projections
if(st_crs(int_sf) != st_crs(layer_sf))
print("Check Projections!!!")
# do intersection
int.res_sf <- st_intersection(int_sf, layer_sf)
# generate area and proportions
int.areas <- st_area(int.res_sf)
layer.areas <- st_area(layer_sf)
# match tract area to the new layer
v1 <- as.vector(data.frame(int.res_sf$LayerID)[,1])
v2 <- as.vector(data.frame(layer_sf$LayerID)[,1])
index <- match(v1, v2)
layer.areas <- layer.areas[index]
layer.prop <- as.vector(int.areas/as.vector(layer.areas))
# create a variable of intersected values
int.res_sf$NewVar <-
```

```
as.vector(data.frame(layer_sf$target.var)[,1][index]) * layer.prop

summarise this and link back to the int.layer_sf

# NewVar <- summarise(group_by(int.res_sf, IntID), count = sum(NewVar))

# create an empty vector

int.layer_sf$NewVar <- 0

# and populate this using ID as the index

int.layer_sf$NewVar[NewVar$IntID] <- NewVar$count

return(int.layer_sf)

}
```

You can test this:

```
# convert blocks to sf

blocks_sf <- st_as_sf(blocks)

# run the function

test.res <- int.count.function(

int_sf <- int.layer_sf,

layer_sf <- blocks_sf,

int.ID <- "ID",

layer.ID <- "NEWH075H_I",

target.var <- "POP1990")

plot(test.res[,"NewVar"])
```

References

Bivand, R.S., Pebesma, E.J. and Gómez-Rubio, V. (2013) Applied Spatial Data: Analysis with R, 2nd edition. New York: Springer.

Comber, A.J., Brunsdon, C. and Green, E. (2008) Using a GIS-based network analysis to determine urban greenspace accessibility for different ethnic

and religious groups. Landscape and Urban Planning, 86: 103–114.

Hijmans, R.J. and van Etten, J. (2014) Raster: Geographic data analysis and modeling. R Package Version 2.6–7. http://cran.r-project.org/package=raster.

Openshaw, S. (1984) The Modifiable Areal Unit Problem, CATMOG 38, Geo Abstracts, Norwich. https://www.uio.no/studier/emner/sv/iss/SGO9010/openshaw1983.pdf.

6 Point Pattern Analysis Using R

6.1 Introduction

In this and the next chapter, some key ideas of spatial statistics will be outlined, together with examples of statistical analysis based on these ideas, via R. The two main areas of spatial statistics that are covered are those relating to *point patterns* (this chapter) and *spatially referenced attributes* (next chapter). One of the characteristics of R, as open source software, is that R packages are contributed by a variety of authors, each using their own individual styles of programming. In particular, for point pattern analysis the spatstat package is often used, while for spatially referenced attributed, spdep is favoured. One the one hand spdep handles spatial data in the same way as sp, maptools and GISTools, while on the other hand spatstat does not. Also, for certain specific tasks, other packages may be called upon whose mode of working differs from either of these packages. While this may seem a daunting prospect, the aim of these two chapters is to introduce the key ideas of spatial statistics, as well as providing guidance in the choice of packages, and help in converting data formats. Fortunately, although some packages use different data formats, conversion is generally straightforward, and examples will appear throughout the chapters, whenever necessary.

6.2 What Is Special About Spatial?

In one sense, the motivations for statistical analysis of spatial data are the same as those for non-spatial data:

- To explore and visualise the data
- To create and calibrate models of the process generating the data
- To test hypotheses related to the processes generating the data

However, a number of these requirements are strongly influenced by the nature of spatial data. The study of mapping and cartography may be regarded as an entire subject area within the discipline of information

visualisation, which focuses exclusively on geographical information. In addition, the kinds of hypotheses one might associate with spatial data are quite distinctive – for example, focusing on the detection and location of spatial clusters of events, or on whether two kinds of event (say, two different types of crime) have the same spatial distribution. Similarly, models that are appropriate for spatial data are distinctive, in that they often have to allow for spatial autocorrelation in their random component – for example, a regression model generally includes a random error term, but if the data are spatially referenced, one might expect nearby errors to be correlated. This differs from a 'standard' regression model where each error term is considered to apply independently, regardless of location. In the remainder of this section, point patterns (one of two key types of spatial data considered in this book) will be considered. First, these will be described.

6.2.1 Point Patterns

Point patterns are collections of geographical points assumed to have been generated by a random process. In this case, the focus of inference and modelling is on model(s) of the random processes and their comparison. Typically, a point dataset consists of a set of observed (x, y) coordinates, say $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where n is the number of observations. As an alternative notation, each point could be denoted by a vector $\mathbf{x}_i$, where $\mathbf{x}_i = (x_i, y_i)$. Using the data formats used in sp, maptools and so on, these data could be represented as SpatialPoints or SpatialPointsDataFrame objects. Since these data are seen as random, many models are concerned with the probability densities of the random points, $\nu(x_i)$.

Another area of interest is the *interrelation* between the points. One way of thinking about this is to consider the probability density of one point $\mathbf{x}i$ conditional on the remaining points $\{\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n\}$. In some situations $\mathbf{x}i$ is independent of the other points. However, for other processes this is not the case. For example, if $\mathbf{x}i$ is the location of the reported address for a contagious disease, then it is more likely to occur near one of the points in the dataset (due to the nature of contagion), and therefore not

independent of the values of $\{\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n\}$.

Also important is the idea of a *marked process*. Here, random sets of points drawn from a number of different populations are superimposed (e.g. household burglaries using force and household burglaries not using force) and the relationship between the different sets is considered. The term 'marked' is used here as the dataset can be viewed as a set of points where each point is tagged (or marked) with its parent population. Using the data formats used by sp, a marked process could be represented as a spatial points data frame – although the spatstat package uses a different format.

6.3 Techniques for Point Patterns Using R

Having outlined the two main data types that will be considered, and the kinds of model that may be applied, in this section more specific techniques will be discussed, with examples of how they may be carried out using R. In this section, we will focus on random point patterns.

6.3.1 Kernel Density Estimates

The simplest way to consider random two-dimensional point patterns is to assume that each random location $\mathbf{x}i$ is drawn independently from an unknown distribution with probability density function $f(\mathbf{x}_i)$. This function maps a location (represented as a two-dimensional vector) onto a probability density. If we think of locations in space as a very fine pixel grid, and assume a value of probability density is assigned to each pixel, then summing the pixels making up an arbitrary region on the map gives the probability that an event occurs in that area. It is generally more practical to assume an *unknown f*, rather than, say, a Gaussian distribution, since geographical patterns often take on fairly arbitrary shapes – for example, when applying the technique to patterns of public disorder, areas of raised risk will occur in a number of locations around a city, rather than a simplistic radial 'bell curve' centred on the city's mid-point.

A common technique used to estimate $f(\mathbf{x}_i)$ is the *kernel density estimate*

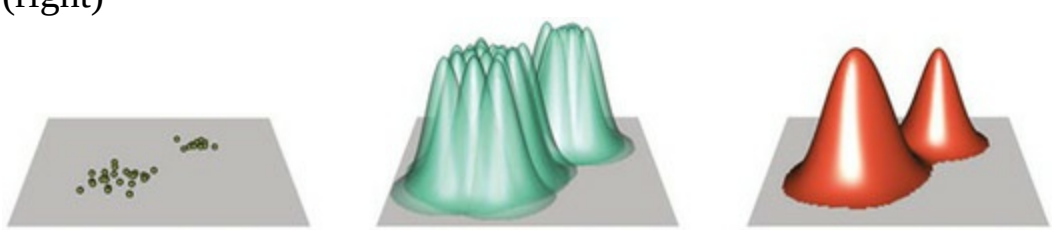
(KDE: Silverman, 1986). KDEs operate by averaging a series of small 'bumps' (probability distributions in two dimensions, in fact) centred on each observed point. This is illustrated in Figure 6.1. In algebraic terms, the approximation to $f(\mathbf{x})$, for an arbitrary location $\mathbf{x} = (x, y)$, is given by

(6.1)

$$\hat{f}(\mathbf{x}) = \hat{f}(x,y) = \frac{1}{nh_x h_y} \sum_i k\left(\frac{x - x_i}{h_x}, \frac{y - y_i}{h_y}\right) \qquad (6.1)$$

Each of the 'bumps' (central panel in Figure 6.1) map onto the kernel function $k\left(\frac{x - x_i}{h_x}, \frac{y - y_i}{h_y}\right)$ in equation (6.1) and the entire equation describes the 'bump averaging' process, leading to the estimate of probability density in the right-hand panel. Note that there are also parameters *hx* and *hy* (frequently referred to as the *bandwidths*) in the *x* and *y* directions; their dimension is length, and they represent the radii of the bumps in each direction. Varying *hx* and *hy* alters the shape of the estimated probability density surface – in brief, low values of *hx* and *hy* lead to very 'spiky' distribution estimates, and very high values, possibly larger than the span of the $\mathbf{x}_i$ locations, tend to 'flatten' the estimate so it appears to resemble the *k*-function itself; effectively this gives a superposition of nearly identical *k*-functions with relatively small perturbations in their centre points.

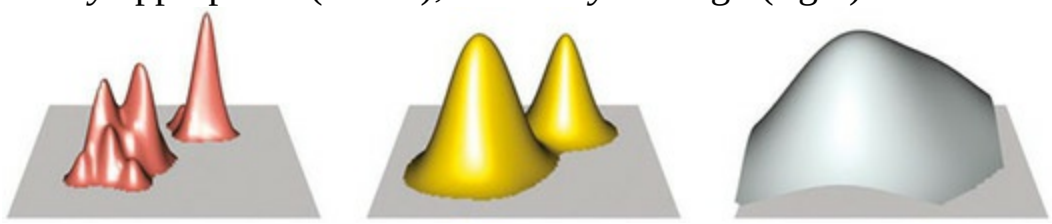
Figure 6.1 Kernel density estimation: initial points (left); bump centred on each point (centre); average of bumps giving estimate of probability density (right)



This effect of varying *hx* and *hy* is shown in Figure 6.2. Typically *hx* and *hy* take similar values. If one of these values is very different in magnitude than the other, kernels elongated in either the *x* or *y* direction result. Although this may be useful when there are strong directional effects, we will focus on the

situation where values are similar for the examples discussed here. To illustrate the results of varying the bandwidths, the same set of points used in Figure 6.1 is used to provide KDEs with three different values of *hx* and *hy*: on the left, they both take a very low value, giving a large number of peaks; in the centre, there are two peaks; and on the right, only one.

Figure 6.2 Kernel density estimation bandwidths: *hx* and *hy* too low (left); *hx* and *hy* appropriate (centre); *hx* and *hy* too high (right)



An obvious problem is that of choosing appropriate *hx* and *hy* given a dataset {**x***i*}. There are a number of formulae to provide 'automatic' choices, as well as some more sophisticated algorithms. Here, a simple rule is used, as proposed by Bowman and Azzalini (1997) and Scott (1992):

(6.2)

$$h_x = \sigma_x \left(\frac{2}{3n} \right)^{1/6} \qquad (6.2)$$

where *σx* is the standard deviation of the *xi*. A similar formula exists for *hy*, replacing *σx* with *σy*, the standard deviation of the *yi*. The central KDE in Figure 6.2 is based on choosing *hx* and *hy* using this method.

6.3.2 Kernel Density Estimation Using R

Here, the breaches of the peace (public disturbances) in New Haven, Connecticut are used as an example; recall that this is provided in the GISTools package, here loaded using data(newhaven). As an initial inspection of the data, look at the locations of breaches of the peace. These can be viewed on an interactive map using the tmap package in view mode. The following code loads the New Haven data and tmap, sets R in view mode and produces a map showing the US Census block boundaries and the locations of breach of the peace, on a backdrop of a *CartoDB* map, provided

your computer is linked to the internet. The two layers can be interactively switched on or off, and the backdrop can be changed. Here, we will generally use the default backdrop as it is monochrome, and the information to be mapped will be in colour. The initial map window is seen in Figure 6.3.

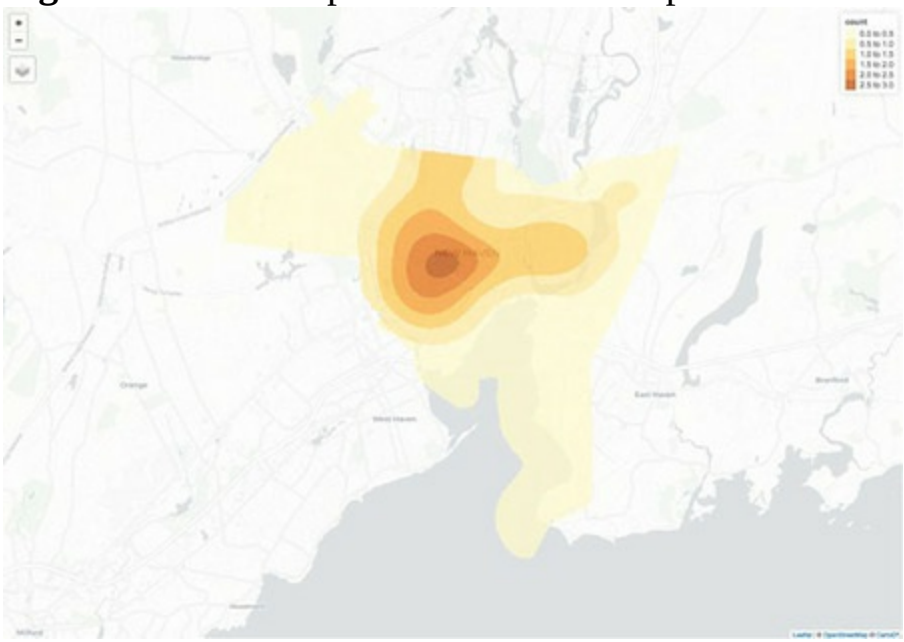
```
# Load GISTools (for the data) and tmap (for the mapping)

require(GISTools)

require(tmap)

# Get the data

data(newhaven)

# look at it

# select 'view' mode

tmap_mode('view')

# Create the map of blocks and incidents

tm_shape(blocks) + tm_borders() + tm_shape(breach) +    tm_dots(col='navyblue')
```

Figure 6.3 Web view mode of tmap

There are a number of packages in R that provide code for computing KDEs. Here, the tmap and tmaptools libraries provide some very useful tools. The

function to compute kernel density estimation is map_smooth from tmaptools. This estimates the value of the density over a grid of points, and returns the result as a list – a raster object – referred to as X$raster (where X is the value returned from map_smooth), a contour object (X$iso) and a polygon object (X$polygon). The first of these is a raster grid of values for the KDEs, and the second and third relate to contour lines associated with the KDE; iso provides a set of lines (the contour lines) which may be plotted. Similarly, the polygons item provides a solid list of polygons that may be plotted (as filled polygons). map_smooth takes several arguments (most notably the set of points to use for the KDE) but also a number of optional arguments. Two key ones here are the bandwidth and the cover. The bandwidth is a vector of length 2 containing *hx* and *hy*, and the cover is a geographical object whose outline forms the boundary of the locations where the KDE is estimated. Both of these have defaults: the default bandwidth is $\frac{1}{50}$ times the shortest side of the bounding box of the points, and the default cover is the bounding box of the points. However, as discussed earlier, more appropriate *hx* and *hy* values may be found using (6.2). This is not provided as part of smooth_map, but a function is easily written. The division of the result by 1000 is because the projected data are measured in metres, but smooth_map expects bandwidths in kilometres.

Figure 6.4 KDE map for breaches of the peace



```
# Function to choose bandwidth according to Bowman and Azzalini / Scott's rule

# for use with smooth_map in tmaptools

choose_bw <- function(spdf) {

X <- coordinates(spdf)

sigma <- c(sd(X[,1]),sd(X[,2]))    * (2 / (3 * nrow(X))) ^ (1/6)

return(sigma/1000)

}
```

Now the code to carry out the KDE and plot the results may be used. Here the raster version of the result is used, and plotted on a web mapping backdrop (Figure 6.4).

```
library(tmaptools)

tmap_mode('view')

breach_dens <- smooth_map(breach,cover=blocks, bandwidth = choose_bw(breach))

tm_shape(breach_dens$raster) + tm_raster()
```

The 'count' caption here indicates that the probability densities have been rescaled to represent intensities – by multiplying the KDE by the number of cases. With this scale, the quantity being mapped is the expected number of cases per unit area in the amount of time of the study period.

It is also possible to use the other forms of result (polygons or isolines) to plot the KDE outcomes. In the following code, isolines are produced, again with a backdrop of a web map (see Figure 6.5).

Figure 6.5 KDE map for breaches of the peace – isoline version

```
tmap_mode('view')

tm_shape(blocks)+ tm_borders(alpha=0.5) +    tm_shape(breach_dens$iso) +
tm_lines(col='darkred',lwd=2)
```

Here, a backdrop of block boundaries has also been added to emphasise the limits of the data collection region. In this and the previous map, it is important to be aware of the boundaries of the data sampling region. Low probability densities outside this region are quite likely due to no data being collected there – not necessarily low incident risk!

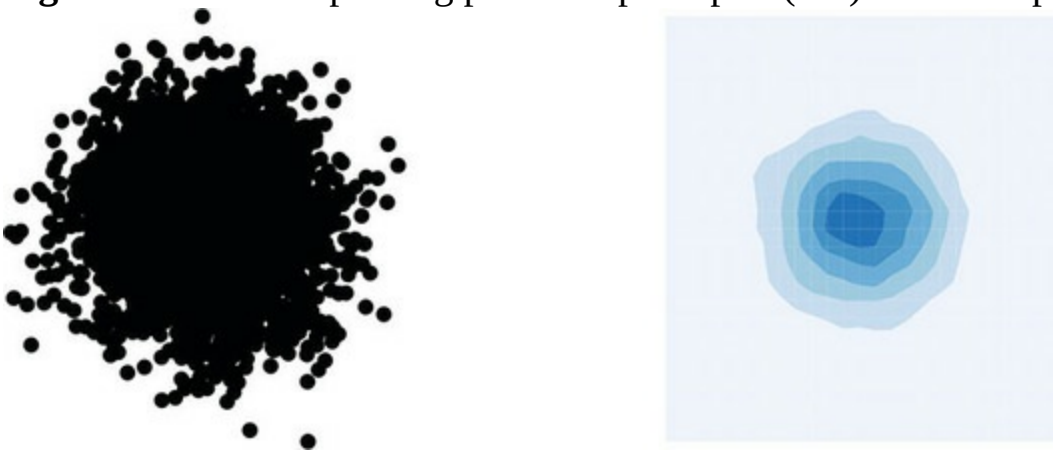
Self-Test Question 1. As a further exercise, create the *polygons* version of the KDE map in the plot mode of tmap – the tm_fill() function will shade the polygons. As there will be no backdrop map, roads and blocks should be added to the map to provide context. Also, add a map scale.

I

As well as estimating the probability density function $f(x, y)$, kernel density estimation also provides a helpful visual tool for displaying point data. Although plotting point data directly can show all of the information in a small dataset, if the dataset is larger it is hard to discriminate between relative densities of points: essentially, when points are very closely packed, the map symbols begin to overprint and exact numbers are hard to determine; this is illustrated in Figure 6.6. On the left is a plot of locations. The points plotted are drawn from a two-dimensional Gaussian distribution, and their relative density increases towards the

centre. However, except for a penumbral region, the intensity of the dot pattern appears to have roughly fixed density. As the KDE estimates relative density, this problem is addressed – as may be seen in the KDE plot in Figure 6.6 (right).

Figure 6.6 The overplotting problem: point plot (left) and KDE plot (right)



6.4 Further Uses of Kernel Density Estimation

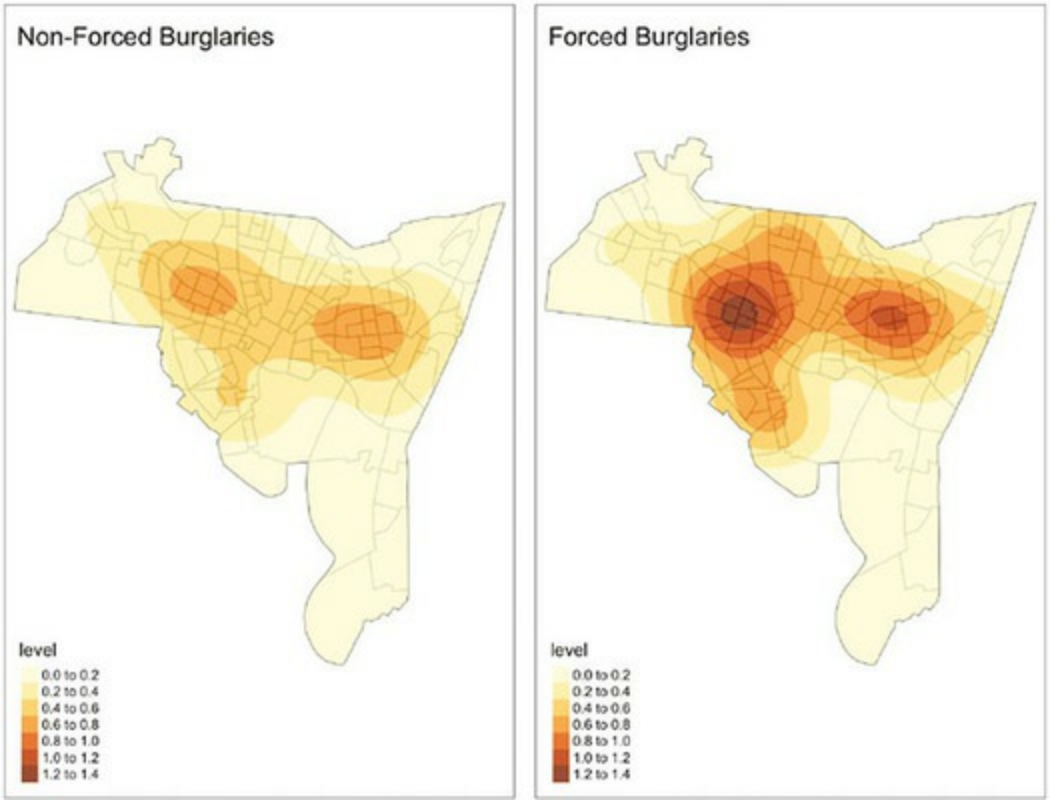
KDEs are also useful for comparative purposes. In the newhaven dataset there are also data relating to burglaries from residential properties. These are divided into two classes: burglaries involving forced entry, and burglaries that do not. It may be of interest to compare the spatial distributions of the two groups. In the newhaven dataset, burgres.f is a SpatialPoints object with points for the occurrence of forced entry residential burglaries, and burgres.n is a SpatialPoints object with points for non-forced entries. Based on the recommendation to compare patterns in data using small multiples of graphical panels (Tufte, 1990), KDE maps for forced and non-forced burglaries may be shown side by side. This is achieved using the R code below, which carries out the following operations:

- Specify a set of levels for the intensity contours. To allow comparison the same levels will be used on both maps
- Compute the KDEs. Here the contours are specified for the iso and polygons results
- Draw each of the two maps and store in variables dn and df. Here the polygon format is used
- Use tmap_arrange to draw the two maps in ‘small multiples’ format

The result is seen in Figure 6.7. Although there are some similarities in the two patterns – likely due to the underlying pattern of housing – it may be seen that for the non-forced entries there are two peaks of roughly equal intensity (Beaver Hills/Edgewood in the west and Fair Haven in the east), while for forced entries the peaks are in similar positions but the stronger peak is to the west, near Edgewood. More generally, there tend to be more forced incidents than non-forced.

```
# R Kernel Density comparison - first make sure the New Haven data are available

require(GISTools)

data(newhaven)

tmap_mode('plot')

# Create the KDEs for the two datasets:

contours <- seq(0,1.4,by=0.2)

brn_dens <- smooth_map(burgres.n,cover=blocks, breaks=contours, style='fixed',
bandwidth = choose_bw(burgres.n))

brf_dens <- smooth_map(burgres.f,cover=blocks, breaks=contours, style='fixed',
bandwidth = choose_bw(burgres.f))

# Create the maps and store them in variables

dn <- tm_shape(blocks) + tm_borders() +    tm_shape(brn_dens$polygons) +
tm_fill(alpha=0.8) +    tm_layout(title="Non-Forced Burglaries")

df <- tm_shape(blocks) + tm_borders() +    tm_shape(brf_dens$polygons) +
tm_fill(alpha=0.8) +    tm_layout(title="Forced Burglaries")

tmap_arrange(dn,df)
```

Figure 6.7 KDE maps to compare forced and non-forced burglary patterns



6.4.1 Hexagonal Binning Using R

An alternative visualisation tool for geographical point datasets with larger numbers of points is *hexagonal binning*. In this approach, a regular lattice of small hexagonal cells is overlaid on the point pattern, and the number of points in each cell is counted. The cells are then shaded according to the counts. This method also overcomes the overplotting problem. However, hexagonal binning is not directly available in GISTools, and it is necessary to use another package. One possibility is the fMultivar package. This provides a routine for hexagonal binning called hexBinning, which takes a two-column matrix of coordinates and provides an object representing the hexagonal grid and the counts of points in each hexagonal cell. Note that this function does not work directly with sp-type spatial data objects. This is mainly because it is designed to apply hexagonal binning to any kind of data (e.g. scatter plot points where the *x* and *y* variables are not geographical coordinates). However, it is perfectly acceptable to subject geographical points to this kind of analysis.

First, make sure that the fMultivar package is installed in R. If not, enter:

```
install.packages("fMultivar",depend=TRUE)
```

A complication here is that the result of the hexBinning function is not a Spatial-PolygonsDataFrame object and not immediately compatible with tmap and other spatial tools in R. To allow for this, a new function hexbin_map is written. This takes a SpatialPointsDataFrame object as input, and returns a SpatialPolygonsDataFrame object consisting of the hexagons in which one or more points occur, together with a data frame with a column z containing the count of points. The code works as follows:

- Extract coordinates from the SpatialPointsDataFrame object
- Run hexBinning on these
- Construct hexagonal polygon coordinates
- Loop through each polygon; construct these according to sp data structures
- Copy the map projection information from the SpatialPointsDataFrame object
- Add the count information giving a SpatialPolygonsDataFrame object

The code is below:

```
hexbin_map <- function(spdf, ...) {

hbins <- fMultivar::hexBinning(coordinates(spdf),...)

# Hex binning code block

# Set up the hexagons to plot, as polygons

u <- c(1, 0, −1, −1, 0, 1)

u <- u ∗ min(diff(unique(sort(hbins$x))))

v <- c(1,2,1,−1,−2,−1)

v <- v ∗ min(diff(unique(sort(hbins$y))))/3

# Construct each polygon in the sp model

hexes_list <- vector(length(hbins$x),mode='list')
```

```
for (i in 1:length(hbins$x)) {

pol <- Polygon(cbind(u + hbins$x[i], v + hbins$y[i]),hole=FALSE)

hexes_list[[i]] <- Polygons(list(pol),i) }

# Build the spatial polygons data frame

hex_cover_sp <\SpatialPolygons(hexes_list,proj4string=CRS(proj4string(spdf)))

hex_cover <- SpatialPolygonsDataFrame(hex_cover_sp,
data.frame(z=hbins$z),match.ID=FALSE)

# Return the result

return(hex_cover)

}
```

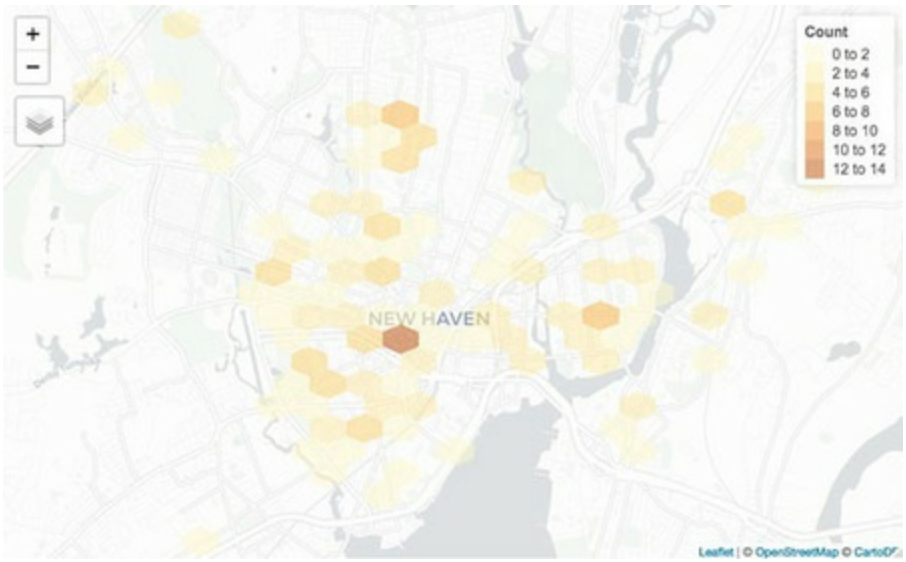
I

Note the reference to fMultivar::hexBinning in the code. This tells R to use the function hexBinning from the package fMultivar without actually loading the package using library. It is useful if it is the only thing used from that package, as it avoids having to load everything else in the package.

It is now possible to create hex binned maps via this function. Here a view mode map is the map of hex binned breach data (Figure 6.8).

```
tmap_mode('view')

breach_hex <- hexbin_map(breach,bins=20)

tm_shape(breach_hex) +    tm_fill(col='z',title='Count',alpha=0.7)
```

Figure 6.8 Hexagonal binning of residential burglaries



As an alternative graphical representation, it is also possible to draw hexagons whose area is proportional to the point count. This is done by creating a variable with which to multiply the relative polygon coordinates (this relates to the *square root* of the count in each polygon, since it is *areas* of the hexagons that should reflect the counts). This is all achieved via a modification of the previous hexbin_map function, called hexprop_map, listed below.

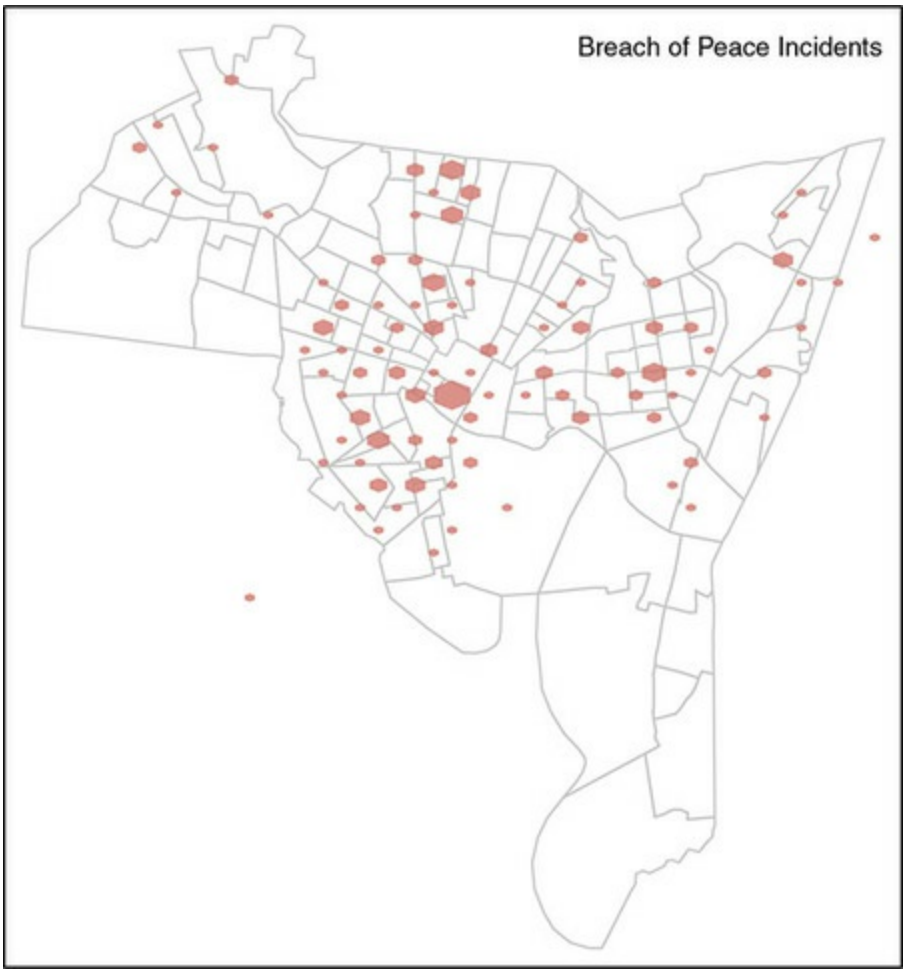
```
hexprop_map <- function(spdf, ...) {

hbins <- fMultivar::hexBinning(coordinates(spdf),...)

# Hex binning code block

# Set up the hexagons to plot, as polygons

u <- c(1, 0, −1, −1, 0, 1)

u <- u ∗ min(diff(unique(sort(hbins$x))))

v <- c(1,2,1,−1,−2,−1)

v <- v ∗ min(diff(unique(sort(hbins$y))))/3

scaler <- sqrt(hbins$z/max(hbins$z))

# Construct each polygon in the sp model

hexes_list <- vector(length(hbins$x),mode='list')

for (i in 1:length(hbins$x)) {
```

```
pol <- Polygon(cbind(u*scaler[i] + hbins$x[i], v*scaler[i] + hbins$y[i]),hole=FALSE)

hexes_list[[i]] <- Polygons(list(pol),i) }

# Build the spatial polygons data frame

hex_cover_sp <- SpatialPolygons(hexes_list,proj4string=CRS(proj4string(spdf)))

hex_cover <- SpatialPolygonsDataFrame(hex_cover_sp,
data.frame(z=hbins$z),match.ID=FALSE)

# Return the result

return(hex_cover)

}
```

It is now possible to create a proportional hex binning map – here in plot mode in Figure 6.9.

```
tmap_mode('plot')

breach_prop <- hexprop_map(breach,bins=20)

tm_shape(blocks) + tm_borders(col='grey') +

tm_shape(breach_prop) +tm_fill(col='indianred',alpha=0.7) +

tm_layout("Breach of Peace Incidents",title.position=c('left','bottom'))
```

Figure 6.9 Hexagonal binning of residential burglaries



6.5 Second-Order Analysis of Point Patterns

In this section an alternative approach to point patterns will be considered. Whereas KDEs assume that the spatial distributions for a set of points are independent but have a varying intensity, the second-order methods considered in this section assume that *marginal* distributions of points have a fixed intensity, but that the *joint* distribution of all points is such that individual distributions of points are not independent.[1] This process describes situations in which the occurrences of events are related in some way – for example, if a disease is contagious, the reporting of an incidence in one place might well be accompanied by other reports nearby. The *K*-function (Ripley, 1981) is a very useful tool for describing processes of this kind. The *K*-function is a function of distance, defined by

[1] A further stage in complication would be the situation where individual

distributions are not independent, but also the marginal distributions vary in intensity – however, this will not be considered here.

$$K(d) = \lambda^{-1}\mathrm{E}(N_d) \quad (6.3)$$

where *Nd* is the number of events **x***i* within a distance *d* of a randomly chosen event from all recorded events $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, and λ is the intensity of the process, measured in events per unit area. Consider the situation where the distributions of **x***i* are independent, and the marginal densities are uniform – often termed a Poisson process, or *complete spatial randomness* (CSR). In this situation one would expect the number of events within a distance *d* of a randomly chosen event to be the intensity λ multiplied by the area of a circle of radius *d*, so that

(6.4)

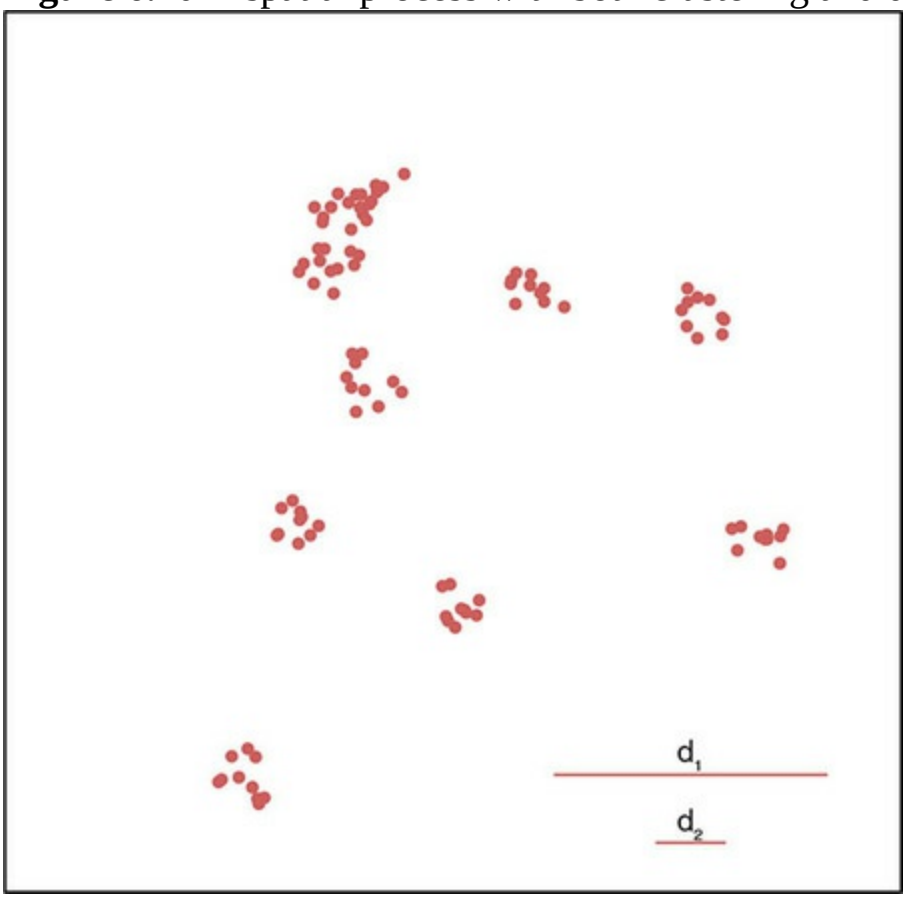
$$K_{\mathrm{CSR}}(d) = \pi d^2 \qquad (6.4)$$

The situation in equation (6.4) can be thought of as a benchmark to assess the clustering of other processes. For a given distance *d*, the function value $K_{\mathrm{CSR}}(d)$ gives an indication of the expected number of events found around a randomly chosen event, under the assumption of a uniform density with each observation being distributed independently of the others. Thus for a process having a *k*-function $k(d)$, if $k(d) > K_{\mathrm{CSR}}(d)$, this suggests that there is an excess of nearby points – or, to put it another way, there is clustering at the spatial scale associated with the distance *d*. Similarly, if $K(d) < K_{\mathrm{CSR}}(d)$, this suggests spatial dispersion at this scale – the presence of one point suggests other points are *less* likely to appear nearby than for a Poisson process.

The consideration of spatial scale is important (many processes exhibit spatial clustering at some scales, and dispersion at others) so that the quantity $K(d) - K_{\mathrm{CSR}}(d)$ may change sign with different values of *d*. For example, the process illustrated in Figure 6.10 shows clustering at low values of *d* – for small distances (such as d_2 in the figure) there is an excess of points near to other points compared to CSR, but for intermediate distances (such as d_1 in the

figure) there is an undercount of points.

Figure 6.10 A spatial process with both clustering and dispersion



When working with a sample of data points {**x***i*}, the *K*-function for the underlying distribution will not usually be known. In this case, an estimate must be made using the sample. If *dij* is the distance between **x***i* and **x***j* then an estimate of *K*(*d*) is given by

(6.5)

$$\hat{K}(d) = \hat{\lambda}^{-1} \sum_{i} \sum_{j \neq i} \frac{I(d_{ij} < d)}{n(n-1)} \qquad (6.5)$$

where $\hat{\lambda}$ is an estimate of the intensity given by

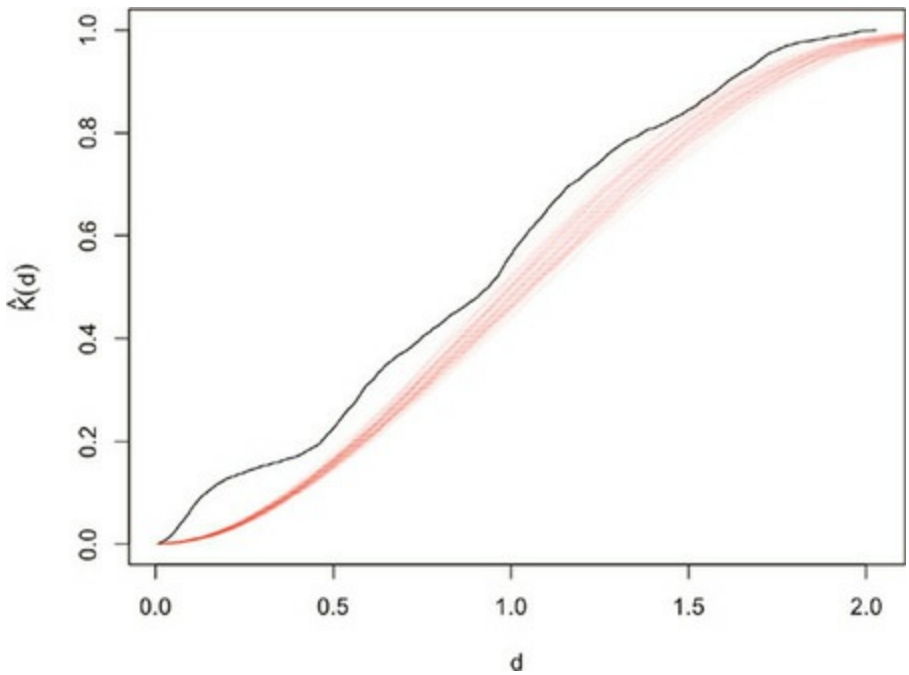
(6.6)

$$\hat{\lambda} = \frac{n}{|A|} \qquad (6.6)$$

$|A|$ being the area of a study region defined by a polygon A. Also $I(\cdot)$ is an indicator function taking the value 1 if the logical expression in the brackets is true, and 0 otherwise. To consider whether this sample comes from a clustered or dispersed process, it is helpful to compare $\hat{K}(d)$ to $K_{\mathrm{CSR}}(d)$.

Statistical inference is important here. Even if the dataset had been generated by a CSR process, an estimate of the K-function would be subject to sampling variation, and could not be expected to match KCSR(d) perfectly. Thus, it is necessary to test whether the sampled $\hat{K}(d)$ is sufficiently unusual with respect to the distribution of $\hat{K}$ estimates one might expect to see under CSR to provide evidence that the generating process for the sample is *not* CSR. The idea is illustrated in Figure 6.11. Here, 100 K-function estimates (based on equation (6.5)) from random CSR samples of 100 points (the same number of points as in Figure 6.10) are superimposed, together with the estimate from the point set shown in Figure 6.10. From this it can be seen that the estimate from the clustered sample is quite different from the range of estimates expected from CSR.

Figure 6.11 Sample K-functions under CSR



Another aspect of sampling inference for *K*-functions is the dependency of $\hat{K}(d)$ on the shape of the study area. The theoretical form $K_{CSR}(d) = \lambda\pi d^2$ is based on an assumption of points occurring in an infinite two-dimensional plane. The fact that a 'real-world' sample will be taken from a finite study area (denoted here by *A*) will lead to further deviation of sample-based estimates of $\hat{K}(d)$ from the theoretical form. This can also be seen in Figure 6.11 – although for the lower values of *d* the CSR estimated *K*-function curves resemble the quadratic shape expected: the curves 'flatten out' for higher values of *d*. This is due to the fact that for larger values of *d*, points will only be observed in the intersection of a circle of radius *d* around a random **x***i* and the study area *A*. This will result in fewer points being observed than the theoretical *K*-function would predict. This effect continues, and when *d* is sufficiently large any circle centred on one of the points will encompass the entirety of *A*. At this point, any further increase in *d* will result in no change in the number of points contained in the circle – this provides an explanation of the flattening-out effect seen in the figure.

Above, the idea is to consider a CSR process constrained to the study area. However, another viewpoint is that the study area defines a subset of all points generated on the full two-dimensional plane. To estimate the *K*-function for the full-plane process some allowance for edge effects on the

study area needs to be made. Ripley (1976) proposed the following modification to equation (6.5):

(6.7)

$$\hat{K}(d) = \hat{\lambda}^{-1} \sum_{i} \sum_{j \neq i} \frac{2I(d_{ij} < d)}{n(n-1)w_{ij}} \qquad (6.7)$$

where w_{ij} is the area of intersection between a circle centred at $\mathbf{x}i$ passing through $\mathbf{x}_j$ and the study area A. Inference about the estimated K-function can then be carried out using the approach used above, but with $\hat{K}(d)$ based on equation (6.7).

I

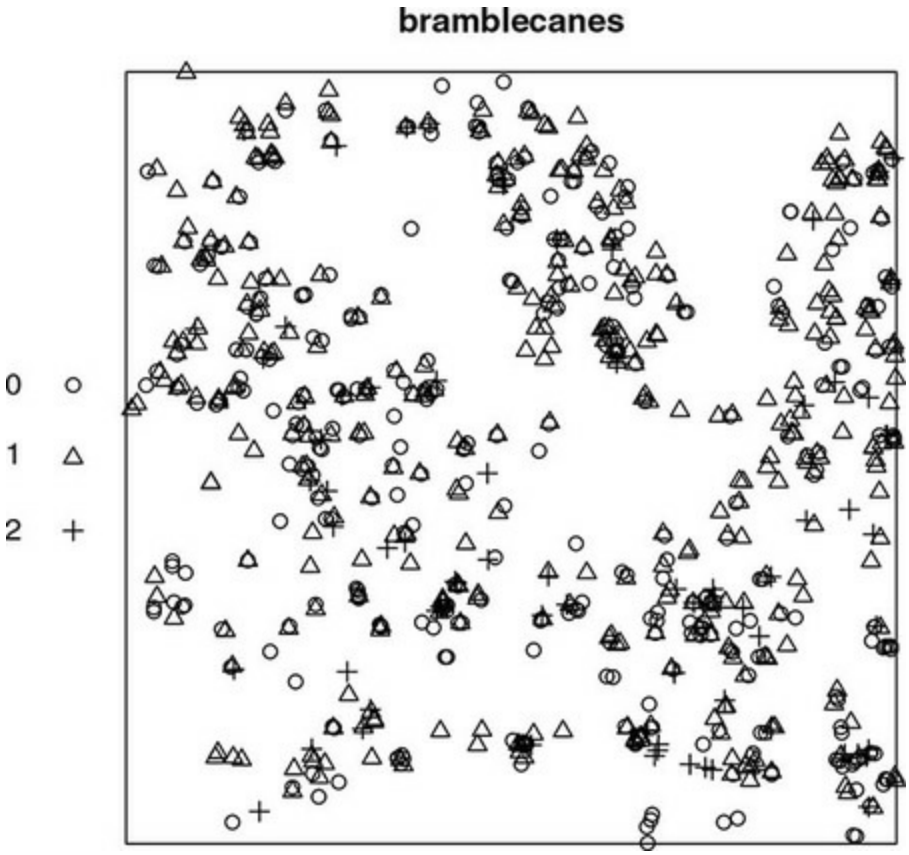
For the data in the example, points were generated with A as the rectangle having lower left corner (−1, −1) and upper right corner (1, 1). In practice A may have a more complex shape (a polygon outline of a county, for example); for this reason, assessing the sampling variability of the K-function under sampling must often be achieved via simulation, as seen in Figure 6.11.

6.5.1 Using the *K*-Function in R

In R, a useful package for computing estimated K-functions (as well as other spatial statistical procedures) is *spatstat*. This is capable of carrying out the kind of simulation illustrated earlier in this section.

The K-function estimation as defined above may be estimated in the spatstat package using the Kest function. Here the locations of bramble canes (Hutchings, 1979; Diggle, 1983) are analysed, having been obtained as a dataset supplied with spatstat via the data(bramblecanes) command. They are plotted in Figure 6.12. Different symbols represent different ages of canes – although initially we will just consider the point pattern for *all* canes.

Figure 6.12 Bramble cane locations



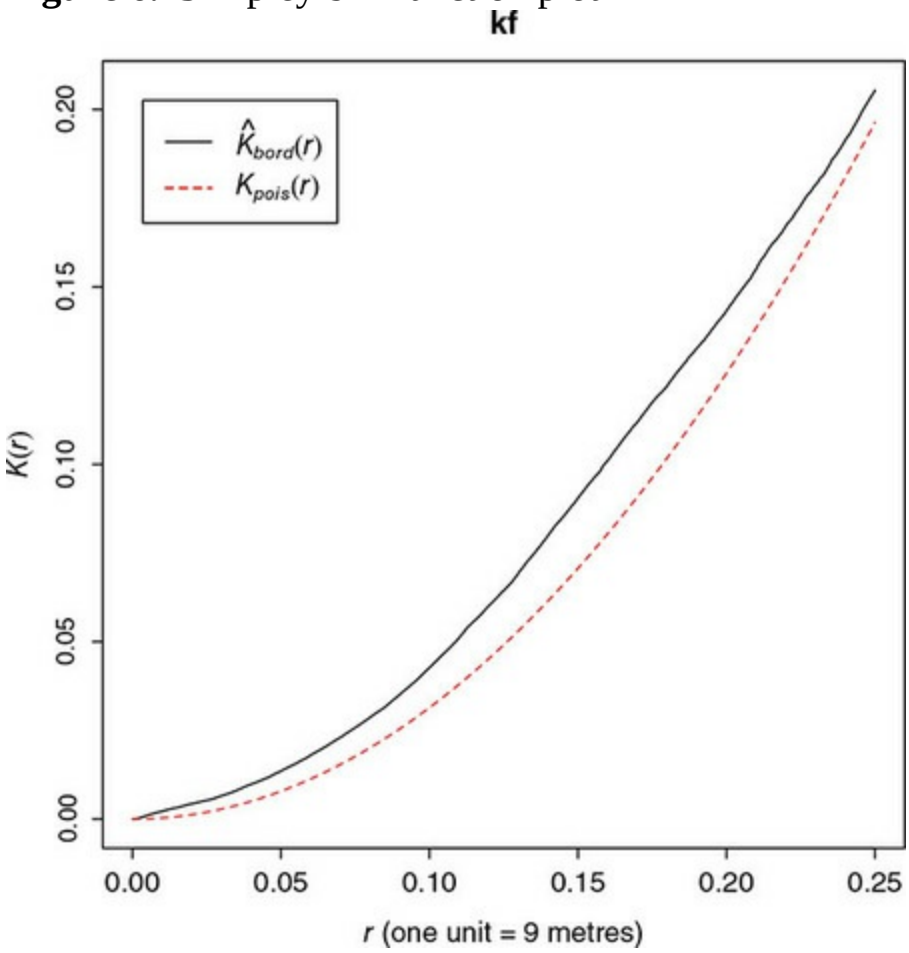

```
# K-function code block

# Load the spatstat package

require(spatstat)

# Obtain the bramble cane data

data(bramblecanes)

plot(bramblecanes)
```

Next, the Kest function is used to obtain an estimate for the *K*-function of the spatial process underlying the distribution of the bramble canes. The correction='border' argument requests that an edge-corrected estimate (as in equation (6.7)) be used.

```
kf <- Kest(bramblecanes,correction='border')

# Plot it
```

```
plot(kf)
```

Figure 6.13 Ripley's *K*-function plot



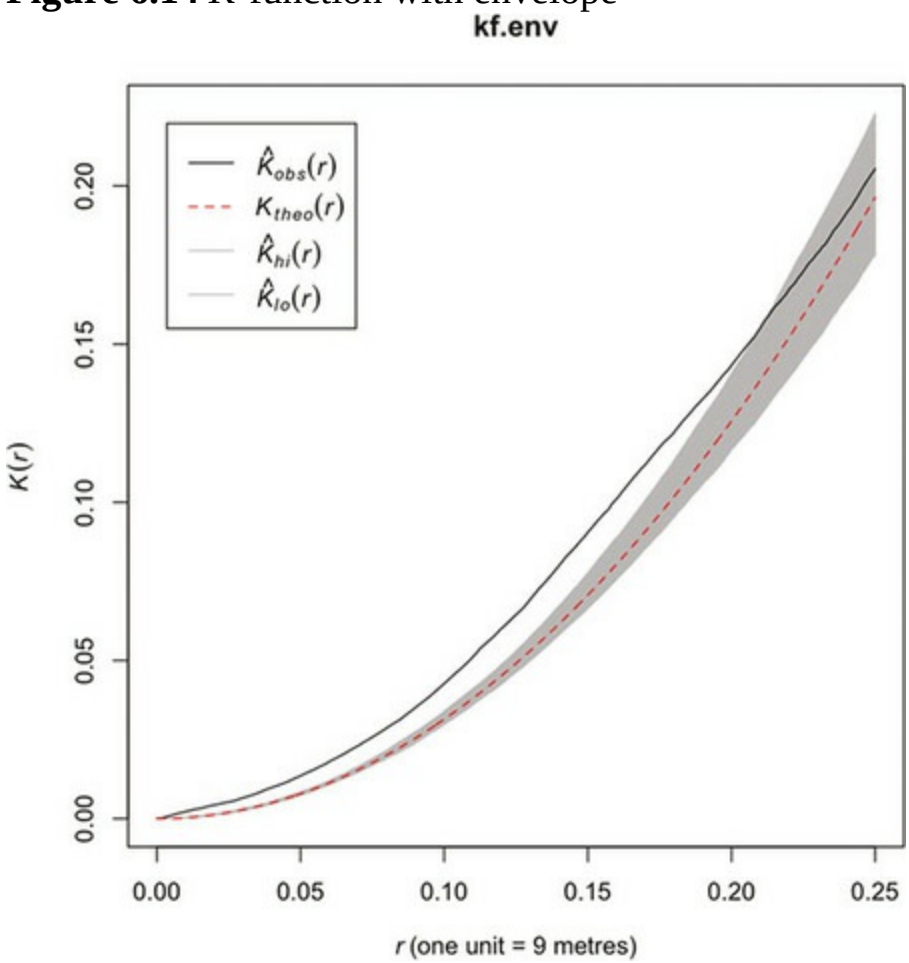
The result of plotting the *K*-function as shown in Figure 6.13 compares the estimated function (labelled $\hat{K}_{\text{bord}}$) to the theoretical function under CSR (labelled $\hat{K}_{\text{pois}}$). It may be seen that the data appear to be clustered (generally the empirical *K*-function is greater than that for CSR, suggesting that more points occur close together than would be expected under CSR). However, this perhaps needs a more rigorous investigation, allowing for sampling variation via simulation as set out above.

This simulation approach is sometimes referred to as *envelope* analysis, the envelope being the highest and lowest values of $\hat{K}(d)$ for a value of *d*. Thus the function for this is called envelope. This takes a ppp object and a further function as an argument. The function here is Kest – there are other functions

also used to describe spatial distributions which will be discussed later, which envelope can use, but for now we focus on Kest. The envelope object may also be plotted, as shown in the following code which results in Figure 6.14:

```
# Code block to produce k-function with envelope

# Envelope function

kf.env <- envelope(bramblecanes,Kest,correction="border")

# Plot it

plot(kf.env)
```

Figure 6.14 *K*-function with envelope



From this it can be seen that the estimated *K*-function for the sample takes on a higher value than the envelope of simulated *K*-functions for CSR until *d* becomes quite large, suggesting strong evidence that the locations of bramble

canes do indeed exhibit clustering. However, it can reasonably be argued that comparing an estimated $\hat{K}(d)$ and an envelope of randomly sampled estimates under CSR is not a formal significance test. In particular, since the sample curve is compared to the envelope for several *d* values, multiple significance testing problems may occur. These are well explained by Bland and Altman (1995) – in short, when carrying out *several* tests, the chance of obtaining a false positive result in *any* test is raised. If the intention is to evaluate a null hypothesis of CSR, then a single number measuring departure of $\hat{K}(d)$ from $K_{\mathrm{CSR}}(d)$, rather than the *K*-function, may be more appropriate – so that a single test can be applied. One such number is the *maximum absolute deviation* (MAD: Ripley, 1977, 1981). This is the absolute value of the largest discrepancy between the two functions:

(6.8)

$$\mathrm{MAD} = \max_{d} \left| \hat{K}(d) - K_{\mathrm{CSR}}(d) \right| \qquad (6.8)$$

In R, we enter:

```
mad.test(bramblecanes,Kest,verbose=FALSE)

Maximum absolute deviation test of CSR

Monte Carlo test based on 99 simulations

Summary function: K(r)

Reference function: theoretical

Alternative: two.sided

Interval of distance values: [0, 0.25] units (one unit = 9 metres)

Test statistic: Maximum absolute deviation

Deviation = observed minus theoretical

data: bramblecanes

mad = 0.016159, rank = 1, p-value = 0.01
```

In this case it can be seen that the null hypotheses of CSR can be rejected at

the 1% level. An alternative test is advocated by Loosmore and Ford (2006) where the test statistic is

(6.9)

$$u_i = \sum_{d_k = d_{min}}^{d_{max}} \left[\hat{K}_i(d_k) - \bar{K}_i(d_k) \right]^2 \delta_k \qquad (6.9)$$

in which $\bar{K}_i(t_k)$ is the average value of $\hat{K}(d)$ over the simulations, the *dk* are a sequence of sample distances ranging from d_{min} to d_{max}, and $\delta k = dk+1 - dk$. Essentially this attempts to measure the sum of the squared distance between the functions, rather than the maximum distance. This is implemented by spatstat via the dclf.test function, which works similarly to mad.test:

```
dclf.test(bramblecanes,Kest,verbose=FALSE)

Diggle-Cressie-Loosmore-Ford test of CSR

Monte Carlo test based on 99 simulations

Summary function: K(r)

Reference function: theoretical

Alternative: two.sided

Interval of distance values: [0, 0.25] units (one unit = 9 metres)

Test statistic: Integral of squared absolute deviation

Deviation = observed minus theoretical

data:    bramblecanes

u = 3.3372e−05, rank = 1, p-value = 0.01
```

Again, results suggest rejecting the null hypothesis of CSR – see the reported *p*-value.

6.5.2 The *L*-function

An alternative to the *K*-function for identifying clustering in spatial processes

is the *L*-function. This is defined in terms of the *K*-function

(6.10)

$$L(d) = \sqrt{\frac{K(d)}{\pi}} \qquad (6.10)$$

Although just a simple transformation of the *K*-function, its utility lies in the fact that under CSR, $L(d) = d$; that is, the *L*-function is linear, having a slope of 1 and passing through the origin. Visually identifying this in a plot of estimated *L*-functions is generally easier than identifying a quadratic function, and therefore *L*-function estimates are arguably a better visual tool. The Lest function provides a sample estimate of the *L*-function (by applying the transform in (6.10) to $\hat{K}(d)$) which can be used in place of Kest. As an example, recall that the envelope function could take alternatives to *K*-functions to create the envelope plot: in the following code, an envelope plot using *L*-functions for the bramble cane data is created (see Figure 6.15):

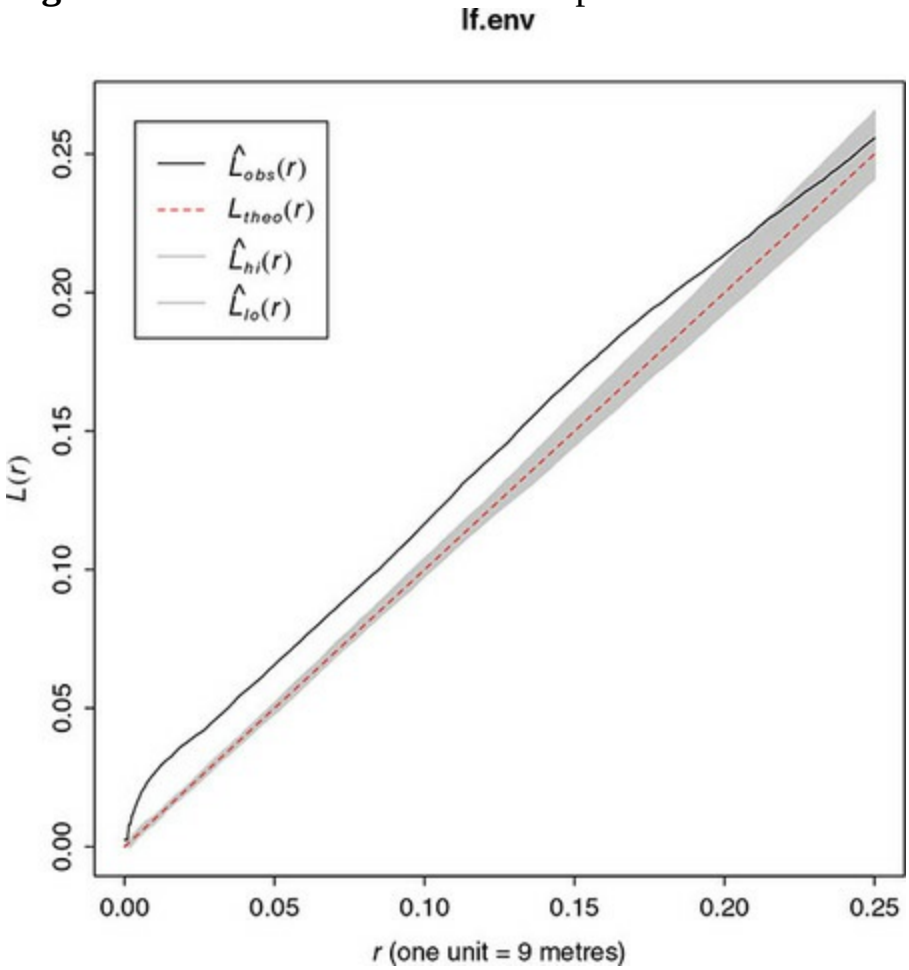
```
# Code block to produce k-function with envelope

# Envelope function

lf.env <- envelope(bramblecanes,Lest,correction="border")

# Plot it

plot(lf.env)
```

Similarly, it is possible to apply MAD tests or Loosmore and Ford tests using *L* instead of *K*. Again mad.test and dclf.test allow an alternative to *K*-functions to be specified. Indeed, Besag (1977) recommends using *L*-functions in place of *K*-functions in this kind of test. As an example, the following code applies the MAD test to the bramble cane data using the *L*-function.

```
mad.test(bramblecanes,Lest,verbose=FALSE)

Maximum absolute deviation test of CSR

Monte Carlo test based on 99 simulations
```

```
Summary function: L(r)

Reference function: theoretical

Alternative: two.sided

Interval of distance values: [0, 0.25] units (one unit = 9 metres)

Test statistic: Maximum absolute deviation

Deviation = observed minus theoretical

data: bramblecanes

mad = 0.017759, rank = 1, p-value = 0.01
```

Figure 6.15 *L*-function with envelope

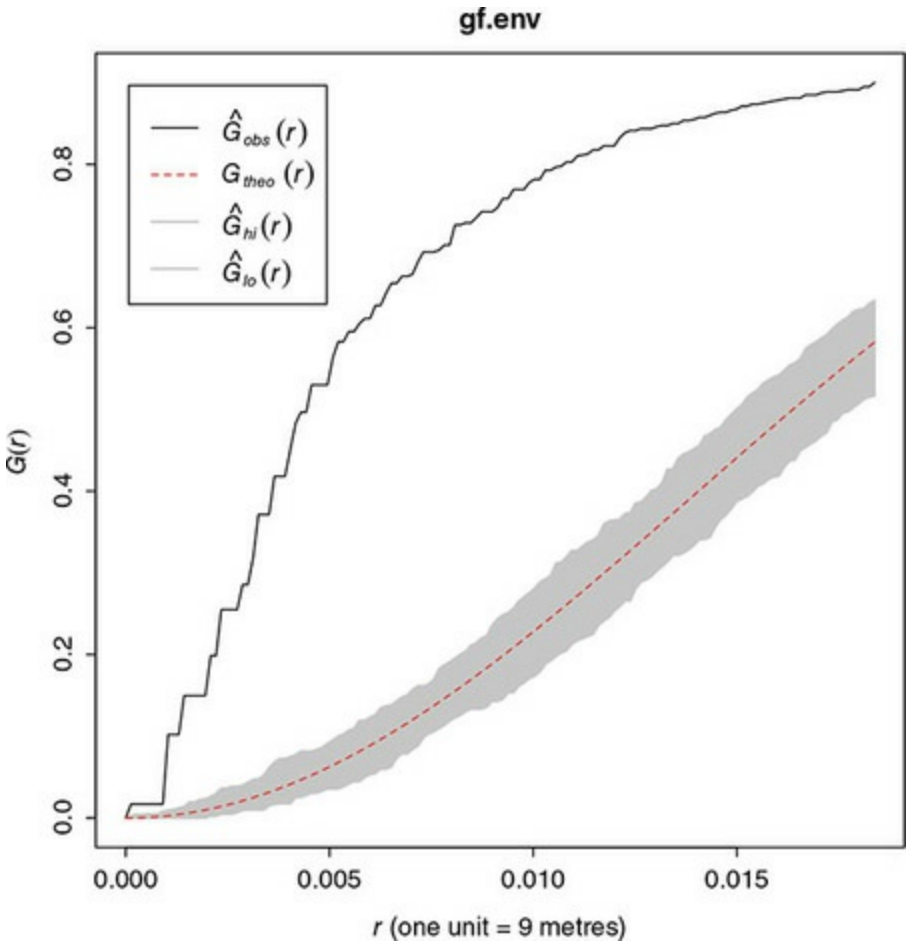


6.5.3 The *G*-Function

Yet another function used to describe the clustering in point patterns is the *G*-function. This is the cumulative distribution of the nearest neighbour distance for a randomly selected **x***i*. Thus, given a distance *d*, *G*(*d*) is the probability that the nearest neighbour distance for a randomly chosen sample point is less than or equal to *d*. Again, this can be estimated using spatstat, using the function Gest. As in the case of Lest and Kest, the functions envelope, mad.test and dclf.test may be used with Gest. Here, again with the bramble cane data, a *G*-function envelope is plotted:

```
# Code block to produce G-function with envelope

# Envelope function

gf.env <- envelope(bramblecanes,Gest,correction="border")

# Plot it

plot(gf.env)
```

Figure 6.16 *G*-function with envelope



The estimate of the *G*-function for the sample is based on the empirical proportion of nearest neighbour distances less than *d*, for several values of *d*. In this case the envelope is the range of estimates for given *d* values, for samples generated under CSR. Theoretically, the expected *G*-function for CSR is

$$G(d) = 1 - \exp(-\lambda\pi d) \quad (6.11)$$

This is also plotted in Figure 6.16, as G_{theo}.

I

One complication is that spatstat stores spatial information in a different way than sp, GISTools and related packages, as noted earlier. This is not a major hurdle, but it does mean that objects of types such as Spatial-PointsDataFrame must be converted to spatstat's ppp

format. This is a compendium format containing both a set of points and a polygon describing the study area *A*, and can be created from a SpatialPoints or SpatialPointsDataFrame object combined with a Spatial-Polygons or SpatialPolygonsDataFrame object. This is achieved via the as and as.ppt functions from the maptools package.

```
require(maptools)

require(spatstat)

# Bramblecanes is a dataset in ppp format from spatstat

data(bramblecanes)

# Convert the data to SpatialPoints, and plot them

bc.spformat <- as(bramblecanes,"SpatialPoints")

plot(bc.spformat)

# It is also possible to extract the study polygon

# referred to as a window in spatstat terminology

# Here it is just a rectangle...

bc.win <- as(bramblecanes$win,"SpatialPolygons")

plot(bc.win,add=TRUE)
```

It is also possible to convert objects in the other direction, via the as.ppp function. This takes two arguments, the coordinates of the SpatialPoints or SpatialPointsDataFrame object (extracted using the coordinates function), and an *owin* object created from a SpatialPolygons or SpatialPolygonsDataFrame via as.win. owin objects are single polygons used by spatstat to denote study areas, and are a component of ppp objects. In the following example, the burgres.n point dataset from GISTools is converted to ppp format and a *G*-function is computed and plotted.

```
require(maptools)

require(spatstat)

# Bramblecanes is a dataset in ppp format from spatstat

# convert burgres.n to a ppp object br.n.ppp <- as.ppp(coordinates(burgres.n),
W=as.owin(gUnaryUnion(blocks)))

br.n.gf <- Gest(br.n.ppp)

plot(br.n.gf)
```

6.6 Looking At Marked Point Patterns

A further advancement of the analysis of patterns of points of a single type is the consideration of *marked* point patterns. Here, several kinds of points are considered in a dataset, instead of only a single kind. For example, in the newhaven dataset there are point data for several kinds of crime. The term 'marked' is used as each point is thought of as being tagged (or marked) with a specific type. As with the analysis of single kinds of points (or 'unmarked' points), the points are still treated as random two-dimensional quantities. It is also possible to apply tests and analyses to each individual kind of point – for example, testing each mark type against a null hypothesis of CSR, or computing the *K*-function for that mark type. However, it is also possible to examine the relationships between the point patterns of different mark types. For example, it may be of interest to determine whether forced entry residential burglaries occur closer to non-forced-entry burglaries than one might expect if the two sets of patterns occurred independently.

One method of investigating this kind of relationship is the *cross-K-function* between marks of type *i* and *j*. This is defined as

$$Kij\ (d) = \lambda_j - 1\mathrm{E}(N_{dij}) \quad (6.12)$$

where N_{dij} is the number of events $\mathbf{x}_k$ of type *j* within a distance *d* of a randomly chosen event from all recorded events $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of type *i*, and λ_j is the intensity of the process marked *j* – measured in events per unit area (Lotwick and Silverman, 1982). If the process for points with mark *j* is CSR, then $K_{ij}(d) = \lambda_j \pi d^2$. A similar simulation-based approach to that set out for *K*, *L* and *G* in earlier sections may be used to investigate $K_{ij}(d)$ and compare it to a hypothesised sample estimate of $K_{ij}(d)$ under CSR.

The empirical estimate of $K_{ij}(d)$ is obtained in a similar way to that in equation (6.5):

(6.13)

$$\hat{K}_{ij}(d) = \hat{\lambda}_j^{-1} \sum_k \sum_l \frac{I(d_{kl} < d)}{n_i n_j} \qquad (6.13)$$

where *k* indexes all of the *i*-marked points and *l* indexes all of the *j*-marked processes, and *ni* and *nj* are the respective numbers of points marked *i* and *j*. A correction (of the form in equation (6.7)) may also be applied. There is also a cross-*L*-function, *Lij* (*d*), which relates to the cross-*K*-function in the same way that the standard *K*-function relates to the standard *L*-function.

6.6.1 Cross-*L*-Function Analysis in R

There is a function in spatstat called Kcross to compute cross-*K*-functions, and a corresponding function called Lcross for cross-*L*-functions. These take a ppp object and values for *i* and *j* as the key arguments. Since *i* and *j* refer to mark types, it is also necessary to identify the marks for each point in a ppp object. This can be done via the marks function. For example, for the bramblecanes object, the points are marked in relation to the age of the cane (see Hutchings, 1979) with three levels of age (labelled as 0, 1 and 2 in increasing order). Note that the marks are factors. These may be listed by entering:

```
marks(bramblecanes)
```

```
  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [28] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [55] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [82] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[109] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[136] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[163] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[190] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[217] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[244] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[271] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[298] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[325] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352] 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[379] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[406] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[433] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[460] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[487] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[514] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[541] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[568] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[595] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[622] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[649] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[676] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[703] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[730] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
[757] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[784] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[811] 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Levels: 0 1 2

I

It is also possible to assign values to marks of a ppp object using the expression:

marks(x) <- ...

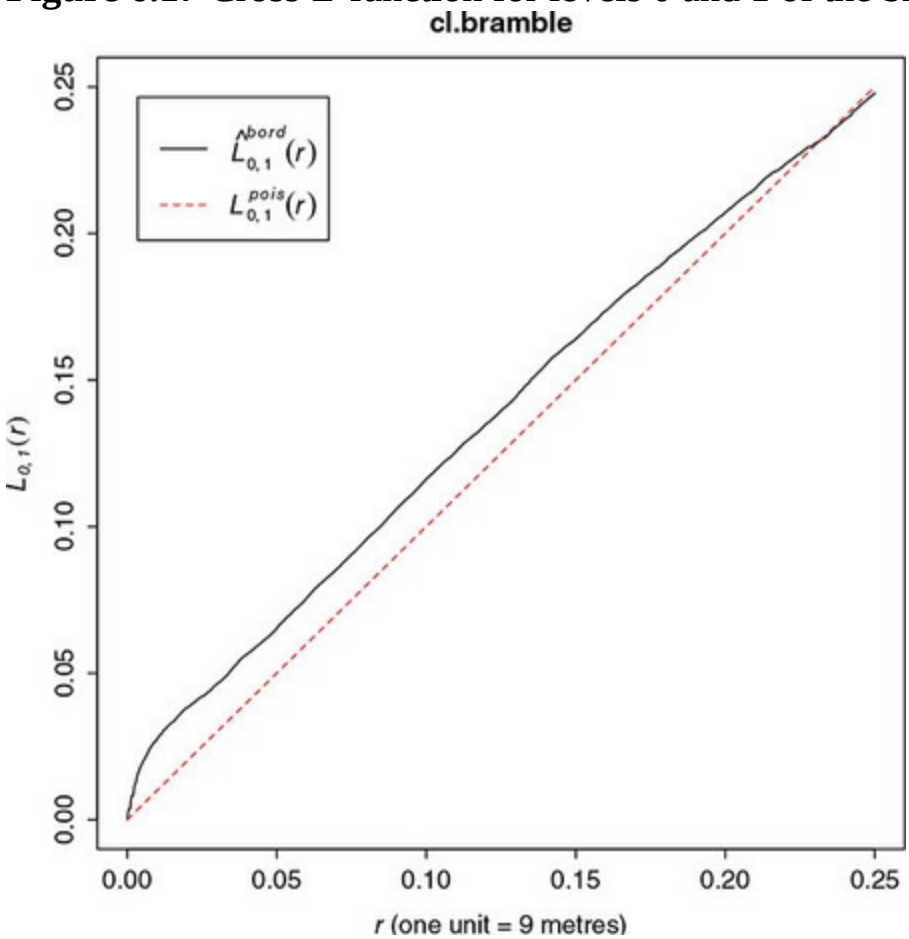
where ... is any valid R expression creating a factor variable with the same length of number elements as there are points in the ppp object x. This is useful if converting a

```
SpatialPointsDataFrame object into a ppp object representing a marked process.
```

As an example here, we compute and plot the cross-*L*-function for levels 0 and 1 of the bramblecanes object (the resultant plot is shown in Figure 6.17):

```
cl.bramble <- Lcross(bramblecanes,i=0,j=1,correction='border')

plot(cl.bramble)
```

Figure 6.17 Cross-*L*-function for levels 0 and 1 of the bramble cane data

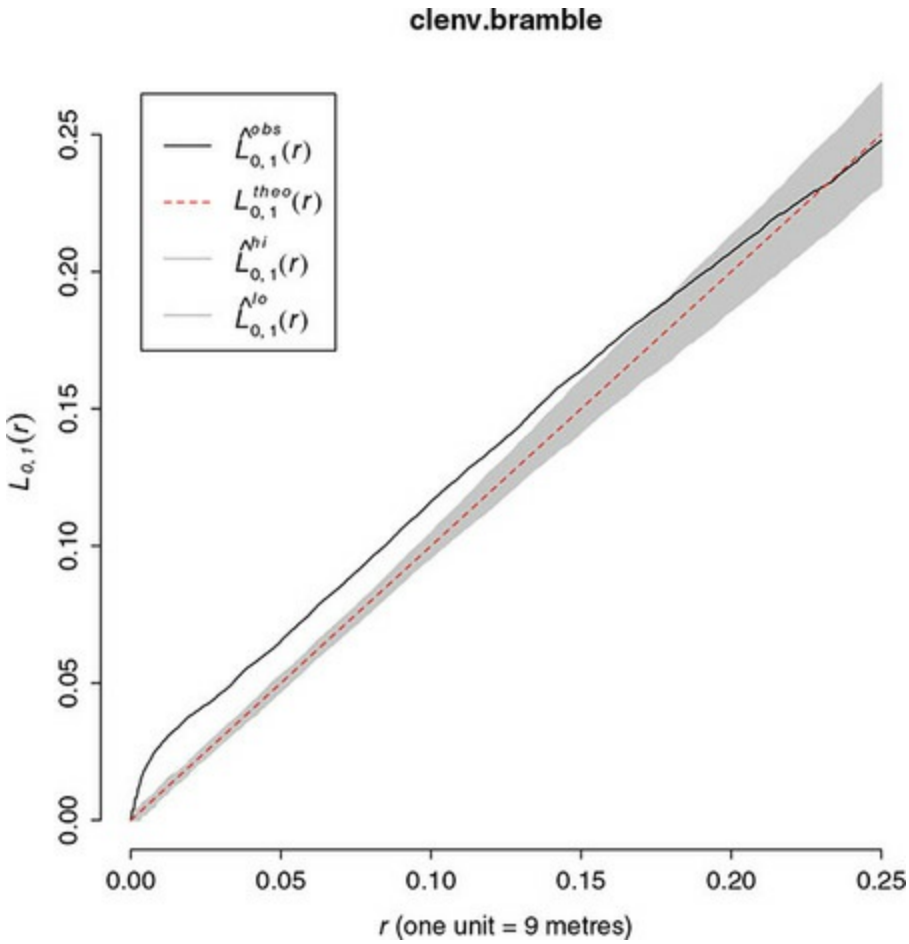


The envelope function may also be used (Figure 6.18):

```
clenv.bramble <- envelope(bramblecanes,Lcross,i=0,j=1,correction='border')

plot(clenv.bramble)
```

Figure 6.18 Cross-*L*-function envelope for levels 0 and 1 of the bramble cane

data



Thus, it would seem that there is a tendency for more young (level 1) bramble canes to occur close to very young (level 0) canes. This can be formally tested, as both mad.test and dclf.test can be used with Kcross and Lcross. Here the use of Lcross with dclf.test is demonstrated:

dclf.test(bramblecanes,Lcross,i=0,j=1,correction='border',verbose=FALSE)

Diggle-Cressie-Loosmore-Ford test of CSR

Monte Carlo test based on 99 simulations

Summary function: L["0", "1"](r)

Reference function: theoretical

Alternative: two.sided

Interval of distance values: [0, 0.25] units (one unit = 9 metres)

```
Test statistic: Integral of squared absolute deviation

Deviation = observed minus theoretical

data: bramblecanes

u = 4.3982e−05, rank = 1, p-value = 0.01
```

6.7 Interpolation of Point Patterns with Continuous Attributes

The previous section can be thought of as outlining methods for analysing point patterns with *categorical*-level attributes. An alternative issue is the analysis of point patterns in which the points have continuous (or measurement scale) attributes, such as height above sea level, soil conductivity or house price. A typical problem here is *interpolation*: given a sample of measurements – say, $\{z_1, \ldots, z_n\}$ at locations $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ – the goal is to estimate the value of *z* at some new point **x**. Possible methods for doing this can be based on fairly simple algorithms, or on more sophisticated spatial statistical models. Here, three key measures will be covered:

- Nearest neighbour interpolation
- Inverse distance weighting
- Kriging

6.7.1 Nearest Neighbour Interpolation

The first of these, nearest neighbour interpolation, is the simplest conceptually, and can be stated as below:

- Find *i* such that |**x***i* − **x**| is minimised
- The estimate of *z* is *zi*

In other words, to estimate *z* at **x**, use the value of *zi* at the observation point closest to **x**. Since the set of closest points to **x***i* for each *i* form the set of Thiessen (Voronoi) polygons for the set of points, an obvious way to

represent the estimates is as a set of Thiessen (Voronoi) polygons corresponding to the **x***i* points, with respective attributes of *zi*. In rgeos there is no direct function to create Voronoi polygons, but Carson Farmer[2] has made some code available to do this, providing a function called voronoipolygons. This has been slightly modified by the authors, and is listed below. Note that the modified version of the code takes the points from a spatial points data frame as the basis for the Voronoi polygons on a spatial points data frame, and carries across the attributes of the points to become attributes of the corresponding Voronoi polygons. Thus, in effect, if the *z* value of interest is an attribute in the input spatial points data frame then the nearest neighbour interpolation is implicitly carried out when using this function.

[2] http://www.carsonfarmer.com/2009/09/voronoi-polygons-with-r/

The function makes use of Voronoi computation tools carried out in another package called deldir – however, this package does not make use of Spatial* object types, and therefore this function provides a 'front end' to allow its integration with the geographical information handling tools in rgeos, sp and maptools. Do not be too concerned if you find the code difficult to interpret – at this stage it is sufficient to understand that it serves to provide a spatial data manipulation function that is otherwise not available.

```
#
# Original code from Carson Farmer
# http://www.carsonfarmer.com/2009/09/voronoi-polygons-with-r/
# Subject to minor stylistic modifications
#
require(deldir)
require(sp)
# Modified Carson Farmer code
voronoipolygons = function(layer) {
crds <- layer@coords
```

```
z <- deldir(crds[,1], crds[,2])
w <- tile.list(z)
polys <- vector(mode='list', length=length(w))
for (i in seq(along=polys)) {
pcrds <- cbind(w[[i]]$x, w[[i]]$y)
pcrds <- rbind(pcrds, pcrds[1,])
polys[[i]] <- Polygons(list(Polygon(pcrds)), ID=as.character(i))
}
SP <- SpatialPolygons(polys)
voronoi <- SpatialPolygonsDataFrame(SP,
data=data.frame(x=crds[,1], y=crds[,2], layer@data,
row.names=sapply(slot(SP, 'polygons'),
function(x) slot(x, 'ID'))))
proj4string(voronoi) <- CRS(proj4string(layer))
return(voronoi)
}
```

6.7.2 A Look at the Data

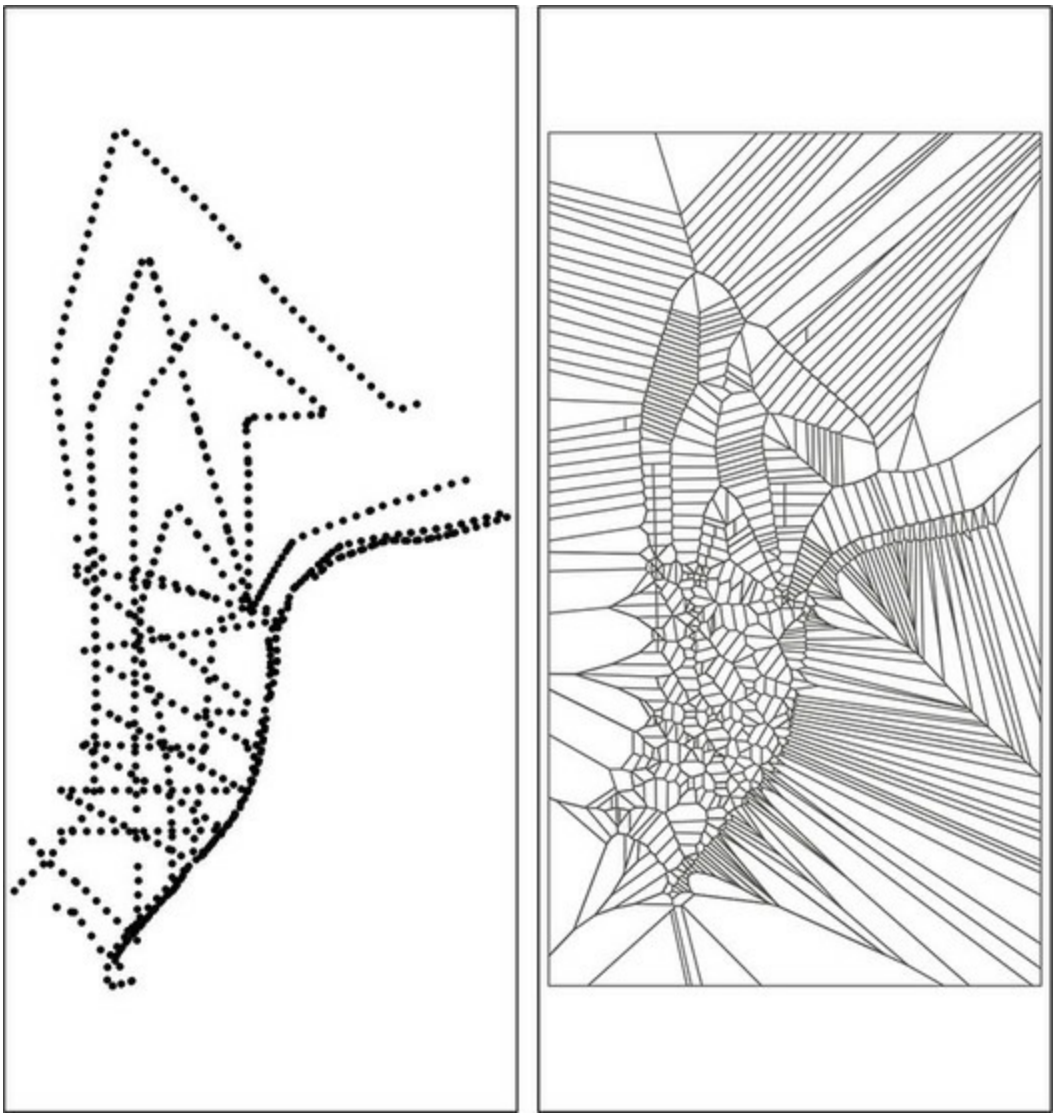
Having defined this function, the next stage is to use it on a test dataset. One such dataset is provided in the gstat package. This package provides tools for a number of approaches to spatial interpolation – including the other two listed in this chapter. Of interest here is a data frame called fulmar. Details of the dataset may be obtained by entering ?fulmar once the package gstat has been loaded. The data are based on airborne counts of the sea bird *Fulmaris glacialis* during August and September of 1998 and 1999, over the Dutch part of the North Sea. The counts are taken along transects corresponding to flight paths of the observation aircraft, and are transformed to densities by dividing counts by the area of observation, 0.5 km^2.

In this and the following sections you will analyse the data described above.

First, however, these data should be read into R, and converted into a Spatial∗ object. The first thing you will need to do is enter the code to define the function voronoipolygons as listed above. The next few lines of code will read in the data (stored in the data frame fulmar) and then convert them into a spatial points data frame. Note that the fulmar sighting density is stored in column fulmar in the data frame fulmar – the location is specified in columns x and y. The point object is next converted into a Voronoi spatial points data frame to provide nearest neighbour interpolations. Having created both the point and Voronoi polygon objects, the code below then plots these (see Figure 6.19):

```
library(gstat)
library(tmap)
data(fulmar)
fulmar.spdf <- SpatialPointsDataFrame(cbind(fulmar$x,fulmar$y),
fulmar)
fulmar.spdf <- fulmar.spdf[fulmar.spdf$year==1999,]
proj4string(fulmar.spdf) <- CRS("+init=epsg:32631")
fulmar.voro <- voronoipolygons(fulmar.spdf)
tmap_mode('plot')
fpt <- tm_shape(fulmar.spdf) + tm_dots(size=0.1)
fvr <- tm_shape(fulmar.voro) + tm_borders()
tmap_arrange(fpt,fvr)
```

Figure 6.19 Fulmar sighting transects: (left) points; (right) Voronoi diagram



The paths of the transects become clear when the data are plotted. For the most part they are linear, although one path follows the Dutch coast. Towards the south-west, north–south-oriented paths are crossed by other zig-zag paths providing a fairly comprehensive coverage. Further north, coverage is sparser. In terms of the Voronoi diagrams, one notable artefact is that the areas of the polygons vary with the density of the points (when the points are internal) – and that edge points have polygons of infinite area (here trimmed to an enclosing rectangle). These are typical features of Voronoi polygons, but they can give rather strange characteristics to spatial interpolation. To see this, a choropleth map of the nearest neighbour densities is created in Figure 6.20. In this case, the break points at densities of 5, 15, 25 and 35 birds per square kilometre are used. The map is specified to be in view mode. Note that

although the outline of the Voronoi polygons forms a perfect rectangle in the UTM zone 31 projection, working with the projection used by the web backdrop (‘web Mercator’) will slightly distort this.

```
library(GISTools)

tmap_mode(’view’)

sh <- shading(breaks=c(5,15,25,35), cols=brewer.pal(5,’Purples’))

tm_shape(fulmar.voro) + tm_fill(col=’fulmar’,style=’fixed’,breaks=c(0,5,15,25,35),
alpha=0.6,title=’Fulmar Density’)
```

Figure 6.20 Nearest neighbour estimate of fulmar density

Again, some of the rather strange characteristics of the Voronoi polygon representation are apparent. In particular, the very large polygons on the edges somewhat dominate the interpolations visually, and the irregular shapes of the polygons lead to a fairly confusing visualisation. Although this approach is sometimes used as a ‘quick and dirty’ estimation tool (possibly as inputs to numerical models or indicators), the visual approach here does demonstrate some of the stranger characteristics of the approach. While it is possible to detect an increased density towards the north-east of the study area, it is harder to identify any subtler patterns due to the distorting effect of the variety of polygon shapes and sizes. Arguably the most problematic aspect of this approach is that the interpolated surfaces are discontinuous, and in particular that the discontinuities are an artefact of the locations of the

samples. For this reason, methods such as the two others covered here are preferred.

6.7.3 Inverse Distance Weighting

In the *inverse distance weighting* (IDW) approach to interpolation, to estimate the value of *z* at location **x** a weighted mean of nearby observations is taken, rather than relying on a single nearest neighbour. To accommodate the idea that observations of *z* at points closer to **x** should be given more importance in the interpolation, greater weight is given to these points – in particular, if *wi* is the weight given to *zi*, then the estimate of *z* at location **x** is

(6.14)

$$\hat{z}(\mathbf{x}) = \frac{\sum_i w_i z_i}{\sum_i w_i} \qquad (6.14)$$

where

(6.15)

$$w_i = \left|\mathbf{x} - \mathbf{x}_i\right|^{-\alpha} \qquad (6.15)$$

and $\alpha \geq 0$. Typically $\alpha = 1$ or $\alpha = 2$, giving an inverse or inverse square relationship.

I

There are some interesting relationships between IDW and other methods. If $\alpha = 0$ then *wi* = 1 for all *i*, and *z* is just the mean of all the *zi* regardless of location. Also note that the ratio of *wi* to *wk*, where *k* is the index of the closest observation to **x**, is

(6.16)

$$\begin{cases} \left(\dfrac{\left|\mathbf{x} - \mathbf{x}_k\right|}{\left|\mathbf{x} - \mathbf{x}_i\right|}\right)^{\alpha} & \text{if } i \neq k \\ 1 & \text{if } i = k \end{cases} \qquad (6.16)$$

and so if $\alpha \to \infty$ then the weighting is dominated by *wk*, and the estimate of *z* tends to the

nearest neighbour estimate.

If the value of **x** coincides with one of the **x***i* values then there is a problem with the weighting, as *wi* is infinite. However, the IDW estimate is then defined to be the value of *zi*. If a number (say, *k*) of distinct observations are all taken at the same location, so that $\mathbf{x}_{i1} = \mathbf{x}_{i2} = \ldots = \mathbf{x}_{ik}$, then the estimate is the mean of $z_{i1}, z_{i2}, \ldots, z_{ik}$.

I

The definition of IDW when **x** coincides with data point **x***i* is understood by noting that the IDW can be written as

(6.17)

$$\hat{z}(\mathbf{x}) = \frac{\sum_i d_i^{-\alpha} z_i}{\sum_i d_i^{-\alpha}}, \quad \text{where } d_i = |\mathbf{x} - \mathbf{x}_i| \tag{6.17}$$

and if the numerator and denominator are multiplied by d_i^{α} then

(6.18)

$$\hat{z}(\mathbf{x}) = \frac{z_i + d_i \sum_{k \neq i} d_k^{-\alpha} z_k}{1 + d_i \sum_{k \neq i} d_k^{-\alpha}} \tag{6.18}$$

and in the limiting case where $di \rightarrow 0$ this expression is just *zi* as in the definition above. Also note that in the case where there are coincident locations, so that $d_{i1} = d_{i2} = \ldots = d_{ik} = d$, say, then by multiplying denominator and numerator by *d* we have

(6.19)

$$\hat{z}(\mathbf{x}) = \frac{z_{i_1} + z_{i_2} + \ldots + z_{i_k} + d \sum_{k \notin \{i_1, i_2, \ldots, i_k\}} d_k^{-\alpha} z_k}{k + d \sum_{k \notin \{i_1, i_2, \ldots, i_k\}} d_k^{-\alpha}} \tag{6.19}$$

and again, as $d \rightarrow 0$ the limit is the mean of $z_{i1}, z_{i2}, \ldots, z_{ik}$.

6.7.4 Computing IDW with the gstat Package

There are a number of ways to compute IDW interpolation. Here, the gstat package will be considered. This package is also useful for kriging, the third approach to spatial interpolation covered here (Section 6.8). Thus knowledge of this package is helpful for both methods. Here, the package is demonstrated using the fulmar data used earlier. The following code carries

out the IDW interpolation, and plots the interpolated surface. First, you will need a set of sample points at which the IDW estimates are computed. The function spsample in the package maptools creates a sample grid.

Given a spatial polygons data frame and a number of points, if the argument type='regular' is provided, it will generate a spatial polygons data frame with a regular grid of points covering the polygon. The density of the grid is such that the number of grid points is as close as possible to the number provided.[3] Here a grid with around 6000 points is created. Since the previous object fulmar.voro has a rectangular footprint, this causes the creation of a rectangular grid.

[3] It is not always possible to find a grid with *exactly* the right number of points.

After this, the IDW estimate is created using the idw function. This requires the model to be specified (here the formula fulmar ~ 1 implies that we are performing a simple interpolation) – the **x***i* locations are in fulmar.spdf and the points at which estimates are made are supplied in s.grid. The parameter idp (interpolation distance parameter) is just the value of α in the IDW – here set to 1.

```
library(maptools) # Required package

library(GISTools) # Required package

library(gstat) # Set up the gstat package

# Define a sample grid then use it as a set of points

# to estimate fulmar density via IDW, with alpha=1

s.grid <- spsample(fulmar.voro,type='regular',n=6000)

idw.est <- gstat::idw(fulmar~1,fulmar.spdf, newdata=s.grid,idp=1.0)
```

I

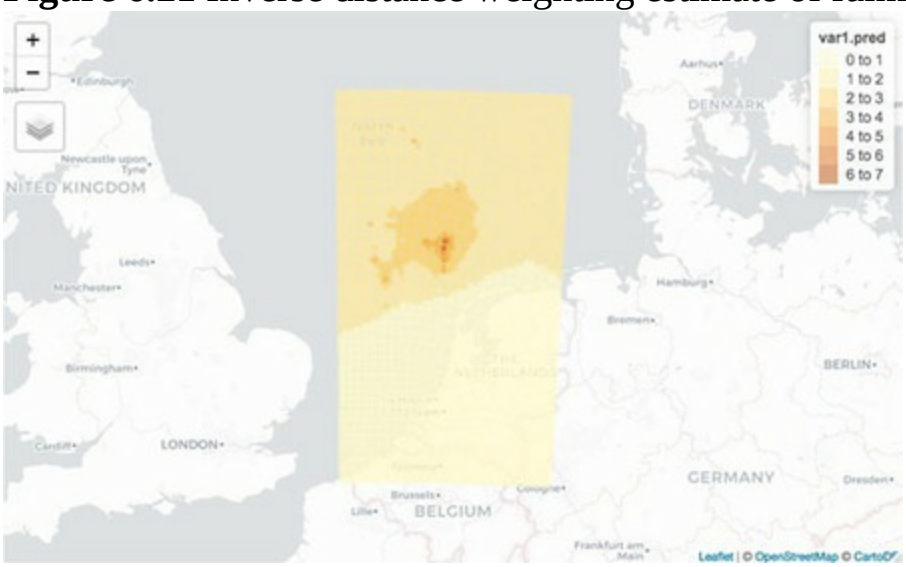
You may wonder why the idw function is referred to as gstat::idw. Recall from earlier that :: allows a function to be called without loading its package. However, here it is helpful because an earlier package you loaded (spatstat) also has a function called idw. This tells R you want to use the function in gstat. If you do not have spatstat loaded but gstat is, then

this notation is not needed – simply idw will do. However, if you are still in the same session where spatstat was used, it is difficult to guarantee which version of the function would be called. Using gstat::idw removes the ambiguity.

The object idw.est is a spatial points data frame containing the IDW estimates at each of the sample points (actually a rectangular grid) in a variable called var1.pred. It is possible to view this directly by choosing point colour to represent the values of var1.pred (Figure 6.21).

```
tmap_mode('view')

tm_shape(idw.est) + tm_dots(col='var1.pred',border.col=NA,alpha=0.7)
```

Figure 6.21 Inverse distance weighting estimate of fulmar density

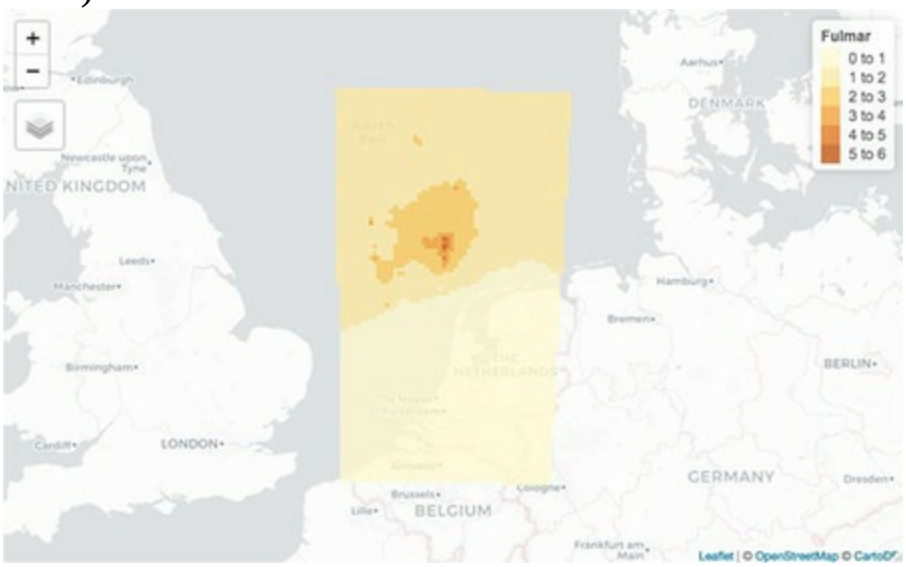


However, when this image is zoomed into, the individual points become visible – possibly a more visually appealing map could be achieved by plotting either contours or a raster overlay. The latter will be dealt with first. To do this, the spatial points data frame should be converted to a spatial pixels data frame – this is similar to a raster coverage, but if there are any pixels that are not in the rectangular bounding box of the coverage region, they are not stored. The conversion is carried out below, with the map shown in Figure 6.22.

```
tmap_mode('view')

idw.grid <- SpatialPixelsDataFrame(idw.est,data.frame(idw.est))
```

```
tm_shape(idw.grid) + tm_raster(col='var1.pred',title='Fulmar')
```

Figure 6.22 Inverse distance weighting estimate of fulmar density (raster view)



It is also possible to depict this as a filled contour plot. To do this, first the contour lines must be derived from the raster data. To do this, the raster package will be used. Although it works fine for mapping, the spatial pixels data frame is not a data type used by this package. However, it can be converted into a standard raster object. When this is done, the rasterToContour function may be used to extract a set of contour lines in a spatial lines data frame object:

```
require(raster)

levs <- c(0,2,4,6,8,Inf)

idw.raster <- raster(idw.grid,layer='var1.pred')

idw.contour <- rasterToContour(idw.raster,levels=levs)
```

The object idw.contour could be used to draw contour *lines* on the map – this can be used directly (Figure 6.23):

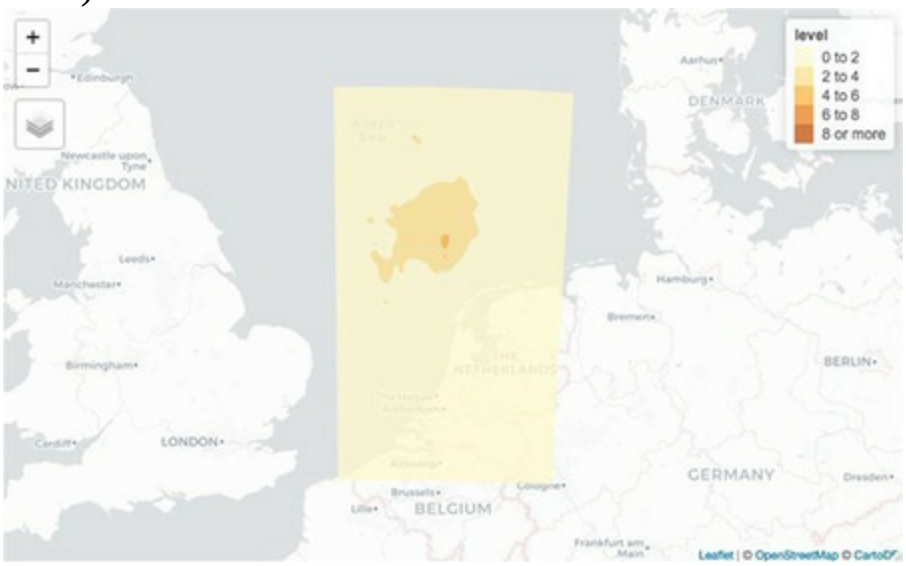
```
tmap_mode('view')

tm_shape(idw.contour) + tm_iso()
```

Figure 6.23 Inverse distance weighting estimate of fulmar density (contour view)

Alternatively, this information could be depicted as a *level plot* or a filled contour plot. This may be achieved by converting the contour lines into a spatial polygons data frame where each polygon is a region between consecutive contour lines. Next, this is shown as a choropleth map. The conversion is currently done in tmap using a *private* function – that is, it is not exported, and so not usually visible. This function is called tmaptools:::lines2polygons – the use of ::: here is similar to that of :: discussed earlier, but implies that the function is one not exported from the library. Apart from using this notation, it is never visible. It should perhaps come with a warning – functions of this sort are usually intended for internal working in the package, and are not necessarily part of the *interface*. Thus, in a future version of tmaptools, if a different way to achieve some of the exported functionality of the package were implemented without this function, it could disappear. However, at the time of writing it does still work with version 1.2 of tmaptools – and is used to create Figure 6.24.

```
tmap_mode('view')

idw.levels <- tmaptools:::lines2polygons(fulmar.voro,
idw.contour,rst=idw.raster,lvls=levs)

tm_shape(idw.levels) + tm_fill(col='level')
```

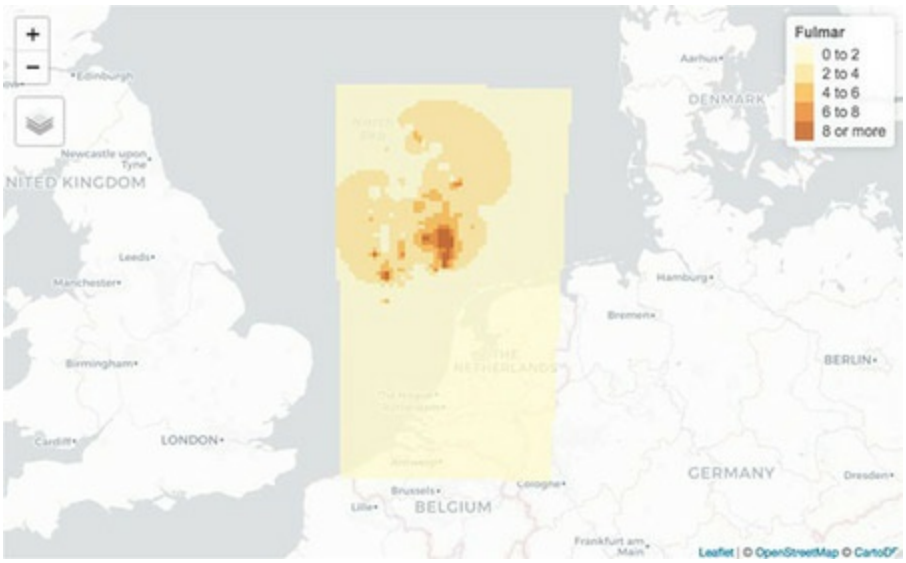
Figure 6.24 Inverse distance weighting estimate of fulmar density (level plot view)



Having explored various ways of mapping the interpolated surface, an alternative interpolation with α = 2 may be created using the same approach. The spatial points data frame IDW is stored in idw.est2, converted to a spatial pixels data frame and mapped using the code here (see Figure 6.25):

```
idw.est2 <- gstat::idw(fulmar~1,fulmar.spdf, newdata=s.grid,idp=2.0)

idw.grid2 <- SpatialPixelsDataFrame(idw.est2,data.frame(idw.est2))

tmap_mode('view')

tm_shape(idw.grid2) + tm_raster(col='var1.pred',title='Fulmar',breaks=levs)
```

Figure 6.25 Inverse distance weighting estimate of fulmar density (α = 2)



Next, the two interpolations are compared. Here this will be done in plot mode.

```
tmap_mode('plot')

idw1 <- tm_shape(idw.grid) + tm_raster(col='var1.pred',title='Alpha = 1',breaks=levs)

idw2 <- tm_shape(idw.grid2) + tm_raster(col='var1.pred',title='Alpha = 2',breaks=levs)

tmap_arrange(idw1,idw2)
```

Figure 6.26 shows that with α = 2 the interpolated values are more inclined to take higher values. To investigate this further, the two interpolated fields can be visualised as surfaces. As tmap does not offer this possibility, here the standard graphics routine persp is used. First, the next few lines of code extract the unique *x* and *y* locations from idw.est, and then format var1.pred into a matrix, predmat.

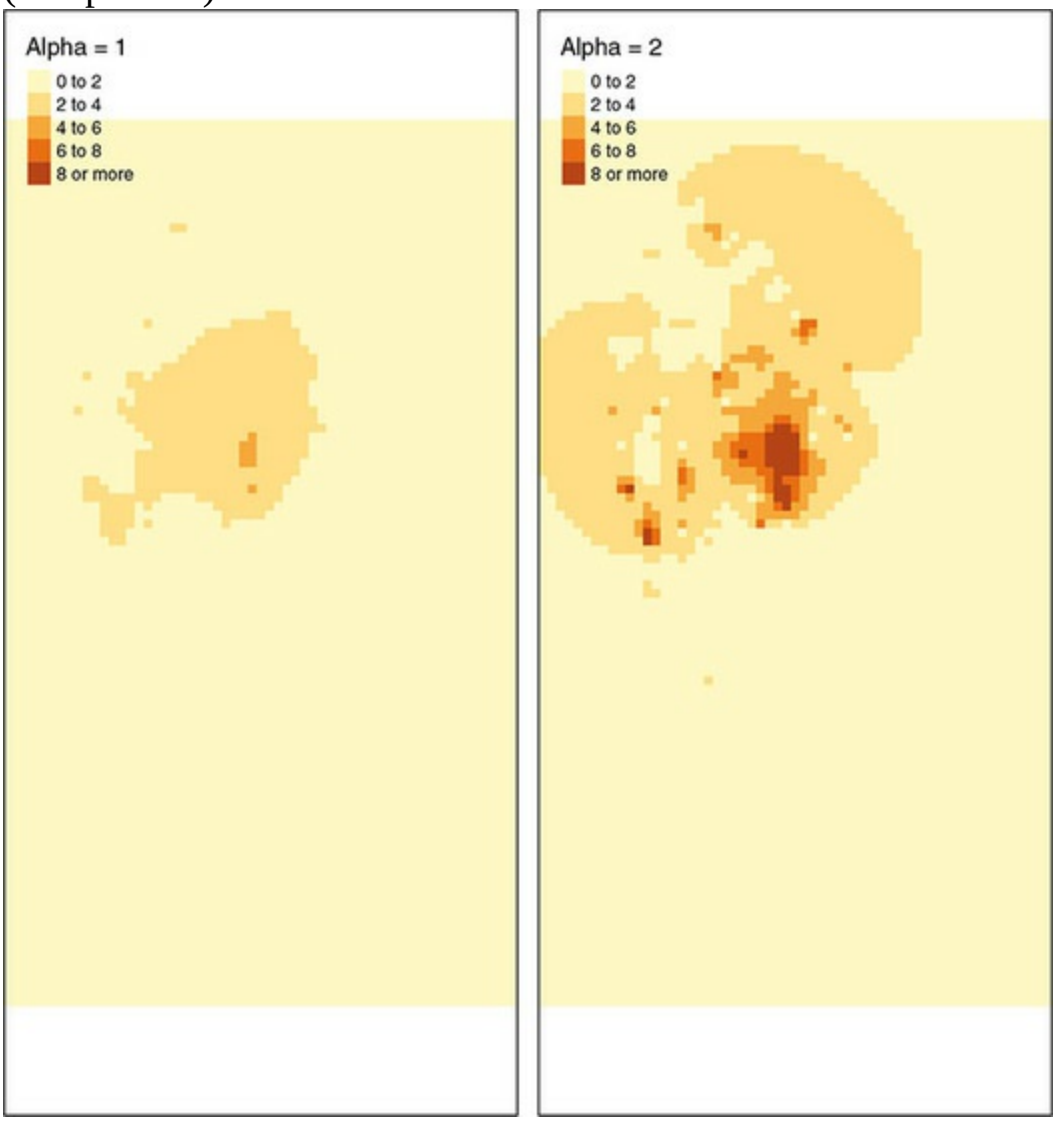
```
# Extract the distinct x and y coordinates of the grid

# Extract the predicted values and form into a matrix

# of gridded values

ux <- unique(coordinates(idw.est)[,1])

uy <- unique(coordinates(idw.est)[,2])
```

```
predmat <- matrix(idw.est$var1.pred,length(ux),length(uy))
```

Next, the same is done for idw.est2.

```
predmat2 <- matrix(idw.est2$var1.pred,length(ux),length(uy))
```

Figure 6.26 Inverse distance weighting estimate of fulmar density (comparison)

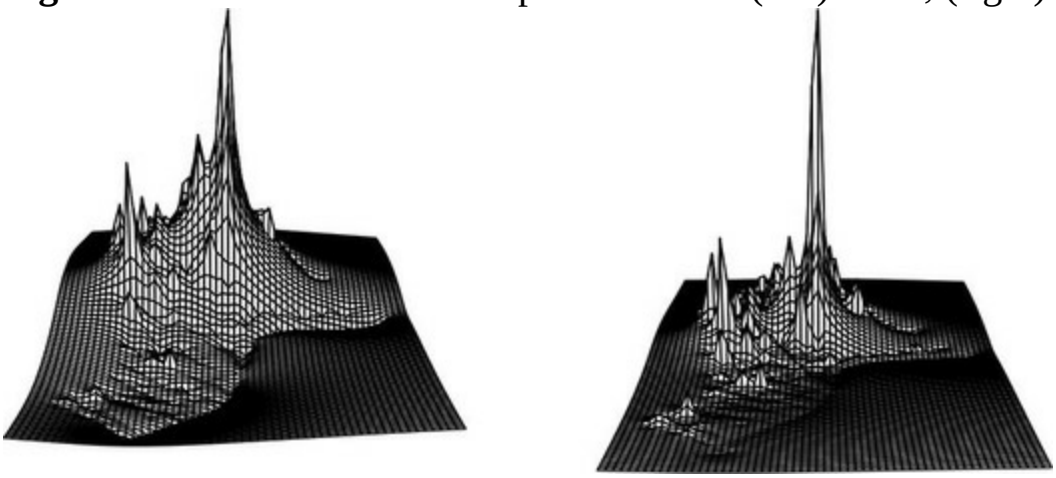


It is now possible to visualise the two interpolations as perspective plots (Figure 6.27).

```
par(mfrow=c(1,2),mar=c(0,0,2,0))
```

```
persp(predmat,box=FALSE)

persp(predmat2,box=FALSE)
```

Figure 6.27 Three-dimensional plots of IDW: (left) α = 1; (right) α = 2)



Although the shapes are similar, it is certainly the case that when α = 2 the surface is slightly more 'spiky', with high-level observations standing out from their surroundings in needle-like protrusions.

6.8 The Kriging Approach

The maps of fulmar density produced by the IDW approach appear to be more satisfactory than those produced by nearest neighbour interpolation, at least in that they do consist of flat regions with a set of arbitrary linear discontinuities. However, one fact to note is that IDW interpolation always passes *exactly* through uniquely located measurement points. If the data are the result of very reliable measurement, and the underlying process is largely deterministic, this is fine. However, if the process is subject to random errors in measurement or sampling, or the underlying process is stochastic, there will be a degree of random variability in the observed *zi* values – essentially, *zi* could be thought of as an expected 'true' value plus some random noise – so that $z_i = T(\mathbf{x}_i) + \varepsilon_i$, where ε_i is a random quantity with mean 0, and $T(\mathbf{x}i)$ is a trend component. In these circumstances it is more useful to estimate the $T(\mathbf{x}i)$ than *zi*. Unfortunately, IDW interpolation does not do this. The problem here is that since this method passes through *zi* it is interpolating the noise in the data as well as the trend. This is illustrated particularly well

with perspective plots in Figure 6.27. The spikes seen in the IDW surfaces for both $\alpha = 1$ and $\alpha = 2$ are a consequence of forcing the surface to go through random noise.

If multiple observations are taken at location $\mathbf{x}i$ then the interpolated value here is the mean of the observed z values, which is a creditable estimator of $T(\mathbf{x}i)$, but in most circumstances only a single observation occurs at each point, and some alternative approach to interpolation should be considered. One possibility here is the use of *kriging* (Matheron, 1963). The theory behind this approach is relatively complex (see, for example, Cressie 1993), but a brief outline will be given here. For another practical overview of the method, see Brunsdon (2009).

6.8.1 A Brief Introduction to Kriging

In kriging, the observed quantity z_i is modelled to be the outcome of a random process:

$$z_i = f(\mathbf{x}_i) + \upsilon(\mathbf{x}_i) + \varepsilon_i \quad (6.20)$$

where $f(\mathbf{x}i)$ is a deterministic trend function, $\upsilon(\mathbf{x}i)$ is a random function and εi is a random error of observation. The deterministic trend function is typical of the sorts of function often encountered in regression models – for example, a planar or quadratic function of $\mathbf{x}$ – or often just a constant mean value function. εi is a random variate, associated with the measurement or sampling error at the point $\mathbf{x}i$. εi is assumed to have a Gaussian distribution with mean 0 and variance $\sigma 2$. This is sometimes called the 'nugget' effect – kriging was initially applied in the area of mining and used to estimate mineral concentration. However, although this was modelled as a continuous quantity, in reality minerals such as gold occur in small nuggets – and exploratory mining samples taken at certain locations would be subject to highly localised variability, depending on whether or not a nugget was discovered. This effect may well be apparent in the fulmar sighting data – an observatory flight at the right time and place may spot a flock of birds, whereas one with a marginally different flight path, or slightly earlier or later,

may miss this.

The final term is the random function υ(**x**). This is perhaps the most complex to explain. If you would like to gain some further insight into this concept, read the next section. If, however, you are happy to take the kriging approach on trust, you can skip this section.

6.8.2 Random Functions

Here, rather than a single random number, the entire function is random.

A simple example of a random function might be $f(x) = a + bx$, where a and b are random numbers (say, independent Gaussian with mean 0 and variances σ_a^2, σ_b^2). Since these functions are straight lines, one can think of υ(x) as a straight line with a random slope and intercept. For any given value of x, one could ask what the expected value of υ(x) is, and also what its variance is.

It is possible to derive the mean value of υ(x) for any value of x by noting that

$$E(\upsilon(x)) = E(a) + E(b)x \qquad (6.21)$$

and that since $E(a) = E(b) = 0$, $E(\upsilon(x)) = 0$. This implies that if a sample of several random straight lines were generated in this way, taking their average value would give something close to zero, regardless of x. However, although the average value of υ(x) may be close to zero, how might its variance change with x? Since a and b are independent,

(6.22)

$$\mathrm{Var}(\upsilon(x)) = \sigma_a^2 + x^2\sigma_b^2 \qquad (6.22)$$

Thus, variance of the expected value of υ(x) increases with large absolute values of x.

Finally, suppose the function was evaluated at two values of x, say x_1 and x_2. Then some similar, but more complex, working shows that the correlation

between $\upsilon(x_1)$ and $\upsilon(x_2)$ is

(6.23)

$$\frac{\sigma_a^2 + x_1 x_2 \sigma_b^2}{\sqrt{(\sigma_a^2 + x_1^2 \sigma_b^2)(\sigma_a^2 + x_2^2 \sigma_b^2)}} \qquad (6.23)$$

The idea here is that it is possible to define a correlation function that is related to the initial random function. It is possible, in some cases, to reverse this notion, and to define a random function in terms of the bivariate correlation function. This idea is central to kriging and geostatistics. In this case, however, a number of extensions to the above idea are applied:

- The function is defined for a vector **x** rather than a scalar x
- Stationarity: The correlation function depends only in terms of the distance between two vectors: say, $\rho(|\mathbf{x}1 - \mathbf{x}2|) = \rho(d)$ for some correlation function ρ
- Typically the relationship is defined in terms of the *variogram*: $\gamma(d) = 2\sigma^2(1 - \rho(d))$
- The function $\upsilon(\mathbf{x})$ is not specified directly, but deduced by 'working backwards' from $\gamma(d)$ and some observed data

The last modification is really just convention – most practitioners of kriging specify the relationship between points in this way, rather than as a correlation or covariance. If the process is stationary, then

(6.24)

$$\gamma(d) = \frac{1}{2}\mathrm{E}\left[\left(\upsilon(\mathbf{x}_1) - \upsilon(\mathbf{x}_2)\right)^2\right] \qquad (6.24)$$

and this can be empirically estimated from data by taking average values of the squared difference of $\upsilon(\mathbf{x}1)$ and $\upsilon(\mathbf{x}2)$ where the distance between **x**1 and **x**2 falls into a specified band.

Not all functional forms are valid semivariograms; however, a number of functions that are valid are well known, such as those shown in Table 6.1.

In all of these functions, the degree of correlation between $\upsilon(\mathbf{x}1)$ and $\upsilon(\mathbf{x}2)$ is

assumed to reduce as distance increases. *a* and *b* are parameters respectively controlling the scale of variance and the extent to which nearby observations are correlated. For the Matérn semivariogram, κ is an additional shape parameter, and $K_\kappa(\cdot)$ is a modified Bessel function of order κ. If $\kappa = \frac{1}{2}$ this is equivalent to an exponential semivariogram, and as $\kappa \to \infty$ it approaches a Gaussian semivariogram.

Table 6.1

Table 6.1 Some semivariogram functions

Name	Functional form
Exponential	$\gamma(d) = a(1 - \exp(-d / b))$
Gaussian	$\gamma(d) = a(1 - \exp(-\frac{1}{2} d^2 / b^2))$
Matérn	$\gamma(d) = a\left(1 - \left(2^{\kappa-1}\Gamma(\kappa)\right)^{-1} (d / b)^\kappa K_\kappa(d / b)\right)$
Spherical	$\gamma(d) = \begin{cases} a\left(\frac{3}{2}(d / b) - \frac{1}{2}(d / b)^3\right) & \text{if } d \le b \\ a & \text{otherwise} \end{cases}$

6.8.3 Estimating the Semivariogram

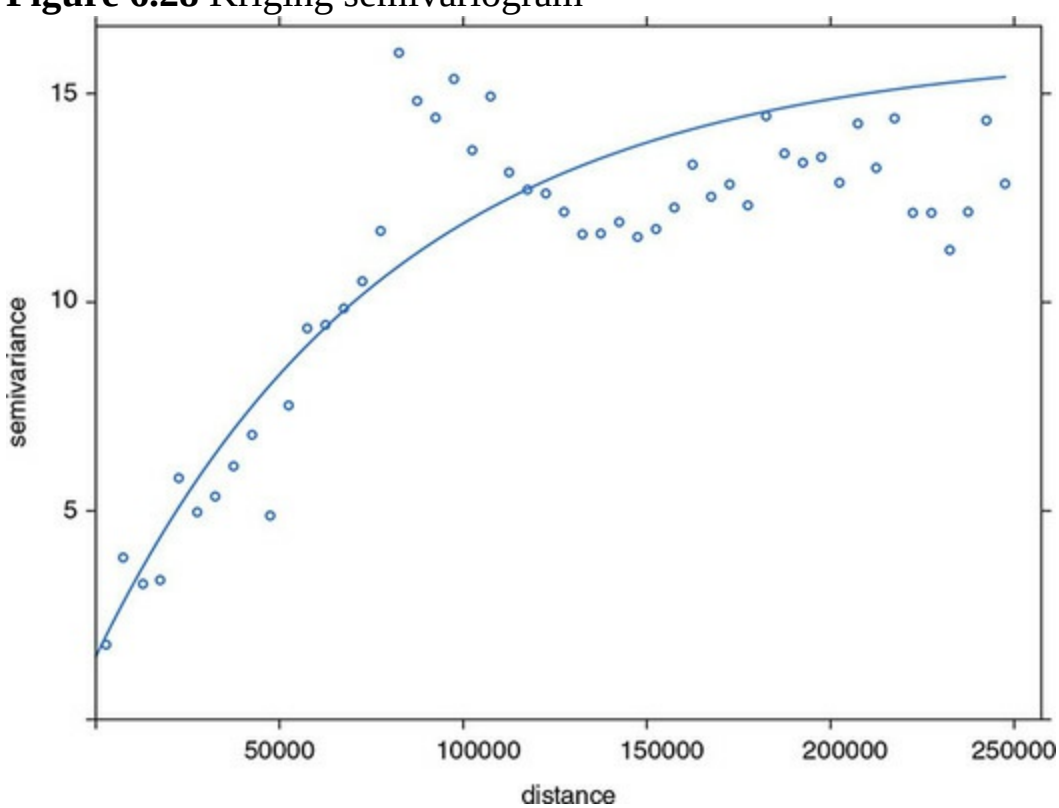
As suggested earlier, equation (6.24) can be used as a way of estimating the semivariogram. Essentially all pairwise point distances are grouped into bands, and the average squared difference between υ(**x**1) and υ(**x**2) is computed for each band. Then, for one of the semivariogram functions listed above (or possibly another), a semivariogram curve is fitted – this involves finding the values of *a* and *b* that best fit the banded average squared differences described above. Note that for the Matérn case, κ is sampled at a small number of values, rather than finding a precise optimal value.

Once this is done, although υ(**x**) has not been explicitly calibrated, an estimate of γ(*d*) is now available. In the R package gstat the semivariogram estimation procedure is carried out with the variogram function. The boundaries argument specifies the distance bands to work with. Here it is used with the fulmar data, and the boundaries are in steps of 5 km up to 250 km. The result of this is stored in evgm. Following the calibration of the estimated semivariogram evgm, by grouped averaging as described above, a semivariogram curve is fitted – in this case a Matérn curve. The kind of curve

to fit is specified in the vgm function. The parameters are, in order, an estimate of *a*, a specification of the kind of semivariogram (Mat is Matérn, Exp is exponential, Gau is Gaussian and Sph is spherical), *b* and κ. Note that the values provided here are initial guesses – the fit.variogram function takes this specification and the evgm and calibrates the parameters to get a best-fit semivariogram. The result is then plotted using the plot function (see Figure 6.28).

```
require(gstat)
evgm <- variogram(fulmar~1,fulmar.spdf,
boundaries=seq(0,250000,l=51))
fvgm <- fit.variogram(evgm,vgm(3,"Mat",100000,1))
plot(evgm,model=fvgm)
```

Figure 6.28 Kriging semivariogram



Once a semivariogram has been fitted it can be used to carry out the interpolation. The fit.variogram function estimates the 'nugget' variance

discussed earlier, as well as the semivariogram parameters. Once this is done, it is possible to carry out the interpolation – essentially, if a set of *zi* values is available at locations **x***i* for *i* = 1,…, *n* and an estimate of γ(*d*) is available, then *f*(**x**) and υ(**x**) can be estimated for arbitrary **x** locations. Until this point, the estimation of the trend *f*(**x**) has not been considered, but it is possible to estimate this (using more conventional regression approaches), or, if the trend is just a constant value μ, say, then the calibration of this value, and the estimation of μ + υ(**x**) (essentially the interpolated value), can be carried out simultaneously using a technique termed *ordinary kriging* (see Wackernagel, 2003: 31, for example).

Operationally the interpolations are achieved by taking a weighted combination of the *zi* values, $\sum_i w_i z_i$. In matrix form, if *wi* is the weight applied to *zi*, $\hat{\mu}$ is an estimate of μ, *dij* is the distance between **x***i* and **x***j*, and *di* is the distance between sample location **x***i* and **x**, an arbitrary location at which it is desired to carry out the interpolation, then

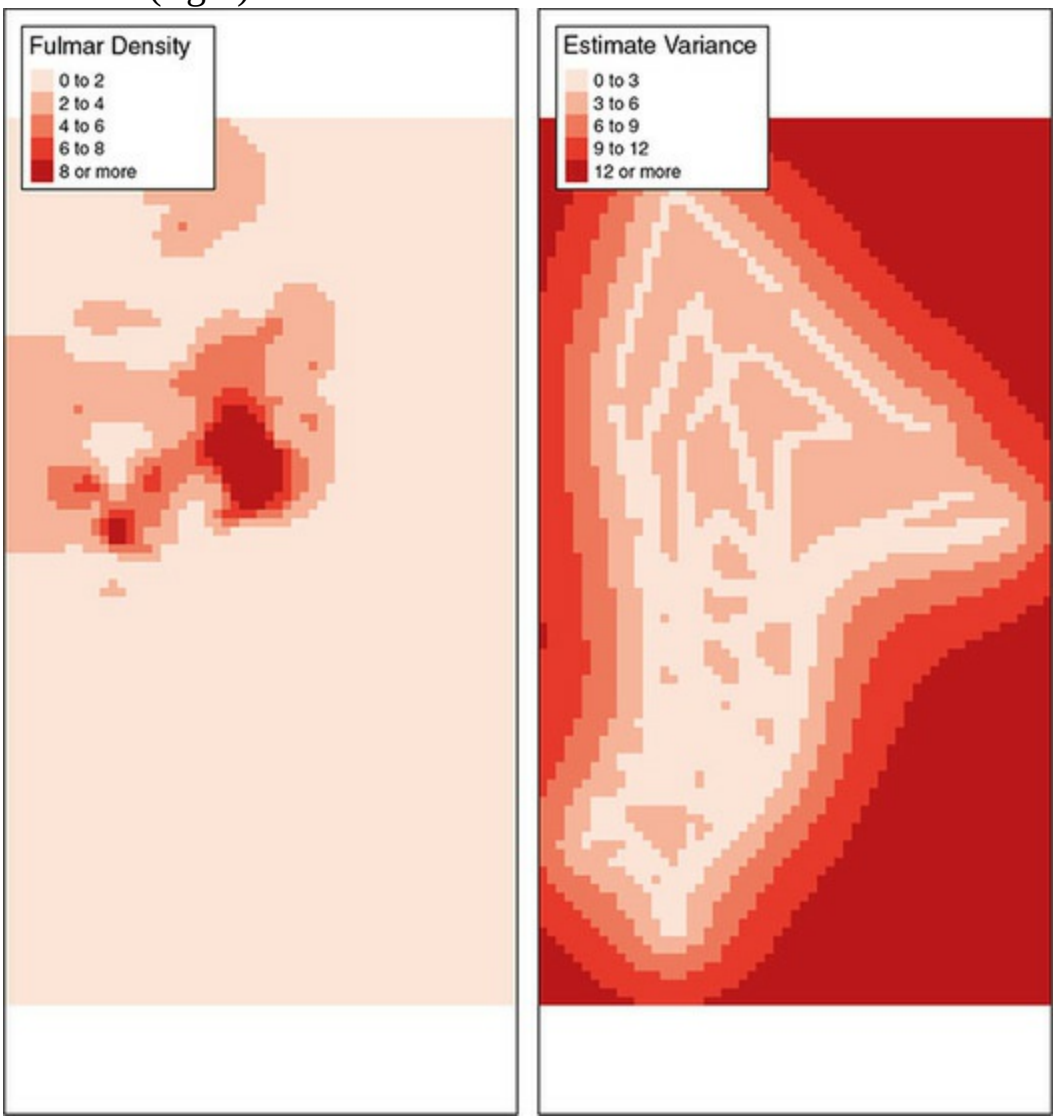
(6.25)

$$\begin{bmatrix} w_1 \\ \vdots \\ w_n \\ 1 \end{bmatrix} = \begin{bmatrix} \gamma(d_{11}) & \cdots & \gamma(d_{1n}) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(d_{n1}) & \cdots & \gamma(d_{nn}) & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \gamma(d_1) \\ \vdots \\ \gamma(d_n) \\ 1 \end{bmatrix} \qquad (6.25)$$

However, users of gstat do not need to implement this explicitly, as it is made available via the krige function. This works in much the same way as the idw function, although the semivariogram model must also be supplied. This is carried out below, followed by a drawing of the fulmar density surface in the same way as before. An added bonus of kriging is that it is possible to obtain variances of the interpolated estimates as well as the estimate itself – these are derived from the statistical model and stored in var1.var, alongside var1.est. They are useful, as they give an indication of the reliability of the estimates. Below, both the interpolated values and their variances are computed and shown in raster plots (Figure 6.29).

Figure 6.29 Kriging estimates of fulmar density (left), and associated

variance (right)



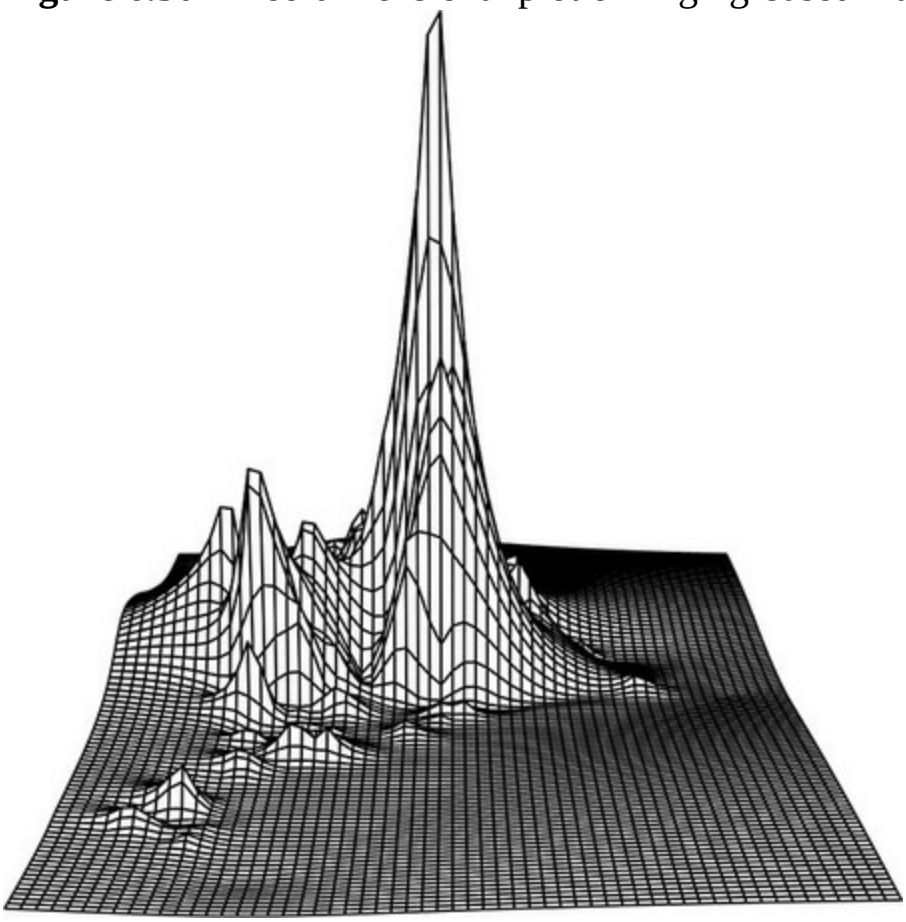

```
krig.est <- krige(fulmar~1,fulmar.spdf,newdata=s.grid,model=fvgm)

krig.grid <- SpatialPixelsDataFrame(krig.est,krig.est@data)

krig.map.est <- tm_shape(krig.grid) +
tm_raster(col='var1.pred',breaks=levs,title='Fulmar Density',palette='Reds') +
tm_layout(legend.bg.color='white',legend.frame = TRUE)

var.levs <- c(0,3,6,9,12,Inf)

krig.map.var <- tm_shape(krig.grid) +
tm_raster(col='var1.var',breaks=var.levs,title='Estimate Variance',palette='Reds') +
tm_layout(legend.bg.color='white',legend.frame = TRUE)

tmap_arrange(krig.map.est,krig.map.var)
```

The plots show the interpolation and the variance. Note that on the variance map, levels are lowest (and hence reliability is highest) near to the transect flight paths – generally speaking, interpolations are at their most reliable when they are close to the observation locations.

Finally, a perspective plot shows that although the interpolated surface is still fairly rough, some of the 'spikiness' of the IDW surface has been removed, as the surface is not forced to pass through all of the *zi* (see Figure 6.30).

```
par ( mfrow=c(1,1)) # reset plotting to make plot fill the entire window

predmat3 <- matrix(krig.est$var1.pred,length(ux),length(uy))

persp(predmat3,box=FALSE)
```

Figure 6.30 Three-dimensional plot of kriging-based interpolation



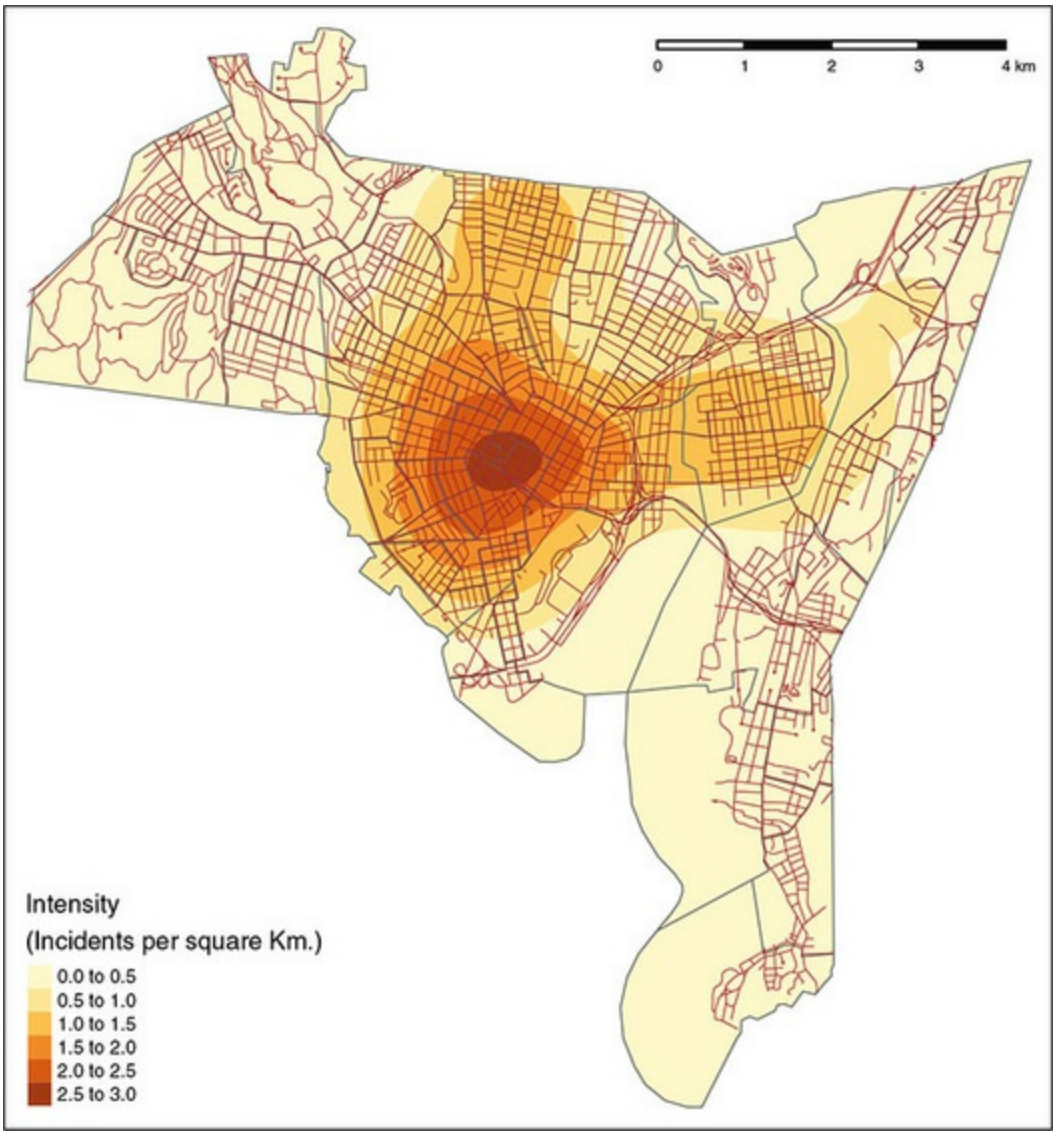
Self-Test Question 2. Try fitting an exponential variogram to the fulmar

data, and creating the surface plot and maps. You may want to look at the help for fit.variogram to find out how to specify alternative variogram models.

6.9 Concluding Remarks

In this chapter, a number of techniques for analysing random patterns of two-dimensional points (with associated measurements in the case of interpolation) have been outlined. The key areas are first-order approaches (where the probability density function for the process is assumed to vary, and an attempt is made to estimate it) and second-order approaches (where the dependency between the spatial distributions of points is considered – this includes *K*-functions and related topics, as well as kriging). Although the chapter does not cover all possible aspects of this, it should provide an overview. As a further exercise, the reader may wish to investigate, for example, *H*-functions (Hansen et al., 1999) and their implementation in spatstat, or *universal kriging* (Wackernagel, 2003) where the deterministic trend function is assumed to be something more complex than a constant, as in ordinary kriging.

Figure 6.31 KDE map for breaches of the peace, for Self-Test Question 1



6.10 Answers to Self-Test Questions

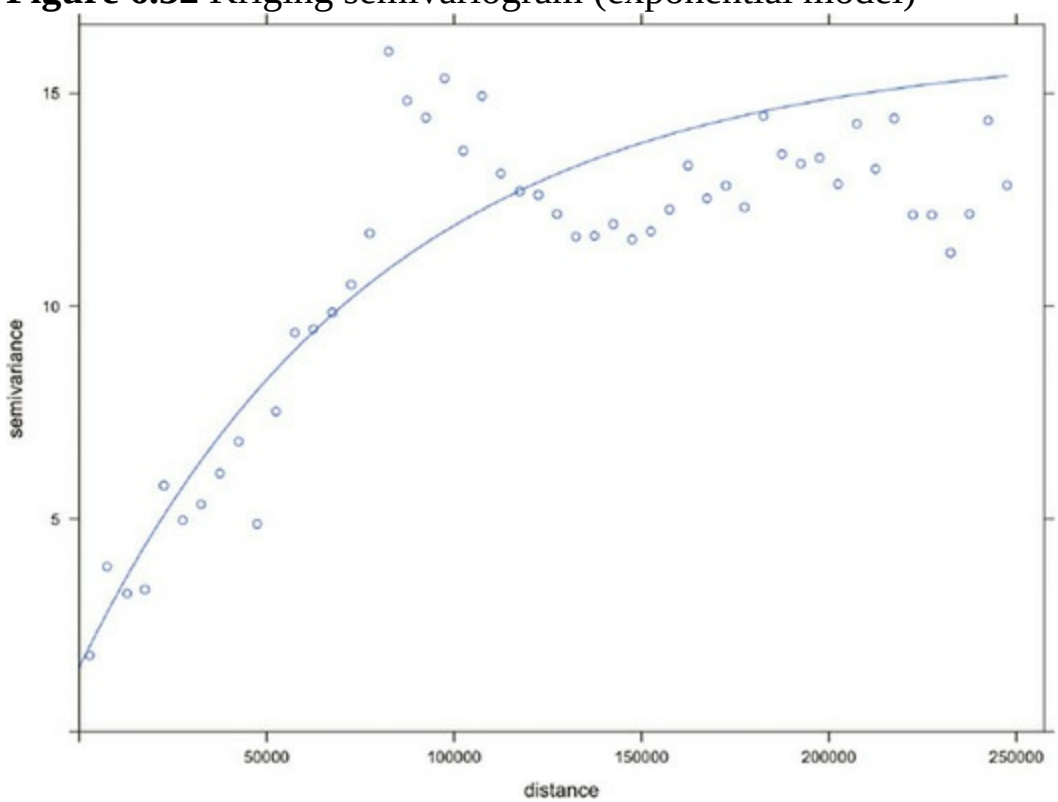
Q1: The suggested map (Figure 6.31) could be achieved with the following code. The title='Intensity' option in tm_fill() changes the label on the legend to a more informative one than the default provides.

```
tmap_mode('plot')

tm_shape(breach_dens$polygons) +tm_fill(title='Intensity \n(Incidents per square Km.)')
+tm_shape(blocks) + tm_borders() +tm_shape(roads) + tm_lines(lwd=0.5,col='brown')
+tm_scale_bar(position=c("right","top"))
```

Q2: The exponential variogram model is specified using the "Exp" argument in fit.variogram – the code to produce the variogram is given below, and the result is shown in Figure 6.32. Following this, the same procedures for producing a perspective plot or contour maps used in the above example may also be applied here.

```
evgm <- variogram(fulmar~1,fulmar.spdf, boundaries=seq(0,250000,l=51))

fvgm <- fit.variogram(evgm,vgm(3,"Exp",100000,1))

plot(evgm,model=fvgm)
```

Figure 6.32 Kriging semivariogram (exponential model)



References

Besag, J. (1977) Discussion of Dr. Ripley's paper. Journal of the Royal Statistical Society, Series B, 39: 193–195.

Bland, J.M. and Altman, D.G. (1995) Multiple significance tests: The

Bonferroni method. British Medical Journal, 310: 170.

Bowman, A. and Azzalini, A. (1997) Applied Smoothing Techniques for Data Analysis: The Kernel Approach with S-Plus Illustrations. Oxford: Oxford University Press.

Brunsdon, C. (2009) Geostatistical analysis of lidar data. In G. Heritage and A. Large (eds), Laser Scanning for the Environmental Sciences. Chichester: Wiley-Blackwell.

Cressie, N. (1993) Statistics for Spatial Data. New York: John Wiley & Sons.

Diggle, P.J. (1983) Statistical Analysis of Spatial Point Patterns. London: Academic Press.

Hansen, M.B., Baddeley, A.J. and Gill, R.D. (1999) First contact distributions for spatial patterns: Regularity and estimation. Advances in Applied Probability, 31: 15–33.

Hutchings, M.J. (1979) Standing crop and pattern in pure stands of *Mercurialis perennis* and *Rubus fruticosus* in mixed deciduous woodland. Oikos, 31: 351–357.

Loosmore, N.B. and Ford, E.D. (2006) Statistical inference using the G or K point pattern spatial statistics. Ecology, 87(8): 1925–1931.

Lotwick, H.W. and Silverman, B.W. (1982) Methods for analysing spatial processes of several types of points. Journal of the Royal Statistical Society, Series B, 44(3): 406–413.

Matheron, G. (1963) Principles of geostatistics. Economic Geology, 58: 1246–1266.

Ripley, B.D. (1976) The second-order analysis of stationary point processes. Journal of Applied Probability, 13: 255–266.

Ripley, B.D. (1977) Modelling spatial patterns (with discussion). Journal of the Royal Statistical Society, Series B, 39: 172–212.

Ripley, B.D. (1981) Spatial Statistics. New York: John Wiley & Sons.

Scott, D. (1992) Multivariate Density Estimation: Theory, Practice, and Visualization. New York: John Wiley & Sons.

Silverman, B.W. (1986) Density Estimation for Statistics and Data Analysis. London: Chapman & Hall.

Tufte, E.R. (1990) Envisioning Information. Cheshire, CT: Graphics Press.

Wackernagel, H. (2003) Multivariate Geostatistics. Berlin: Springer.

7 Spatial Attribute Analysis with R

7.1 Introduction

Spatially referenced observations are a kind of data that is very similar to an ordinary dataset, for example a set of observations[1] $\{z_1, z_2, \ldots, z_n\}$. The only difference is that each observation is associated with some form of spatial reference – typically a point or a polygon. Unlike the processes modelled in point pattern analysis, the polygons or points are considered as fixed, non-random entities. Here, the observations $\{z_1, z_2, \ldots, z_n\}$ are the random quantities. In one kind of frequently used model, the probability distributions of the z_i depend on their spatial references, and some other parameters, which may need to be estimated from the data. For example, if each observation z_i is referenced by a spatial location (x_i, y_i) then it may be modelled by a normal distribution with variance σ^2 and mean $a + bx_i + cy_i$; thus the distribution of z_i depends on the spatial location and the parameters a, b, c and σ. A model of this kind is useful for modelling broad geographical trends – for example, whether house prices to the east of a state in the USA tend to be lower or higher than those to the west. This situation might be the case if the state is on the coast, and housing closer to the coast generally fetches a higher price.

[1] Here we use z for the data values, as x and y are sometimes used to denote location.

An alternative approach is to model the correlation between observations z_i and z_j as dependent on their spatial references. For example, the variables $\{z_1, z_2, \ldots, z_n\}$ may have a multivariate normal distribution whose variance–covariance matrix (giving the covariances between each pair of z variables) depends on the distances between points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. Alternatively, if the observations are associated with polygons rather than

points (e.g. this would be the case if the z_i were unemployment rates for counties, with county boundaries expressed as polygons) then correlations or covariances could be a function of the *adjacency matrix* of the polygons, a 0–1 matrix indicating whether each polygon pair share any common boundary. This can be the case where processes can be thought of as random (unlike the fixed pattern due to proximity to the coast in the last example) but still exhibiting spatial patterns – for example, the measurement of crop yields, where groups of nearby fields may exhibit similar values due to shared soil characteristics.

7.2 The Pennsylvania Lung Cancer Data

In this section, the dataset that will be used in the first set of examples will be introduced. This is a set of county-level lung cancer counts for 2002. The counts are stratified by ethnicity (with rather broad categories 'white' and 'other'), gender and age ('under 40', '40 to 59', '60 to 69' and 'over 70'). In addition, a table of proportion of smokers per county is provided. Population data were obtained from the 2000 decennial census, and lung cancer and smoking data were obtained from the Pennsylvania Department of Health website.[2] All of these data are provided by the SpatialEpi package – so it will be necessary to install the package and its dependencies before trying the code segments in this chapter. To do this from the command line in R, ensure your computer is connected to the internet, and that you have appropriate permissions, and then enter:

[2] http://www.dsf.health.state.pa.us/

```
install.packages('SpatialEpi', depend=TRUE)
```

In conjunction with tmap, it is then possible to use this dataset – which is stored in an object called pennLC. This is a list with a number of components:

- geo A table of county IDs, with longitudes and latitudes of the geographical centroid of each county
- data A table of county IDs, number of cases, population subdivided by

race, gender and age
- smoking A table of county IDs and proportion of smokers
- spatial.polygon A SpatialPolygons object giving the boundaries of each county in latitude and longitude (geographical coordinates)

Using the packages tmap and tmaptools, for example, standard methods may be used to produce a choropleth map of smoking uptake in Pennsylvania. In the code below (all making use of techniques from earlier chapters), the map of Pennsylvania is transformed from geographical coordinates to UTM projection[3] for zone 17. Note that this has European Petroleum Survey Group (EPSG) reference number[4] 3724, as is used in the set_projection function. These are then used to create a choropleth map as seen in Figure 7.1. Note that this is produced on a notional window of 11 cm × 6 cm – you may have to resize the window or set par(mar=c(0,0,0,0)) to ensure the legend is visible.

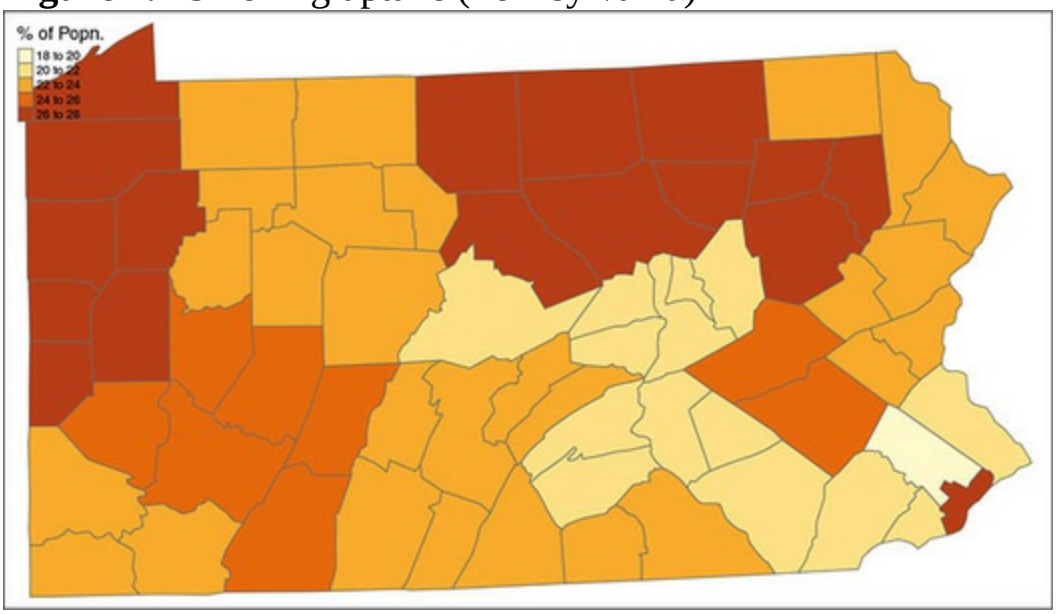
[3] http://www.history.noaa.gov/stories_tales/geod1.html

[4] http://www.epsg.org

```
# Make sure the necessary packages have been loaded

library(tmap)

library(tmaptools)

library(SpatialEpi)

# Read in the Pennsylvania lung cancer data

data(pennLC)

# Extract the SpatialPolygon info

penn.state.latlong <- pennLC$spatial.polygon

# Convert to UTM zone 17N

penn.state.utm <- set_projection(penn.state.latlong, 3724)

if ("sf" %in% class(penn.state.latlong))

penn.state.utm <- as(penn.state.utm, "Spatial")
```

```
# Obtain the smoking rates

penn.state.utm$smk <- pennLC$smoking$smoking * 100

# Draw a choropleth map of the smoking rates

tm_shape(penn.state.utm) + tm_polygons(col='smk',title= '% of Popn.')
```

Figure 7.1 Smoking uptake (Pennsylvania)



This produces a basic choropleth map of smoking rates in Pennsylvania. From this, it may be seen that these tend to show some degree of spatial clustering – counties having higher rates of uptake are generally near to other counties with higher rates of uptake, and similarly for lower rates of uptake. This is quite a common occurrence – and this kind of spatial clustering will be seen in the coming sections, for smoking rates, patterns in death rates, and in the classes used in the stratification of the population.

7.3 A Visual Exploration of Autocorrelation

An important descriptive statistic for spatially referenced attribute data – and, in particular, measurement scale data – is spatial autocorrelation. In Figure 7.1 it was seen that counties in Pennsylvania tended to have similar smoking uptake rates to their neighbours. This is a way in which spatial attribute data are sometimes different from other data, and it suggests that models used for other data are not always appropriate. In particular, many statistical tests and

models are based on the assumption that each observation in a set of measurements is distributed independently of the others – so that in a set of observations $\{z_1, z_2, \ldots, z_n\}$, each z_i is modelled as being drawn from, say, a Gaussian distribution, with probability density

(7.1)

$$\phi(z_i \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(z_i - \mu)^2}{2\sigma^2}\right] \qquad (7.1)$$

where μ and σ are respectively the population mean and standard deviation of the distribution of the data. However, the distribution itself is not a key issue here. More important is the assumption that for each z_i the distribution is independent of the other observations $\{z_1, z_2, \ldots, z_{i-1}, z_{t+1}, \ldots, z_n\}$, so that the joint distribution is

(7.2)

$$\Phi(\mathbf{z}) = \prod_{i=1}^{n} \phi(z_i \mid \mu, \sigma) \qquad (7.2)$$

where **z** denotes the vector $(z_1, z_2, \ldots, z_n)^T$. The reason why this common assumption is important here is that it is frequently untrue for spatial data. Figure 7.1 suggests that it is unlikely that, say, two observations *zi* and *zj* are independent, if *i* and *j* index adjacent counties in Pennsylvania. It seems that a more realistic model would allow for some degree of correlation between nearby observations. Correlation of this kind is referred to as *spatial autocorrelation*. There are a number of ways in which spatial autocorrelation can be modelled, but in this section visual exploration will be considered.

We begin by scrutinising the claim that the image in Figure 7.1 really does demonstrate correlation. This can be done via significance testing in later sections, but here some useful visual approaches will be outlined. The first of these is to compare the pattern seen in the map to a set of random patterns, where the observed smoking rates are assigned randomly to counties. Here, six maps are drawn, one based on the actual data and the rest created using random permutations. These are drawn in a 2 × 3 rectangular grid

arrangement. For the random part of this, the sample function is used. Given a single argument, which is a vector, this function returns a random permutation of that argument. Thus, sample(penn.state.utm$smk) will permute the smoking uptakes among the counties. This produces five alternative allocations of rates in addition to the actual one. The sample(c('smk','smk_rand1',… ,'smk_rand5')) expression in the following code block scrambles the six variable names, and these in turn are given to the tm_polygons function in that scrambled order. Not only are five of the six maps based on random permutations, but also the position in the figure of the actual data map is chosen at random. The idea of this second randomisation is that not even the coder will know which of the six maps represents the true data. If on inspection there is one clearly different pattern – and it appears obvious which of the maps this is – then there is strong visual evidence of autocorrelation.

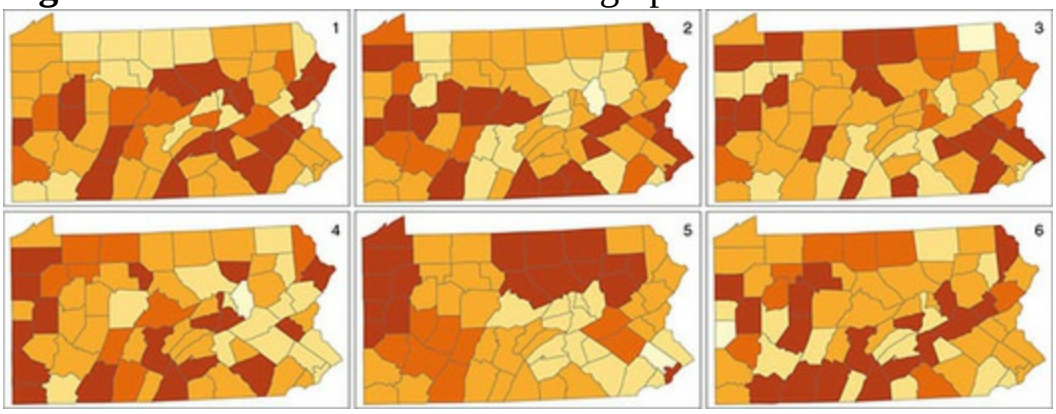
```
# Set up a set of five 'fake' smoking update rates as well as the real one

# Create new columns in penn.state.utm for randomised data

# Here the seed 4676 is used. Use a different one to get an unknown outcome.

set.seed(4676)

penn.state.utm$smk_rand1 <- sample(penn.state.utm$smk)

penn.state.utm$smk_rand2 <- sample(penn.state.utm$smk)

penn.state.utm$smk_rand3 <- sample(penn.state.utm$smk)

penn.state.utm$smk_rand4 <- sample(penn.state.utm$smk)

penn.state.utm$smk_rand5 <- sample(penn.state.utm$smk)

# Scramble the variables used in terms of plotting order

vars <- sample(c('smk','smk_rand1','smk_rand2','smk_rand3','smk_rand4','smk_rand5'))

# Which one will be the real data?

# Don't look at this variable before you see the maps!

real.data.i <- which(vars == 'smk')

# Draw the scrambled map grid

tm_shape(penn.state.utm) + tm_polygons(col=vars,legend.show=FALSE) +
```

```
tm_layout(title= 1: 6, title.position= c("right","top"))
```

Having drawn these maps (see Figure 7.2), an informal self-test is to reveal which map is the real data:

```
real.data.i

[1] 5
```

Figure 7.2 Randomisation of smoking uptake rates



One point worth making is that the ‘random’ maps do show some groups of similar neighbours – commonly this is the case: the human eye tends to settle on regularities in maps, giving a tendency to identify clusters, even when the data are generated by a process without spatial clustering. This is why procedures such as the previous one are necessary, to make visual cluster identification more robust. This approach is a variant on that due to Wickham et al. (2010).

7.3.1 Neighbours and Lagged Mean Plots

An alternative visual approach is to compare the value in each county with the average value for its neighbours. This can be achieved via the lag.listw function in the spdep library. This library provides a number of tools for handling data with spatial referencing, particularly data that are attributes of SpatialPolygons such as the Pennsylvania data here. A lagged mean plot can be generated if we have a list of which counties each county has as neighbours. Neighbours can be defined in several ways, but a common

definition is that a pair of counties (or other polygons in different examples) which share some part of their boundaries are neighbours. If this is the *queen's case* definition, then even a pair of counties meeting at a single corner point are considered neighbours. The more restrictive *rook's case* requires that the common boundary must be a linear feature. This is illustrated in Figure 7.3. Neighbour lists of either case can be extracted using the poly2nb function in spdep. These are stored in an nb object – basically a list of neighbouring polygons for each polygon.

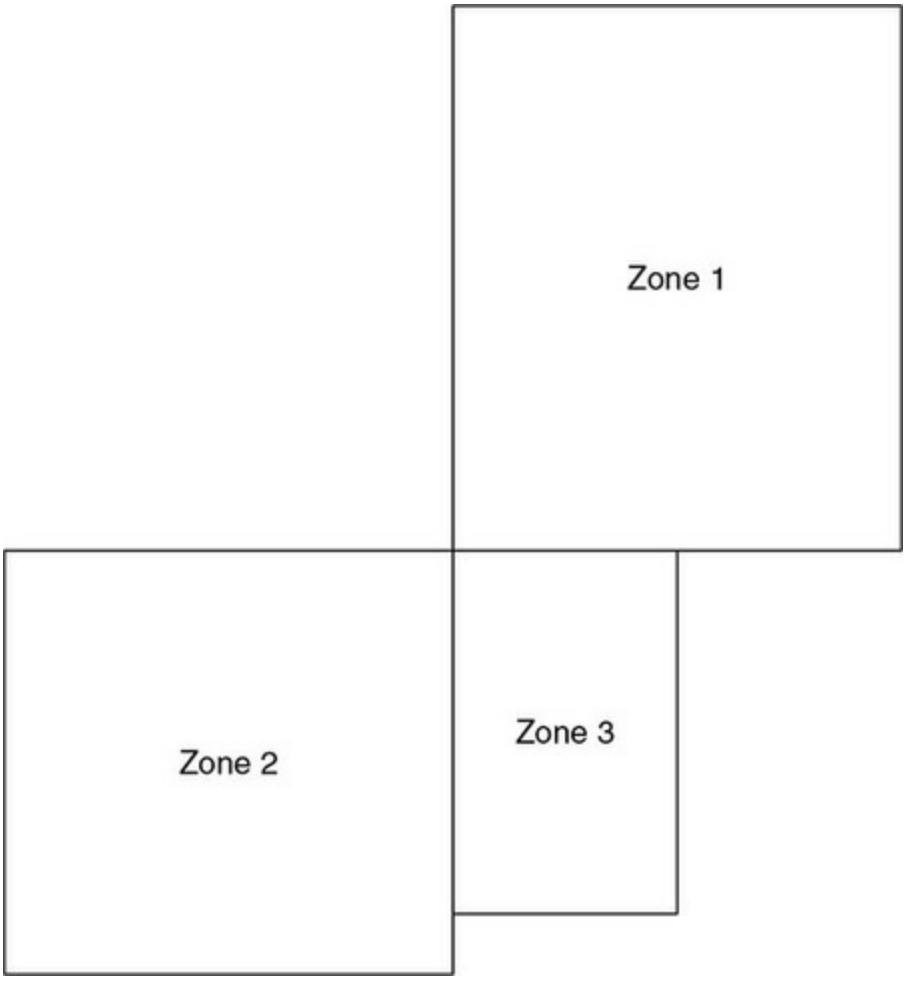
```
require(spdep)

penn.state.nb <- poly2nb(penn.state.utm)

penn.state.nb

Neighbour list object:

Number of regions: 67

Number of nonzero links: 346

Percentage nonzero weights: 7.70773

Average number of links: 5.164179
```

As seen in the block above, printing out an nb object lists various characteristics, such as the average number of neighbours each polygon has – in this case 5.164. Note that in the default situation, Queen's case neighbours are computed. To compute the Rook's case, the optional argument queen=FALSE is added to poly2nb.

It is also possible to plot an nb object – this represents neighbours as a network (see Figure 7.4). First, it is turned into a SpatiaLinesDataFrame object by nb2lines, a function in spdep. The lines join the centroids of the polygons regarded as neighbours, and may be plotted as a map. The centroid locations are provided by the coordinates function. One further thing to note is that by default, nb2lines sets the projection of the result to NA. This is overwritten using the current.projection option in set_projection, so the result is EPSG 3724, the correct UTM projection here.

Figure 7.3 Rook's case and queen's case neighbours: zones 1 and 2 are

neighbours only under queen's case; zone pairs 1,3 and 2,3 are neighbours under both cases

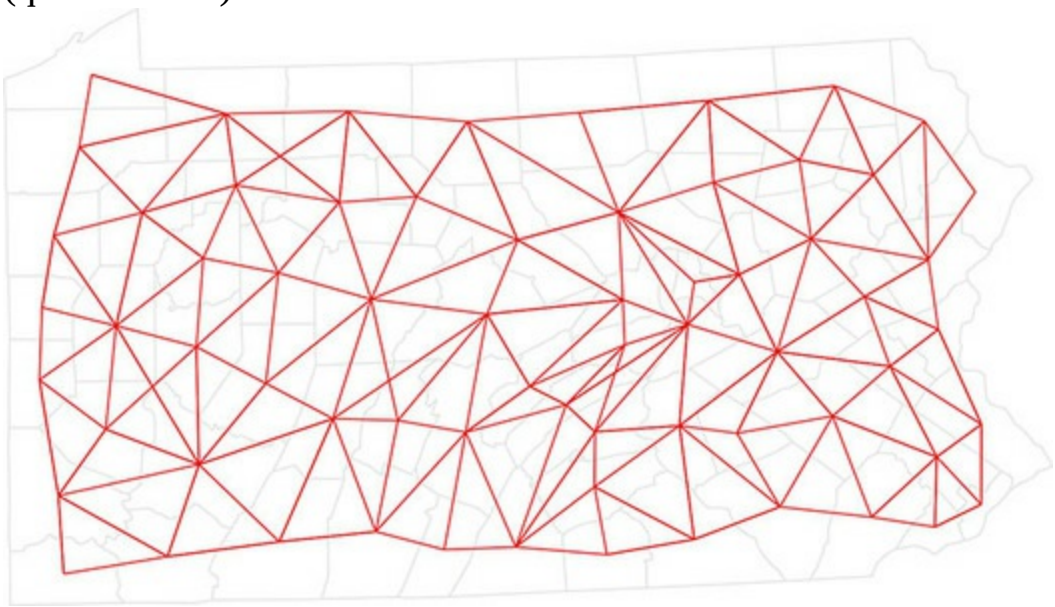



```
# Create a SpatialLinesDataFrame showing the Queen's case contiguities
penn.state.net <- nb2lines(penn.state.nb,coords=coordinates(penn.state.utm))
# Default projection is NA, can change this as below
penn.state.net <- set_projection(penn.state.net,current.projection = 3724)
# Draw the projections
tm_shape(penn.state.utm) + tm_borders(col='lightgrey') +
   tm_shape(penn.state.net) + tm_lines(col='red')
```

Finally, the plots are also useful to compare the rook's case to the queen's case neighbourhoods (see Figure 7.5):

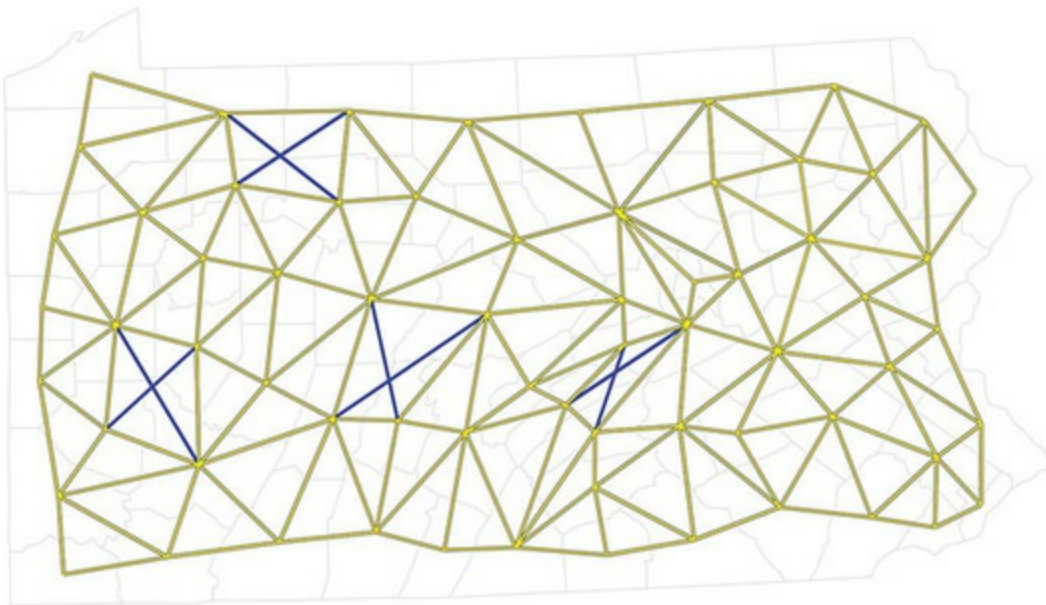
```
# Calculate the Rook's case neighbours

penn.state.nb2 <- poly2nb(penn.state.utm, queen=FALSE)

# Convert this to a SpatialLinesDataFrame

penn.state.net2 <- nb2lines(penn.state.nb2, coords=coordinates(penn.state.utm))

# Update projection

penn.state.net2 <- set_projection(penn.state.net2, current.projection = 3724)

# Plot the counties in background, then the two networks to compare:

tm_shape(penn.state.utm) + tm_borders(col='lightgrey') +     tm_shape(penn.state.net) +
tm_lines(col='blue', lwd = 2) +     tm_shape(penn.state.net2) + tm_lines(col='yellow')
```

Figure 7.4 Depiction of neighbouring counties of Penn State as a network (queen's case)



Here the queen's case only neighbours are apparent (these are the blue links on the network) – there are eight of these. For now, we will work with rook's case neighbours.

Figure 7.5 Comparison of neighbouring counties of Penn State (rook's vs. queen's case)



The next stage is to consider the lagged mean plot. As discussed above, this is a plot of the value of z_i for each polygon i against the mean of the z values for the neighbours of polygon i. If δ_i is the set of indices of the neighbours of polygon i, and $|\delta_i|$ is the number of elements in this set, then this mean (denoted by $\tilde{z}_i$) is defined by

(7.3)

$$\tilde{z}_i = \sum_{j \in \delta_i} \frac{1}{|\delta_i|} z_i \qquad (7.3)$$

Thus, the lagged mean is a weighted combination of values of the neighbours. In this case, the weights are the same within each neighbour list, but in some cases they may differ (e.g. if weighting were inversely related to distance between polygon centres).

In spdep another kind of object – the listw object – is used to store a list of neighbours, together with their weights. A listw object can be created from an nb object using the nb2listw function in spdep:

```
# Convert the neighbour list to a listw object - use Rook's case...

penn.state.lw <- nb2listw(penn.state.nb2)
```

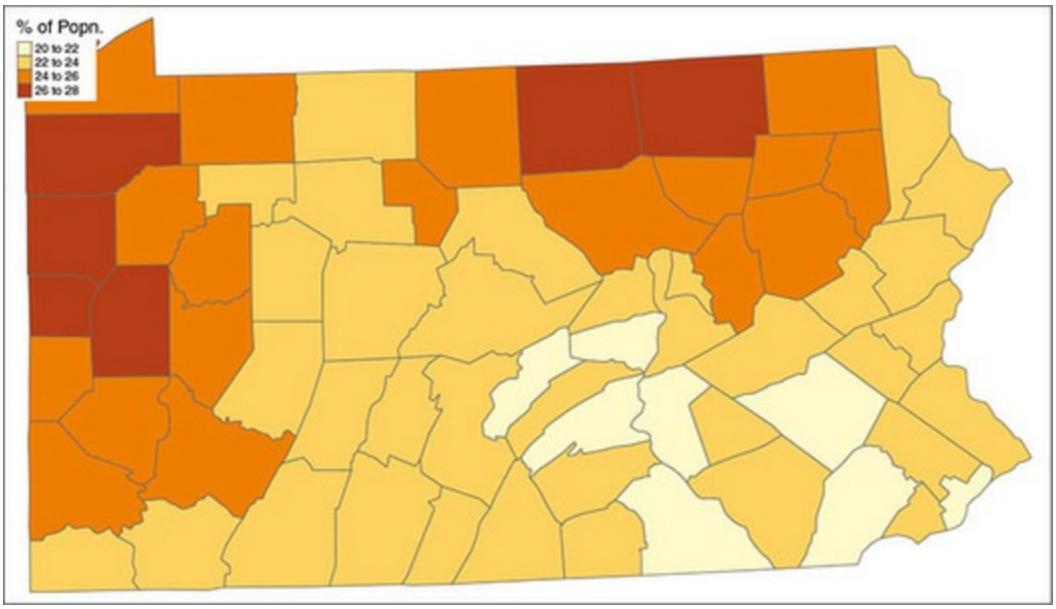
```
penn.state.lw

Characteristics of weights list object:

Neighbour list object:

Number of regions: 67

Number of nonzero links: 330

Percentage nonzero weights: 7.351303

Average number of links: 4.925373

Weights style: W

Weights constants summary:

   n    nn S0       S1       S2

W 67 4489 67 28.73789 274.6157
```

As a default, this function creates weights as given in equation (7.3) – this is the Weights style: W in the printout above. Other possible approaches to weights are possible – use ?nb2listw if you wish to investigate this further.

Having obtained a listw object, the function lag.listw computes a spatially lagged mean (i.e. a vector of $\tilde{z}_i$ values) – here, these are calculated and then mapped (see Figure 7.6):

```
penn.state.utm$smk.lagged.means <- lag.listw(penn.state.lw,penn.state.utm$smk)
tm_shape(penn.state.utm) + tm_polygons(col= 'smk.lagged.means',title= '% of Popn.') +

tm_layout(legend.bg.color = "white")
```

Figure 7.6 Lagged means of smoking uptake rates



Finally, a lagged mean plot is produced – as described, this is a scatter plot of z_i against $\tilde{z}_i$. Here the line $x = y$ is added to the plot as a point of reference. The idea is that when nearby polygons tend to have similar z_i values, there should be a linear trend in the plots. However, if each z_i is independent, then $\tilde{z}_i$ will be uncorrelated to z_i and the plots will show no pattern. Below, code is given to produce the plot shown in Figure 7.7:

```
with(data.frame(penn.state.utm), {
plot(smk,smk.lagged.means, asp= 1, xlim=range(smk), ylim=range(smk))
abline(a= 0, b= 1)
abline(v=mean(smk), lty= 2)
abline(h=mean(smk.lagged.means), lty= 2)
})
```

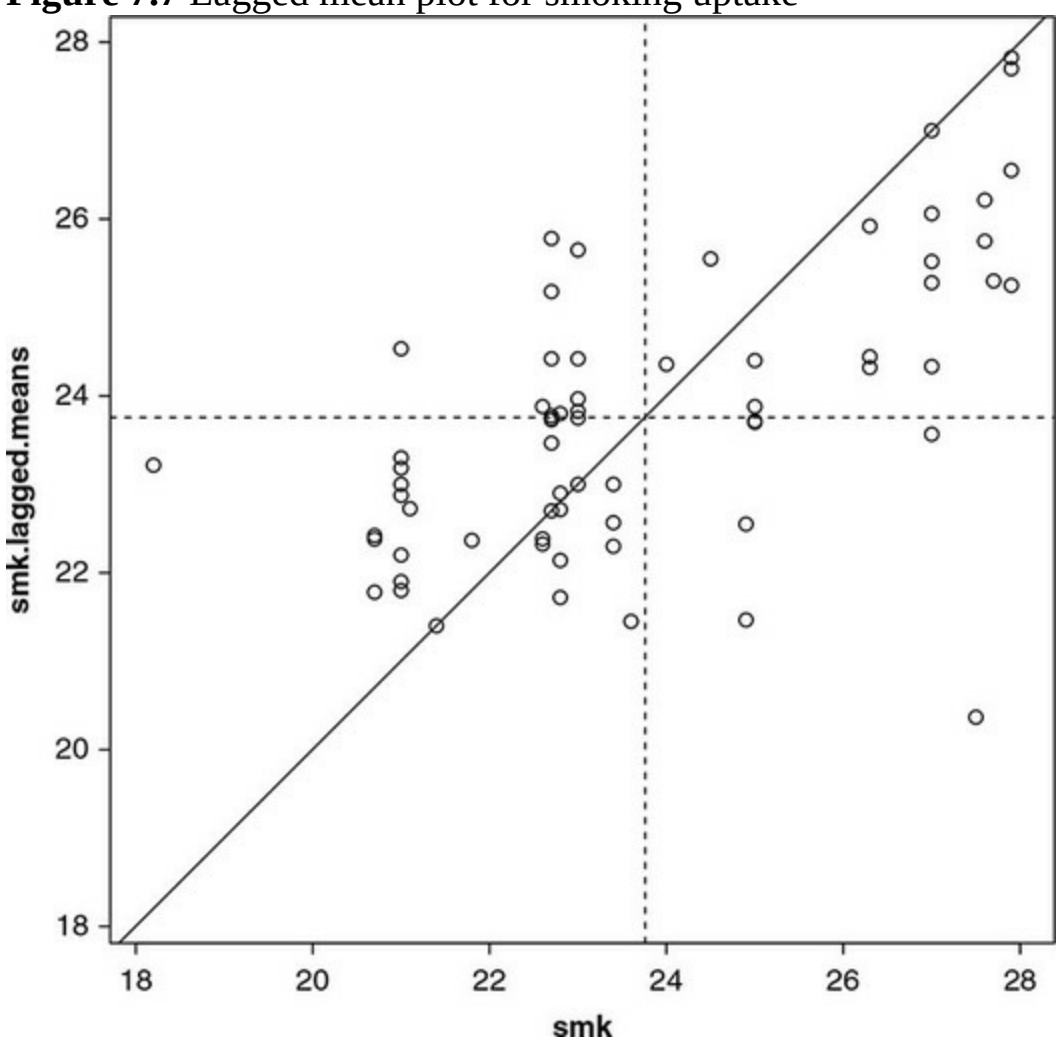
The abline(a=0,b=1) command adds the line $x = y$ to the plot. The two following abline commands add dotted horizontal and vertical lines through the mean values of the variables. The fact that more points lie in the bottom left and upper right quadrants created by the two lines suggests that there is some degree of positive association between z_i and $\tilde{z}_i$; this means generally

that when z_i is above average, so is $\tilde{z}_i$, and when one is below average, so is the other.

Note that this procedure is also termed a *Moran plot* or *Moran scatter plot* (see Anselin, 1995, 1996). In fact there is a function that combines the above steps, and adds some functionality, called moran.plot. However, working through the steps is helpful in demonstrating the ways in which spdep handles neighbour-based data. Below, the moran.plot approach is demonstrated. The output may be seen in Figure 7.8.

```
moran.plot(penn.state.utm$smk,penn.state.lw)
```

Figure 7.7 Lagged mean plot for smoking uptake

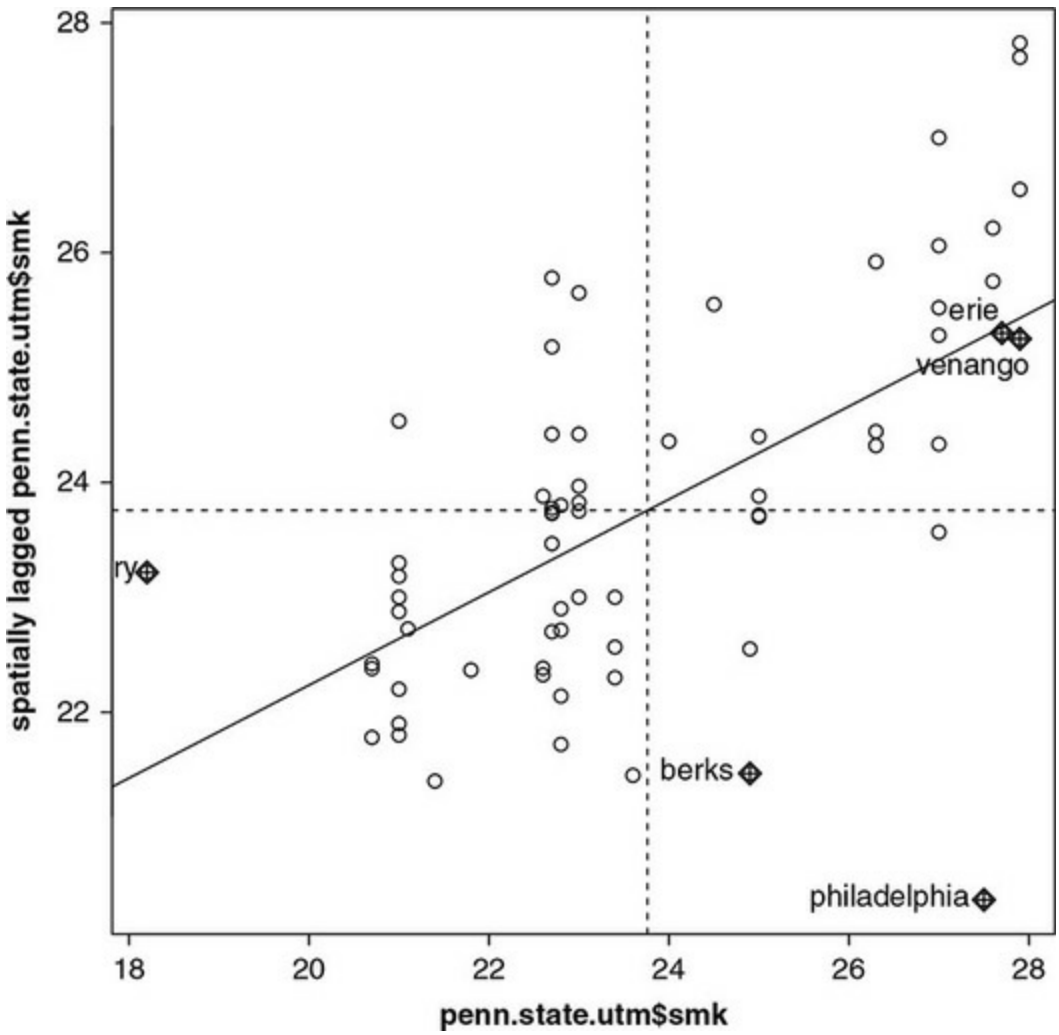


In addition to the code earlier, this approach also identifies points with a high influence in providing a best-fit line to the plot (see, for example, Belsley et al., 1980; Cook and Weisberg, 1982).

> **Self-Test Question 1.** One further modification of this approach is based on the observation that although the permutation approach does simulate no spatial influence on correlation, observations are in fact correlated – since the randomised data are a *permutation* of the actual data, the fact that *zi* gets assigned one particular value implies that no other variables can take this value.[5] An alternative simulation would assign values to counties based on sampling *with replacement*. In this case we are no longer conditioning on the exact set of observed uptake rates, but on an empirical estimate of the cumulative distribution function of the data, assuming that observations are independent. Modify the above code to carry out this alternative approach. *Hint*: use the help facility to find the optional arguments to the sample function.

[5] Unless there are repeated values in the data, but even then a similar argument applies.

Figure 7.8 Lagged mean plot for smoking uptake – alternative method



7.4 Moran's *I*: An Index of Autocorrelation

In this section, the exploratory approaches of Section 7.3 will be taken a step further. As stated earlier, autocorrelation is the tendency of z_i values of nearby polygons to be related. Rather like the Pearson correlation coefficient, which measures the dependency between a pair of variables, there are also coefficients (or indices) to measure autocorrelation. One that is very commonly used is *Moran's I* coefficient (Moran, 1950), defined by

(7.4)

$$I = \frac{n}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(z_i - \bar{z})(z_j - \bar{z})}{\sum_i (z_i - \bar{z})^2} \qquad (7.4)$$

where w_{ij} is the (i, j)th element of a weights matrix **W**, specifying the degree of dependency between polygons i and j. As before, this could be a neighbour indicator, so that $w_{ij} = 1$ if polygons i and j are neighbours and $w_{ij} = 0$ otherwise, or the rows of the matrix could be standardised to sum to 1, in which case **Wz** is the vector of lagged means $\tilde{z}_i$ as defined in Section 7.3.

I

You may have noticed that the matrix **W** contains the same information as the listw objects discussed earlier. This is certainly true, but the latter stores information in quite a different, and usually more compact, way. The listw object notes, for each polygon, a list of its neighbours and their associated weights. For the polygon's non-neighbours, nothing needs to be stored. On the other hand, the matrix **W** has $n \times n$ elements. Each row of **W** contains information for all n polygons – although for many of them, $w_{ij} = 0$. Computation using the matrix form in R is generally less efficient – for example, although **Wz** is the vector of spatially lagged means, computing it directly as W %*% z would result in several numbers being multiplied by zero, and these resultant zeros being added up. If there are a lot of polygons, and typically they have, say, four or five neighbours, then the matrix format would also have a much higher storage overhead – and much of this would be filled with zeros.

Given the computational advantages of listw, why consider matrices? There are two important reasons. First, when considering the algebraic properties of quantities like Moran's *I*, matrix expressions are easy to manipulate. Second, although it is mostly the case that the listw form is more compact, it does store two items of data for every neighbour – the index of the neighbouring polygon, and the associated weight. Thus, if neighbours were defined in a very permissive way, so that **W** had few zero elements, the storage overheads might exceed that for standard matrices. A matrix with no zeros requires $n2$ items of information, but the listw form requires $2n2$. In this situation, calculations would also take longer. This is also true when the result of a computation has few zeros even if the supplied input does, as is the case in matrix inversion, for example.

It is also worth noting that, if **W** is standardised so that its rows sum to 1, then $\sum_i \sum_j w_{ij} = n$. In this case, equation (7.4) simplifies to

(7.5)

$$I = \frac{\sum_i \sum_j w_{ij} q_i q_j}{\sum_i q_i^2} \qquad (7.5)$$

where $q_i = z_i - \bar{z}$; that is, q_i is z_i recentred around the mean value of z. If the vector of q_i values is written as **q**, then equation (7.5) may be written in

vector–matrix form as

(7.6)

$$I = \frac{\mathbf{q}^T \mathbf{W} \mathbf{q}}{\mathbf{q}^T \mathbf{q}} \qquad (7.6)$$

It may be checked that if the q_i are plotted in a lagged mean plot – as in Figure 7.8 or 7.7 – and a regression line is fitted, then I is the slope of this line. This helps to interpret the coefficient. Larger values of I suggest that there is a stronger relationship between nearby z_i values. Furthermore, I may be negative in some circumstances – suggesting that there can be a degree of inverse correlation between nearby z_i values, giving a checkerboard pattern on the map. For example, a company may choose to site a chain of stores to be spread evenly across the state, so that occurrence of a store in one county may imply that there is no store in a neighbouring county.

7.4.1 Moran's *I* in R

The package spdep provides functions to evaluate Moran's *I* for a given dataset and **W** matrix. As noted in the earlier information box, it is sometimes more effective to store the **W** matrix in listw form – and this is done for the computation of Moran's *I* here. The function used to compute Moran's *I* is called moran.test – and can be used as below:

```
moran.test(penn.state.utm$smk,penn.state.lw)

Moran I test under randomisation

data: penn.state.utm$smk

weights: penn.state.lw

Moran I statistic standard deviate = 5.4175, p-value = 3.022e−08

alternative hypothesis: greater

sample estimates:

Moran I statistic      Expectation        Variance
```

```
0.404431265      −0.015151515      0.005998405
```

The above code supplies more than the actual Moran's *I* estimate itself – but for now note that the value is about 0.404 for the Penn State smoking uptake data.

This is fine, but one problem is deciding whether the above value of *I* is sufficiently high to suggest that an autocorrelated process model is a plausible alternative to an assumption that the smoking uptake rates are independent. There are two issues here:

1. Is this value of *I* a relatively large level on an absolute scale?
2. How likely is the observed *I* value, or a larger value, to be observed if the rates *were* independent?

The first of these is a benchmarking problem. Like correlation, Moran's *I* is a dimensionless property – so that, for example, with a given set of polygons and associated **W** matrix, area-normalised rates of rainfall would have the same Moran's *I* regardless of whether rainfall was measured in millimetres or inches. However, while correlation is always restricted to lie within the range [−1, 1] – making, say, a value of 0.8 reasonably easy to interpret – the range of Moran's *I* varies with the **W** matrix. The maximum and minimum values of *I* are shown (de Jong et al., 1984) to be the maximum and minimum values of the eigenvalues (Marcus and Minc, 1988) of $(\mathbf{W} + \mathbf{W}^T)/2$.[6] For the **W** matrix here, *I* can range between −0.579 and 1.020. Thus on an absolute scale the reported value suggests a reasonable degree of spatial autocorrelation.

[6] Don't worry too much if you don't know what an eigenvalue or eigenvector is – but if curious, see Marcus and Minc (1988).

An R function to find the maximum and minimum *I* values from a listw object is defined below. listw2mat converts a listw function to a matrix.

```
moran.range <- function(lw) {

   wmat <- listw2mat(lw)

return(range(eigen((wmat + t(wmat))/ 2)$values))
```

```
}

moran.range(penn.state.lw)
```

However, the null hypothesis statement of ‘no spatial autocorrelation’ is quite broad, and two more specific hypotheses will be considered here. The first is the assumption that each z_i is drawn from an independent Gaussian distribution, with mean μ and variance σ. Under this assumption, it can be shown that I is approximately normally distributed with mean $E(I) = -1/(n-1)$. The variance of this distribution is quite complex – readers interested in seeing the formula could consult, for example, Fotheringham et al. (2000). If the variance is denoted $V_{\text{norm}}(I)$ then the test statistic is

(7.7)

$$\frac{I - E(I)}{V_{\text{norm}}(I)} \tag{7.7}$$

This will be approximately normally distributed with mean 0 and variance 1, so that p-values may be obtained by comparison with the standard normal distribution.

The other form of the test is a more formal working of the randomisation idea set out in Section 7.3. In this case, no assumption is made about the distribution of the z_i, but it is assumed that any permutation of the z_i against the polygons is equally likely. Thus, the null hypothesis is still one of ‘no spatial pattern’, but it is conditional on the observed data. Under this hypothesis, it is also possible to compute the mean and variance of I. As before, the expected value of I is $E(I) = -1/(n-1)$, and the formula for the variance is different from that for the normality assumption, but also complex – again the formula is given in Fotheringham et al. (2000). If this variance is denoted by $V_{\text{rand}}(I)$ then the test statistic is

(7.8)

$$\frac{I - E(I)}{V_{\text{rand}}(I)} \tag{7.8}$$

In this case, the distribution of the text statistic in expression (7.8) is also

close to the normal distribution – and the quantity in this expression can also be compared to the normal distribution with mean 0 and variance 1, to obtain *p*-values. Both kinds of test are available in R via the moran.test function shown earlier. As noted earlier, as well as the Moran's *I* statistic itself, this function prints out some further information. In particular, looking again at this output, it can be seen that the expectation, variance and test statistic for the Moran's *I* statistic is output (the test statistic is labelled 'Moran *I* statistic standard deviate'), as well as the associated *p*-value. As a default, the output refers to the randomised hypotheses – that is, $V_{\text{rand}}(I)$ is used. Thus, looking at the output from moran.test(penn.state.utm$smk,penn.state.lw) again, it can be seen that there is strong evidence to reject the randomisation null hypothesis in favour of an alternative hypothesis of $I > 0$ for the smoking uptake rates.

The argument randomisation allows the normal distribution assumption, and hence $V_{\text{norm}}(I)$, to be used instead:

```
moran.test(penn.state.utm$smk,penn.state.lw,randomisation=FALSE)

        Moran I test under normality

data: penn.state.utm$smk

weights: penn.state.lw

Moran I statistic standard deviate = 5.4492, p-value = 2.53e−08

alternative hypothesis: greater

sample estimates:

Moran I statistic      Expectation         Variance

    0.404431265     −0.015151515     0.005928887
```

From this, it can be seen that there is also strong evidence to reject the null hypothesis of the *zi* being independently normally distributed, again in favour of an alternative that $I > 0$.

7.4.2 A Simulation-Based Approach

The previous tests approximate the test statistic by a normal distribution with mean 0 and variance 1. However, this distribution is asymptotic – that is, as *n* increases, the actual distribution of the test statistic gets closer to the normal distribution. The rate at which this happens is affected by the arrangement of the polygons – essentially, in some cases, the value of *n* for which a normal approximation is reasonable is lower than for others (Cliff and Ord, 1973, 1981).

For this reason, it may be reasonable to employ a simulation-based approach here, instead of using a theoretical, but approximate, approach. In this approach – which applies to the permutation-based hypothesis – a number of random permutations (say 10,000) of the data are drawn and assigned to polygons, using the sample function in R, as in Section 7.3. For each randomly drawn permutation, Moran's *I* is computed. This provides a simulated sample of draws of Moran's *I* from the randomisation null hypothesis. The true Moran's *I* is then computed from the data. If the null hypothesis is true, then the probability of drawing the observed data is the same as any other permutation of the *zi* among the polygons. Thus, if *m* is just the number of simulated Moran's *I* values exceeding the observed one, and *M* is the total number of simulations, then the probability of getting the observed Moran's *I* or a greater one is

(7.9)

$$p = \frac{m+1}{M+1} \qquad (7.9)$$

This methodology is due to Hope (1968). The function moran.mc in spdep allows this to be computed:

```
moran.mc(penn.state.utm$smk,penn.state.lw,10000 )

        Monte-Carlo simulation of Moran I

data: penn.state.utm$smk

weights: penn.state.lw

number of simulations + 1: 10001

statistic = 0.40443, observed rank = 10001, p-value = 9.999e−05
```

```
alternative hypothesis: greater
```

Note that the third argument provides the number of simulations. Once again, there is evidence to reject the null hypothesis that any permutation of the *zi* is equally likely in favour of the alternative that $I > 0$.

7.5 Spatial Autoregression

Moran's *I*, discussed in the previous section, can be thought of as a measure of spatial autocorrelation. However, up to this point no consideration has been given to a *model* of a spatially autocorrelated process. In this section, two spatial models will be considered – these are termed *spatial autoregressive* models. Essentially, they regress the z_i value for any given polygon on values of z_j for neighbouring polygons. The two models that will be considered are the *simultaneous autoregressive* (SAR) and *conditional autoregressive* (CAR) models. In each case, the models can also be thought of as multivariate distributions for **z**, with the variance–covariance matrix being dependent on the **W** matrix considered earlier.

The SAR model may be specified as

(7.10)

$$z_i = \mu + \sum_{j=1}^{n} b_{ij}\left(z_j - \mu\right) + \varepsilon_i \qquad (7.10)$$

where ε_i has a Gaussian distribution with mean 0 and variance σ_i^2 (often $\sigma^{i2} = \sigma^2$ for all *i*, so that the variance of ε_i is constant across zones), $E(z_i) = \mu$ and b_{iji} are constants, with $b_{ii} = 0$ and usually $b_{ij} = 0$ if polygon *i* is not adjacent to polygon *j*; thus, one possibility is that b_{ij} is λw_{ij}. Here, λ is a parameter specifying the degree of spatial dependence. When $\lambda = 0$ there is no dependence; when it is positive, positive autocorrelation exists; and when it is negative, negative correlation exists. μ is an overall level constant (as it is in a standard normal distribution model). If the rows of **W** are normalised to sum to 1, then the deviation from μ for z_i is dependent on the deviation from μ for the z_j values for its neighbours.

The CAR model is specified by

(7.11)

$$z_i \mid \{z_j : j \neq i\} \sim N\left(\mu + \sum_{j=1}^{n} c_{ij}\left(z_j - \mu\right), \tau_i^2\right) \qquad (7.11)$$

where, in addition to the above definitions, $N(\cdot,\cdot)$ denotes a normal distribution with the usual mean and variance parameters, τ_i^2 is the conditional variance of z_i given $\{z_j : j \neq i\}$ and c_{ij} are constants such that $c_{ii} = 0$ and, as with b_{ij} in the SAR model, typically $c_{ij} = 0$ if polygon i is not adjacent to polygon j. Again, a common model is to set $c_{ij} = \lambda w_{ij}$. μ and λ have similar interpretations to the SAR model. A detailed discussion in Cressie (1991) refers to the matrices $\mathbf{B} = [b_{ij}]$ and $\mathbf{C} = [c_{ij}]$ as 'spatial dependence' matrices. Note that this model can be expressed as a multivariate normal distribution in **z** as

(7.12)

$$\mathbf{z} \sim N(\mu\mathbf{1}, (\mathbf{I} - \mathbf{C})^{-1}\mathbf{T}) \qquad (7.12)$$

where **1** is a column vector of 1s (of size n) and **T** is a diagonal matrix composed of the τ_i (see, for example, Besag, 1974). Note that this suggests that the matrix $(\mathbf{I} - \mathbf{C})^{-1}\mathbf{T}$ must be symmetrical (as well as positive definite). If the **W** matrix is row-normalised, and the $c_{ij} = \lambda w_{ij}$ model is used, then this implies that τ_i must be proportional to $\left[\sum_j c_{ij}\right]^{-1}$.

7.6 Calibrating Spatial Regression Models in R

The SAR model may be calibrated using the spautolm function from spdep. This uses the notation also used in the lm function – and related functions – to specify models. In the next section, the SAR and CAR models will be expanded to consider further predictor variables, rather than just neighbouring values of z_i. However, for now the basic model may be specified by using the notation for a linear model with just a constant term for the mean of the predicted variable – this is μ in equation (7.11) or (7.10).

This is simply Var.Name ~ 1, with Var.Name replaced with the actual variable name of interest (e.g. penn.state.utm$smk in the smoking rate examples used in previous sections).[7] A further parameter, family, specifies whether a SAR or a CAR model is fitted. The function returns a regression model object – among other things, this allows the values of coefficients, fitted values and so on to be extracted. An example of use is as follows:

[7] Or use smk and supply penn.state.utm as the data parameter.

```
sar.res <- spautolm (smk~ 1, listw=penn.state.lw, data=penn.state.utm)

sar.res

Call:

spautolm(formula = smk ~ 1, data = penn.state.utm, listw = penn.state.lw)

Coefficients:

(Intercept)       lambda

23.7689073     0.6179367

Log likelihood: −142.8993
```

From this it can be seen that $\lambda = 0.618$ and $\mu = 23.769$, to 3 decimal places. While the estimate for μ is easily interpretable, deciding where the reported level of λ is of importance is harder. One possibility is to find the standard error of λ – this is reported as the lambda.se component of the spatial autoregression object:

```
sar.res$lambda.se

[1] 0.1130417
```

An approximate 5% confidence interval can be found in the standard way – by finding a band given by the estimate of λ plus or minus twice the standard error:

```
sar.res$lambda + c(−2,2)*sar.res$lambda.se

[1] 0.3918532 0.8440201
```

As before, this suggests that a null hypothesis of λ = 0 is highly unlikely.

I

It is also possible to calibrate CAR models in the same way, and similarly obtain an approximate confidence interval for λ. This is achieved – in our example – via the family parameter in spautolm:

```
car.res <- spautolm(smk~1,listw=penn.state.lw, family='CAR',data=penn.state.utm)

car.res
```

However, at the time of writing, the help document for this function points out:

the function does not (yet) prevent asymmetric spatial weights being used with 'CAR' family models. It appears that both numerical issues (convergence in particular) and uncertainties about the exact spatial weights matrix used make it difficult to reproduce ... results.

Experimentation with the above code suggests similar convergence issues occur here, hence attention will be focused on SAR models for the R examples.

7.6.1 Models with Predictors: A Bivariate Example

Both the CAR and SAR models can be modified to include predictor variables as well as incorporate autocorrelationeffects. This is achieved by replacing a constant μ by an observation specific μ_i for each z_i, where μ_i is some function of a predictor variable (say, P_i). If the relationship between μ_i and P_i is linear, we can write, for the SAR case,

(7.13)

$$z_i = a_0 + a_1 P_i + \sum_{j=1}^{n} b_{ij}\left(z_j - a_0 - a_1 P_i\right) + \varepsilon_i \qquad (7.13)$$

where a_0 and a_1 are effectively intercept and slope terms in a regression model. The key difference between this kind of model and a standard ordinary least squares (OLS) model is that for the OLS case the z_i values are assumed to be independent, whereas, here, nearby z_j values influence z_i as well as the predictor variable.

Calibrating models such as that in equation (7.13) in R is straightforward, and involves including predictor variables in the model argument for spautolm. In the following example, a new data item, the per-county lung cancer rate for Penn State in 2002, is computed and used as the *zi* variable. This time the role of the smoking uptake variable is changed to that of the predictor variable, *Pi*. This is achieved via a two-stage process:

1. Compute the per-county lung cancer rates.
2. Compute the regression model.

For stage 1, the plyr package is used to manipulate the data. Recall that pennLC is a list, and one of the elements (called data) is a data frame giving the counts of population, and lung cancer incidence, for each county in Penn State subdivided by race (‘white’ or ‘other’), gender (‘male’ or ‘female’), and age (‘under 40’, ‘40 to 59’, ‘60 to 69’, and ‘over 70’). The format of the data frame uses a county column and three *substrata* columns – together specifying a combination of county, age, gender and ethnicity.[8] Two further columns then specify the count of cases for that county–substrata combination, and also the overall population for the same county substrata combination:

[8] We would prefer to use the term ‘ethnicity’ – unfortunately the supplied data use ‘race’.

```
head(pennLC$data)

      county cases population race gender     age
1    adams     0       1492    o     f Under.40
2    adams     0       365     o     f    40.59
3    adams     1        68     o     f    60.69
4    adams     0        73     o     f     70+
5    adams     0     23351     w     f Under.40
6    adams     5     12136     w     f    40.59
```

For example, it may be seen that Adams County has 0 incidents of lung

cancer for non-white[9] females under 40 out of a total population of 1492 female non-white people under 40 in Adams County. Using the plyr package (Wickham, 2011), it is possible to create a data frame showing the total number of cases over all combinations of age, ethnicity and gender for each county:

[9] Here ‘o’ denotes ‘other’ – that is, ‘non-white’.

```
require(plyr)

totcases <- ddply(pennLC$data,c("county"),numcolwise(sum))
```

I

plyr is a very powerful package very much worth reading more about (see Wickham, 2011). It applies a *split–apply–combine* approach to data manipulation. A number of functions are supplied to apply this approach for various formats of variable. Here, ddply is used. A dataset is supplied (pennLC$data) and one of the factor (or character) column names is given (county) in this example. The data frame is *split* into a list of smaller data frames, one for each value of the county variable. Next, a function is *applied* to each of these data frames, giving a list of transformed data frames – quite often the new data frame is a smaller one, often having only one row consisting of summary statistics (or sums or counts) for some selected rows of the data frames arising from the split. Finally, the list of transformed data frames is *combined* by row-wise stacking to create a new data frame. Hence *split–apply–combine*.

In the code above, the function applied to a subset data frame for each county is created via numcolwise(sum). This transforms the basic sum function, which applies to vectors, to a new function which sums all numeric columns in a data frame, yielding a one-row data frame with sums of numeric columns. Here these columns are the number of incidents of lung cancer, and the population. After applying this function to each subset of the data, the countywise totals for lung cancer incidents and populations are recombined to give a data frame with county name, county total lung cancer cases and county total population – in the data frame totcases:

```
head(totcases)
```

```
        county   cases  population
1        adams      55       91292
2    allegheny    1275     1281666
3    armstrong      49       72392
4       beaver     172      181412
5      bedford      37       49984
6        berks     308      373638
```

I

The expression numcolwise(sum) may look a little strange. numcol-wise is a function, but, unusually, it takes another function as its input, and returns yet another function as output. The input function is assumed to apply to standard R numerical vectors – it is modified by numcolwise to produce a new function that applies the input function to data frames on a row-by-row basis, and returns a single-row data frame of the results. Note that since in this example sum is the input function, it is only valid for numerical data columns. The numcolwise column allows for this, and the modified function only returns entries in the output data frame for numerical columns. Although it would not make much sense in this example, functions like mean and median could also be used as inputs to numcolwise – or indeed user-defined numeric functions.

In the example the output function is then fed into ddply to provide the *apply* stage function in the split–apply–combine procedure.

Having created a data frame of county-based lung cancer incident and population counts, the cancer rates per 10,000 population are computed. These are added as a new column to the totcases data frame:

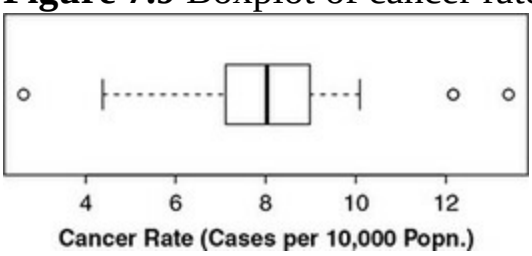
```
totcases <- transform(totcases, rate= 10000*cases/population)
```

Thus, totcases now has three columns, and is ready to provide input to the regression model – below this variable is inspected (using head) and a boxplot drawn in Figure 7.9:

```
head(totcases)

county cases population    rate
```

```
Cancer Rate (Cases per 10,000 Popn.)
  1       adams    55    91292   6.024624
  2  allegheny  1275  1281666   9.947990
  3  armstrong    49    72392   6.768704
  4     beaver   172   181412   9.481181
  5    bedford    37    49984   7.402369
  6      berks   308   373638   8.243273
# Check the distribution of rates
boxplot(totcases$rate, horizontal=TRUE,
xlab='Cancer Rate (Cases per 10,000 Popn.)')
```

Figure 7.9 Boxplot of cancer rates (Penn State, 2002)



It is now possible to calibrate the spatial regression model. As stated earlier, the z_i variable here is related to the cancer rate, and the predictor is smoking uptake. Note that in this case an additional weighting variable is added, based on the population variable, and also that z_i is actually the square root of the cancer rate. This allows for the fact that the random variable here is actually the count of cancer cases – and that this is possibly a Poisson distributed variable – since the square root transform can stabilise variance of Poisson count data (Bartlett, 1936). Since the square root rate is essentially

(7.14)

$$\sqrt{\frac{\text{No. of cases}}{\text{Population}}} \qquad (7.14)$$

and population is assumed to be a fixed quantity, the numerator above will have an approximately fixed variance and be reasonably approximated by a

normal distribution. Dividing this by the square root of population then makes the variance inversely proportional to the population. Hence, weighting by population is also appropriate here. Taking these facts into account, the SAR model may be calibrated and assessed:

```
sar.mod <- spautolm (rate~sqrt(penn.state.utm$smk), listw=penn.state.lw,
weight=population, data=totcases)

summary(sar.mod)

Call:

spautolm(formula = rate ~ sqrt(penn.state.utm$smk), data = totcases,

        listw = penn.state.lw, weights = population)

Residuals:

Min              1Q    Median        3Q     Max

−5.45183 −1.10235 −0.31549 0.59901 5.00115

Coefficients:

Estimate Std.      Error    z value

(Intercept)                                −0.35263    2.26795    −0.1555

sqrt(penn.state.utm$smk)     1.80976    0.46064    3.9288

Pr(>|z|)

(Intercept)                 0.8764

sqrt(penn.state.utm$smk)    8.537e−05

Lambda: 0.38063 LR test value: 6.3123 p-value: 0.01199

Numerical Hessian standard error of lambda: 0.13984

Log likelihood: −123.3056

ML residual variance (sigma squared): 209030, (sigma: 457.19)

Number of observations: 67

Number of parameters estimated: 4

AIC: 254.61
```

The ‘coefficients’ section in the output may be interpreted in a similar way to a standard regression model. From this it can be seen that the rate of smoking does influence the rate of occurrence of lung cancer – or at least that there is evidence to reject a null hypothesis that it does not affect cancer rates, with $p = 8.54 \times 10^{-5}$. The ‘lambda’ section provides a p-value for the null hypothesis that $\lambda = 0$: that is, that there is a degree of spatial autocorrelation in the cancer rates. Here, $p = 0.012$, so that at the 5% level there is evidence to reject the null hypothesis, although the strength of evidence just falls short of the 1% level.

Thus, the analysis here suggests that smoking is linked to lung cancer, but that lung cancer rates are spatially autocorrelated. This is possibly because other factors that influence lung cancer (possibly age, or risk associated with occupation) are geographically clustered. Since these factors are not included in the model, information about their spatial arrangement might be inferred via nearby occurrence of lung cancer.

7.6.2 Further Issues

The above analysis gave a reasonable insight into the occurrence of lung cancer in Pennsylvania as a spatial process. However, a number of approximations were made. A more exact model could have been achieved if a direct Poisson model had been used, rather than using an approximation via square roots. Indeed, if an independent z_i model were required, where the z_i were case counts, then a straightforward Poisson regression via glm could have achieved this. However, a Poisson model with an autocorrelated error term is less straightforward. One approach might be to use a Bayesian Markov chain Monte Carlo (MCMC) approach for this kind of model (for an example, see Wolpert and Ickstadt, 1998). In R, this type of approach can be achieved using the RJags[10] or RStan package.[11]

[10] http://cran.r-project.org/web/packages/rjags/index.html

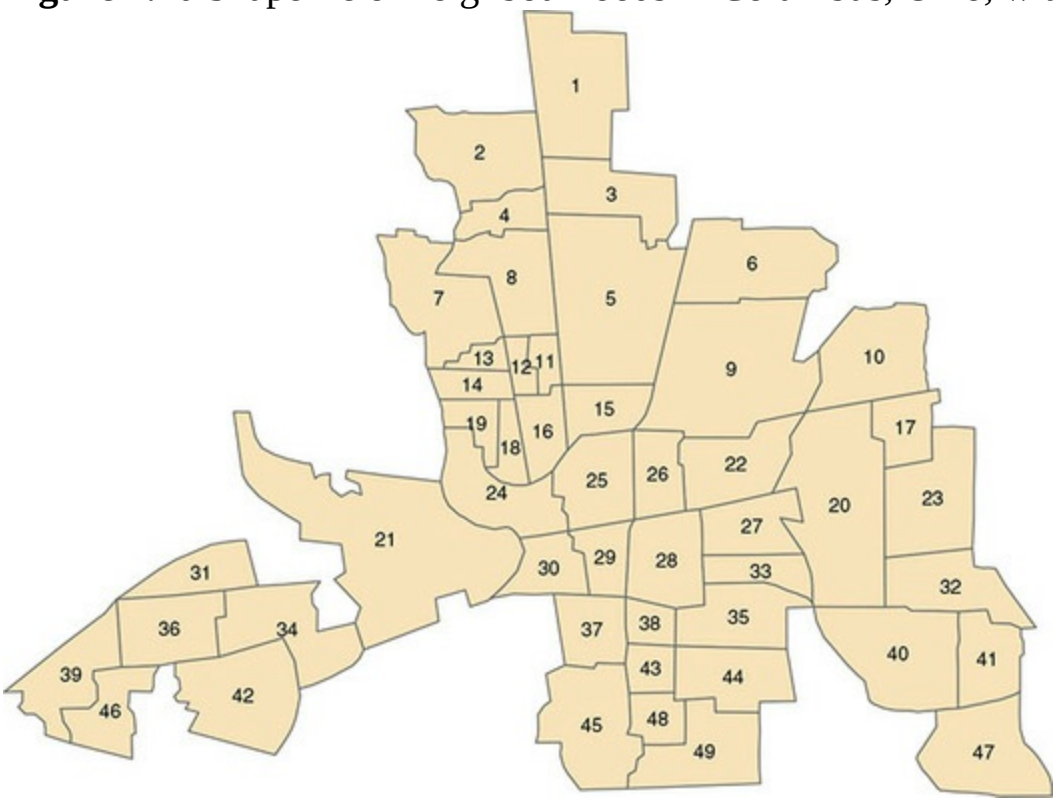
[11] http://mc-stan.org/users/interfaces/rstan

7.6.3 Troubleshooting Spatial Regression

In this section, a set of the issues with spatial models based on **W** matrices will be explored. These issues are identified in Wall (2004). The issues identify certain strange characteristics in some spatial models – and possibly interactive exploration via R is an important way of identifying whether these issues affect a particular study. For this exercise you will look at the Columbus crime data supplied with the spdep package.[12] Typing in the following will load the shapefile of neighbourhoods in Columbus, Ohio, and create a map (Figure 7.10). Note that the mapping command might issue a warning as columbus does not have a defined map projection. This is not a problem if the shapefile is only considered in isolation – if it is to be combined with other sources of geographical information, this could be more problematic.

[12] http://www.rri.wvu.edu/WebBook/LeSage/spatial/anselin.html

Figure 7.10 Shapefile of neighbourhoods in Columbus, Ohio, with labels




```
columbus <- readShapePoly(system.file("etc/shapes/columbus.shp",package= "spdep")
[1])

# Create a plot of columbus and add labels for each of the zones
```

```
tm_shape(columbus) + tm_polygons(col='wheat') +
tm_text(text='POLYID', size= 0.7)
```

This dataset has been used in a number of studies. For each neighbourhood, a number of attributes are provided, including 'average house price', 'burglary rate' and 'average income'. However, here these will not be considered, as the focus will be on the correlation structure implied by the **W** matrix. Here, a queen's case matrix is extracted from the data. Adjacency plays an important role in SAR models. Recall that there are also several options in terms of specifying the definition of polygon adjacency – in particular, the rook's case and queen's case. Both of these can be computed from columbus, which is a SpatialPolygonsDataFrame object.

```
# Extract a 'queen's case' adjacency object and print it out
col.queen.nb <- poly2nb(columbus, queen=TRUE)
col.queen.nb
Neighbour list object:
Number of regions: 49
Number of nonzero links: 236
Percentage nonzero weights: 9.829238
Average number of links: 4.816327
# Extract a 'rook's case' adjacency object and print it out
col.rook.nb <- poly2nb(columbus, queen=FALSE)
col.rook.nb
Neighbour list object:
Number of regions: 49
Number of nonzero links: 200
Percentage nonzero weights: 8.329863
Average number of links: 4.081633
```

The two variables col.queen.nb and col.rook.nb respectively contain the

adjacency information for the queen's and rook's case adjacency. It can be seen that the queen's case has 36 more adjacencies than the rook's case.

Wall (2004) and others demonstrate that for the SAR model with a constant σ^2 term

(7.15)

$$\mathrm{Var}(\mathbf{z}) = (\mathbf{I} - \lambda\mathbf{W})^{-1}\left[(\mathbf{I} - \lambda\mathbf{W})^{-1}\right]^T \sigma^2 \qquad (7.15)$$

provided $(\mathbf{I} - \lambda\mathbf{W})$ is invertible. Thus, as stated before, the SAR model is essentially a regression model with non-independent error terms, unless $\lambda = 0$, in which case it is equivalent to a model with independent observations. The variance–covariance matrix is therefore a function of the variables **W**, σ^2 and λ. Without loss of generality, we can assume that T is scaled so that $\sigma^2 = 1$. Then, for any given definition of adjacency for the study area, it is possible to investigate the correlation structure for various values of λ. In R, the following code defines a function to compute a variance–covariance matrix from λ and **W**. Here, the adjacency object is used (rather than supplying a **W** matrix), but this contains the same information.

```
covmat <- function(lambda,adj) {
   solve(tcrossprod(diag(length(adj)) − lambda∗ listw2mat(nb2listw(adj))))
}
```

The tcrossprod function takes a matrix **X** and returns $\mathbf{XX}^T$. The function solve finds the inverse of a matrix. This can also be used as the basis for finding the *correlation* matrix (rather than the variance–covariance matrix).

```
cormat <- function(lambda,adj) {
   cov2cor(covmat(lambda,adj))
}
```

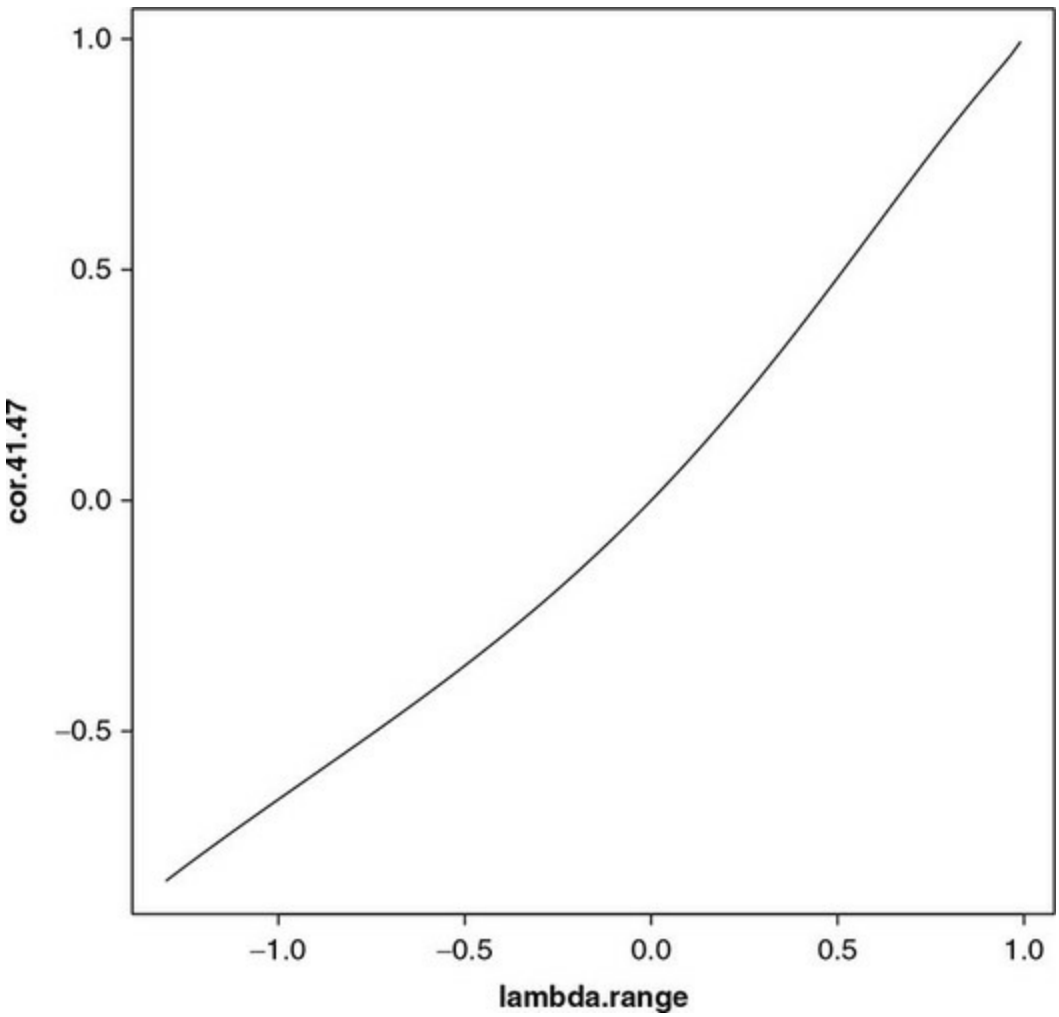
We can now examine the relationship between, say, the correlation between zones 41 and 47, and λ – the plot created is shown in Figure 7.11.

```
# Create a range of valid lambda values

lambda.range <- seq(−1.3,0.99, l= 101)

# Create an array to store the corresponding correlations

cor. 41.47 <- lambda.range∗ 0

# ... store them

for (i in 1: 101) cor. 41.47[i] <- cormat(lambda.range[i],col.rook.nb)[41,47]

# ... plot the relationship

plot(lambda.range,cor. 41.47, type='l')
```

This seems reasonable – larger values of λ lead to higher correlation between the zones, λ = 0 implies no correlation, and the sign of λ implies the sign of the correlation. However, now consider the same curve, but between zones 40 and 41 (see Figure 7.12).

```
# First, add the line from the previous figure for reference

plot(lambda.range,cor. 41.47, type='l', xlab=expression(lambda), ylab= 'Correlation',
lty= 2)

# Now compute the correlation between zones 40 and 41.

cor. 40.41 <- lambda.range∗ 0

for (i in 1: 101) cor. 40.41[i] <- cormat(lambda.range[i],col.rook.nb)[40,41]

# ... and add these to the plot

lines(lambda.range,cor. 40.41)
```

Figure 7.11 Relationship between λ and the correlation between zones 41 and 47

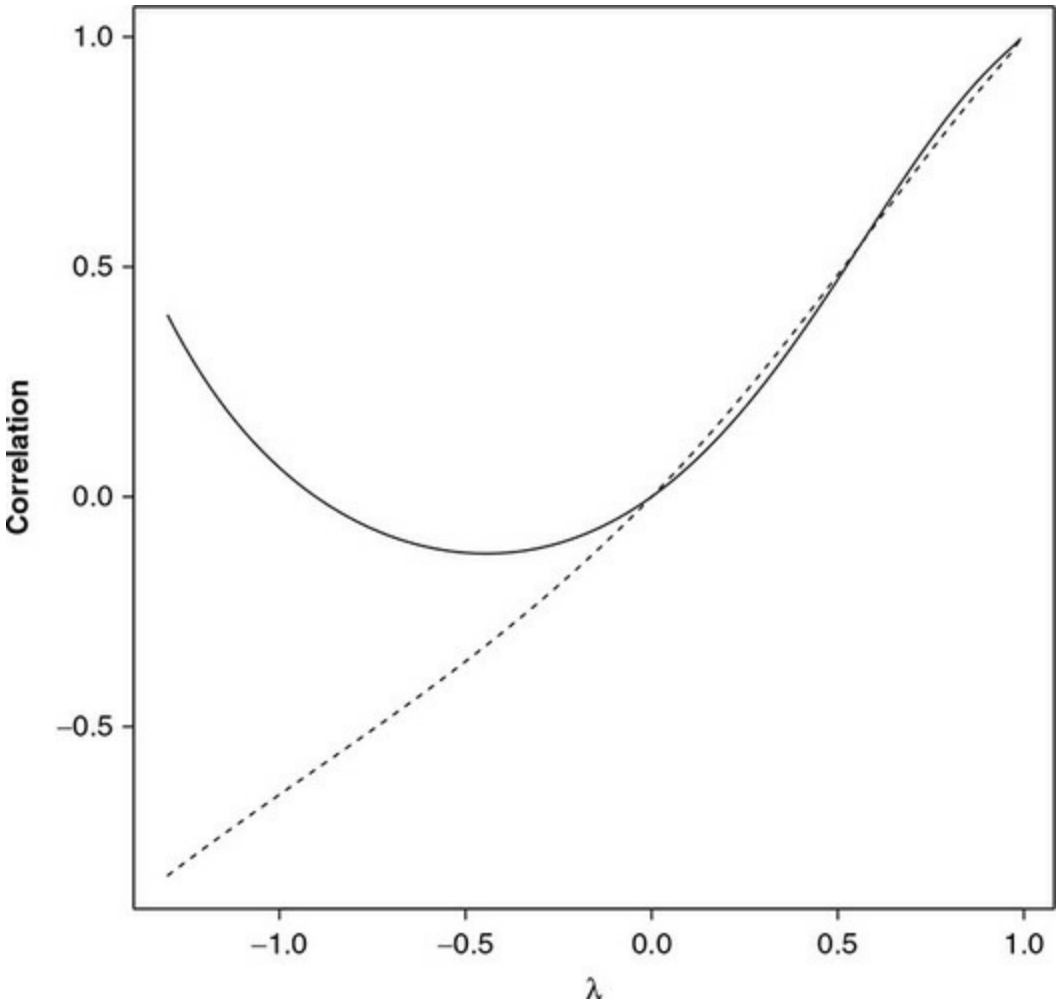


Here, something strange is happening. When λ drops below around −0.5 the correlation between zones 40 and 41 begins to increase, and at around −0.7 it becomes positive again. This is somewhat counter-intuitive, particularly as λ is often referred to as an indicator of spatial association. For example, Ord (1975) states that w_{ij} 'represents the degree of possible interaction of location *j* on location *i*'. Although initially for positive λ the correlation between zones 40 and 41 is less than that for zones 41 and 47, when λ exceeds around 0.5 the situation is reversed (although this is a less pronounced effect than the sign change noted earlier). A useful diagnostic plot is a parametric curve of the two correlations, with parameter λ (see Figure 7.13):

```
# First, plot the empty canvas (type='n')

plot(c(−1,1), c(−1,1), type= 'n', xlim= c(−1, 1), ylim= c(−1,1), xlab= 'Corr1', ylab=
'Corr2')
```

```
# Then the quadrants
rect(−1.2, −1.2,1.2, 1.2, col= 'pink', border=NA)
rect(−1.2, −1.2,0, 0, col= 'lightyellow', border=NA)
rect(0, 0, 1.2,1.2, col= 'lightyellow', border=NA)
# Then the x=y reference line
abline(a=0, b=1, lty=3)
# Then the curve
lines(cor. 40.41,cor. 41.47)
```

Figure 7.12 Relationship between λ and the correlation between zones 41 and 47

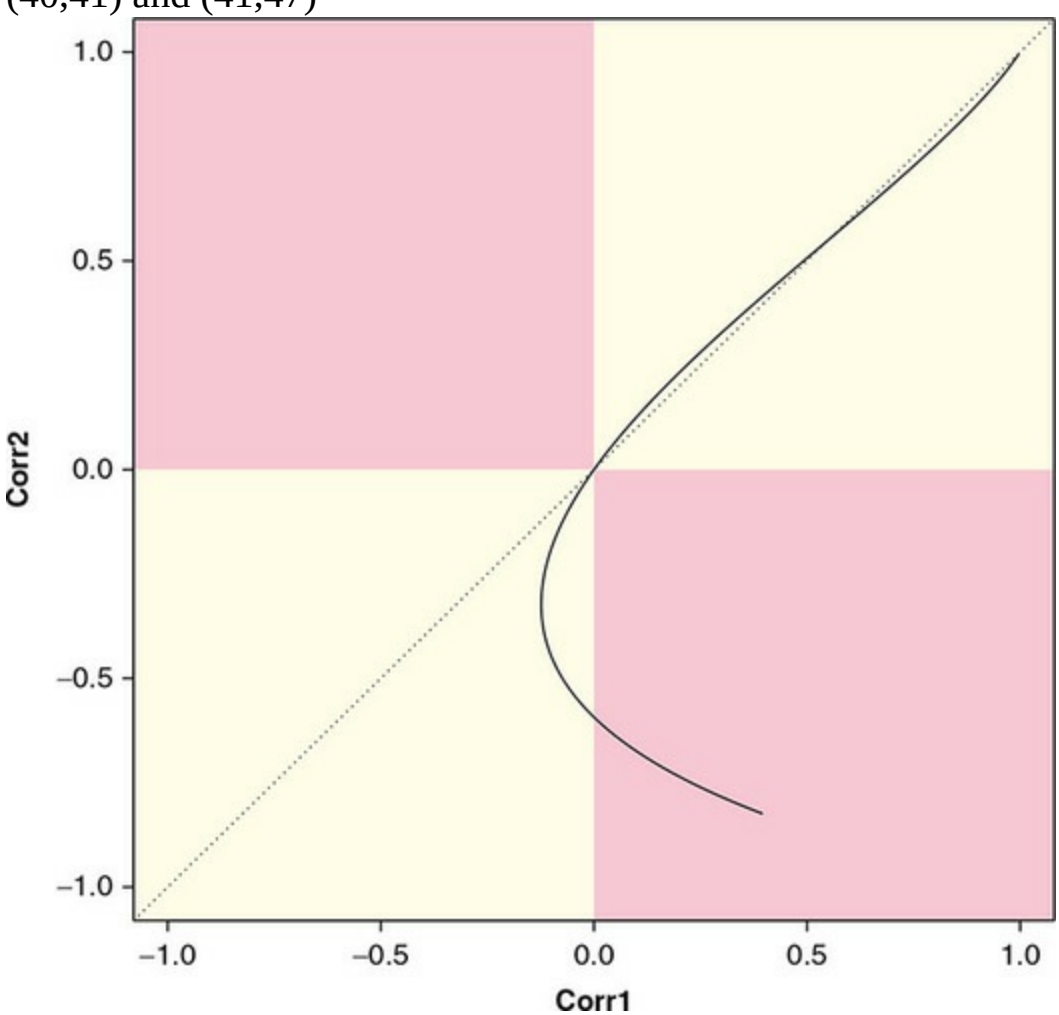


We term this a *Battenberg*[13] plot. Tracing along this line from top right

shows the relationship between the two correlations as λ decreases from its maximum value. The dotted line is the $x = y$ reference point – whenever the curve crosses this, the values of the two correlations change order. Perhaps the key feature is that the curve 'doubles back' on itself – so that for some ranges of λ one of the correlations increases while the other decreases. The quadrants are also important – if a curve enters one of the pink quadrants this suggests that one of the correlations is positive, while the other is negative. Again, this is perhaps counter-intuitive, given the interpretation of λ as a measure of spatial association. Note in this case that after the 'doubling back' of the curve it does enter the pink quadrant.

[13] https://www.thespruceeats.com/best-british-battenberg-cake-recipe-434905

Figure 7.13 Parametric plot of correlations between two polygon pairs (40,41) and (41,47)

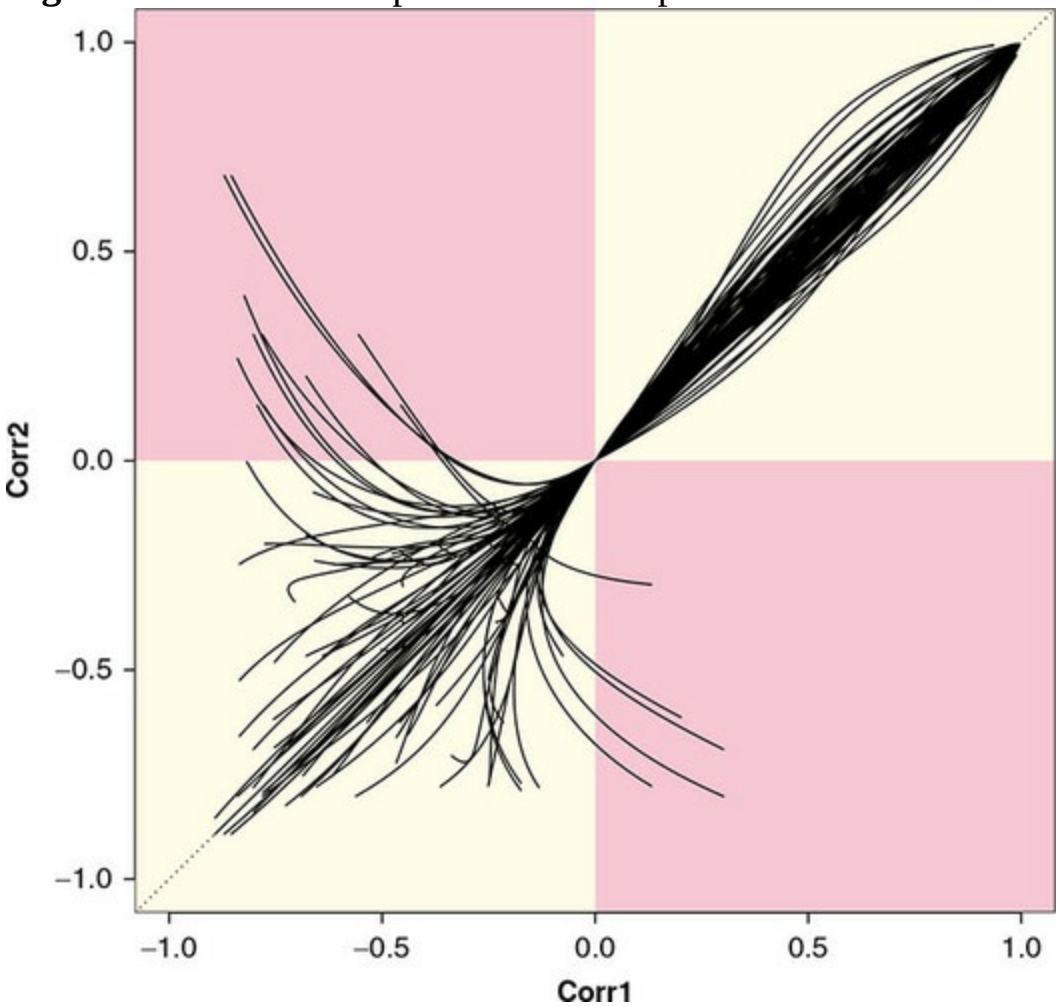


A selection of 100 random pairs of correlations (chosen so that each pair has one zone in common) can be drawn (see Figure 7.14). This seems to suggest that 'doubling back' and curves going inside the pink quadrants are not uncommon problems. In addition, for positive λ values, there is a fair deal of variation in the values of correlation for given λ values. In addition, the variability is not consistent, so that the order of values of correlation changes frequently.

```
# First, plot the empty canvas (type='n)

plot(c(−1,1), c(−1, 1), type= 'n', xlim= c(−1,1), ylim= c(−1, 1), xlab= 'Corr1', ylab=
'Corr2')

# Then the quadrants

rect(−1.2, −1.2,1.2, 1.2, col= 'pink', border=NA)

rect(−1.2, −1.2,0, 0, col= 'lightyellow', border=NA)

rect(0, 0, 1.2,1.2, col= 'lightyellow', border=NA)

# Then the x=y reference line

abline(a= 0, b= 1, lty= 3)

# Then the curves

# First, set a seed for reproducibility

set.seed(310712)

for (i in 1: 100) {

r1 <- sample(1:length(col.rook.nb), 1)

r2 <- sample(col.rook.nb[[r1]], 2)

cor.ij1 <- lambda.range∗ 0

cor.ij2 <- lambda.range∗ 0

for (k in 1: 101)

cor.ij1[k] <- cormat(lambda.range[k],col.rook.nb)[r1,r2[1]]

for (k in 1: 101)

cor.ij2[k] <- cormat(lambda.range[k],col.rook.nb)[r1,r2[2]]
```

```
lines(cor.ij1,cor.ij2)

}
```

Figure 7.14 Parametric plots of 100 sampled correlations



This shows a pattern very similar to those seen in Wall (2004). Essentially, for negative λ values, some correlations become positive while others remain negative. The ordering can also change as λ changes, as noted earlier, so that some adjacent zones are more correlated than others for certain λ values, but this situation can alter. Finally, some adjacent zone pairs experience a sign change for negative values of λ, while others do not. The aim of this section has been in part to highlight the issues identified by Wall, but also to suggest some visual techniques in R that could be used to explore these – and identify situations in which the counter-intuitive behaviour seen here may be occurring. As a general rule, the authors have found this not to happen a great

deal when working with zones based on a regular grid, but that the problems seen here occur quite often for irregular lattices. This provides empirical back-up to the more theoretical arguments of Besag and Kooperberg (1995) for CAR models.

7.7 Answer to Self-Test Question

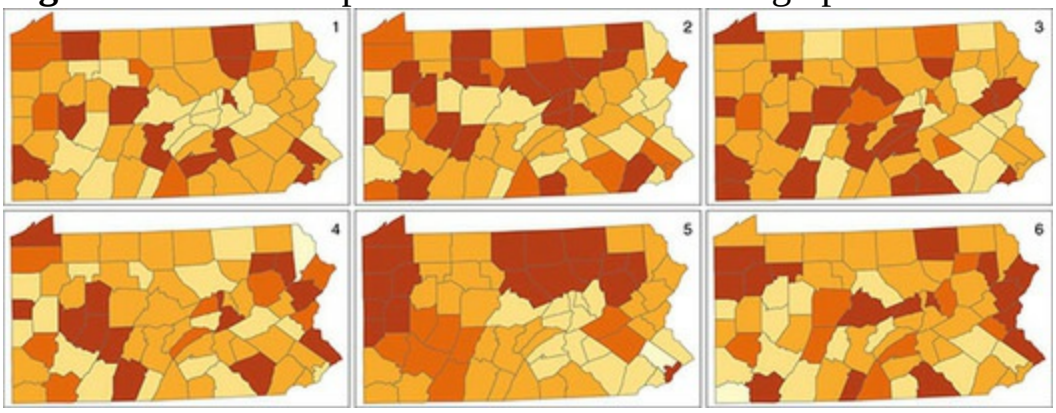
Q1: The following code will apply the modified approach asked for in the question:

```
# Set up a set of five 'fake' smoking update rates as well as the real one

# Create new columns in penn.state.utm for randomised data

# Here the seed 4676 is used. Use a different one to get an unknown outcome.

set.seed(4676)

penn.state.utm$smk_rand1 <- sample(penn.state.utm$smk, replace=TRUE)

penn.state.utm$smk_rand2 <- sample(penn.state.utm$smk, replace=TRUE)

penn.state.utm$smk_rand3 <- sample(penn.state.utm$smk, replace=TRUE)

penn.state.utm$smk_rand4 <- sample(penn.state.utm$smk, replace=TRUE)

penn.state.utm$smk_rand5 <- sample(penn.state.utm$smk, replace=TRUE)

# Scramble the variables used in terms of plotting order

vars <- sample(c('smk','smk_rand1','smk_rand2','smk_rand3','smk_rand4','smk_rand5'))

# Which one will be the real data?

# Don't look at this variable before you see the maps!

real.data.i <- which(vars == 'smk')

# Draw the scrambled map grid

tm_shape(penn.state.utm) +

   tm_polygons(col=vars, legend.show=FALSE, breaks=c(18,20,22,24,26,28)) +
tm_layout(title= 1: 6, title.position= c("right","top"))
```

The only difference between this and the previous code block is the inclusion

of the optional parameter replace=TRUE in the sample function – which tells the function to return *n* random samples from the list of smoking take-up rates *with* replacement. Also the breaks are now explicitly specified – these would always be the same for a permutation since the default break-choosing algorithm depends on the range of the values – but for a sample with replacement these may change as the largest and smallest values in the dataset may not be chosen. This is essentially the technique to simulate the drawing of random samples used by Efron (1979) to carry out the *bootstrap* approach to non-parametric estimations of standard error. Thus, here it is referred to as 'bootstrap randomisation'.

Figure 7.15 Bootstrap randomisation of smoking uptake rates



The multiple map drawing algorithm in tmap decides automatically on the layout, depending on the shape of the window it is drawing into (Figure 7.15). Thus, you may get the layout as 'long and thin' (i.e. 2 columns and 3 rows) rather than 'short and wide' (3 rows and 2 columns).

References

Anselin, L. (1995) Local indicators of spatial association – LISA. Geographical Analysis, 27: 93–115.

Anselin, L. (1996) The Moran scatterplot as an ESDA tool to assess local stability in spatial association. In M.M. Fischer, H.J. Scholten and D.J. Unwin (eds), Spatial Analytical Perspectives on GIS, pp. 111–125. London: Taylor & Francis.

Bartlett, M.S. (1936) The square root transformation in analysis of variance. Supplement to the Journal of the Royal Statistical Society, 3(1): 68–78.

Belsley, D.A., Kuh, E. and Welsch, R.E. (1980) Regression Diagnostics. New York: John Wiley & Sons.

Besag, J. (1974) Spatial interaction and the statistical analysis of lattice systems (with discussion). Journal of the Royal Statistical Society, Series B, 36: 192–236.

Besag, J. and Kooperberg, C. (1995) On conditional and intrinsic autoregressions. Biometrika, 82(4): 733–746.

Cliff, A.D. and Ord, J. K. (1973) Spatial Autocorrelation. London: Pion.

Cliff, A.D. and Ord, J.K. (1981) Spatial Processes: Methods and Applications. London: Pion.

Cook, R.D. and Weisberg, S. (1982) Residuals and Influence in Regression. London: Chapman & Hall.

Cressie, N. (1991) Statistics for Spatial Data. New York: John Wiley & Sons.

de Jong, P., Sprenger, C. and van Veen, F. (1984) On extreme values of Moran's *I* and Geary's *c*. Geographical Analysis, 16(1): 17–24.

Efron, B. (1979) Bootstrap methods: Another look at the jackknife. Annals of Statistics, 7: 1–26.

Fotheringham, A., Brunsdon, C. and Charlton, M. (2000) Quantitative

Geography: Perspectives on Spatial Analysis. London: Sage.

Hope, A.C.A. (1968) A simplified Monte Carlo significance test procedure. Journal of the Royal Statistical Society, Series B, 30(3): 582–598.

Marcus, H. and Minc, H. (1988) Introduction to Linear Algebra. New York: Dover.

Moran, P. (1950) Notes on continuous stochastic phenomena. Biometrika, 37: 17–23.

Ord, J.K. (1975) Estimation methods for models of spatial interaction. Journal of the American Statistical Association, 70(349): 120–126.

Wall, M.M. (2004) A close look at the spatial structure implied by the CAR and SAR models. Journal of Statistical Planning and Inference, 121(2): 311–324.

Wickham, H. (2011) The split-apply-combine strategy for data analysis. Journal of Statistical Software, 40(1): 1–29.

Wickham, H., Cook, D., Hofmann, H. and Buja, A. (2010) Graphical inference for infovis. IEEE Transactions on Visualization and Computer Graphics, 16(6): 973–979.

Wolpert, R. and Ickstadt, K. (1998) Poisson/gamma random field models for spatial statistics. Biometrika, 85(2): 251–267.

8 Localised Spatial Analysis

8.1 Introduction

In the previous chapters, a number of models of spatial processes have been used to analyse data. One characteristic of many of the models used was an assumption of homogeneity in the way that spatial data interacted. For example, *K*-functions and related ideas model the interdependence *between* points, in terms of the distance between them – *K*-functions model the number of points one might encounter at a radius *r* from a particular point. However, a general assumption is that these relations depend only on *relative* distance. Thus, the expected number of points within a circle of radius *r* centred around a point at location **x** will depend only on the value of *r* and not on **x**. Similarly, in the SAR models considered in the previous chapter, the coefficient λ specified the degree to which an attribute at polygon *i* depended on the values of nearby polygons. However, λ takes the same value for all polygons – suggesting again that the degree of spatial interdependency is the same regardless of location.

This has an effect on the kind of hypothesis testing that may take place. For example, in the previous chapter, the hypothesis that $\lambda = 0$ was tested – and in the examples given it was rejected at the 5% level. This tells us that there is spatial dependency in the process under investigation (in the example, rates of smoking) – but of itself it supplies no inference as to where high or low levels occur geographically, or whether the dependency occurs in some regions but not in others.[1] In this chapter, a number of approaches that attempt to highlight geographical variation in spatial processes will be introduced. Two key ideas here are *index decomposition*, in which indices such as Moran's *I* are decomposed according to the contribution of data from each locality to identify local effects, and *moving window* approaches, where data will be analysed in a moving circular window, to identify variation in relationships within the data over space.

[1] Indeed, adopting this model requires a prior assumption of homogeneity in spatial dependency.

8.2 Setting Up the Data Used in this Chapter

The main dataset used in this chapter will be the North Carolina sudden infant death syndrome (SIDS) data, appearing in Getis and Ord (1992). The data are supplied with the spdep package.[2] The package supplies this as a shapefile, and read_shape from tmaptools is used to read it in. Initially, the shapefile is supplied in geographical (i.e. latitude and longitude) coordinates – although this information is not supplied with the shapefile and is specified (as EPSG:4326) in the current.projection argument. However, some of the examples in this chapter will require distances between county centroids, and so a projected coordinate system should be used. Here the geodetic parameters with ID 2264 from the European Petroleum Survey Group are used – using miles as the units of distance. The code to carry out this operation follows. The map (in projected coordinates) is shown in Figure 8.1.

[2] Here we do not need to load the package, just determine the location of the data file via system.file.

```
# Load tmap, tmaptools packages

require(tmap)

require(tmaptools)

# read in the shapefile for North Carolina SIDS (it's in epsg:4326)

nc.sids <- read_shape(system.file("shapes/sids.shp",     package= "spData")[1],
current.projection= 4326)

# Transform to EPSG 2264 - and units in miles. We need the full proj4 string here to specify
units

nc.sids.p <- set_projection(nc.sids, "+init=epsg:2264 +units=mi")

# Plot North Carolina

tm_shape(nc.sids.p,unit= 'miles') + tm_borders() + tm_scale_bar(position = c("left",
"bottom"))
```

Figure 8.1 North Carolina SIDS Data, County Map

8.3 Local Indicators of Spatial Association

Recalling that the purpose of this chapter is to consider localised forms of spatial data analysis, Anselin (1995) proposed the idea of *local indicators of spatial association* (LISAs). He states two requirements for a LISA:

- The LISA for each observation gives an indication of the extent of significant spatial clustering of similar values around that observation
- The sum (or mean) of LISAs for all observations is proportional to a global indicator of spatial association

It should be possible to apply a statistical test to the LISA for each observation, and thus test whether the local contribution to clustering around observation *i* is significantly different from zero. This provides a framework for identifying localities where there is a significant degree of spatial clustering (or repulsion). A good initial example of a LISA may be derived from the Moran's *I* index. Recall that this is defined by

(8.1)

$$I = \frac{n}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(z_i - \overline{z})(z_j - \overline{z})}{\sum_i (z_i - \overline{z})^2} \qquad (8.1)$$

where *zi* is a measurement associated with polygon *i*, and *wij* is a binary indicator as to whether polygons *i* and *j* are neighbours taking the value 0 if they are not, and the value $1/|\delta_i|$ if they are, with $|\delta i|$ being the number of polygon neighbours that polygon *i* has. This expression can be written as

(8.2)

$$I = n\left[\sum_k (z_k - \bar{z})^2\right]^{-1}\left[\sum_k \sum_j w_{kj}\right]^{-1} \sum_i I_i \qquad (8.2)$$

where

(8.3)

$$I_i = (z_i - \bar{z})\sum_j w_{ij}(z_j - \bar{z}) \qquad (8.3)$$

Noting that $n\left[\sum_k (z_k - \bar{z})^2\right]^{-1}\left[\sum_k \sum_j w_{kj}\right]^{-1}$ does not depend on *i*, so that for a given set of z_i it may be regarded as a constant, we have

(8.4)

$$I = \text{const.} \times \sum_i I_i \qquad (8.4)$$

so that I_i is a LISA. As previously, writing $q_i = z_i - \bar{z}$, so that the q_i are mean centred values, we can write

(8.5)

$$I_i = q_i \sum_j w_{ij} q_j \qquad (8.5)$$

so that I_i is the product of q_i and the mean of the q_j values for the neighbours of polygon *i*. If both q_i and the average value of q_j for polygon *i*'s neighbours are all above average, this quantity will be large, indicating a cluster of above- average values focused on polygon *i*. This is also the case if polygon *i* and its neighbours all have values below average. Thus, it can be seen that I_i is a local measure of clustering (either above or below the average value). Also, if the signs of q_i and $\sum_j w_{ij} q_j$ differ, and I_i is a large negative value, this suggests that a local 'repulsion' effect may be occurring, where neighbouring values take opposite extremes. Finally, if the magnitude of I_i is not particularly large (for either positive or negative values) this suggests that there is little evidence for either clustering or repulsion.

For each *Ii*, a significance test may be carried out against a hypothesis of no spatial association. Anselin (1995) provides formulae for the sampling mean and variance of I_i given a randomisation hypothesis as discussed in the previous chapter (essentially this assumes that any permutation of z_i values among polygons is equally likely), and from these, the quantity

(8.6)

$$\frac{I_i - \mathrm{E}[I_i]}{\mathrm{Var}[I_i]^{1/2}} \qquad (8.6)$$

may be used as a test statistic. The R function localmoran in spdep computes *Ii* values, given a set of *zi* values and a listw object providing neighbour weighting information for the polygons associated with the z_i. This function returns a matrix of values whose columns are:

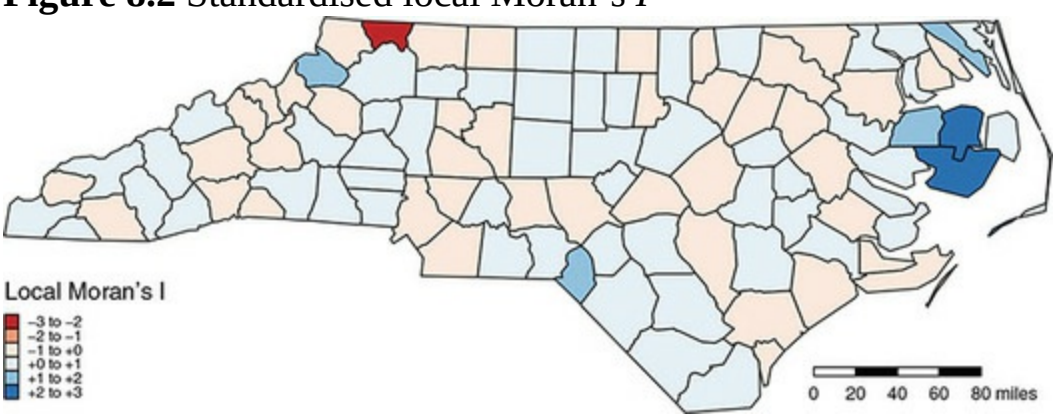
1. The local Moran's *I* statistic, I_i
2. E(I_i) under the randomisation hypothesis
3. Var(I_i) under the randomisation hypothesis
4. The test statistic from equation (8.6)
5. The *p*-value of the above statistic, assuming an approximate normal distribution

The following code computes the rates of SIDS for 1979 per 1000 births, then computes the local Moran's *I* and then produces a map (Figure 8.2) – here the basic I_i values are plotted. Note that tmap detects that the local Moran's *I* has negative and positive values, and chooses a colour shading scheme centred on zero with different hues for positive and negative values. Unfortunately the hues are red and green, which are unhelpful for red–green colour-blind people, hence the tm_style_col_blind() operation at the end to alter this. Finally, the legend.format parameter supplies a list of formatting parameters. The flag="+" argument requires positive values to be preceded with a plus sign. This is mainly an aesthetic consideration as it then balances well with the minus sign for negative values on the legend.

```
require(spdep)
```

```
# Compute the listw object for the North Carolina polygons

# Make sure nc.sids.p is in SpatialPolygonsDataFrame format

if ("sf" %in% class(nc.sids.p))

nc.sids.p <- as(nc.sids.p, "Spatial")

nc.lw <- nb2listw(poly2nb(nc.sids.p))

# Compute the SIDS rates (per 1000 births) for 1979

nc.sids.p$sidspm79 <- 1000*nc.sids.p$SID79/nc.sids.p$BIR79

# Compute the local Moran's I

nc.sids.p$lI <- localmoran(nc.sids.p$sidspm79,nc.lw)[, 1]tm_shape(nc.sids.p,unit=
'miles') + tm_polygons(col= 'lI',title= "Local Moran's I",legend.format=list(flag= "+")) +
tm_style('col_blind') + tm_scale_bar(width= 0.15) +

tm_layout(legend.position = c("left", "bottom"), legend.text.size= 0.4)
```

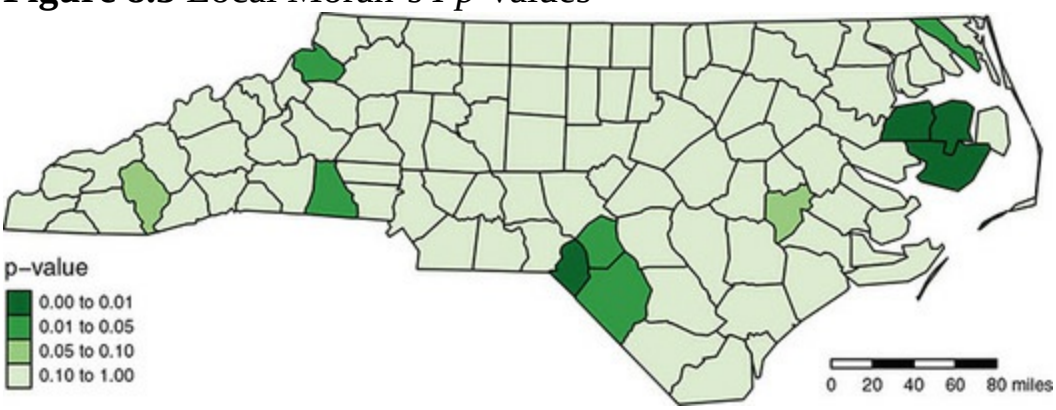
Figure 8.2 Standardised local Moran's *I*



The map shows there is some evidence for both positive and negative I_i values. However, it is useful to consider the *p*-values for each of these values, as considered above. These are mapped below. In this case a manual shading scheme (i.e. one in which the shading interval breaks are specified directly) is used, based on conventional 'critical' *p*-values. The code below produces this (see Figure 8.3). Also, here the palette has been selected manually, via the palette parameter. The minus sign in front of Greens signifies that the palette is used in reverse order (higher numbers are lighter) to reflect the fact that lower *p*-values are more important. Finally, to contrast with the Greens palette, borders are specified to be black.

```
# Create the local p-values

nc.sids.p$pval <- localmoran(nc.sids.p$sidspm79,nc.lw)[, 5]

# Draw the map

tm_shape(nc.sids.p,unit= 'miles') +

tm_polygons(col= 'pval' , title= "p-value" , breaks= c(0, 0.01, 0.05, 0.10, 1),
border.col = "black",palette = "−Greens") +

tm_scale_bar(width=0.15) +

tm_layout(legend.position = c("left", "bottom"))
```

Figure 8.3 Local Moran's *I p*-values



From the resultant map, a number of places can be seen where the *p*-value is notably low (e.g. Washington) – suggesting the possibility of a cluster of either high or low values. Inspecting the actual rate for Washington (which is zero) suggests there may be a cluster of very low rates here. Another region where the *p*-value is low is Scotland County – although in this case the rate is very high, suggesting a cluster of higher values here.

Self-Test Question 1. Verify the significance figures above by selecting and listing the counties for which $p < 0.05$.

8.4 Further Issues with the Above Analysis

The above analysis shows a way in which notable counties – or possibly clusters of neighbouring counties (in terms of their SIDS rates) – can be identified via mapping the *p*-values of local Moran's *I* statistics. However,

there are two notable difficulties with using this approach in an unmodified form. These are:

- Multiple hypothesis testing
- Assuming that the *Ii* are normally distributed

Although these can be thought of as specific issues for this particular study, many are relevant in the general case. It is therefore useful to consider them in turn.

8.4.1 Multiple Hypothesis Testing

In the previous study, there were 100 counties. Using the categories of shading for the map in Figure 8.3 it may be seen that seven counties have $p \leq 0.05$. However, if it is proposed to carry out testing at the 5% level, and if the null hypothesis is true, then the probability of obtaining a false positive result (i.e. a significant value of I_i when in fact the null hypothesis – of randomisation – is true) is 0.05. Thus, even if no spatial process is occurring, we can expect to obtain $100 \times 0.05 = 5$ counties flagged as 'significant'. Thus, even when no effect is present, this approach can generate several false positives. One way this could be dealt with is by comparing the number of significant results observed in the data to the binomial distribution – but ultimately this loses sight of the main objective of the local Moran's *I* approach, since it then just provides a 'whole-study-area' test of whether a spatial process occurs, rather than considering specific localities. If that is all that is required, there is no advantage in the suggested approach over a test based on the standard Moran's *I*.

However, the advertised advantage of a LISA-based approach is its ability to identify *where* clustering is occurring, not just *whether* it occurs. The problem happens because often the method is required to answer both of these questions. If the 'false positive rate' – that is, the probability of detecting a significant *Ii* if the null hypothesis were true – were zero, then *any* significant *Ii* would imply with certainty that clustering does occur. But the false positive rate is not zero – and given that inconvenient fact, one useful approach is to determine the probability of falsely stating that clustering exists on the basis of finding one or more significant *Ii* values. The individual

p-values, and associated tests, apply to individual counties. Assuming the tests are applied independently, and each has a false positive probability *p*, then the probability of not getting a false positive is $1 - p$ for each county. If there are *n* counties, then the probability of getting *no* false positives is $(1 - p)n$, and therefore the probability of getting one or more false positives when looking at *all* counties, denoted by $p*$, is the complement of this, so that

(8.7)

$$p^* = 1 - (1 - p)^n \qquad (8.7)$$

Thus, the $p*$-value can be regarded as a *p*-value for the ensemble of tests on each county – and as a false positive rate for a general test of a 'no clustering' null hypothesis. A further simplification may be made by noting that for small *p*,

$$p^* \approx np \qquad (8.8)$$

Now, if it were desired to find the individual county *p*-value required to give a specified overall $p*$-value, equation (8.7) could be rearranged to give

(8.9)

$$p = 1 - (1 - p^*)^{1/n} \qquad (8.9)$$

or, using the approximation above,

(8.10)

$$p \approx \frac{p^*}{n} \qquad (8.10)$$

Here, $n = 100$ and so if a $p*$-value of 0.05 is required, R can be used as a desk calculator to obtain the countywise *p*:

```
1 − (1 − 0.05) ^(1/100)

[1] 0.0005128014
```

Thus, to make the overall chance of falsely rejecting the null hypothesis of no clustering anywhere equal to 0.05, individual counties should be tested against a *p*-value of around approximately 5.00 × 10–4. The approach using the approximation in equations (8.8) and (8.10) is known as the Bonferroni *p*-value adjustment (see, for example, Šidák, 1967). In R, instead of the 'desktop calculator' approach set out above, the function p.adjust may be used. This takes a slightly different approach – instead of adjusting the threshold for countywise *p*-values to be significant, it adjusts the *p*-values themselves, so they may be compared to the critical value of *p*∗ required. Thus to apply the test above, using p.adjust(pvals,method='bonferroni') on a set of countywise *p*-values returns a set of *adjusted* countywise *p*-values that may be compared against the critical value for *p*∗. Using this approach, anomalous localities can be identified, but the overall probability of any false positives is controlled. For example, comparing adjusted county *p*-values against 0.05 will provide a test where the overall chance of erroneously rejecting the overall hypothesis of no spatial pattern is 0.05.

This idea may now be used to provide a map of *adjusted* local Moran's *I p*-values for the SIDS data analysed earlier (see Figure 8.4).

```
# Create the adjusted p-value

nc.sids.p$pval_bonf <- p.adjust(nc.sids.p$pval, method= 'bonferroni')

# Draw the map

tm_shape(nc.sids.p, unit= 'miles') +

tm_polygons(col= 'pval_bonf', title= "p-value", breaks= c(0, 0.01, 0.05, 0.10,
1),border.col = "black", palette = "−BuGn") +

   tm_scale_bar(width=0.15) +

tm_layout(legend.position = c("left", "bottom"))
```

This reveals that there is in fact a significant pattern (some counties are still significant even after *p*-values are adjusted), and that it is the pattern around Washington County that contributes notably to the departure from an aspatial process. Interestingly, it is a group of very low rates that is detected here.

Figure 8.4 Local Moran's *I* Bonferroni adjusted *p*-values

I

A slightly different approach to explaining the idea of $p*$ is to note that the probability of erroneously rejecting a null hypothesis of no spatial association is equivalent to the probability of erroneously rejecting the smallest *p*-value of all of the counties. Assuming the same threshold is applied to all tests, if the smallest *p*-value falls below this threshold, this is equivalent to the event that at least one county is erroneously flagged as significant. Noting that typically one is testing with a one-tailed (upper-tail) alternative hypothesis, so that large *Ii* values relate to small *p*-values, an alternative way to compute adjusted *p*-values is to compare local standardised Moran's *I* against the distribution of the largest of *n* standard normal variates.

References

Anselin, L. (1995) Local indicators of spatial association – LISA. Geographical Analysis, 27: 93–115.

Getis, A. and Ord, J.K. (1992) The analysis of spatial association by use of distance statistics. Geographical Analysis, 24(3): 189–206.

Šidák, Z. (1967) Rectangular confidence region for the means of multivariate normal distributions. Journal of the American Statistical Association, 62: 626–633.

9 R and Internet Data

9.1 Introduction

While the last few chapters have focused on spatial data analysis, and particularly on statistical approaches, this chapter considers another important aspect of working with spatial data, that of obtaining it from the internet. There are a number of ways that R can access internet data. Had this chapter been written some time ago, much emphasis would have been placed on *web scraping*, where output intended to create human-readable material (such as HTML files) is 'mined' using a computer program to extract relevant information. For example, the underlying HTML for webpages containing weather forecasts or share prices would be downloaded as they would for a web browser, but then the content would be scanned for patterns in their content that contained the information of interest – such as a formatted table with share prices or a table of forecasted maximum and minimum daily temperatures. Although this is still possible in some cases, the situation has perhaps bifurcated – some institutions have taken on board calls for open data, and provided *application program interfaces* (APIs) where direct requests may be made for data in machine-readable form, and others, perhaps initially unaware that data they published could be 'scraped' in this way and used in bulk, have adopted more complex practices in website provision so that a basic 'scraping' approach is no longer possible.

For the situations in which an API is used, R provides generic tools for working with APIs, and also a large number of packages intended to work directly with specific data providers, such as Google. In addition to this, web scraping can also be achieved, and in some cases data files may be directly accessed over the web. The latter is perhaps the simplest option, when available. However, it is perhaps necessary to add a note of caution here. One issue with data from the internet is that data providers may occasionally change the format of API requests, or filenames, or of the dataset itself. Some providers also supply real-time (or near-real-time) data on a rolling basis, so older files disappear after some time period. Thus an approach that works at one point in time is not guaranteed to work indefinitely. In general, the

problem is not irreparable – usually a reorganisation of the website has meant that some files are differently named, or the API has been modified, and existing code can be modified and become usable once more. However, for this reason it is important to have an understanding of the general principles involved in producing R code for accessing the data, rather than regarding code snippets as mystical incantations that can make certain items of data magically appear. The other consequence of this changeability leads the authors to issue a warning – what has just been said about changes in format means that although the examples given here work at the time of writing, we cannot guarantee that they will work indefinitely without modification.

9.2 Direct Access to Data

Let us begin with the situation in which data may be directly downloaded from the internet. R can deal with this in a number of ways. If the dataset on the internet is simply a text file, then the URL can sometimes be substituted for a filename with a number of commands. This works with read.csv and read.table, for example. A simple demonstration is provided here, via a Princeton University website, recording birth rates, an index of social setting and an index of family planning effort for a number of countries.[1] Here the data are read from the remote URL using read.table into a data frame called fpe:

[1] See http://data.princeton.edu/R/readingData.html

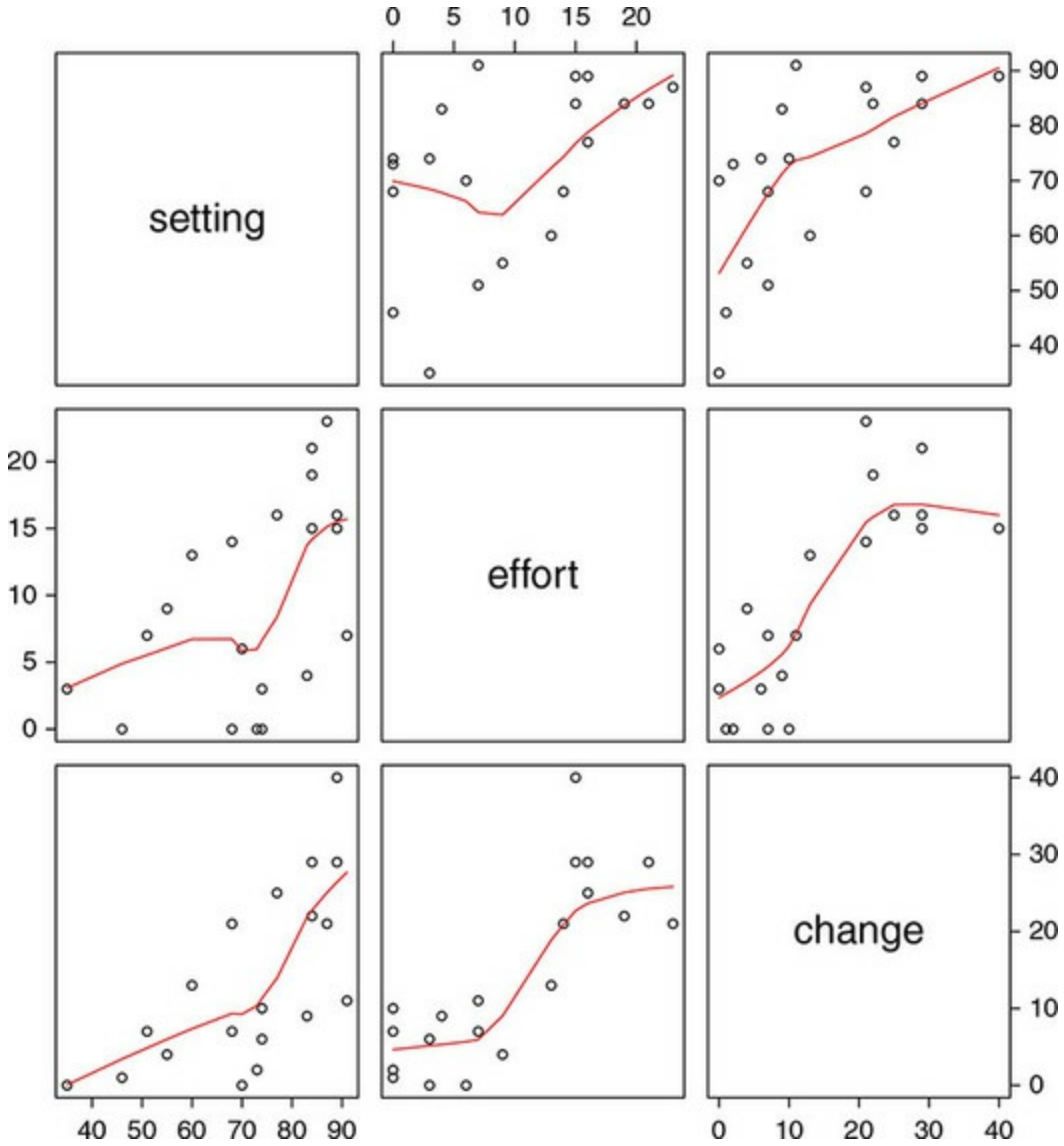
```
fpe <- read.table("http://data.princeton.edu/wws509/datasets/effort.dat")

head(fpe)
```

```
            setting   effort   change
Bolivia          46        0        1
Brazil           74        0       10
Chile            89       16       29
Colombia         77       16       25
CostaRica        84       21       29
Cuba             89       15       40
```

It is then possible to analyse these data in the same way as any other data – here a scatter plot matrix of the variables is drawn (see Figure 9.1) to investigate relationships between the three variables. The panel=panel.smooth option causes a loess smooth (Cleveland, 1979) to be added to each scatter plot.

```
pairs(fpe,panel=panel.smooth)
```

Figure 9.1 Scatter plot matrix of Princeton data



This approach can also be used to access code over the internet, via the source command. An example of this is found on the Bioconductor project[2] – on their website they describe themselves thus: 'Bioconductor is an open source, open development software project to provide tools for the analysis and comprehension of high-throughput genomic data. It is based primarily on the R programming language.' They provide a number of R packages, including a basic collection and a number of further optional packages. Although their aim is to develop code for a relatively specific area of application, some of these packages are of more general use – for example, the Rgraphviz library for visualising graphs, as an alternative to the iGraph package. To install the base Bioconductor package collection (for R version 3.0.1 and later) enter:

[2] http://www.bioconductor.org

```
source("http://bioconductor.org/biocLite.R")

biocLite()
```

Then, to install Rgraphviz, enter:

```
source("http://bioconductor.org/biocLite.R")

biocLite("Rgraphviz")
```

This runs some code located on the Bioconductor website to install the packages. Note also that this is not running a remote process. The remote code is read into R on the user's machine, and then executed on that machine. Although the intention here is to focus on running the remote code, rather than on the use of Rgraphviz, a brief example follows. This uses the state.x77 data frame, supplied with the datasets package. This contains a number of variables recorded for each US state:

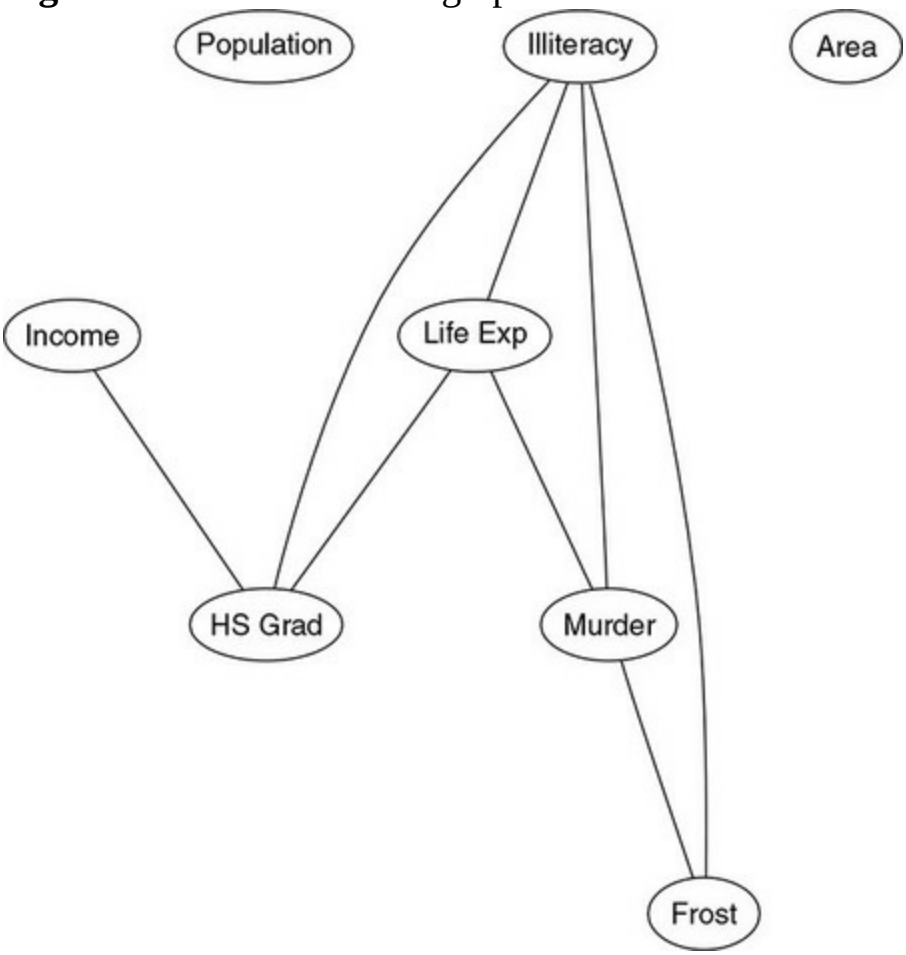
- Population estimate as of 1 July 1975 (Population)
- Income per capita, 1974 (Income)
- Illiteracy, 1970, as a percentage of population (Illiteracy)
- Life expectancy in years, 1969–1971 (Life Exp)
- Murder and non-negligent manslaughter rate per 100,000 population, 1976 (Murder)
- Percentage high-school graduates, 1970 (HS Grad)
- Mean number of days with minimum temperature below freezing, 1931–1960, in capital or large city (Frost)
- Land area in square miles (Area)

The correlations between each of these variables are computed, and the variable pairs whose absolute correlation exceeds 0.5 are noted. A graph is then created whose edges correspond to these pairs. Next, the graph is 'configured' – essentially, locations for the nodes are specified, in a layout designed to limit the number of edge crossings. Finally, it is drawn.[3] The code below executes this procedure, and results in the graph seen in Figure

9.2.

[3] As stated earlier, the aim here is not to provide a detailed tutorial on Rgraphviz, but the code block is commented, and further details are available at http://www.bioconductor.org/packages/2.12/bioc/vignettes/Rgraphviz/inst/doc

Figure 9.2 Illustration of Rgraphviz




```
# The following two packages are required:
require(Rgraphviz)
require(datasets)
# Load the state.x77 data
data(state)
# Which ones are 'connected' - i.e. abs correlation above 0.5
```

```
connected <- abs(cor(state.x77)) > 0.5

# Create the graph - node names are the column names

conn <- graphNEL(colnames(state.x77))

# Populate with edges - join variables that are TRUE in 'connected'

for (i in colnames(connected)) {

for (j in colnames(connected)) {

if (i < j) {

if (connected[i,j]) {

conn <- addEdge(i,j,conn, 1)}}}}

# Create a layout for the graph

conn <- layoutGraph(conn)

# Specify some drawing parameters

attrs <- list(node=list(shape="ellipse",                    fixedsize=FALSE,
fontsize= 12))

# Plot the graph

plot(conn, attrs=attrs)
```

As can be seen, the population and area variables do not connect strongly with other variables (it is of note that these depend directly on the size of the state, while the others are state averages or per capita rates). Illiteracy is connected to the greatest number of other variables.

9.3 Using Rcurl

The package RCurl [4] provides quite a lot of extra functionality for accessing data from the web. Essentially, it provides a set of tools to allow R to act as a web client. This is done by providing a number of helper functions. Perhaps the most basic is getURL: given a URL (including secure HTTPS and a number of other protocols, such as FTPS), this function returns the information located at the URL. In some cases this might be the content of an HTML file specifying a webpage (which might possibly be used for web

scraping), while in others it may be plain text (e.g. the content of a CSV file). The latter situation will be considered here. The file 1871702.csv is located on the UK government's server and contains the 2010 English Index of Multiple Deprivation (IMD) scores[5] in CSV format. This index combines measures of deprivation via a number of different dimensions (see Department for Communities and Local Government, 2012, for example), and the CSV file contains deprivation scores and ranks for each dimension, as well as an overall score[6] and rank for each Lower Super Output Area (LSOA) in England – there are 32,482 LSOAs in all. It also provides a look-up from LSOAs to larger areal units (such as Government Office Regions, GORs).

[4] See http://www.omegahat.org/RCurl/ for full details.

[5] Contains public sector information licensed under the Open Government Licence v2.0.

[6] Higher scores imply a greater degree of deprivation.

The full URL for this file appears in the code below. As can be seen, this URL uses the HTTPS protocol. The getURL function is used to read the contents of the file into temp. The getURL function always returns the information of the URL into a single-character variable – including control characters such as carriage returns. The information is not particularly helpful in this form. The content is in fact a CSV file, and a command such as read.csv would ideally be used to read this content into a data frame. Fortunately, R offers this possibility, via the textConnection function. Given a character argument (such as that returned by getURL), this function creates a *connection* – a kind of pseudofile that can be read by a function usually requiring a filename as input. The content of the pseudofile is just the character content of the argument to textConnection. Thus, by storing the information obtained from getURL in a temporary variable, and then using this variable as input to textConnection and finally inputting this to read.csv, the file may be read into a data frame. This is set out below:

```
library(RCurl) # Load RCurl

# Get the content of the URL and store it into 'temp'
```

```
stem <- ’https://www.gov.uk/government/uploads/system/uploads’

file1 <- ’/attachment_data/file/15240/1871702.csv’

temp <- getURL(paste0(stem,file1))

# Use textConnection to read the content of temp

# as though it were a CSV file

imd <- read.csv(textConnection(temp))

# Check - this gives the first 10 column names of the data frame

head(colnames(imd), n= 10)

[1]    "LSOA.CODE"

[2]    "PRE.2009.LA.CODE"

[3]    "PRE.2009.LA.NAME"

[4]    "POST.2009.LA.CODE"

[5]    "POST.2009.LA.NAME"

[6]    "GOR.CODE"

[7]    "GOR.NAME"

[8]    "IMD.SCORE"

[9]    "RANK.OF.IMD.SCORE..where.1.is.most.deprived."

[10]    "INCOME.SCORE"
```

MS Windows users should ignore the line:

```
temp <- getURL(paste0(stem,file1))
```

and replace:

```
imd <- read.csv(textConnection(temp))
```

with:

```
imd <- read.csv(paste0(stem,file1))
```

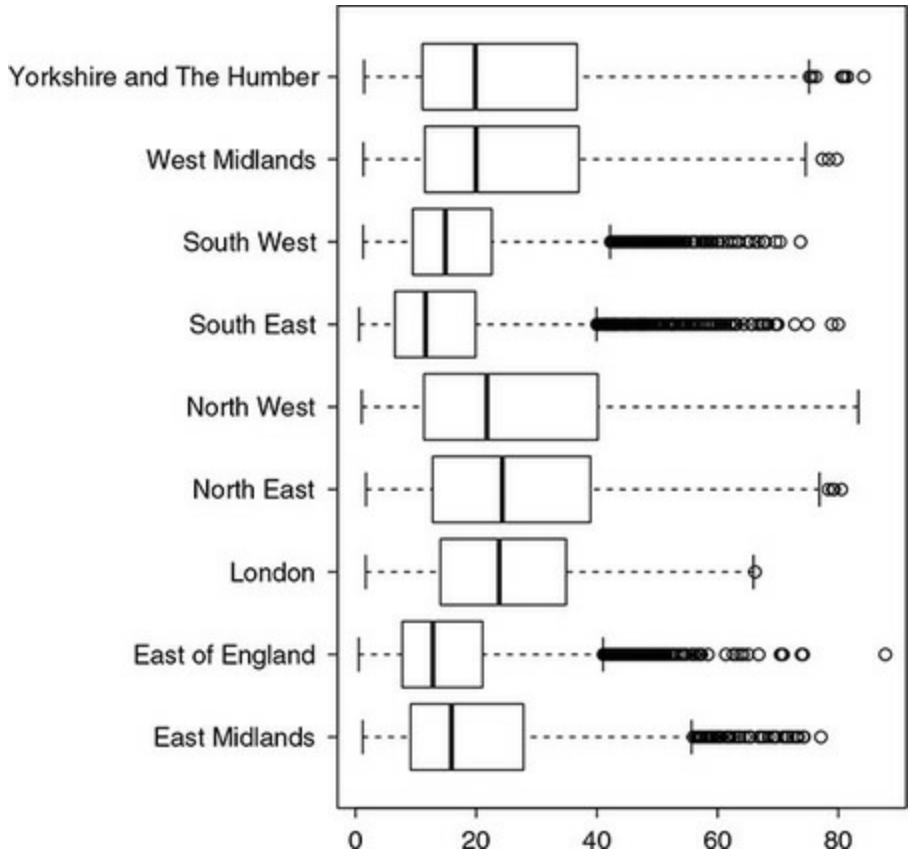
If a number of CSV files are going to be read in this way (i.e. via a URL using the HTTPS protocol) it may be helpful to define a function read.csv.https to do this in a single command:

```
read.csv.https <- function(url) {
temp <- getURL(url)
return(read.csv(textConnection(temp)))
}
```

This function is then used in the following code block to create a boxplot of the IMD for each GOR in England. The result is reproduced in Figure 9.3.

```
# Download the csv data and put them into 'imd2'
imd2 <- read.csv.https(paste0(stem,file1))
# Modify the margins around the plot, to fit the GOR
# names into the left-hand margin
par(mar=c(5,12,4,2) + 0.1)
# Create the boxplot. The 'las' parameter specifies y-axis
# labelling is horizontal, x-axis is vertical
boxplot(IMD.SCORE~GOR.NAME, data=imd, horizontal=TRUE, las= 2)
```

Figure 9.3 Boxplot of IMDs by Government Office Regions



The boxplots show patterns varying across England – the lowest median levels of the IMD are in the East of England and South-East GORs, although London has a higher level. The North-East has the highest median level. However, the East of England actually has the highest IMD for an individual LSOA, and it can also be seen that some GORs have a more prominent upper tail than others, suggesting that some parts of England are more prone to small ‘pockets of deprivation’ than others. The variation in distribution shape across the UK also suggests that geographically weighted summary statistics may be a useful exploratory tool here.

9.4 Working with Apis

As well as providing ‘raw’ text files, many websites provide ‘bespoke’ data in response to requests – for example, all of the detached houses for sale in a given locality, or all of the crimes occurring in a given rectangular region. Obtaining data often takes the form of a client request, followed by a server response. The requests specify the information to be returned – for example,

a location which is the centre of an area for crimes to be returned. It is often possible to specify these requests as part of a URL. The protocol for these specifications is essentially the API. For example, the police.uk website allows requests of the form:

```
http://data.police.uk/
```

where the items in angle brackets are replaced by actual values – such as:

```
http://data.police.uk/
```

Note that the lines are not broken in practice – this was done here just to fit the URLs on the page. These return all crimes in a 1 mile radius of the specified latitude and longitude, for the month specified by <date>. It can therefore be seen that the requests consist of a number of named arguments (here, lat, lng and date).

It is quite possible to construct requests as above, and simply use getURL to retrieve the results. However, it is also possible to use getForm which accepts the named parameters in the same format as named parameters in R. For example:

```
crimes.buf <- getForm( "http://data.police.uk/", lat=53.401422, lng=−2.965075,
date="2016−04")
```

Although functionally identical, this format is easier to read – a useful characteristic when revisiting old code. Note, however, that as with getURL, the information returned is not quite ready to be used. As before, it is returned as a single character string, although this time the data are in JavaScript Object Notation (JSON) format, rather than CSV. JSON is a more sophisticated format, allowing lists with named elements, arrays, and other forms of data to be represented. In particular, it allows lists within lists, and so on, to be represented. The crime data here are returned in this format. The data item that the JSON format character string represents consists of a list of items, one for each crime. Each item is itself a list, which contains at least the following items:

- Crime category (category)
- Unique crime identifier (persistent_id)
- Month of the crime (month)
- A list item (location) containing the approximate crime latitude (latitude), approximate crime longitude (longitude), and street details (street), the latter itself a list including an ID number for the street (id) and a street name (name) specifying a degree of uncertainty (e.g. ‘On or near Florizel Street’)
- A location type (location_type), either ‘Force’ or ‘BTP’ (British Transport Police)

In fact some further items are present, but we will focus on the ones listed, as they supply location, date and crime type.

To convert the variable crimes.buf from a single character into a usable R object, the function fromJSON in the package jsonlite will be used.

```
require(jsonlite)

crimes <- fromJSON(crimes.buf)
```

The variable crimes is now an R list object, whose items meet the description above. To check this, enter:

```
crimes[[1]]

$category

[1] "anti-social-behaviour"

$location_type[1] "Force"

$location

$location$latitude

[1] "53.390927"

$location$street

$location$street$id

[1] 908665
```

```
$location$street$name

[1] "On or near Geraint Street"

$location$longitude

[1] "−2.964659"

$context[1] ""

$outcome_status

NULL

$persistent_id

[1] "69db2c85a755b3ed67e6fa3649fbc6e82e612d33f0a846964e911a6d773f41a1"

$id

[1] 48497941

$location_subtype

[1] ""

$month

[1] "2016−04"
```

This is the first item describing a crime, in a list of all crimes within a 1 mile radius of the location 53.401422°N, 2.965075°W, in April 2016. This is a helpful object format for storing complex information – for example, the street name and ID are in a list within a list within a list. However, most data analysis in R uses the more familiar vector, matrix and data frame formats.

The next step is to extract the relevant information from crimes. This is a two-stage process:

1. Create a function to extract the information needed from an individual list item.
2. Use sapply to apply the function to each item in the list, and recombine the results into an array.

The method is illustrated below – here the function getLonLat extracts longitude and latitude.

```
getLonLat <- function(x) as.numeric(c(x$location$longitude,
x$location$latitude))

crimes.loc <- t(sapply(crimes,getLonLat))

head(crimes.loc)
```

```
         [,1]     [,2]
[1,] -2.964659 53.39093
[2,] -2.968009 53.40600
[3,] -2.951200 53.39023
[4,] -2.947242 53.40495
[5,] -2.979260 53.40598
[6,] -2.944165 53.40867
```

The as.numeric function is used, since the latitude and longitude are stored as characters. Note that the sapply function returns the listwise results as one *column* per item, whereas data frames and matrices are usually formatted as one row per item. Thus the t function is applied, which transposes rows and columns.

Next, some of the attribute data for each of the crimes is extracted. This is done in the same way:

```
getAttr <- function(x) c(

x$category, x$location$street$name, x$location_type)

crimes.attr <- as.data.frame(t(sapply(crimes,getAttr)))

colnames(crimes.attr) <- c("category","street","location_type")

head(crimes.attr)

          category

1   anti-social-behaviour

2   anti-social-behaviour

3   anti-social-behaviour

4   anti-social-behaviour
```

```
5   anti-social-behaviour
6   anti-social-behaviour
  street
1   On or near Geraint Street
2   On or near Further/higher Educational Building
3   On or near Greenheys Road
4   On or near Ivatt Way
5   On or near Back Lime Street
6   On or near Botanic Place
  location_type
1   Force
2   Force
3   Force
4   Force
5   Force
6   Force
```

Here, the matrix created is converted to a data frame (with as.data.frame). Inspecting the top of the data frame created, it may be seen that the crimes are located in Liverpool, UK, and that the BTP crimes are generally associated with railway stations (such as Liverpool Central). Although the crimes have a spatial identifier, in this case they are not associated with streets. Finally, the location and attribute information can be combined to provide a SpatialPointsDataFrame:

```
library(sp)
crimes.pts <- SpatialPointsDataFrame(crimes.loc,crimes.attr)
# Specify the projection - in this case just geographical coordinates
proj4string(crimes.pts) <- CRS("+proj=longlat")
# Note that 'head' doesn't work on SpatialPointsDataFrames
```

```
crimes.pts[ 1: 6,]

                    coordinates                     category
1        (-2.964659, 53.39093)        anti-social-behaviour
2          (-2.968009, 53.406)        anti-social-behaviour
3           (-2.9512,53.39024)        anti-social-behaviour
4         (-2.947242,53.40495)        anti-social-behaviour
5          (-2.97926,53.40598)        anti-social-behaviour
6         (-2.944165,53.40867)        anti-social-behaviour

  street
1  On or near Geraint Street
2  On or near Further/higher Educational Building
3  On or near Greenheys Road
4  On or near Ivatt Way
5  On or near Back Lime Street
6  On or near Botanic Place
  location_type
1  Force
2  Force
3  Force
4  Force
5  Force
6  Force
```

It is also possible to create further SpatialPointsDataFrames by taking subsets of the data. The following creates a set with incidents of anti-social behaviour only:

```
asb.pts <- crimes.pts[crimes.pts$category=="anti-social-behaviour",]
```

This creates a set with criminal damage and arson cases:

```
cda.pts <- crimes.pts[crimes.pts$category=="criminal-damage-arson",]
```

These data will be revisited, but for now a plot contrasting the locations of anti-social behaviour against criminal damage and arson crimes can be created. First, the two SpatialPolygonsDataFrames are joined together (using rbind) and then the category column is converted from factor to character. This stops tmap from producing a key with *all* possible category types when only two are examined here. The result is shown in Figure 9.4.

Figure 9.4 Locations of anti-social behaviour and criminal damage/arson incidents

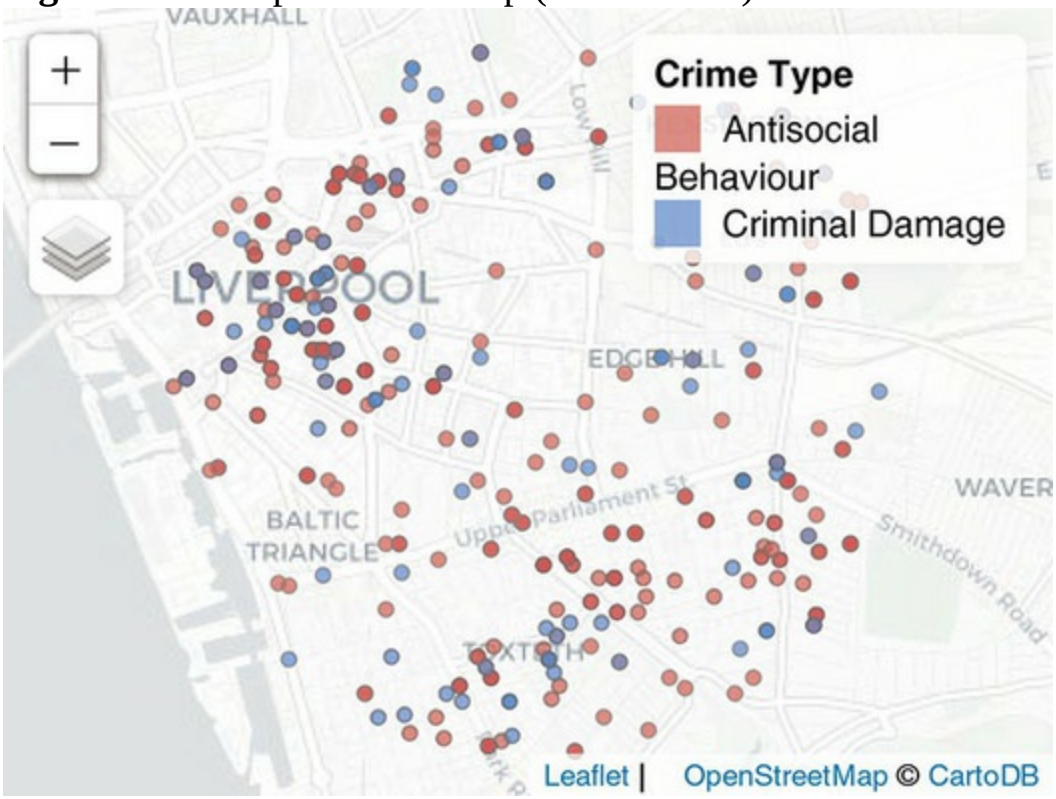
One thing to note here is that although the locations are stored as latitude and longitude, so that directly plotting coordinates as *x* and *y* locations would give

a distorted map, the plot method recognises this (from the proj4string) and corrects for this. As can be seen, both sets of points lie within a circle; however, without this correction they would appear to lie within an ellipse.

This dataset is also one that can usefully be viewed on a zoomable map (although be aware that the locations are not exact, so that zooming too closely may lead to incorrectly located crimes (Singleton and Brunsdon, 2014)). R code to achieve this is given below, resulting in a zoomable map as seen in Figure 9.5.

```
tmap_mode('view')

tm_shape(asb_cda.pts) + tm_dots(col='category', title='Crime Type',

labels=c("Antisocial Behaviour","Criminal Damage"),

palette=c('indianred','dodgerblue'), size=0.02)
```

Figure 9.5 Liverpool crime map (web version)



9.5 Creating a Statistical 'Mashup'

In this section another API will be accessed, and the information obtained from it will be used in conjunction with the police.uk API. The new API is provided by Nestoria[7] and supplies lists of housing properties that are currently for sale. The API can be accessed via getForm in much the same way as the police.uk site.[8] Among other things, this supplies the asking price, latitude and longitude of properties. In the example below, a sample of 50 three-bedroom terraced houses is retrieved, and decoded from JSON form into an R object:

[7] www.nestoria.co.uk

[8] When using this API, follow the guidelines at http://www.nestoria.co.uk/help/api

```
terr3bed.buf <- getForm("https://api.nestoria.co.uk/api",
action='search_listings',
place_name='liverpool',
encoding='json',
listing_type='buy',
number_of_results=50,
bedroom_min=3, bedroom_max=3,
keywords='terrace')
Warning in testCurlOptionsInFormParameters(.params): Found
possible curl options in form parameters: encoding
terr3bed <- fromJSON(terr3bed.buf)
```

The warning appears because the keyword encoding is also used by the getForm function directly, as well as a keyword in the Nestoria API. However, in this case it is interpreted as a Nestoria keyword, which is required in the example, therefore the warning may be ignored.

The results are now stored as a list in terr3bed$response$listings. Inside each item in the list, a number of other items are stored. Among those of interest

here are price, longitude and latitude (further details of the keywords in the API and the returned information can be found at http://www.nestoria.co.uk/help/api). The following code extracts these and stores them in a data frame:

```
getHouseAttr <- function(x) {
   as.numeric(c(x$price/ 1000,x$longitude,x$latitude))
}
terr3bed.attr <- as.data.frame(t(sapply(terr3bed$response$listings,
getHouseAttr)))
colnames(terr3bed.attr) <- c("price","longitude","latitude") head(terr3bed.attr)
    price    longitude    latitude
1   100.00   −2.928117    53.46545
2    85.00   −2.949700    53.40997
3   120.00   −2.953319    53.41240
4   209.95   −2.985383    53.38715
5   170.00   −2.954813    53.37964
6   315.00   −2.916008    53.36753
```

Note that prices are divided by 1000 – this is just to return simpler numbers for formatting. Next, these data are combined with police.uk data. Essentially, the aim here is to provide a further data item to terr3bed.attr – a count of the number of household burglaries occurring within a 1 mile radius of each house during April 2016. This gives a measure of the frequency of burglaries that occur close to each house. This rate will then be compared to the price of each house. By restricting the study to three-bedroom terraced houses, it is hoped that much of the price variation attributed to the characteristics of the house itself will be controlled.

The code to create this extra variable is shown below:

```
# Create an extra column to contain burglary rates
```

```
terr3bed.attr <- transform(terr3bed.attr, burgs= 0)
# For each house in the data frame
for (i in 1: 50) {
# Firstly obtain crimes in a 1-mile radius of the house's
# latitude and longitude and decode it from JSON form
crimes.near <- getForm("http://data.police.uk/",
lat=terr3bed.attr$latitude[i],
lng=terr3bed.attr$longitude[i],
date="2016−04")
crimes.near <- fromJSON(crimes.near)
    crimes.near <- as.data.frame(t(sapply(crimes.near,getAttr)))
# Then from the 'category' column count the number of burglaries
# and assign it to the 'burgs' column
terr3bed.attr$burgs[i] <- sum(crimes.near[, 1] == 'burglary')
# Pause before running next API request - to avoid overloading
# the server
Sys.sleep(0.7)
# Note this stage may cause the code to take a minute or two to run
}
```

Essentially this code makes a call to the police.uk API for each entry in the house price data frame. Once it has run, the relationship between price and burglary rate may be plotted as a scatter plot. As one or two house prices are very high, a log scale is used for the *y*-axis (the house price axis). The following code produces Figure 9.6:

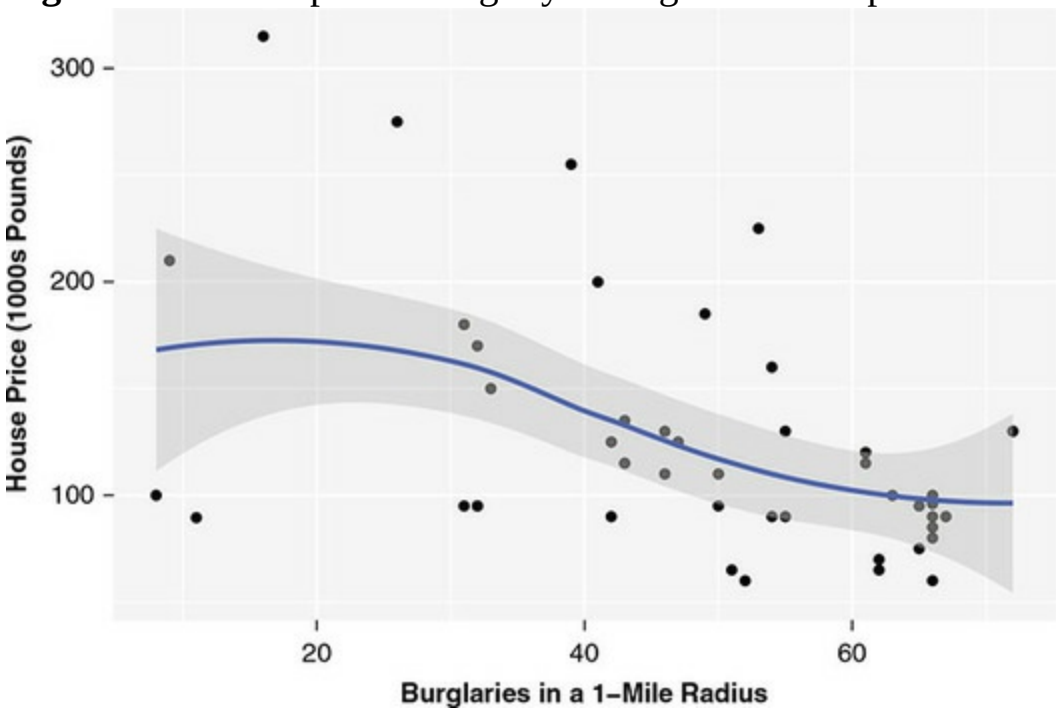
```
library(ggplot2)
ggplot(terr3bed.attr,aes(x=burgs, y=price)) + geom_point() +
geom_smooth(span= 1) +
```

```
labs(x='Burglaries in a 1-Mile Radius',
y='House Price (1000s Pounds)')
```

A scatter plot of house price against nearby burglary count is created (via geom_point) and then a smooth line (with error bands) is added via geom_smooth. This suggests that there is some relationship – possibly a drop in price in the range of about 30–60 burglaries in the 1 mile radius. However, the graph demonstrates there are some outliers[9] – and it should be noted that this is a relatively small sample.

[9] Although the smoothing technique used (loess) does take outliers into account.

Figure 9.6 Scatter plot of burglary rate against house price



One final issue to be aware of with the Nestoria API is that it operates in real time – so that a request returns a list of houses that are currently on the market. This implies that running the code above at a future point may well not return the same results as listed here, since different data will be returned. It is suggested also that runs of the analysis in the future should use a more recent month for recorded crimes than April 2016.

9.6 Using Specific Packages

Although it is possible to access many APIs using the RCurl package as a toolkit, there are a number of R packages that are designed to access specific APIs, such as Google Maps, Twitter or Eurostat. Here we focus on Eurostat, via the R eurostat package (Lahti et al., 2017). This package provides a set of tools for accessing and manipulating the data from the Eurostat open data service. It essentially provides a 'shell' around the API for this service, so that data are accessed via higher level R functions.

Here the dataset tgs00026 is downloaded and read into R. This dataset is the disposable income of private households by NUTS2 regions.

I

From the Eurostat website (http://ec.europa.eu/eurostat/en/web/products-datasets/-/TGS00026): 'The disposable income of private households is the balance of primary income (operating surplus/mixed income plus compensation of employees plus property income received minus property income paid) and the redistribution of income in cash.'

The code to download this is given below. The time_format option specifies that time is stored in character format.

```
library(eurostat)

tgs00026 <- get_eurostat("tgs00026", time_format = "raw")

head(tgs00026)

# A tibble: 6 x 5
      unit    na_item      geo      time     values
     <fctr>   <fctr>    <fctr>    <chr>     <dbl>
1    PPCS_HAB B6N       AT11      2003      15800
2    PPCS_HAB B6N       AT12      2003      16900
3    PPCS_HAB B6N       AT13      2003      17800
4    PPCS_HAB B6N       AT21      2003      15700
5    PPCS_HAB B6N       AT22      2003      15900
6    PPCS_HAB B6N       AT31      2003      16200
```

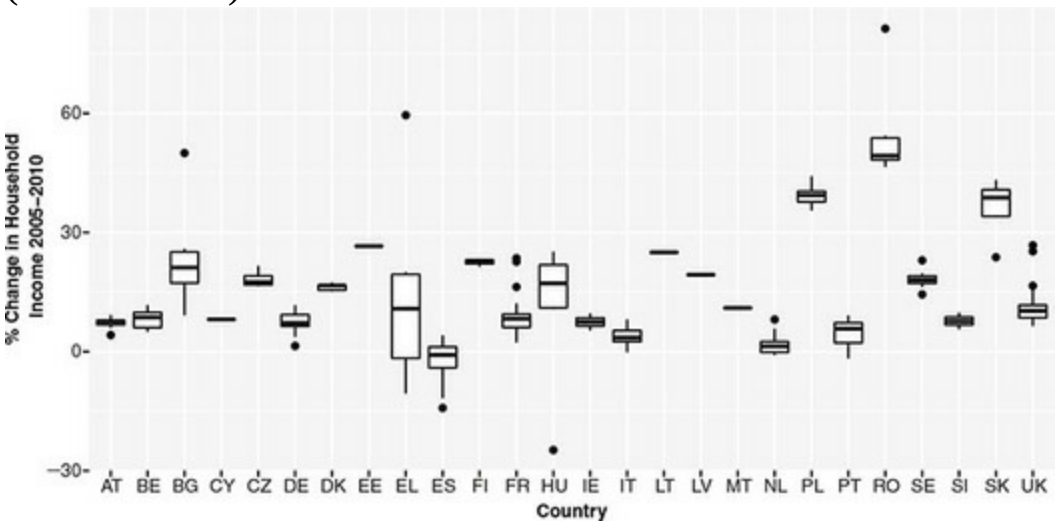
Also, rather than the absolute income, here we will focus attention on the change in household income between 2005 (pre-recession) and 2010, expressed as a percentage of 2005 household income. To do this, two subsets of the data are taken, for the years 2005 and 2010, in data frames called tgs00026_05 and tgs00026_10. The percentage change is computed and added as a new column in the 2010 data frame.

```
tgs00026_05 <- tgs00026[tgs00026$time=='2005',]

tgs00026_10 <- tgs00026[tgs00026$time=='2010',]

tgs00026_10$delta <-

100*(tgs00026_10$values − tgs00026_05$values)/tgs00026_05$values
```

The geo column provides the NUTS2 code for each area. Next, the country code is extracted from this. As the NUTS2 code is a factor, this is converted to character mode, and the first two letters are extracted – these signify the country level.

```
tgs00026_10$country <- substr(as.character(tgs00026_10$geo),1,2)
```

Figure 9.7 Boxplots of disposable income change, 2005–2010, by country (Eurostat data)



With these data, it is possible (via ggplot) to create boxplots of NUTS2 region household income change by country. The code below produces

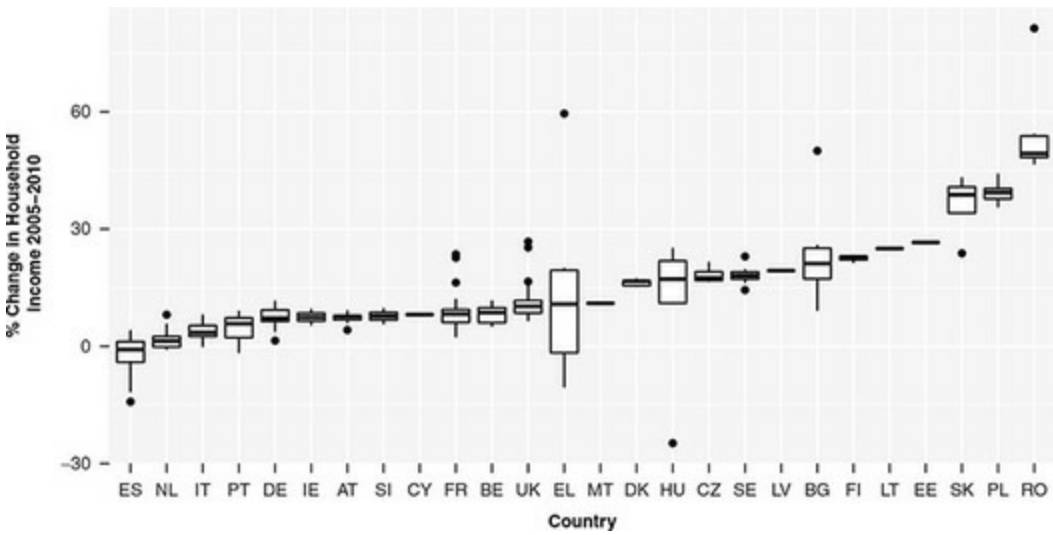
Figure 9.7.

```
ggplot(tgs00026_ 10,aes(x=country, y=delta)) + geom_boxplot() +
labs(x="Country", y="% Change in Household\n Income 2005−2010")
```

A problem with this is that because the boxplots are ordered alphabetically by country code, the diagram is fairly difficult to read. A more helpful ordering might be based on the order of the median income change of NUTS2 region by country. However, ggplot orders categorical variables by the order of the factor levels that are specified if the variables are of type factor – and by default these are alphabetical. The forcats library provides some helpful tools for working with factors – in particular, it offers an option to reorder them according to some user-defined criterion, with the fct_reorder function. Given a factor variable, an associated variable and a summery function, it computes the summary function on a level-by-level basis and then reorders the levels in accordance with these summary values. With the following code a similar boxplot is achieved, but ordered by median NUTS2 level (see Figure 9.8).

```
library(forcats)
tgs00026_10$country <-fct_reorder(tgs00026_10$country,tgs00026_10$delta,median)
ggplot(tgs00026_10,aes(x=country,y=delta)) + geom_boxplot() +
labs(x="Country", y="% Change in Household\n Income 2005−2010")
```

Figure 9.8 Median ordered boxplots of disposable income change, 2005–2010, by country



There are a number of interesting patterns in these data – here we will focus on the fact that Spain (ES) has one of the most negative levels of change of household income. A useful function in the eurostat package is the ability to make SpatialPolygonsDataFrame objects from ordinary data frames, based on the region codes. This is achieved with the function merge_eurostat_geodata.

An example is given here. First, the subset of tgs00026_10 for Spain (country=='ES') is selected, and then merge_eurostat_geodata is used to create the SpatialPolygonsDataFrame (you should check whether the eurostat package is installed). All of the options for the function can be seen by entering ?merge_eurostat_geodata, but one notable argument is geocolumn, stating which column in the data frame specifies the geography. The all_regions option specifies whether to provide non-specified regions with NA values or simply remove them from the SpatialPolygonsDataFrame. The code below creates the map in Figure 9.9.

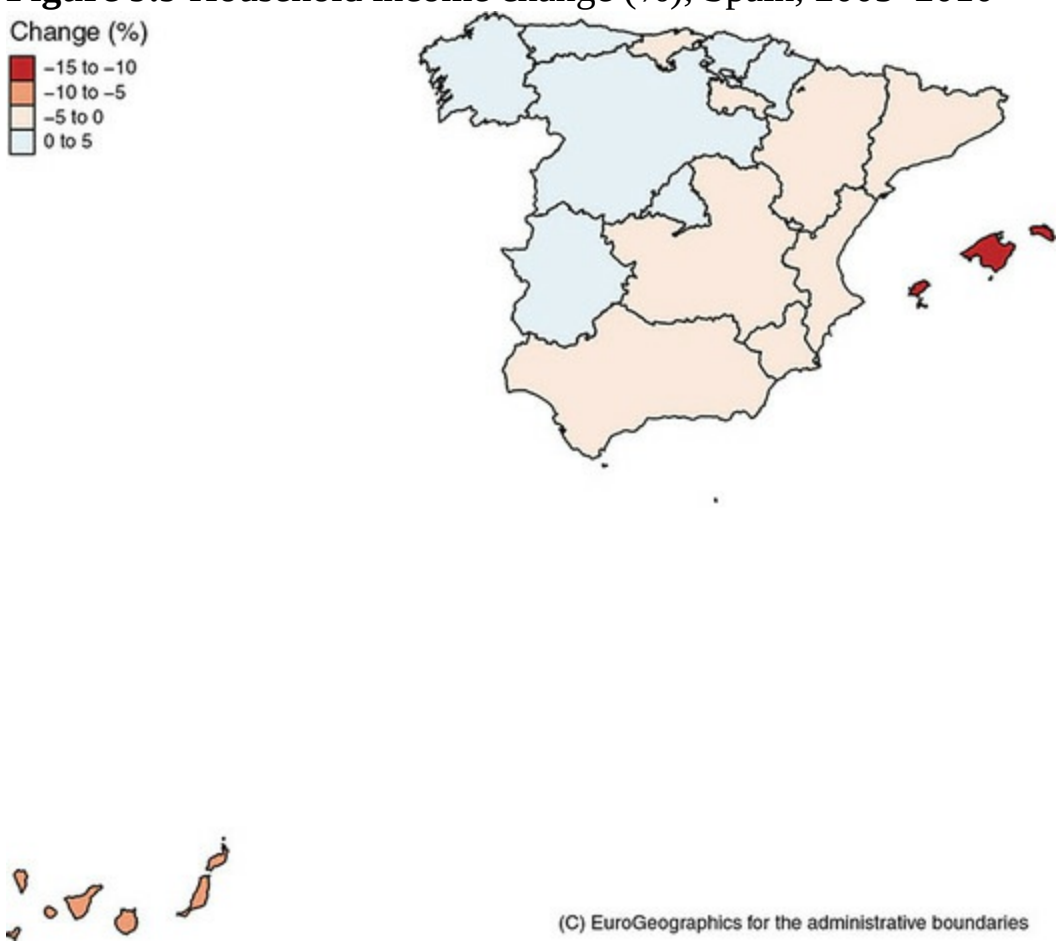
```
library(eurostat)

tmap_mode('plot')

tgs_es <- tgs00026_10[tgs00026_10$country=='ES',]

tgs_map <- merge_eurostat_geodata(data = tgs_es, geocolumn = "geo",
                                  resolution = "1", output_class = "spdf", all_regions = FALSE)

tm_shape(tgs_map) + tm_polygons(col='delta', title="Change (%)") +

tm_style('col_blind') + tm_credits("(C) EuroGeographics for the administrative
```

```
boundaries")
```

This makes it clear that there are strong regional patterns in the change in household income – with negative changes to the south and east of the mainland and in Gran Canaria, and most strongly in the Balearic Islands.

To focus on the mainland, it is possible to filter out Gran Canaria and the Balearics by noting that their NUTS2 IDs are ES70 and ES53. The code to do this follows, giving rise to Figure 9.10.

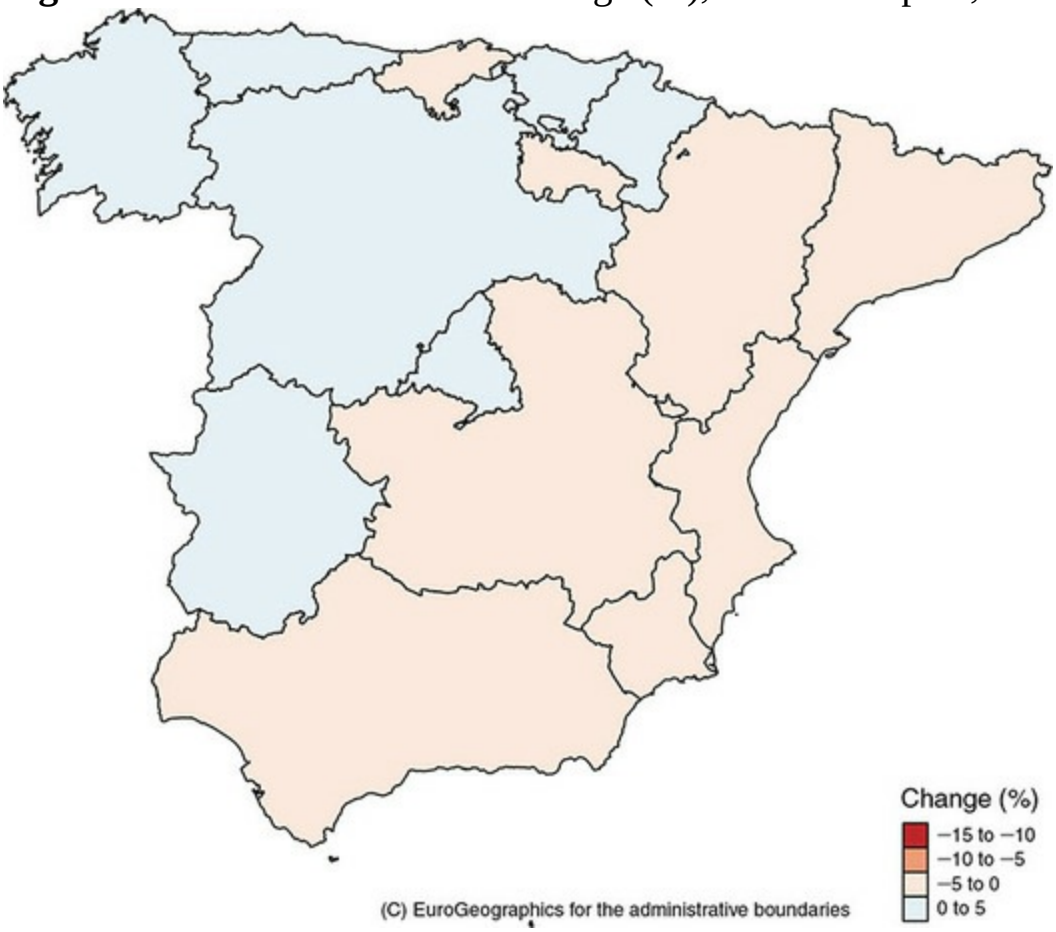
Figure 9.9 Household income change (%), Spain, 2005–2010




```
tmap_mode('plot')

tgs_mlmap <- tgs_map[!(tgs_map$NUTS_ID %in% c("ES70","ES53")),]
tm_shape(tgs_mlmap) + tm_polygons(col='delta', title="Change (%)") +

tm_style('col_blind') + tm_layout(legend.position=c("right","bottom")) +
```

```
tm_credits("(C) EuroGeographics for the administrative boundaries")
```

Figure 9.10 Household income change (%), mainland Spain, 2005–2010



9.7 Web Scraping

The final aspect of web-based information gathering to be covered here is *web scraping*. As mentioned earlier, this is perhaps the oldest approach to obtaining information from the web, and involves directly reading information from HTML code used to create human-readable content. This process is the successor to early techniques to extract information from Teletext pages through USB TV tuners for computers.[10] The arrival of APIs means that this technique has been less frequently used in recent years, but it is still needed occasionally. The method used for web scraping in R typically involves techniques for text pattern searching and pattern extraction, mostly achieved through the use of *regular expressions* (see Aho, 1990, for example). Regular expressions are a way of specifying patterns to search for

in character data. The very simplest expression is just a direct string – for example, the pattern ’chris’ simply specifies the five letters c, h, r, i and s appearing in sequence in a string. Thus, the string ’chris brunsdon’ would match this expression, since it has these five letters appearing in sequence. However, the string ’lex comber’ would not prove a match. It should also be noted that ’Chris Brunsdon’ does not match, because regular expressions discriminate between upper- and lower-case characters.

[10] See http://nxtvepg.sourceforge.net/man-ttx_grab.html, for example.

If one wanted to find strings that contain either ’Chris’ or ’chris’, one possible regular expression might be ’[Cc]hris’ – a sequence of characters inside square brackets will match a stream with *any* of these characters where the square-bracketed list occurs. Any number of characters may lie within the square brackets. Sequences may also be specified: for example, ’[0−9]’ matches any single numeric digit. Some postfix modifiers may be used: a character or pattern followed by ’+’ means that pattern may be repeated one or more times – thus ’ [0−9]+ ’ matches a whole number preceded and succeeded by a space (note that spaces are also matched). A number of other modifiers and pattern specifiers exist. Some notable ones are listed here:

- ’∗’: A pattern preceding this symbol may be repeated zero or more times – for example, ’Chris[0−9]∗’ matches ’Chris’, ’Chris1’, ’Chris2013’ and so on, since the pattern preceding the ’∗’ is ’[0−9]’
- ’?’: A pattern preceding this symbol may be repeated zero times or once – for example, ’-?[0−9]+’ matches a positive or negative whole number, since the ’?’ is preceded by ’-’
- ’.’: This pattern matches any character – for example, ’#.∗’ would match a Twitter hashtag, since this is a hash character followed by any combination of characters
- ’^’: This pattern matches the beginning of a line – for example, ’^[0−9]’ matches a line starting with a numeric character
- ’$’: This pattern matches the end of the line – for example, ’!$’ matches a line ending with an exclamation mark
- ’\’: If this symbol is placed in front of one of the special symbols it implies the special symbol should be matched literally rather than take on its special meaning – for example, ’\.doc’ matches ’.doc’, the ‘dot’

being taken literally rather than matching any character

The above list is a very brief overview of a fairly involved topic. A comprehensive treatment is given in Friedl (2002).

These patterns may be used in R via a number of related functions. The function grepl takes two arguments: the first is a pattern, and the second is an array of one or more character variables. It returns TRUE for each character value that matches the pattern, and FALSE for each one that does not:

```
grepl('Chris[0−9]*',c('Chris','Lex','Chris1999','Chris Brunsdon'))
[1]    TRUE FALSE    TRUE    TRUE
```

Also, the grep function returns the indices of the items in the list that match:

```
grep('Chris[0−9]*',c('Chris','Lex','Chris1999','Chris Brunsdon'))
[1] 1 3 4
```

Finally, grep with the value=TRUE option returns the actual matching character strings:

```
grep('Chris[0−9]*',c('Chris','Lex','Chris1999','Chris Brunsdon'),
value=TRUE)
[1] "Chris"    "Chris1999"    "Chris Brunsdon"
```

These functions are the key tools in R for finding the lines in the HTML code that contain the information of interest when web scraping. This is best illustrated with a practical example.

9.7.1 Scraping Train Times

The Accessible UK Train Timetables website[11] provides a simple interface to UK train timetable information. This is an unofficial site, but acknowledges National Rail Enquiries for allowing the site to use information from the

official site.[12] The main advantage of the unofficial site here is that it provides a very simple query system. For example, http://traintimes.org.uk/durham/leicester/00:00/monday returns a webpage listing train times for journeys from Durham to Leicester after midnight on the Monday following the date of the request.

[11] http://traintimes.org.uk

[12] http://www.nationalrail.co.uk

Suppose it is desired to extract the departure and arrival times from this web-page. The HTML content of the webpage can be read into a character array (one element of the array being one line of the HTML file) using readLines.

```
web.buf <- readLines( "https://traintimes.org.uk/durham/leicester/00:00/monday")
```

If you are curious you might type in web.buf to see what the HTML looks like. Next, grep can be used to select the lines in the HTML corresponding to the train departure and arrival times. These all emphasise the times of interest with the HTML tag, and then take the form dd:dd - dd:dd where each dd is a two-digit number (e.g. 04:59 − 08:23), and an appropriate regular expression is ’strong.*[0−2][0−9]:[0−5][0−9].*[0−2][0−9]:[0−5][0−9]’ – this accounts for the fact that the first digit of the hour must be 0, 1 or 2, and the first digit of the minute cannot exceed 5.

```
times <- grep("strong.*[0−2][0−9]:[0−5][0−9].*[0−2][0−9]:[0−5][0−9]",

web.buf,value=TRUE)
```

Again, typing in times will verify that the lines of interest have been selected. The previous stages have selected the lines actually containing the information of interest. The next stage is to extract the specific information required.

Each line has two times, with the pattern ’[0−2][0−9]:[0−5][0−9]’. A companion function to grep is gregexpr – this returns a list giving the locations in the input strings where the part of the string actually matching

the pattern begins. It also supplies information about the length of the pattern match. If the pattern occurs more than once, a vector of locations is given:

```
locs <- gregexpr("[0−2][0−9]:[0−5][0−9]",times)

# Show the match information for times[1]

locs[[1]]

[1] 46 60

attr(,"match.length")

[1] 5 5

attr(,"useBytes")

[1] TRUE
```

Thus, for the strings selected in times there are two matches, one for the departure time and one for the arrival time. If *x* is the location of the time within the string, then characters *x* to *x* + 1 supply the hours, and characters *x* + 3 and *x* + 4 supply the minutes. In the final section of code these pieces of information are extracted, and converted into numerical values. Finally, a new column is added to the data frame which is the duration of the journey in decimal hours:

```
timedata <- matrix(0,length(locs), 4)

ptr <- 1

for (loc in locs) {

timedata[ptr, 1] <- as.numeric(substr(times[ptr],loc[1],loc[1]+1))

timedata[ptr, 2] <- as.numeric(substr(times[ptr],loc[1]+3,loc[1]+4))

timedata[ptr, 3] <- as.numeric(substr(times[ptr],loc[2],loc[2]+1))

timedata[ptr, 4] <- as.numeric(substr(times[ptr],loc[2]+3,loc[2]+4))

ptr <- ptr + 1

}

colnames(timedata) <- c('h1','m1','h2','m2')
```

```
timedata <- transform(timedata, duration = h2 + m2/60 − h1 − m1/60 )

timedata
```

	h1	m1	h2	m2	duration
1	6	38	9	23	2.750000
2	6	44	9	46	3.033333
3	7	38	10	23	2.750000
4	8	48	11	23	2.583333
5	9	49	12	23	2.566667

Although this is a fairly basic example, it provides an indication of the approach used to extract information from 'raw' HTML data. However, it is important to realise that if the design of the webpage is changed, it may be necessary to revisit any web-scraping code, since the patterns specifying the data of interest may need to be altered. This would be the case, for example, if the website in the example altered the format for displaying times from dd:dd to dddd (i.e. 0823 instead of 08:23).

References

Aho, A. (1990) Algorithms for finding patterns in strings. In J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, pp. 255–300. Cambridge, MA: MIT Press.

Cleveland, W.S. (1979) Robust locally weighted regression and smoothing scatterplots. Journal of the American Statistical Association, 74: 829–836.

Department for Communities and Local Government (2012) Tracking economic and child income deprivation at neighbourhood level in England, 1999–2009. Neighbourhoods Statistical Release. London: DCLG. https://www.gov.uk/government/uploads/system/uploads/attachment_data/f

Friedl, J. (2002) Mastering Regular Expressions. Sebastopol, CA: O'Reilly.

Lahti, L., Huovari, J., Kainu, M. and Biecek, P. (2017) Retrieval and analysis of Eurostat open data with the eurostat package. The R Journal, 9(1): 385–392. https://journal.r-project.org/archive/2017/RJ-2017-019/index.html.

Singleton, A. and Brunsdon, C. (2014) Escaping the pushpin paradigm in geographic information science: (Re)presenting national crime data. Area, 46(3): 294–304.

10 Epilogue

10.1 The Future of R as a Tool For Geocomputation

Considering the future of R is not an easy task – it is an already complex language, with what some may argue is an even more complex collection of libraries, covering a wide range of techniques and application areas, many of which extend beyond the original purpose of R as a programming language and interactive environment for statistical data analysis. The existence of this book – much of which is about using R as a tool for manipulating geographical information and the production of maps – is evidence of this. If the authors had considered the future of R at the point of its first public release in 1995, we admit it would be highly unlikely that we could have predicted the current situation. Indeed, now the book is in its second edition, we are in a position to reflect on notable changes that have occurred even in the time period between the two editions.

In the first edition we identified three aspects of potential future trajectories for R:

- Extensions of R as a language
- Improvements ‘under the bonnet’
- Coexistence with other software

We still argue that this is a helpful categorisation, although with the benefit of hindsight, we may update or revise some of our original opinions as to what these pathways may contain.

10.2 Extensions of R As a Language

Almost certainly this is the category that has seen most development in recent years, mainly due to the development of dplyr and related packages – colloquially referred to as the ‘Hadleyverse’ due to the input of Hadley Wickham. More formally, the idea is based on the notion of a ‘tidyverse’

(Wickham and Grolemund, 2016), broadly described as ‘an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs.’

Essentially, data are generally stored in such a way that columns represent *key-value* collections – for example, for monthly rainfall data, one row would represent one month’s rainfall figure, with a year-and-month value in one column and the rainfall value in another. Note that there could be multiple keys (e.g. year in one column, and month in an adjacent column, and possibly a location in a third column) and multiple values (e.g. total rainfall and average temperature for that month). This differs, for example, from a situation where rainfall is stored in a matrix, with rows as years, and columns as months.

The ‘tidy data’ idea is essentially to create operators that take key-value data frames and return other key-value data frames. These operators can do things like filter the data (e.g. remove observations after a certain year) or summarise it (e.g. compute yearly total rainfall). In both of those examples, the output data could be represented in key-value format. For the summarised data, the keys would be modified (i.e. only the year column would remain) as well as the values (these would now be yearly sums, grouped by years).

This approach lends itself well to a *fluent* or *method chaining* style of programming where operations on the data frame can be thought of as similar to a pipeline, where data frames are passed through a series of operators in order to carry out the data processing. The final result is then fed into some kind of analysis or visualisation method which terminates the pipeline. The pipelining operator is %>% so that, for example, to compute yearly mean rainfall for all observations in years 2010 and later, when rainfall is in the data frame rainfall_data, one could write:

```
rainfall_data %>% filter(year >= 2010) %>%
group_by(year) %>% summarise(mean_rainfall=mean(rainfall)) ->
yearly_data
```

and then carry out visualisation techniques (possibly via ggplot) on

yearly_data. Whereas a full discussion of this style of coding is beyond the scope of this book, it can be seen how this pipeline style can make data manipulation code easy to follow. More formally, a pipeline operator applies the right-hand argument (a function) to the left-hand argument (a key-value data frame). Thus:

```
f(x,y,z,...)
```

and:

```
x %>% f(y,z,...)
```

are equivalent. In the special case where function f has only one argument:

```
f(x)
```

is equivalent to:

```
x %>% f
```

Functions can be thought of as internal (key value to key value), initiators (anything to key value) or terminators (key value to anything). The idea is that a pipeline is an initiator, one or more internal functions, and finally a terminator. An initiator might, for example, take a filename and read in the file returning a key-value data frame, the internal functions transform this in some way, and the terminator will create a graphic or possibly the output of some kind of statistical analysis.

This approach separates data cleaning, data selection and other kinds of pre-processing from analysis or visualisation – and also offers a consistent and extendable model for data transformation. To create new internal functions, a certain set of conventions must be followed, but if this is done, then one can be sure they may be generally used in this framework.

It could be argued that this approach has effectively extended the language. The syntax and coding style used here are quite different from standard (or

base) R. At least one colleague of the authors has commented that they do not really recognise this as R! Second, the uptake of this approach is significant, both for carrying out ‘real-world’ analyses and as a teaching tool for data science. Thus, the impact of the approach is strong.

In terms of the future, at the time of writing, although not all kinds of data analysis or manipulation are well addressed by the ‘tidyverse’, the list is growing. Perhaps of most relevence to this book, the sf package (Pebesma, 2016) allows users to handle spatial objects in this way. Although, as of now, not all spatial analysis or manipulations have been adapted to work with these kind of objects, work is certainly progressing. For example, tmap will now work with ‘traditional’ R spatial objects, such as spatial polygons data frames, as well as sf objects.

Thus, at least outwardly, the appearance of R and the way it is used could well change notably in the coming years.

10.3 Improvements ‘Under the Bonnet’

‘Tidyverse’-style changes have a very visible and user-facing influence on R – adding to the R language will change the way coders interact with R as a tool. However, another aspect of R that undergoes change is its internal design. The most obvious kind of ‘invisible’ change is when internal algorithms or memory management are made more efficient. The only difference in user experience is that issuing the same commands leads to a faster result or more effective memory usage by the R process. Although on release 3.4.3 at the time of writing, some notable changes in memory management occurred in release 3.0.0, speeding up several operations. As incremental version upgrades have occurred, other gradual improvements have taken place. For example, as part of the numerical processing software embedded in R, the system-level LAPACK library is used. As this is improved, updated versions are incorporated into R.

The ‘tidyverse’ has some influence here as well. Among other things, a new set of routines for reading in files (such as read_csv) are provided – these perform similar tasks to exiting routines (such as read.csv) but do so considerably faster. Although it has been some time since the ‘quantum leap’

in speed due to version 3.0.0 happened, incremental changes are ongoing, and are likely to continue to do so.

10.4 Coexistence with Other Software

A final area where R is currently extending, and we feel will continue to extend, is its ability to work with other software. This can occur in a number of ways. In terms of big data, one path forward might be to manipulate and summarise very large datasets using some other software best suited to that task and pass on the pre-processed data to R. The Rcpp package facilitates such an approach; it provides a framework for creating functions in C++ (a compiled language that has interfaces to a number of alternative data-processing tools) and creating an interface to R, so that the C++ functions can be called directly from R, and can act as a bridge. Of course, C++ is also a powerful tool in itself, and, since it is compiled, can also offer more rapid execution of algorithms that are prohibitively slow in R.

A further example is RStudio. This is an integrated development environment for R that runs on Windows, Mac or Linux computers with a number of user-friendly features. It provides a graphical front end for R and includes a console, a scripting window, a graphics window, and an R workspace window. Some of its key features are a colour-coded text editor (also present on the Mac R package), an integrated help and graphics and an interactive debugger. It also has tools that aid the development of packages. Once installed, it has the same functionality as R, uses exactly the same code and draws from the packages installed in normal R libraries. It provides a standard interface to R (i.e. the Windows, Linux and Mac versions are the same), and many users find this environment easier for developing their code, especially users who are new to R. Since the first edition of this book, its functionality has grown massively. It now offers the facility to create reproducible documents (documents that contain the code used to carry out any data analysis they report) through the Sweave, knitr and Rmarkdown packages. These allow R to be embedded in LaTEX or *Markdown* – both are tools for creating documents. The code is executed, and the output automatically included. When the files are compiled in RStudio (or R) all the outputs of the data analysis such as the code itself, any maps, tables or graphs are created on the fly and inserted into the final document.

Sweave, knitr and Rmarkdown come with the standard R installation and are described in full at http://www.stat.uni-muenchen.de/~leisch/Sweave/. The first two both compile .Rnw scripts (Rmarkdown compiles .Rmd scripts) and generate LaTEX files that can be directly converted to PDF files in RStudio. In fact, this book, including all the code snippets, examples, exercises and figures, was created and compiled in Rmarkdown. Embedding the code in the document in this way has a number of advantages. First, it supports dynamic data analysis, allowing analyses to be updated automatically if the data or the analysis change. Second, it provides a transparent and reproducible research environment: rather than inserting a graph or table from Excel, for example, the Sweave document contains the R code necessary to generate each figure or table, allowing for reproducible research. At the time of writing, reproducibility in research is seen as an important issue, and so tools of this kind are also likely to become important.

A further way in which RStudio enables linkages with other software tools is through the use of *HTML widgets* – essentially interactive tools written for web browsers that allow users to interact with data. For example, these are used in tmap when tmap_mode is set to 'view' – where access to the *Leaflet* JavaScript map viewing library is exploited.

As a final example, the shiny framework and package provide a tool for creating interactive webpages using R.[1] To do this, shiny defines *reactive expressions* – chunks of R code that are linked to the values of sliders, buttons and other widgets, so that they are re-evaluated whenever the user interacts – these in turn are connected to output widgets such as graph panels, so that graphical and textual outputs and so on may be interactively linked. It has two components: a user interface definition (defining buttons, sliders and so on) and a 'server' definition that specifies the actions associated with these components of the interface. In terms of spatial data analysis, shiny offers the opportunity to generate interactive web mapping using R, without any knowledge of HTML, style sheets or JavaScript. However, it is sufficiently flexible that it may be augmented with HTML to modify the default styles of the interface or functionality. This may now be combined with many HTML widget packages, so that Leaflet maps, plotly graphics and many other interactive data visualisation tools can be incorporated, and used in combination. We expect the range of widgets to increase notably over time.

Since HTML widgets were launched there are 93 R packages providing links to different types of widget.

[1] http://shiny.rstudio.com

10.5 Finally…

If you have worked through this book, you are now proficient in the use of R for the analysis and visualisation of spatial data. The discussion in this chapter will allow you to extend and develop your R-based projects and applications, hopefully acting as a springboard for further exploration and development. Perhaps one of the best ways to further your understanding of R is to explore possibilities such as these. The discussion may also help you to form an impression of how R as a tool for spatial data analysis will develop in the coming years. We hope that you will enjoy doing this as much as we have enjoyed exploring these possibilities in order to produce this book, and that some of you will play a major role in the future of R.

References

Pebesma, E., Bivand, R., Cook, I., Keitt, T., Sumner, M., Lovelace, R., Wickham, H., Ooms, J. and Racine, E. (2016) sf: Simple features for R. R Package Version 0. 6–3. http://cran.r-project.org/package=sf.

Wickham, H. and Grolemund, G. (2016) R for Data Science. Sebastopol, CA: O'Reilly.

Index

adjacency matrices, 245–246
Altman, D.G., 214
AND operations, 186
Anselin, L., 283–284
application program interfaces (APIs), 290, 306, 310–311, 317
apply functions, 6, 137–140
area of a polygon, calculation of, 157–158
argument lists, 124
arrow pointing upward (^), meaning of, 128
assignment of *numeric*, *character* or *logical* values, 16
attributes, 81–84, 90
    continuous, 222–223
    see also distance attributes; projection attributes; spatial attribute analysis
autocorrelation, 260
    visual exploration of, 248–257
Azzalini, A., 197

bandwidths, 196, 199
Battemberg plots, 274
Besag, J., 215, 277–279
‘bespoke’ data from websites, 297
‘big data’, 319
Bioconductor project, 292
Bivand, R.S., 3
bivariate regression, 168
Bland, J.M., 214
Bonferoni *p*-value adjustment, 288–289
Bowman, A., 197
box-and-whisker plots, 100–101
boxplots, 49, 100–101, 297, 308
braces *see* curly brackets
Brewer, Cynthia, 85

buffers, 153–156

calc function, 188
Cartesian minimum distances, 164
CartoDB maps, 198
categorical data, 23
character variables, 16–18
choropleth maps, 84–89, 92, 226, 246–248
classified data, 23
clipping of spatial data, 151–152, 155
closing of files, 80
clustering, 57, 162–163, 195, 208–216, 248–251, 283–288
code blocks, 122–126, 296
    nested, 131
code snippets, 4, 10, 13–14, 22, 59, 69, 118, 291, 320
coding in “R”, 4–5
colour codes, 85
Comber, A.J., 165
command line interfaces, 10
comma-separated variable (csv) files, 50–51
complex spatial randomness (CSR), 208–219
conditional autoregressive (CAR) models, 262–265, 277
conditional loops, 126, 132
conditional statements, 119–122
copying to clipboard, 78
cor function, 158–159
C++ language, 319
cropping *see* clipping
cross-*K* function, 219
cross-*L* function, 219–222
cube roots, 129–132
curly brackets ({}), 122–124

data, direct access to, 291–295
data capture devices, 2–3
data checking, 129–132
data classes, 16–17, 19–29

in lists, 26
user-defined, 25–26
data files, 51–52
data frames, 26–28, 56, 59, 63, 301, 308–309, 317–318
data types, 16–19
debugging, 126–127, 130–131
declaration of variables, 124
deldir package, 223
dependencies, 58
deterministic loops, 132–134
distance attributes, 163–175
in raster analysis, 189
downloading of “R” and RStudio, 7
dplyr package, 6, 140–141, 149, 316

embedded exercises in text, 4
envelope analysis, 213–215
equals (=) sign, 14
Euclid’s algorithm, 132–133
Eurostat, 306–309

factor function, 22–23
false positives, 287–288
fields in databases, 56
for loops, 125, 138–139
Ford, E.D., 214–215
Fotheringham, A., 260
Friedl, J., 312
functions, 119–120, 123–125
writing of, 127–135
see also logical functions

gDistance and *gWithinDistance* functions, 164–166
Gentleman, Robert, 7
geocomputation, the future of “R” as a tool for, 316
geographic information systems *see* GIS software
geometry, spatial, 61

Geospatial Data Abstraction Library (GDAL), 64, 183
geostatics, 237
Getis, A., 282
gets arrow, 14
G-function, 216–218
ggplot package, 5–6, 43–50
GIS software in general 4, 148, 175
GISTools package, 2, 6, 38, 58–60, 150, 157
Google, 290
Google Maps, 93, 95
great circle distances, 164
greatest common divisor (GCD), 132
grep function, 312
gstat package, 228–229

hash (#) sign, 15
help files, 22
hexagonal binding, 203–207
H-function, 242
histograms, 47–48, 100
Hope, A.C.A., 262
HTML widgets, 320
hypothesis testing, 281
    multiple, 287–288

Ickstadt, K., 279
if statements, 120, 129–131
Ihaka, Ross, 7
image command, 42
index decomposition, 281
information boxes in text, 5
installation of packages, 8, 56–57
internet data, use of, 290–314
interpolation of point patterns, 222–235
    nearest-neighbour type of, 223–224
interrelations between points, 195
intersection, 152–156, 170–173, 186

inverse distance weighting (IDW), 227–235, 241
isolines, 200

Javascript Object Notation (JSON), 298

Kcross function, 219
kernel density estimates, 196–207
    KDE maps, 202–203
Kest function, 211–213
key value collections, 317–318
K-functions, 207–215, 242, 281
Kooperberg, C., 277–279
kriging, 235–242
    universal, 242

lagged means, 254–256
lazy data format, 1
Lcross function, 219
Leaflet library, 76–77, 320
Lest function, 15
L-function, 215–216
lines function, 36
lists, 24–26
loading of packages, 57
local indicators of spatial association (LISAs), 283–287
localised spatial analysis, 281–289
location data, 3
locator function, 41
logical functions, 18–19, 122
logical operators, 92–93, 122
loops, 6, 119–120, 125–126, 132–134, 138
Loosmore, N.B., 214–215
lower case, use of, 15

Mac computers, “R” on, 8, 319
map algebra, 182
mapping, 66–98
    with lines, 94–96

with points, 89–94
with polygons, 84–89
with raster data, 96–98
maps, saving of, 78–80
maptools functions, 5–6
marked point patterns, 218–222
marked processes, 195
Markov chain Monte Carlo (MCMC) approach, 279
Martin, Douglas, 7
matrix function, 20–22
maximum absolute deviation, 214–215
merging of spatial features, 155–156
method chaining in programs, 317
mirror sites, 8
Moran plots, 255
Moran's '*I*' statistic, 257–262, 281–289
mosaic plots, 104–107, 167
moving window approaches, 281
multi-criteria analysis, 182
Multivar package, 203–205

nearest-neighbour interpolation, 223–224
neighbours, definitions of, 251
'nugget' effect and variance, 236, 238
numcolwise function, 267
numeric data variables, 16, 18
NUTS2 code, 307–308

opening of files, 80
OpenStreetMap, 74
OR operations, 186
Ord, J.K., 273, 282
ordered function, 23
overlay function, 188–189
overlay operations, 148–149
overplotting problem, 201, 203
Owen Guide, 11, 13–14

packages, commonly used, 8–9
PBSMapping package, 3
Pebesma, E., 57
percentages (%%) symbol, 133
pipeline style of data handling, 317–318
piping syntax, 141
plots, 34–43
    basic tools for, 35–39
    colour in, 39–43
    different types, 35–37
    of raster data, 42–43
    text placement in, 41
plyr package, 265–266
point data, display of, 201
point-in-polygon calculations, 157, 169
point pattern analysis, 194–243
    interpolation, 222–235
    second-order, 207–218
points function, 36
Poisson processes, 208
Poisson regression, 159–160, 270
polygon function, 37–38
predictor variables, 265
pre-processing steps, 149, 318
probability densities, 47–48, 195–196, 200
programming in “R”, 118–119, 127
programs, building blocks for, 120–127
projection attributes, 170
pseudofiles, 295

qplot () function, 44
queen’s case neighbours, 251–253, 271
quick maps, 67–68

“R”
    availability of package in, 292, 295, 306
    basic ingredients of, 14–16

coexistence with other software, 319–320
extensions of, 316–318
features of, 1–2, 10–11, 158, 194
freely available resources for, 11–12
future prospects for, 316, 320–321
improvements made to, 318–319
obtaining and running software, 7–10
project website, 7
reading, writing, loading and saving data, 50–52
used for statistical computing, 7
user-facing influence on, 318
random functions, 236–238
randomisation, 260–261
raster analysis, 2–3, 21, 182–189
data preparation for, 182–183
distance functions in, 189
layers in, 65, 187–188
mapping of attributes for, 96–98
mathematical functions applied to layers, 187–188
overlays in, 184–187
plots of data, 42–43
reclassification of data, 183–187
raster package, 180, 188–189
raster and vector environments, conversion between, 175–182
RColorBrewer package, 42
Rcop package, 319
RCurl package, 295–297, 306
RData files, 51
reactive expressions, 320
records in databases, 56
rect tool, 41–42
regression, 102–104, 159–160, 168, 195, 270–277
regular expressions, 311
repeated application of commands, 118–119
reproducibility of research, 320
‘repulsion’ effects, 283–284
residuals, 160

rgdal package, 64
Ripley, B.D., 210, 212
rnorm function, 36
rook's case neighbours, 251–253, 271
RStudio, 7, 10–11, 14, 319–320

sample function, 249
S and S+ language, 7
saving as an image, 78
scatter plots, 35, 102, 291–292, 305–306
Scott, D., 197
scripts, 10–12, 14
self-test questions, 4, 29–34, 107–110, 128, 132–135, 140–143, 149, 158, 163, 167–168, 174, 190–192, 201, 241, 256, 277–279, 286
    answers to, 52–54, 110–116, 128, 143–147, 243
semivariograms, 237–239
seq function, 125
sf format, 57, 60–63, 149, 175
    reading to and writing from, 65–66
sf package, 2–6, 57, 60–62, 318
shading schemes, 160–163, 285
shapefiles, 46–47, 63–64, 270–271
shiny framework and package, 320
sieve mapping, 182
Simple Features format, 3
simulation-based approaches, 261–262
simultaneous autoregressive (SAR) models, 262–265, 269, 281
social media sites, 11
sort function, 23–24
source functions, 128–129, 292
sp class of objects, 60
sp format, 2, 6, 57–58, 84, 149
    reading to and writing from, 64–65
sp package, 57–58, 61, 189
spatial attribute analysis, 245–279
spatial autocorrelation, 248
spatial autoregressive models, 262–277

calibration of, 263–265, 268–269
troubleshooting for, 270–277
spatial data
nature of, 194–195
formats for, 61
in GISTools, 58–60
reading and writing of, 63–66
spatial data analysis, 1–4, 55–66
recent changes in 1–2
spatial data files, 51–52
spatial data structures, 135–140
spatial datasets, creation of, 91
spatial distances and their attributes, combination of, 169–175
spatstat package, 194, 211, 214–219, 229
spdep package, 194, 254–255, 259, 280, 284
standardised residuals, 160
statistical analysis, descriptive, 98–107
subsetting, 16, 27, 72, 152–153
sum function, 22–23
supply and demand modelling, 163–164

table function, 23
thematic data attributes, 66
Thiessen polygons *see* Voronoi polygons
tibble class, 26–29
tibble format, 56
tidy data format, 1
tidyverse collection, 1–2, 43, 56, 140–141, 316–319
tmap package, 1, 6, 66, 77, 84, 246, 318
full version of, 69–74
parameters in, 70
train times, 313–314
transparency of point data, 90
Twitter, 11

union operations, 156
upper case, use of, 15

vector–raster conversion, 175–182
vectors, 2–3, 15–16, 19–20
vignettes, 62, 140
Voronoi polygons, 223, 224–226

Wald statistics, 169
Wall, M.M., 270–272, 276–277
web scraping, 290, 295, 310–314
website for present book, 5, 10, 12, 15
Wickham, Hadley, 251, 316
widgets, 320
Windows computers, “R” on, 8, 319
Wolpert, R., 279
working directories, 78–79, 135

xlim option, 36

ylim option, 36

zoomable maps, 303
Z-statistics, 169