Geocomputation with R

Robin Lovelace, Jakub Nowosad, Jannes Muenchow

2019-02-26

Contents

List of Tables

List of Figures

Welcome

This is the online home of *Geocomputation with R*, a book on geographic data analysis, visualization and modeling.

Note: This book has now been published by CRC Press in the R Series[1]. You can buy the book from CRC Press[2], Wordery[3], or Amazon[4].

Inspired by **bookdown**[5] and the Free and Open Source Software for Geospatial (FOSS4G[6]) movement, this book is open source. This ensures its contents are reproducible and publicly accessible for people worldwide.

The online version of the book is hosted at geocompr.robinlovelace.net[7] and kept up-to-date by Travis[8], which provides information on its 'build status' as follows:

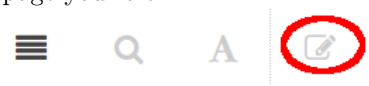
The version of the book you are reading now was built on 2019-02-26 and was built on Travis[9].

How to contribute?

bookdown makes editing a book as easy as editing a wiki, provided you have a GitHub account (sign-up at github.com[10]). Once logged-in to GitHub, click on the 'edit me' icon highlighted with a red ellipse in the image below. This will take

[1] https://www.crcpress.com/Chapman--HallCRC-The-R-Series/book-series/CRCTHERSER
[2] https://www.crcpress.com/9781138304512
[3] https://wordery.com/geocomputation-with-r-robin-lovelace-9781138304512
[4] https://www.amazon.com/Geocomputation-Chapman-Hall-Robin-Lovelace/dp/1138304514/
[5] https://github.com/rstudio/bookdown
[6] http://foss4g.org/
[7] https://geocompr.robinlovelace.net
[8] https://travis-ci.org/Robinlovelace/geocompr
[9] https://travis-ci.org/Robinlovelace/geocompr
[10] https://github.com/

you to an editable version of the the source R Markdown[11] file that generated the page you're on:



[12]

To raise an issue about the book's content (e.g. code not running) or make a feature request, check-out the issue tracker[13].

Reproducibility

To reproduce the code in the book, you need a recent version of R[14] and up-to-date packages. These can be installed with the following command (which requires **devtools**[15]):

```
devtools::install_github("geocompr/geocompkg")
```

To build the book locally, clone or download[16] the geocompr repo[17], load R in root directory (e.g. by opening geocompr.Rproj[18] in RStudio) and run the following lines:

```
bookdown::render_book("index.Rmd") # to build the book
browseURL("_book/index.html") # to view it
```

Supporting the project

If you find the book useful, please support it by:

- Recommending, citing[19] or linking-to[20] it
- 'Starring[21]' the geocompr GitHub repository[22]
- Reviewing it, e.g. on Amazon or Goodreads[23]
- Buying a copy

[11] http://rmarkdown.rstudio.com/
[12] https://github.com/Robinlovelace/geocompr/edit/master/index.Rmd
[13] https://github.com/Robinlovelace/geocompr/issues
[14] https://cran.r-project.org/
[15] https://github.com/hadley/devtools
[16] https://github.com/Robinlovelace/geocompr/archive/master.zip
[17] https://github.com/Robinlovelace/geocompr/
[18] https://github.com/Robinlovelace/geocompr/blob/master/geocompr.Rproj
[19] https://github.com/Robinlovelace/geocompr/raw/master/cite-geocompr.bib
[20] https://geocompr.robinlovelace.net/
[21] https://help.github.com/articles/about-stars/
[22] https://github.com/robinlovelace/geocompr
[23] https://www.goodreads.com/book/show/42780859-geocomputation-with-r

Further details can be found at github.com/Robinlovelace/geocompr[24].

Foreword

Doing 'spatial' in R has always been about being broad, seeking to provide and integrate tools from geography, geoinformatics, geocomputation and spatial statistics for anyone interested in joining in: joining in asking interesting questions, contributing fruitful research questions, and writing and improving code. That is, doing 'spatial' in R has always included open source code, open data and reproducibility.

Doing 'spatial' in R has also sought to be open to interaction with many branches of applied spatial data analysis, and also to implement new advances in data representation and methods of analysis to expose them to cross-disciplinary scrutiny. As this book demonstrates, there are often alternative workflows from similar data to similar results, and we may learn from comparisons with how others create and understand their workflows. This includes learning from similar communities around Open Source GIS and complementary languages such as Python, Java and so on.

R's wide range of spatial capabilities would never have evolved without people willing to share what they were creating or adapting. This might include teaching materials, software, research practices (reproducible research, open data), and combinations of these. R users have also benefitted greatly from 'upstream' open source geo libraries such as GDAL, GEOS and PROJ.

This book is a clear example that, if you are curious and willing to join in, you can find things that need doing and that match your aptitudes. With advances in data representation and workflow alternatives, and ever increasing numbers of new users often without applied quantitative command-line exposure, a book of this kind has really been needed. Despite the effort involved, the authors have supported each other in pressing forward to publication.

So, this fresh book is ready to go; its authors have tried it out during many tutorials and workshops, so readers and instructors will be able to benefit from knowing that the contents have been and continue to be tried out on people like them. Engage with the authors and the wider R-spatial community, see value in having more choice in building your workflows and most important, enjoy applying what you learn here to things you care about.

[24] `https://github.com/Robinlovelace/geocompr#geocomputation-with-r`

Roger Bivand
Bergen, September 2018

Preface

Who this book is for

This book is for people who want to analyze, visualize and model geographic data with open source software. It is based on R, a statistical programming language that has powerful data processing, visualization and geospatial capabilities. The book covers a wide range of topics and will be of interest to a wide range of people from many different backgrounds, especially:

- People who have learned spatial analysis skills using a desktop Geographic Information System (GIS) such as QGIS[25], ArcMap[26], GRASS[27] or SAGA[28], who want access to a powerful (geo)statistical and visualization programming language and the benefits of a command-line approach (Sherman 2008):

> With the advent of 'modern' GIS software, most people want to point and click their way through life. That's good, but there is a tremendous amount of flexibility and power waiting for you with the command line.

- Graduate students and researchers from fields specializing in geographic data including Geography, Remote Sensing, Planning, GIS and Geographic Data Science
- Academics and post-graduate students working on projects in fields including Geology, Regional Science, Biology and Ecology, Agricultural Sciences (precision farming), Archaeology, Epidemiology, Transport Modeling, and broadly defined Data Science which require the power and flexibility of R for their research
- Applied researchers and analysts in public, private or third-sector organizations who need the reproducibility, speed and flexibility of a command-line

[25] http://qgis.org/en/site/
[26] http://desktop.arcgis.com/en/arcmap/
[27] https://grass.osgeo.org/
[28] http://www.saga-gis.org/en/index.html

language such as R in applications dealing with spatial data as diverse as Urban and Transport Planning, Logistics, Geo-marketing (store location analysis) and Emergency Planning

The book is designed for intermediate-to-advanced R users interested in geocomputation and R beginners who have prior experience with geographic data. If you are new to both R and geographic data, do not be discouraged: we provide links to further materials and describe the nature of spatial data from a beginner's perspective in Chapter **??** and in links provided below.

How to read this book

The book is divided into three parts:

1. Part I: Foundations, aimed at getting you up-to-speed with geographic data in R.
2. Part II: Extensions, which covers advanced techniques.
3. Part III: Applications, to real-world problems.

The chapters get progressively harder in each so we recommend reading the book in order. A major barrier to geographical analysis in R is its steep learning curve. The chapters in Part I aim to address this by providing reproducible code on simple datasets that should ease the process of getting started.

An important aspect of the book from a teaching/learning perspective is the **exercises** at the end of each chapter. Completing these will develop your skills and equip you with the confidence needed to tackle a range of geospatial problems. Solutions to the exercises, and a number of extended examples, are provided on the book's supporting website, at geocompr.github.io[29].

Impatient readers are welcome to dive straight into the practical examples, starting in Chapter **??**. However, we recommend reading about the wider context of *Geocomputation with R* in Chapter **??** first. If you are new to R, we also recommend learning more about the language before attempting to run the code chunks provided in each chapter (unless you're reading the book for an understanding of the concepts). Fortunately for R beginners R has a supportive community that has developed a wealth of resources that can help. We particularly recommend three tutorials: R for Data Science[30] (Grolemund and Wickham 2016) and Efficient R Programming[31] (Gillespie and Lovelace 2016), especially Chapter 2[32] (on installing and setting-up R/RStudio) and Chapter 10[33] (on learning to learn), and

[29] `https://geocompr.github.io/`
[30] `http://r4ds.had.co.nz/`
[31] `https://csgillespie.github.io/efficientR/`
[32] `https://csgillespie.github.io/efficientR/set-up.html#r-version`
[33] `https://csgillespie.github.io/efficientR/learning.html`

An introduction to R[34] (Venables, Smith, and Team 2017). A good interactive tutorial is DataCamp's Introduction to R[35].

Why R?

Although R has a steep learning curve, the command-line approach advocated in this book can quickly pay off. As you'll learn in subsequent chapters, R is an effective tool for tackling a wide range of geographic data challenges. We expect that, with practice, R will become the program of choice in your geospatial toolbox for many applications. Typing and executing commands at the command-line is, in many cases, faster than pointing-and-clicking around the graphical user interface (GUI) a desktop GIS. For some applications such as Spatial Statistics and modeling R may be the *only* realistic way to get the work done.

As outlined in Section **??**, there are many reasons for using R for geocomputation: R is well-suited to the interactive use required in many geographic data analysis workflows compared with other languages. R excels in the rapidly growing fields of Data Science (which includes data carpentry, statistical learning techniques and data visualization) and Big Data (via efficient interfaces to databases and distributed computing systems). Furthermore R enables a reproducible workflow: sharing scripts underlying your analysis will allow others to build-on your work. To ensure reproducibility in this book we have made its source code available at github.com/Robinlovelace/geocompr[36]. There you will find script files in the `code/` folder that generate figures: when code generating a figure is not provided in the main text of the book, the name of the script file that generated it is provided in the caption (see for example the caption for Figure **??**).

Other languages such as Python, Java and C++ can be used for geocomputation and there are excellent resources for learning geocomputation *without R*, as discussed in Section **??**. None of these provide the unique combination of package ecosystem, statistical capabilities, visualization options, powerful IDEs offered by the R community. Furthermore, by teaching how to use one language (R) in depth, this book will equip you with the concepts and confidence needed to do geocomputation in other languages.

Real-world impact

Geocomputation with R will equip you with knowledge and skills to tackle a wide range of issues, including those with scientific, societal and environmental implications, manifested in geographic data. As described in Section **??**, geocomputation

[34] `http://colinfay.me/intro-to-r/`
[35] `https://www.datacamp.com/courses/free-introduction-to-r`
[36] `https://github.com/Robinlovelace/geocompr#geocomputation-with-r`

is not only about using computers to process geographic data: it is also about real-world impact. If you are interested in the wider context and motivations behind this book, read on; these are covered in Chapter **??**.

Acknowledgements

Many thanks to everyone who contributed directly and indirectly via the code hosting and collaboration site GitHub, including the following people who contributed direct via pull requests: katygregg[37], erstearns[38], eyesofbambi[39], tyluRp[40], marcosci[41], giocomai[42], mdsumner[43], rsbivand[44], pat-s[45], gisma[46], ateucher[47], annakrystalli[48], gavinsimpson[49], Henrik-P[50], Himanshuteli[51], yutannihilation[52], jbixon13[53], katiejolly[54], layik[55], mvl22[56], nickbearman[57], ganes1410[58], richfitz[59], SymbolixAU[60], wdearden[61], yihui[62], chihinl[63]. Special thanks to Marco Sciaini, who not only created the front cover image, but also published the code that generated it (see `frontcover.R` in the book's GitHub repo). Dozens more people contributed online, by raising and commenting on issues, and by providing feedback via social media. The `#geocompr` hashtag will live on!

We would like to thank John Kimmel from CRC Press, who has worked with us over two years to take our ideas from an early book plan into production via four rounds of peer review. The reviewers deserve special mention here: their

[37] `https://github.com/katygregg`
[38] `https://github.com/erstearns`
[39] `https://github.com/eyesofbambi`
[40] `https://github.com/tyluRp`
[41] `https://github.com/marcosci`
[42] `https://github.com/giocomai`
[43] `https://github.com/mdsumner`
[44] `https://github.com/rsbivand`
[45] `https://github.com/pat-s`
[46] `https://github.com/gisma`
[47] `https://github.com/ateucher`
[48] `https://github.com/annakrystalli`
[49] `https://github.com/gavinsimpson`
[50] `https://github.com/Henrik-P`
[51] `https://github.com/Himanshuteli`
[52] `https://github.com/yutannihilation`
[53] `https://github.com/jbixon13`
[54] `https://github.com/katiejolly`
[55] `https://github.com/layik`
[56] `https://github.com/mvl22`
[57] `https://github.com/nickbearman`
[58] `https://github.com/ganes1410`
[59] `https://github.com/richfitz`
[60] `https://github.com/SymbolixAU`
[61] `https://github.com/wdearden`
[62] `https://github.com/yihui`
[63] `https://github.com/chihinl`

detailed feedback and expertise substantially improved the book's structure and content.

We thank Patrick Schratz and Alexander Brenning from the University of Jena for fruitful discussions on and input into Chapters **??** and **??**. We thank Emmanuel Blondel from the Food and Agriculture Organization of the United Nations for expert input into the section on web services; Michael Sumner for critical input into many areas of the book, especially the discussion of algorithms in Chapter 10; Tim Appelhans and David Cooley for key contributions to the visualization chapter (Chapter 8); and Katy Gregg, who proofread every chapter and greatly improved the readability of the book.

Countless others could be mentioned who contributed in myriad ways. The final thank you is for all the software developers who make geocomputation with R possible. Edzer Pebesma (who created the **sf** package), Robert Hijmans (who created **raster**) and Roger Bivand (who laid the foundations for much R-spatial software) have made high performance geographic computing possible in R.

0.1 Introduction

This book is about using the power of computers to *do things* with geographic data. It teaches a range of spatial skills, including: reading, writing and manipulating geographic data; making static and interactive maps; applying geocomputation to solve real-world problems; and modeling geographic phenomena. By demonstrating how various geographic operations can be linked, in reproducible 'code chunks' that intersperse the prose, the book also teaches a transparent and thus scientific workflow. Learning how to use the wealth of geospatial tools available from the R command line can be exciting, but creating *new ones* can be truly liberating. Using the command-line driven approach taught throughout, and programming techniques covered in Chapter **??**, can help remove constraints on your creativity imposed by software. After reading the book and completing the exercises, you should therefore feel empowered with a strong understanding of the possibilities opened up by R's impressive geographic capabilities, new skills to solve real-world problems with geographic data, and the ability to communicate your work with maps and reproducible code.

Over the last few decades free and open source software for geospatial (FOSS4G) has progressed at an astonishing rate. Thanks to organizations such as

TABLE 0.1: Differences in emphasis between software packages (Graphical User Interface (GUI) of Geographic Information Systems (GIS) and R).

Attribute	Desktop GIS (GUI)	R
Home disciplines	Geography	Computing, Statistics
Software focus	Graphical User Interface	Command line
Reproducibility	Minimal	Maximal

OSGeo[64], geographic data analysis is no longer the preserve of those with expensive hardware and software: anyone can now download and run high-performance spatial libraries. Open source Geographic Information Systems (GIS), such as QGIS[65], have made geographic analysis accessible worldwide. GIS programs tend to emphasize graphical user interfaces (GUIs), with the unintended consequence of discouraging reproducibility (although many can be used from the command line as we'll see in Chapter **??**). R, by contrast, emphasizes the command line interface (CLI). A simplistic comparison between the different approaches is illustrated in Table **??**.

This book is motivated by the importance of reproducibility for scientific research (see the note below). It aims to make reproducible geographic data analysis workflows more accessible, and demonstrate the power of open geospatial software available from the command-line. "Interfaces to other software are part of R" (Eddelbuettel and Balamuta 2018). This means that in addition to outstanding 'in house' capabilities, R allows access to many other spatial software libraries, explained in Section **??** and demonstrated in Chapter **??**. Before going into the details of the software, however, it is worth taking a step back and thinking about what we mean by geocomputation.

Reproducibility is a major advantage of command-line interfaces, but what does it mean in practice? We define it as follows: "A process in which the same results can be generated by others using publicly accessible code."

This may sound simple and easy to achieve (which it is if you carefully maintain your R code in script files), but has profound implications for teaching and the scientific process (Pebesma, Nüst, and Bivand 2012).

[64] `https://www.osgeo.org/`
[65] `http://qgis.org/en/site/`

0.1.1 What is geocomputation?

Geocomputation is a young term, dating back to the first conference on the subject in 1996.[66] What distinguished geocomputation from the (at the time) commonly used term 'quantitative geography', its early advocates proposed, was its emphasis on "creative and experimental" applications (Longley et al. 1998) and the development of new tools and methods (Openshaw and Abrahart 2000): "GeoComputation is about using the various different types of geodata and about developing relevant geo-tools within the overall context of a 'scientific' approach." This book aims to go beyond teaching methods and code; by the end of it you should be able to use your geocomputational skills, to do "practical work that is beneficial or useful" (Openshaw and Abrahart 2000).

Our approach differs from early adopters such as Stan Openshaw, however, in its emphasis on reproducibility and collaboration. At the turn of the 21st Century, it was unrealistic to expect readers to be able to reproduce code examples, due to barriers preventing access to the necessary hardware, software and data. Fast-forward two decades and things have progressed rapidly. Anyone with access to a laptop with ~4GB RAM can realistically expect to be able to install and run software for geocomputation on publicly accessible datasets, which are more widely available than ever before (as we will see in Chapter **??**).[67] Unlike early works in the field, all the work presented in this book is reproducible using code and example data supplied alongside the book, in R packages such as **spData**, the installation of which is covered in Chapter **??**.

Geocomputation is closely related to other terms including: Geographic Information Science (GIScience); Geomatics; Geoinformatics; Spatial Information Science; Geoinformation Engineering (Longley 2015); and Geographic Data Science (GDS). Each term shares an emphasis on a 'scientific' (implying reproducible and falsifiable) approach influenced by GIS, although their origins and main fields of application differ. GDS, for example, emphasizes 'data science' skills and large datasets, while Geoinformatics tends to focus on data structures. But the overlaps between the terms are larger than the differences between them and we use geocomputation as a rough synonym encapsulating all of them: they all seek to use geographic data for applied scientific work. Unlike early users of the term, however, we do not seek to imply that there is any cohesive academic field called

[66] The conference took place at the University of Leeds, where one of the authors (Robin) is currently based. The 21st GeoComputation conference was also hosted at the University of Leeds, during which Robin and Jakub presented, led a workshop on 'tidy' spatial data analysis and collaborated on the book (see www.geocomputation.org for more on the conference series, and papers/presentations spanning two decades).

[67] A laptop with 4GB running a modern operating system such as Ubuntu 16.04 onward should also be able to reproduce the contents of this book. A laptop with this specification or above can be acquired second-hand for ~US$100 in many countries nowadays, reducing the financial/hardware barrier to geocomputation far below the levels in operation in the early 2000s, when high-performance computers were unaffordable for most people.

'Geocomputation' (or 'GeoComputation' as Stan Openshaw called it). Instead, we define the term as follows: working with geographic data in a computational way, focusing on code, reproducibility and modularity.

Geocomputation is a recent term but is influenced by old ideas. It can be seen as a part of Geography, which has a 2000+ year history (Talbert 2014); and an extension of *Geographic Information Systems* (GIS) (Neteler and Mitasova 2008), which emerged in the 1960s (Coppock and Rhind 1991).

Geography has played an important role in explaining and influencing humanity's relationship with the natural world long before the invention of the computer, however. Alexander von Humboldt's travels to South America in the early 1800s illustrates this role: not only did the resulting observations lay the foundations for the traditions of physical and plant geography, they also paved the way towards policies to protect the natural world (Wulf 2015). This book aims to contribute to the 'Geographic Tradition' (Livingstone 1992) by harnessing the power of modern computers and open source software.

The book's links to older disciplines were reflected in suggested titles for the book: *Geography with R* and *R for GIS*. Each has advantages. The former conveys the message that it comprises much more than just spatial data: non-spatial attribute data are inevitably interwoven with geometry data, and Geography is about more than where something is on the map. The latter communicates that this is a book about using R as a GIS, to perform spatial operations on *geographic data* (Bivand, Pebesma, and Gómez-Rubio 2013). However, the term GIS conveys some connotations (see Table **??**) which simply fail to communicate one of R's greatest strengths: its console-based ability to seamlessly switch between geographic and non-geographic data processing, modeling and visualization tasks. By contrast, the term geocomputation implies reproducible and creative programming. Of course, (geocomputational) algorithms are powerful tools that can become highly complex. However, all algorithms are composed of smaller parts. By teaching you its foundations and underlying structure, we aim to empower you to create your own innovative solutions to geographic data problems.

0.1.2 Why use R for geocomputation?

Early geographers used a variety of tools including barometers, compasses and sextants[68] to advance knowledge about the world (Wulf 2015). It was only with the invention of the marine chronometer[69] in 1761 that it became possible to calculate longitude at sea, enabling ships to take more direct routes.

Nowadays such lack of geographic data is hard to imagine. Every smartphone has a global positioning (GPS) receiver and a multitude of sensors on devices rang-

[68] `https://en.wikipedia.org/wiki/Sextant`
[69] `https://en.wikipedia.org/wiki/Marine_chronometer`

ing from satellites and semi-autonomous vehicles to citizen scientists incessantly measure every part of the world. The rate of data produced is overwhelming. An autonomous vehicle, for example, can generate 100 GB of data per day (The Economist 2016). Remote sensing data from satellites has become too large to analyze the corresponding data with a single computer, leading to initiatives such as OpenEO[70].

This 'geodata revolution' drives demand for high performance computer hardware and efficient, scalable software to handle and extract signal from the noise, to understand and perhaps change the world. Spatial databases enable storage and generation of manageable subsets from the vast geographic data stores, making interfaces for gaining knowledge from them vital tools for the future. R is one such tool, with advanced analysis, modeling and visualization capabilities. In this context the focus of the book is not on the language itself (see Wickham 2014a). Instead we use R as a 'tool for the trade' for understanding the world, similar to Humboldt's use of tools to gain a deep understanding of nature in all its complexity and interconnections (see Wulf 2015). Although programming can seem like a reductionist activity, the aim is to teach geocomputation with R not only for fun, but for understanding the world.

R is a multi-platform, open source language and environment for statistical computing and graphics (r-project.org/[71]). With a wide range of packages, R also supports advanced geospatial statistics, modeling and visualization. New integrated development environments (IDEs) such as RStudio have made R more user-friendly for many, easing map making with a panel dedicated to interactive visualization.

At its core, R is an object-oriented, functional programming language[72] (Wickham 2014a), and was specifically designed as an interactive interface to other software (Chambers 2016). The latter also includes many 'bridges' to a treasure trove of GIS software, 'geolibraries' and functions (see Chapter **??**). It is thus ideal for quickly creating 'geo-tools', without needing to master lower level languages (compared to R) such as C, FORTRAN or Java (see Section **??**). This can feel like breaking free from the metaphorical 'glass ceiling' imposed by GUI-based or proprietary geographic information systems (see Table **??** for a definition of GUI). Furthermore, R facilitates access to other languages: the packages **Rcpp** and **reticulate** enable access to C++ and Python code, for example. This means R can be used as a 'bridge' to a wide range of geospatial programs (see Section **??**).

Another example showing R's flexibility and evolving geographic capabilities is interactive map making. As we'll see in Chapter **??**, the statement that R

[70] `http://r-spatial.org/2016/11/29/openeo.html`
[71] `https://www.r-project.org/`
[72] `http://adv-r.had.co.nz/Functional-programming.html`

has "limited interactive [plotting] facilities" (Bivand, Pebesma, and Gómez-Rubio 2013) is no longer true. This is demonstrated by the following code chunk, which creates Figure **??** (the functions that generate the plot are covered in Section **??**).

```
library(leaflet)
popup = c("Robin", "Jakub", "Jannes")
leaflet() %>%
  addProviderTiles("NASAGIBS.ViirsEarthAtNight2012") %>%
  addMarkers(lng = c(-3, 23, 11),
             lat = c(52, 53, 49),
             popup = popup)
```

\begin{figure}[t]

{

It would have been difficult to produce Figure **??** using R a few years ago, let alone as an interactive map. This illustrates R's flexibility and how, thanks to developments such as **knitr** and **leaflet**, it can be used as an interface to other software, a theme that will recur throughout this book. The use of R code, therefore, enables teaching geocomputation with reference to reproducible examples such as that provided in Figure **??** rather than abstract concepts.

0.1.3 Software for geocomputation

R is a powerful language for geocomputation but there are many other options for geographic data analysis providing thousands of geographic functions. Awareness of other languages for geocomputation will help decide when a different tool may be more appropriate for a specific task, and place R in the wider geospatial ecosystem. This section briefly introduces the languages C++[73], Java[74] and Python[75] for geocomputation, in preparation for Chapter **??**.

An important feature of R (and Python) is that it is an interpreted language. This is advantageous because it enables interactive programming in a Read–Eval–Print Loop (REPL): code entered into the console is immediately executed and the result is printed, rather than waiting for the intermediate stage of compilation. On the other hand, compiled languages such as C++ and Java tend to run faster (once they have been compiled).

C++ provides the basis for many GIS packages such as QGIS[76], GRASS[77] and SAGA[78] so it is a sensible starting point. Well-written C++ is very fast, making it a good choice for performance-critical applications such as processing large geographic datasets, but is harder to learn than Python or R. C++ has become more accessible with the **Rcpp** package, which provides a good 'way in' to C programming for R users. Proficiency with such low-level languages opens the possibility of creating new, high-performance 'geoalgorithms' and a better understanding of how GIS software works (see Chapter **??**).

Java is another important and versatile language for geocomputation. GIS packages gvSig[79], OpenJump[80] and uDig[81] are all written in Java. There are many GIS libraries written in Java, including GeoTools[82] and JTS, the Java Topology Suite[83] (GEOS is a C++ port of JTS). Furthermore, many map server applications use Java including Geoserver/Geonode[84], deegree[85] and 52°North WPS[86].

Java's object-oriented syntax is similar to that of C++. A major advantage of Java is that it is platform-independent (which is unusual for a compiled

[73] `https://isocpp.org/`
[74] `https://www.oracle.com/java/index.html`
[75] `https://www.python.org/`
[76] `www.qgis.org`
[77] `https://grass.osgeo.org/`
[78] `www.saga-gis.org`
[79] `http://www.gvsig.com/en/products/gvsig-desktop`
[80] `http://openjump.org/`
[81] `http://udig.refractions.net/`
[82] `http://docs.geotools.org/`
[83] `https://www.locationtech.org/projects/technology.jts`
[84] `http://geonode.org/`
[85] `http://www.deegree.org/`
[86] `http://52north.org/communities/geoprocessing/wps/`

language) and is highly scalable, making it a suitable language for IDEs such as RStudio, with which this book was written. Java has fewer tools for statistical modeling and visualization than Python or R, although it can be used for data science (Brzustowicz 2017).

Python is an important language for geocomputation especially because many Desktop GIS such as GRASS, SAGA and QGIS provide a Python API (see Chapter **??**). Like R, it is a popular[87] tool for data science. Both languages are object-oriented, and have many areas of overlap, leading to initiatives such as the **reticulate** package that facilitates access to Python from R and the Ursa Labs[88] initiative to support portable libraries to the benefit of the entire open source data science ecosystem.

In practice both R and Python have their strengths and to some extent which you use is less important than the domain of application and communication of results. Learning either will provide a head-start in learning the other. However, there are major advantages of R over Python for geocomputation. This includes its much better support of the geographic data models vector and raster in the language itself (see Chapter **??**) and corresponding visualization possibilities (see Chapters **??** and **??**). Equally important, R has unparalleled support for statistics, including spatial statistics, with hundreds of packages (unmatched by Python) supporting thousands of statistical methods.

The major advantage of Python is that it is a *general-purpose* programming language. It is used in many domains, including desktop software, computer games, websites and data science. Python is often the only shared language between different (geocomputation) communities and can be seen as the 'glue' that holds many GIS programs together. Many geoalgorithms, including those in QGIS and ArcMap, can be accessed from the Python command line, making it well-suited as a starter language for command-line GIS.[89]

For spatial statistics and predictive modeling, however, R is second-to-none. This does not mean you must choose either R or Python: Python supports most common statistical techniques (though R tends to support new developments in spatial statistics earlier) and many concepts learned from Python can be applied to the R world. Like R, Python also supports geographic data analysis and manipulation with packages such as **osgeo**, **Shapely**, **NumPy** and **PyGeoProcessing** (Garrard 2016).

[87] `https://stackoverflow.blog/2017/10/10/impressive-growth-r/`

[88] `https://ursalabs.org/`

[89] Python modules providing access to geoalgorithms include `grass.script` for GRASS, `saga-python` for SAGA-GIS, `processing` for QGIS and `arcpy` for ArcGIS.

0.1.4 R's spatial ecosystem

There are many ways to handle geographic data in R, with dozens of packages in the area.[90] In this book we endeavor to teach the state-of-the-art in the field whilst ensuring that the methods are future-proof. Like many areas of software development, R's spatial ecosystem is rapidly evolving (Figure **??**). Because R is open source, these developments can easily build on previous work, by 'standing on the shoulders of giants', as Isaac Newton put it in 1675[91]. This approach is advantageous because it encourages collaboration and avoids 'reinventing the wheel'. The package **sf** (covered in Chapter **??**), for example, builds on its predecessor **sp**.

A surge in development time (and interest) in 'R-spatial' has followed the award of a grant by the R Consortium for the development of support for Simple Features, an open-source standard and model to store and access vector geometries. This resulted in the **sf** package (covered in Section **??**). Multiple places reflect the immense interest in **sf**. This is especially true for the R-sig-Geo Archives[92], a long-standing open access email list containing much R-spatial wisdom accumulated over the years.

It is noteworthy that shifts in the wider R community, as exemplified by the data processing package **dplyr** (released in 2014[93]) influenced shifts in R's spatial ecosystem. Alongside other packages that have a shared style and emphasis on 'tidy data' (including, e.g., **ggplot2**), **dplyr** was placed in the **tidyverse** 'metapackage' in late 2016[94]. The **tidyverse** approach, with its focus on long-form data and fast intuitively named functions, has become immensely popular. This has led to a demand for 'tidy geographic data' which has been partly met by **sf** and other approaches such as **tabularaster**. An obvious feature of the **tidyverse** is the tendency for packages to work in harmony. There is no equivalent **geoverse**, but there are attempts at harmonization between packages hosted in the r-spatial[95] organization and a growing number of packages use **sf** (Table **??**).

0.1.5 The history of R-spatial

There are many benefits of using recent spatial packages such as **sf**, but it also important to be aware of the history of R's spatial capabilities: many functions,

[90] An overview of R's spatial ecosystem can be found in the CRAN Task View on the Analysis of Spatial Data (see `https://cran.r-project.org/web/views/Spatial.html`).

[91] `http://digitallibrary.hsp.org/index.php/Detail/Object/Show/object_id/9285`

[92] `https://stat.ethz.ch/pipermail/r-sig-geo/`

[93] `https://cran.r-project.org/src/contrib/Archive/dplyr/`

[94] `https://cran.r-project.org/src/contrib/Archive/tidyverse/`

[95] `https://github.com/r-spatial/discuss/issues/11`

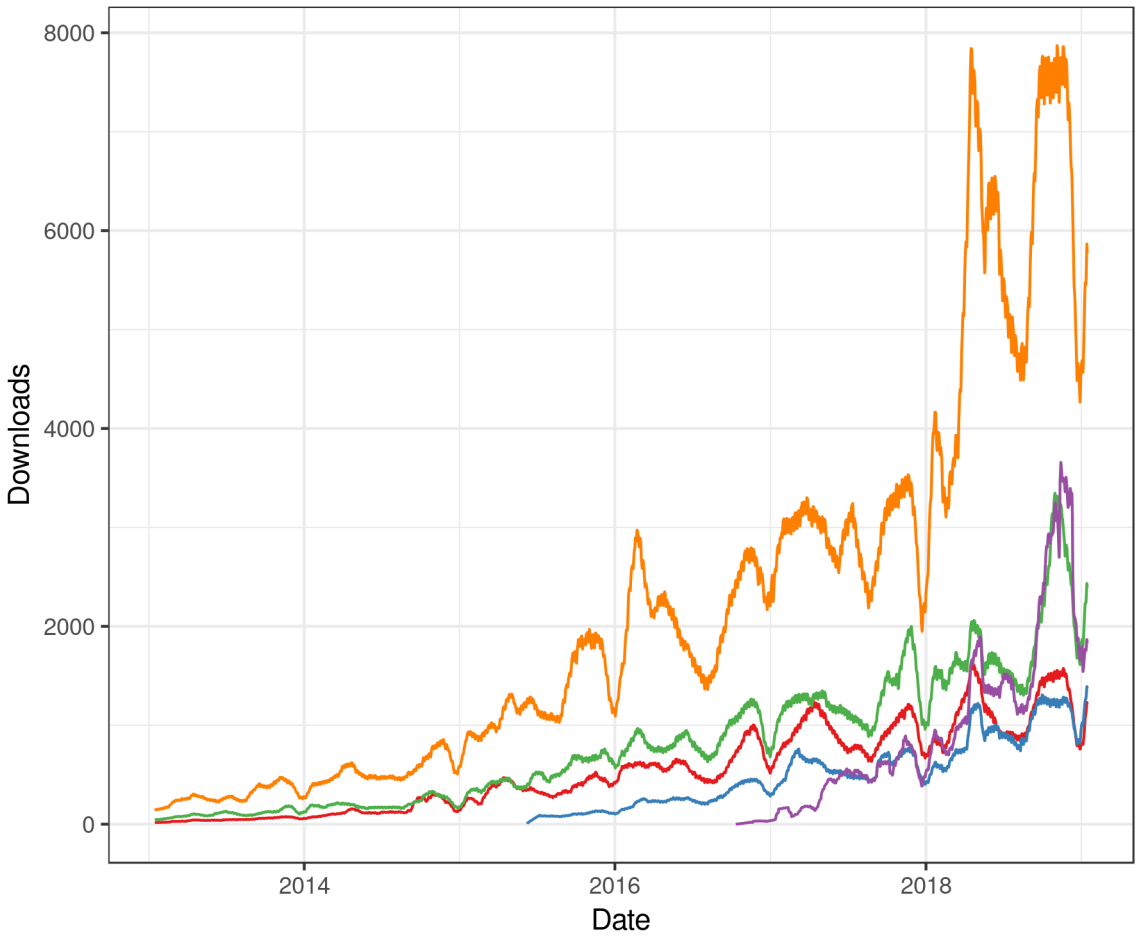


FIGURE 1: The popularity of spatial packages in R. The y-axis shows average number of downloads per day, within a 30-day rolling window, of prominent spatial packages.

use-cases and teaching material are contained in older packages. These can still be useful today, provided you know where to look.

R's spatial capabilities originated in early spatial packages in the S language (Bivand and Gebhardt 2000). The 1990s saw the development of numerous S scripts and a handful of packages for spatial statistics. R packages arose from these and by 2000 there were R packages for various spatial methods "point pattern analysis, geostatistics, exploratory spatial data analysis and spatial econometrics", according to an article[96] presented at GeoComputation 2000 (Bivand and Neteler 2000). Some of these, notably **spatial**, **sgeostat** and

[96] `http://www.geocomputation.org/2000/GC009/Gc009.htm`

TABLE 0.2: The top 5 most downloaded packages that depend on sf, in terms of average number of downloads per day over the previous month. As of 2019-01-29 there are 117 packages which import sf.

package	Downloads
ggplot2	22593
plotly	3628
raster	2433
leaflet	1391
spData	1174

splancs are still available on CRAN (Rowlingson and Diggle 1993, 2017; Venables and Ripley 2002; Majure and Gebhardt 2016).

A subsequent article in R News (the predecessor of The R Journal[97]) contained an overview of spatial statistical software in R at the time, much of which was based on previous code written for S/S-PLUS (Ripley 2001). This overview described packages for spatial smoothing and interpolation, including **akima** and **geoR** (Akima and Gebhardt 2016; Jr and Diggle 2016), and point pattern analysis, including **splancs** (Rowlingson and Diggle 2017) and **spatstat** (Baddeley, Rubak, and Turner 2015).

The following R News issue (Volume 1/3) put spatial packages in the spotlight again, with a more detailed introduction to **splancs** and a commentary on future prospects regarding spatial statistics (Bivand 2001). Additionally, the issue introduced two packages for testing spatial autocorrelation that eventually became part of **spdep** (Bivand 2017). Notably, the commentary mentions the need for standardization of spatial interfaces, efficient mechanisms for exchanging data with GIS, and handling of spatial metadata such as coordinate reference systems (CRS).

maptools (written by Nicholas Lewin-Koh; Bivand and Lewin-Koh 2017) is another important package from this time. Initially **maptools** just contained a wrapper around shapelib[98] and permitted the reading of ESRI Shapefiles into geometry nested lists. The corresponding and nowadays obsolete S3 class called "Map" stored this list alongside an attribute data frame. The work on the "Map" class representation was nevertheless important since it directly fed into **sp** prior to its publication on CRAN.

In 2003 Roger Bivand published an extended review of spatial packages. It proposed a class system to support the "data objects offered by GDAL", including 'fundamental' point, line, polygon, and raster types. Furthermore, it suggested interfaces to external libraries should form the basis of modular R

[97] `https://journal.r-project.org/`
[98] `http://shapelib.maptools.org/`

packages (Bivand 2003). To a large extent these ideas were realized in the packages **rgdal** and **sp**. These provided a foundation for spatial data analysis with R, as described in *Applied Spatial Data Analysis with R* (ASDAR) (Bivand, Pebesma, and Gómez-Rubio 2013), first published in 2008. Ten years later, R's spatial capabilities have evolved substantially but they still build on ideas set-out by Bivand (2003): interfaces to GDAL and PROJ, for example, still power R's high-performance geographic data I/O and CRS transformation capabilities (see Chapters **??** and **??**, respectively).

rgdal, released in 2003, provided GDAL bindings for R which greatly enhanced its ability to import data from previously unavailable geographic data formats. The initial release supported only raster drivers but subsequent enhancements provided support for coordinate reference systems (via the PROJ library), reprojections and import of vector file formats (see Chapter **??** for more on file formats). Many of these additional capabilities were developed by Barry Rowlingson and released in the **rgdal** codebase in 2006 (see Rowlingson et al. 2003 and the R-help[99] email list for context).

sp, released in 2005, overcame R's inability to distinguish spatial and non-spatial objects (Pebesma and Bivand 2005). **sp** grew from a workshop[100] in Vienna in 2003 and was hosted at sourceforge before migrating to R-Forge[101]. Prior to 2005, geographic coordinates were generally treated like any other number. **sp** changed this with its classes and generic methods supporting points, lines, polygons and grids, and attribute data.

sp stores information such as bounding box, coordinate reference system and attributes in slots in `Spatial` objects using the S4 class system, enabling data operations to work on geographic data (see Section **??**). Further, **sp** provides generic methods such as `summary()` and `plot()` for geographic data. In the following decade, **sp** classes rapidly became popular for geographic data in R and the number of packages that depended on it increased from around 20 in 2008 to over 100 in 2013 (Bivand, Pebesma, and Gómez-Rubio 2013). As of 2018 almost 500 packages rely on **sp**, making it an important part of the R ecosystem. Prominent R packages using **sp** include: **gstat**, for spatial and spatio-temporal geostatistics; **geosphere**, for spherical trigonometry; and **adehabitat** used for the analysis of habitat selection by animals (E. Pebesma and Graeler 2018; Calenge 2006; Hijmans 2016).

While **rgdal** and **sp** solved many spatial issues, R still lacked the ability to do geometric operations (see Chapter **??**). Colin Rundel addressed this issue by developing **rgeos**, an R interface to the open-source geometry library (GEOS) during a Google Summer of Code project in 2010 (Bivand and Rundel 2018).

[99] `https://stat.ethz.ch/pipermail/r-help/2003-January/028413.html`
[100] `http://spatial.nhh.no/meetings/vienna/index.html`
[101] `https://r-forge.r-project.org`

rgeos enabled GEOS to manipulate **sp** objects, with functions such as `gIntersection()`.
Another limitation of **sp** — its limited support for raster data — was overcome by **raster**, first released in 2010 (Hijmans 2017). Its class system and functions support a range of raster operations as outlined in Section **??**. A key feature of **raster** is its ability to work with datasets that are too large to fit into RAM (R's interface to PostGIS supports off-disc operations on vector geographic data). **raster** also supports map algebra (see Section **??**).
In parallel with these developments of class systems and methods came the support for R as an interface to dedicated GIS software. **GRASS** (R. S. Bivand 2000) and follow-on packages **spgrass6** and **rgrass7** (for GRASS GIS 6 and 7, respectively) were prominent examples in this direction (Bivand 2016a, 2016b). Other examples of bridges between R and GIS include **RSAGA** (Brenning, Bangs, and Becker 2018, first published in 2008), **RPyGeo** (Brenning 2012a, first published in 2008), and **RQGIS** (Muenchow, Schratz, and Brenning 2017, first published in 2016) (see Chapter **??**).
Visualization was not a focus initially, with the bulk of R-spatial development focused on analysis and geographic operations. **sp** provided methods for map making using both the base and lattice plotting system but demand was growing for advanced map making capabilities, especially after the release of **ggplot2** in 2007. **ggmap** extended **ggplot2**'s spatial capabilities (Kahle and Wickham 2013), by facilitating access to 'basemap' tiles from online services such as Google Maps. Though **ggmap** facilitated map-making with **ggplot2**, its utility was limited by the need to `fortify` spatial objects, which means converting them into long data frames. While this works well for points it is computationally inefficient for lines and polygons, since each coordinate (vertex) is converted into a row, leading to huge data frames to represent complex geometries. Although geographic visualization tended to focus on vector data, raster visualization is supported in **raster** and received a boost with the release of **rasterVis**, which is described in a book on the subject of spatial and temporal data visualization (Lamigueiro 2018). As of 2018 map making in R is a hot topic with dedicated packages such as **tmap**, **leaflet** and **mapview** all supporting the class system provided by **sf**, the focus of the next chapter (see Chapter **??** for more on visualization).

0.1.6 Exercises

1. Think about the terms 'GIS', 'GDS' and 'geocomputation' described above. Which (if any) best describes the work you would like to do using geo* methods and software and why?
2. Provide three reasons for using a scriptable language such as R for

geocomputation instead of using an established GIS program such as QGIS.

3. Name two advantages and two disadvantages of using mature vs recent packages for geographic data analysis (for example **sp** vs **sf**).

Part I

Foundations

0.2 Geographic data in R

Prerequisites

This is the first practical chapter of the book, and therefore it comes with some software requirements. We assume that you have an up-to-date version of R installed and that you are comfortable using software with a command-line interface such as the integrated development environment (IDE) RStudio.

If you are new to R, we recommend reading Chapter 2 of the online book *Efficient R Programming* by Gillespie and Lovelace (2016) and learning the basics of the language with reference to resources such as Grolemund and Wickham (2016) or DataCamp[102] before proceeding. Organize your work (e.g., with RStudio projects) and give scripts sensible names such as `chapter-02.R` to document the code you write as you learn.

The packages used in this chapter can be installed with the following commands:[103]

```
install.packages("sf")
install.packages("raster")
install.packages("spData")
devtools::install_github("Nowosad/spDataLarge")
```

If you're running Mac or Linux, the previous command to install **sf** may not work first time. These operating systems (OSs) have 'systems requirements' that are described in the package's README[104]. Various OS-specific instructions can be found online, such as the article *Installation of R 3.5 on Ubuntu 18.04* on the blog rtask.thinkr.fr[105].

All the packages needed to reproduce the contents of the book can be installed with the following command: `devtools::install_github("geocompr/geocompkg")`. The necessary packages can be 'loaded' (technically they are attached) with the `library()` function as follows:

[102] `https://www.datacamp.com/courses/free-introduction-to-r`

[103] **spDataLarge** is not on CRAN, meaning it must be installed via **devtools** or with the following command: `install.packages("spDataLarge", repos = "https://nowosad.github.io/drat/", type = "source")`.

```
library(sf)          # classes and functions for vector data
library(raster)      # classes and functions for raster data
```

The other packages that were installed contain data that will be used in the book:

```
library(spData)        # load geographic data
library(spDataLarge)   # load larger geographic data
```

0.2.1 Introduction

This chapter will provide brief explanations of the fundamental geographic data models: vector and raster. We will introduce the theory behind each data model and the disciplines in which they predominate, before demonstrating their implementation in R.

The *vector data model* represents the world using points, lines and polygons. These have discrete, well-defined borders, meaning that vector datasets usually have a high level of precision (but not necessarily accuracy as we will see in Section **??**). The *raster data model* divides the surface up into cells of constant size. Raster datasets are the basis of background images used in web-mapping and have been a vital source of geographic data since the origins of aerial photography and satellite-based remote sensing devices. Rasters aggregate spatially specific features to a given resolution, meaning that they are consistent over space and scalable (many worldwide raster datasets are available).

Which to use? The answer likely depends on your domain of application:

- Vector data tends to dominate the social sciences because human settlements tend to have discrete borders.
- Raster often dominates in environmental sciences because of the reliance on remote sensing data.

There is much overlap in some fields and raster and vector datasets can be used together: ecologists and demographers, for example, commonly use both vector and raster data. Furthermore, it is possible to convert between the two forms (see Section **??**). Whether your work involves more use of vector or raster datasets, it is worth understanding the underlying data model before using them, as discussed in subsequent chapters. This book uses **sf** and **raster** packages to work with vector data and raster datasets, respectively.

0.2.2 Vector data

Take care when using the word 'vector' as it can have two meanings in this book: geographic vector data and the `vector` class (note the `monospace` font) in R. The former is a data model, the latter is an R class just like `data.frame` and `matrix`. Still, there is a link between the two: the spatial coordinates which are at the heart of the geographic vector data model can be represented in R using `vector` objects.

The geographic vector model is based on points located within a coordinate reference system (CRS). Points can represent self-standing features (e.g., the location of a bus stop) or they can be linked together to form more complex geometries such as lines and polygons. Most point geometries contain only two dimensions (3-dimensional CRSs contain an additional z value, typically representing height above sea level).

In this system London, for example, can be represented by the coordinates `c(-0.1, 51.5)`. This means that its location is -0.1 degrees east and 51.5 degrees north of the origin. The origin in this case is at 0 degrees longitude (the Prime Meridian) and 0 degree latitude (the Equator) in a geographic ('lon/lat') CRS (Figure **??**, left panel). The same point could also be approximated in a projected CRS with 'Easting/Northing' values of `c(530000, 180000)` in the British National Grid[106], meaning that London is located 530 km *East* and 180 km *North* of the *origin* of the CRS. This can be verified visually: slightly more than 5 'boxes' — square areas bounded by the gray grid lines 100 km in width — separate the point representing London from the origin (Figure **??**, right panel).

The location of National Grid's origin, in the sea beyond South West Peninsular, ensures that most locations in the UK have positive Easting and Northing values.[107] There is more to CRSs, as described in Sections **??** and **??** but, for the purposes of this section, it is sufficient to know that coordinates consist of two numbers representing distance from an origin, usually in x then y dimensions.

sf is a package providing a class system for geographic vector data. Not only does **sf** supersede **sp**, it also provides a consistent command-line interface to GEOS and GDAL, superseding **rgeos** and **rgdal** (described in Section **??**). This section introduces **sf** classes in preparation for subsequent chapters (Chapters **??** and **??** cover the GEOS and GDAL interface, respectively).

[106] `https://en.wikipedia.org/wiki/Ordnance_Survey_National_Grid`

[107] The origin we are referring to, depicted in blue in Figure **??**, is in fact the 'false' origin. The 'true' origin, the location at which distortions are at a minimum, is located at 2° W and 49° N. This was selected by the Ordnance Survey to be roughly in the center of the British landmass longitudinally.

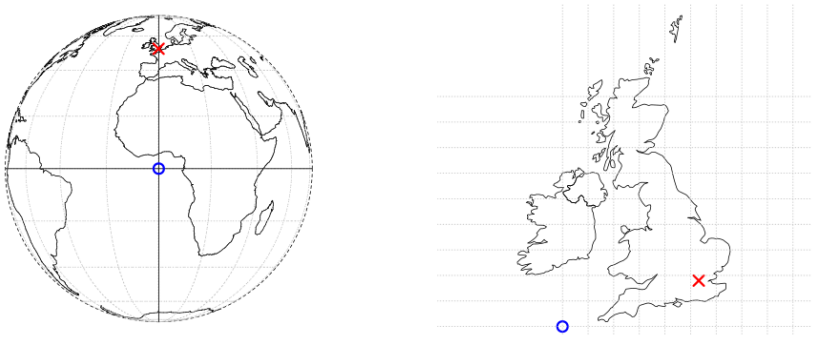


FIGURE 2: Illustration of vector (point) data in which location of London (the red X) is represented with reference to an origin (the blue circle). The left plot represents a geographic CRS with an origin at 0° longitude and latitude. The right plot represents a projected CRS with an origin located in the sea west of the South West Peninsula.

0.2.2.1 An introduction to simple features

Simple features is an open standard[108] developed and endorsed by the Open Geospatial Consortium (OGC), a not-for-profit organization whose activities we will revisit in a later chapter (in Section **??**). Simple Features is a hierarchical data model that represents a wide range of geometry types. Of 17 geometry types supported by the specification, only 7 are used in the vast majority of geographic research (see Figure **??**); these core geometry types are fully supported by the R package **sf** (E. Pebesma 2018).[109]

sf can represent all common vector geometry types (raster data classes are not supported by **sf**): points, lines, polygons and their respective 'multi' versions (which group together features of the same type into a single feature). **sf** also supports geometry collections, which can contain multiple geometry types in a single object. **sf** largely supersedes the **sp** ecosystem, which comprises **sp** (E. Pebesma and Bivand 2018), **rgdal** for data read/write (Bivand, Keitt, and Rowlingson 2018) and **rgeos** for spatial operations (Bivand and Rundel 2018). The package is well documented, as can be seen on its website and in 6 vignettes, which can be loaded as follows:

[108] `http://portal.opengeospatial.org/files/?artifact_id=25355`

[109] The full OGC standard includes rather exotic geometry types including 'surface' and 'curve' geometry types, which currently have limited application in real world applications. All 17 types can be represented with the **sf** package, although (as of summer 2018) plotting only works for the 'core 7'.

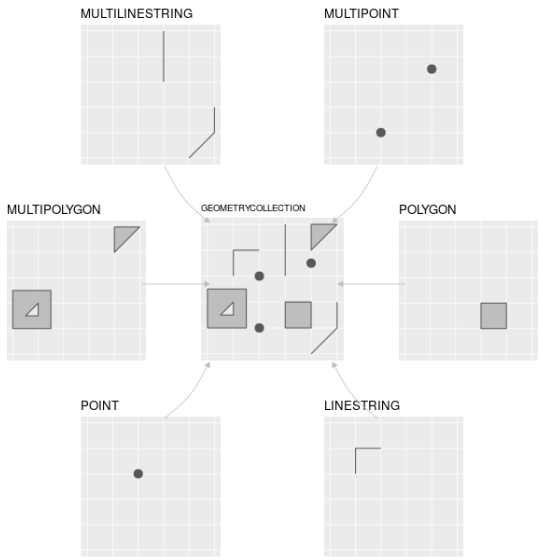


FIGURE 3: Simple feature types fully supported by sf.

```
vignette(package = "sf") # see which vignettes are available
vignette("sf1")          # an introduction to the package
```

As the first vignette explains, simple feature objects in R are stored in a data frame, with geographic data occupying a special column, usually named 'geom' or 'geometry'. We will use the `world` dataset provided by the **spData**, loaded at the beginning of this chapter (see nowosad.github.io/spData[110] for a list of datasets loaded by the package). `world` is a spatial object containing spatial and attribute columns, the names of which are returned by the function `names()` (the last column contains the geographic information):

```
names(world)
#>  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion"
#>  [6] "type"      "area_km2"  "pop"       "lifeExp"   "gdpPercap"
#> [11] "geom"
```

The contents of this `geom` column give `sf` objects their spatial powers: `world$geom` is a 'list column[111]' that contains all the coordinates of the country polygons. The **sf** package provides a `plot()` method for visualizing geographic data: the following command creates Figure **??**.

[110] `https://nowosad.github.io/spData/`
[111] `https://jennybc.github.io/purrr-tutorial/ls13_list-columns.html`

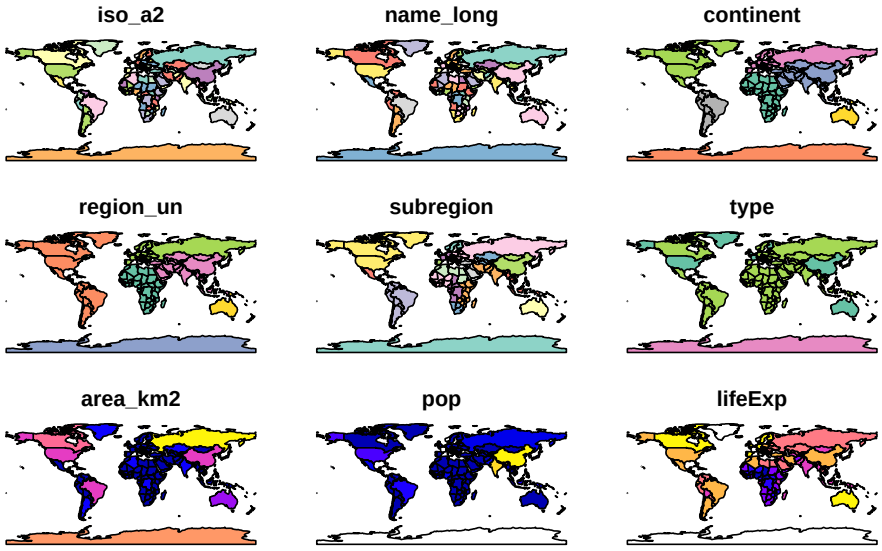


FIGURE 4: A spatial plot of the world using the sf package, with a facet for each attribute.

```
plot(world)
```

Note that instead of creating a single map, as most GIS programs would, the `plot()` command has created multiple maps, one for each variable in the `world` datasets. This behavior can be useful for exploring the spatial distribution of different variables and is discussed further in Section **??** below.

Being able to treat spatial objects as regular data frames with spatial powers has many advantages, especially if you are already used to working with data frames. The commonly used `summary()` function, for example, provides a useful overview of the variables within the `world` object.

```
summary(world["lifeExp"])
#>     lifeExp                   geom
#>  Min.   :50.6   MULTIPOLYGON :177
#>  1st Qu.:65.0   epsg:4326    :  0
#>  Median :72.9   +proj=long...:  0
#>  Mean   :70.9
#>  3rd Qu.:76.8
#>  Max.   :83.6
#>  NA's   :10
```

Although we have only selected one variable for the `summary` command, it also outputs a report on the geometry. This demonstrates the 'sticky' behavior of the geometry columns of **sf** objects, meaning the geometry is kept unless the user deliberately removes them, as we'll see in Section **??**. The result provides a quick summary of both the non-spatial and spatial data contained in `world`: the mean average life expectancy is 71 years (ranging from less than 51 to more than 83 years with a median of 73 years) across all countries.

The word `MULTIPOLYGON` in the summary output above refers to the geometry type of features (countries) in the `world` object. This representation is necessary for countries with islands such as Indonesia and Greece. Other geometry types are described in Section **??**.

It is worth taking a deeper look at the basic behavior and contents of this simple feature object, which can usefully be thought of as a '**s**patial data **f**rame'. `sf` objects are easy to subset. The code below shows its first two rows and three columns. The output shows two major differences compared with a regular `data.frame`: the inclusion of additional geographic data (`geometry type`, `dimension`, `bbox` and CRS information - `epsg (SRID)`, `proj4string`), and the presence of a `geometry` column, here named `geom`:

```
world_mini = world[1:2, 1:3]
world_mini
#> Simple feature collection with 2 features and 3 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -180 ymin: -18.3 xmax: 180 ymax: -0.95
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
#>   iso_a2 name_long continent                              geom
#> 1     FJ      Fiji   Oceania MULTIPOLYGON (((180 -16.1, ...
#> 2     TZ  Tanzania    Africa MULTIPOLYGON (((33.9 -0.95,...
```

All this may seem rather complex, especially for a class system that is supposed to be simple. However, there are good reasons for organizing things this way and using **sf**.

Before describing each geometry type that the **sf** package supports, it is worth taking a step back to understand the building blocks of `sf` objects. Section **??** shows how simple features objects are data frames, with special geometry columns. These spatial columns are often called `geom` or `geometry`: `world$geom`

refers to the spatial element of the `world` object described above. These geometry columns are 'list columns' of class `sfc` (see Section **??**). In turn, `sfc` objects are composed of one or more objects of class `sfg`: simple feature geometries that we describe in Section **??**.
To understand how the spatial components of simple features work, it is vital to understand simple feature geometries. For this reason we cover each currently supported simple features geometry type in Section **??** before moving on to describe how these can be represented in R using `sfg` objects, which form the basis of `sfc` and eventually full `sf` objects.

The preceding code chunk uses `=` to create a new object called `world_mini` in the command `world_mini = world[1:2, 1:3]`. This is called assignment. An equivalent command to achieve the same result is `world_mini <- world[1:2, 1:3]`. Although 'arrow assigment' is more commonly used, we use 'equals assignment' because it's slightly faster to type and easier to teach due to compatibility with commonly used languages such as Python and JavaScript. Which to use is largely a matter of preference as long as you're consistent (packages such as **styler** can be used to change style).

0.2.2.2 Why simple features?

Simple features is a widely supported data model that underlies data structures in many GIS applications including QGIS and PostGIS. A major advantage of this is that using the data model ensures your work is cross-transferable to other set-ups, for example importing from and exporting to spatial databases. A more specific question from an R perspective is "why use the **sf** package when **sp** is already tried and tested"? There are many reasons (linked to the advantages of the simple features model) including:

- Fast reading and writing of data.
- Enhanced plotting performance.
- **sf** objects can be treated as data frames in most operations.
- **sf** functions can be combined using `%>%` operator and works well with the tidyverse[112] collection of R packages.
- **sf** function names are relatively consistent and intuitive (all begin with `st_`).

Due to such advantages, some spatial packages (including **tmap**, **mapview** and **tidycensus**) have added support for **sf**. However, it will take many years for most packages to transition and some will never switch. Fortunately, these can still be used in a workflow based on `sf` objects, by converting them to the `Spatial` class used in **sp**:

[112] `http://tidyverse.org/`

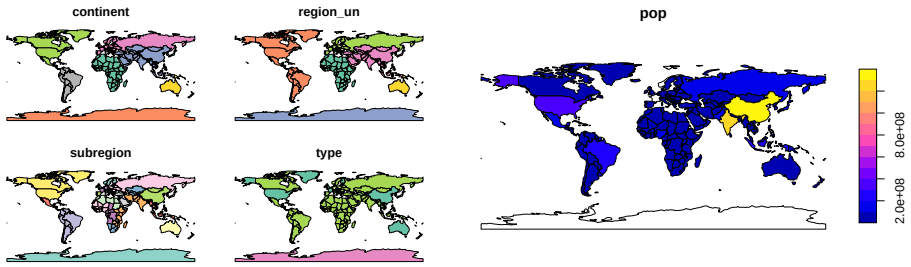


FIGURE 5: Plotting with sf, with multiple variables (left) and a single variable (right).

```
library(sp)
world_sp = as(world, Class = "Spatial")
# sp functions ...
```

`Spatial` objects can be converted back to `sf` in the same way or with `st_as_sf()`:

```
world_sf = st_as_sf(world_sp, "sf")
```

0.2.2.3 Basic map making

Basic maps are created in **sf** with `plot()`. By default this creates a multi-panel plot (like **sp**'s `spplot()`), one sub-plot for each variable of the object, as illustrated in the left-hand panel in Figure **??**. A legend or 'key' with a continuous color is produced if the object to be plotted has a single variable (see the right-hand panel). Colors can also be set with `col =`, although this will not create a continuous palette or a legend.

```
plot(world[3:6])
plot(world["pop"])
```

Plots are added as layers to existing images by setting `add = TRUE`.[113] To demonstrate this, and to provide a taster of content covered in Chapters **??** and **??** on attribute and spatial data operations, the subsequent code chunk combines countries in Asia:

[113] `plot()`ing of **sf** objects uses `sf:::plot.sf()` behind the scenes. `plot()` is a generic method that behaves differently depending on the class of object being plotted.

xl

```
world_asia = world[world$continent == "Asia", ]
asia = st_union(world_asia)
```

We can now plot the Asian continent over a map of the world. Note that the first plot must only have one facet for `add = TRUE` to work. If the first plot has a key, `reset = FALSE` must be used (result not shown):

```
plot(world["pop"], reset = FALSE)
plot(asia, add = TRUE, col = "red")
```

Adding layers in this way can be used to verify the geographic correspondence between layers: the `plot()` function is fast to execute and requires few lines of code, but does not create interactive maps with a wide range of options. For more advanced map making we recommend using dedicated visualization packages such as **tmap** (see Chapter **??**).

0.2.2.4 Base plot arguments

There are various ways to modify maps with **sf**'s `plot()` method. Because **sf** extends base R plotting methods `plot()`'s arguments such as `main =` (which specifies the title of the map) work with `sf` objects (see `?graphics::plot` and `?par`).[114]

Figure **??** illustrates this flexibility by overlaying circles, whose diameters (set with `cex =`) represent country populations, on a map of the world. A basic version of the map can be created with the following commands (see exercises at the end of this chapter and the script `02-contplot.R`[115] to create Figure **??**):

```
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
world_cents = st_centroid(world, of_largest = TRUE)
plot(st_geometry(world_cents), add = TRUE, cex = cex)
```

The code above uses the function `st_centroid()` to convert one geometry type (polygons) to another (points) (see Chapter **??**), the aesthetics of which are varied with the `cex` argument.

sf's plot method also has arguments specific to geographic data. `expandBB`, for example, can be used plot an `sf` object in context: it takes a numeric vector of

[114] Note: many plot arguments are ignored in facet maps, when more than one `sf` column is plotted.

[115] `https://github.com/Robinlovelace/geocompr/blob/master/code/02-contpop.R`

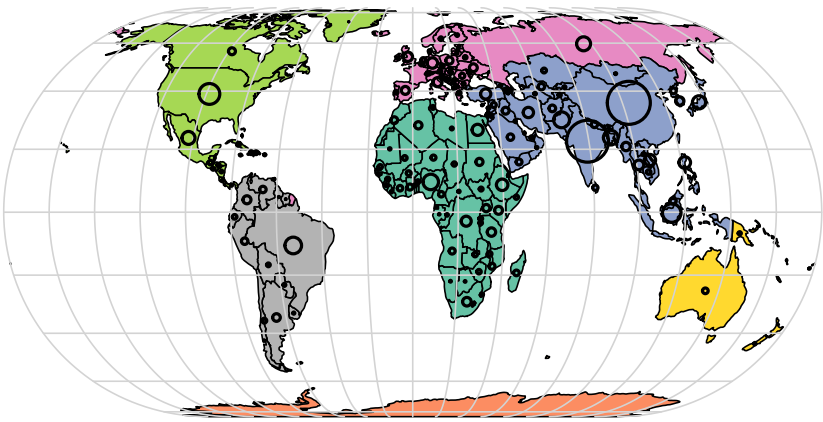


FIGURE 6: Country continents (represented by fill color) and 2015 populations (represented by circles, with area proportional to population).

FIGURE 7: India in context, demonstrating the expandBB argument.

length four that expands the bounding box of the plot relative to zero in the following order: bottom, left, top, right. This is used to plot India in the context of its giant Asian neighbors, with an emphasis on China to the east, in the following code chunk, which generates Figure **??** (see exercises below on adding text to plots):

```
india = world[world$name_long == "India", ]
plot(st_geometry(india), expandBB = c(0, 0.2, 0.1, 1), col = "gray", lwd = 3)
plot(world_asia[0], add = TRUE)
```

Note the use of `[0]` to keep only the geometry column and `lwd` to emphasize

India. See Section **??** for other visualization techniques for representing a range of geometry types, the subject of the next section.

0.2.2.5 Geometry types

Geometries are the basic building blocks of simple features. Simple features in R can take on one of the 17 geometry types supported by the **sf** package. In this chapter we will focus on the seven most commonly used types: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`. Find the whole list of possible feature types in the PostGIS manual[116].

Generally, well-known binary (WKB) or well-known text (WKT) are the standard encoding for simple feature geometries. WKB representations are usually hexadecimal strings easily readable for computers. This is why GIS and spatial databases use WKB to transfer and store geometry objects. WKT, on the other hand, is a human-readable text markup description of simple features. Both formats are exchangeable, and if we present one, we will naturally choose the WKT representation.

The basis for each geometry type is the point. A point is simply a coordinate in 2D, 3D or 4D space (see `vignette("sf1")` for more information) such as (see left panel in Figure **??**):

- `POINT (5 2)`

A linestring is a sequence of points with a straight line connecting the points, for example (see middle panel in Figure **??**):

- `LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)`

A polygon is a sequence of points that form a closed, non-intersecting ring. Closed means that the first and the last point of a polygon have the same coordinates (see right panel in Figure **??**).[117]

- Polygon without a hole: `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`

So far we have created geometries with only one geometric entity per feature. However, **sf** also allows multiple geometries to exist within a single feature (hence the term 'geometry collection') using "multi" version of each geometry type:

- Multipoint: `MULTIPOINT (5 2, 1 3, 3 4, 3 2)`

[116] `http://postgis.net/docs/using_postgis_dbmanagement.html`

[117] By definition, a polygon has one exterior boundary (outer ring) and can have zero or more interior boundaries (inner rings), also known as holes. A polygon with a hole would be, for example, `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))`

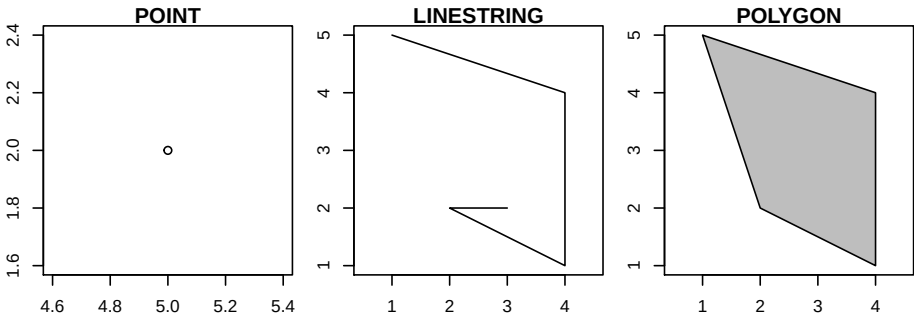


FIGURE 8: Illustration of point, linestring and polygon geometries.

- Multilinestring: `MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))`
- Multipolygon: `MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))`

Finally, a geometry collection can contain any combination of geometries including (multi)points and linestrings (see Figure **??**):

- Geometry collection: `GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))`

0.2.2.6 Simple feature geometries (sfg)

The `sfg` class represents the different simple feature geometry types in R: point, linestring, polygon (and their 'multi' equivalents, such as multipoints) or geometry collection.

Usually you are spared the tedious task of creating geometries on your own since you can simply import an already existing spatial file. However, there are a set of functions to create simple feature geometry objects (`sfg`) from scratch if needed. The names of these functions are simple and consistent, as they all start with the `st_` prefix and end with the name of the geometry type in lowercase letters:

- A point: `st_point()`

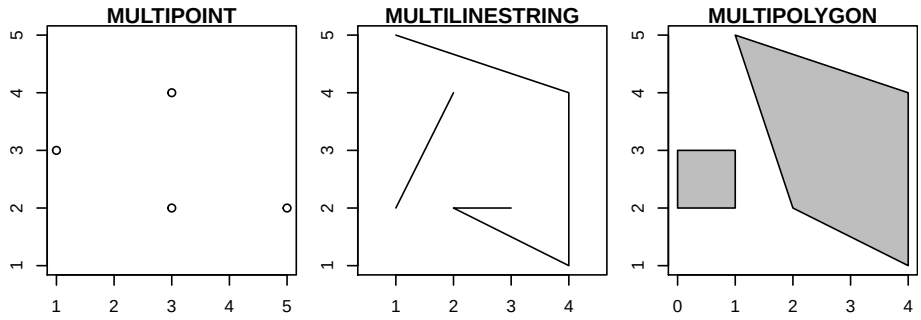


FIGURE 9: Illustration of multi* geometries.

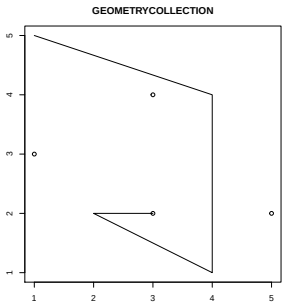


FIGURE 10: Illustration of a geometry collection.

- A linestring: `st_linestring()`
- A polygon: `st_polygon()`
- A multipoint: `st_multipoint()`
- A multilinestring: `st_multilinestring()`
- A multipolygon: `st_multipolygon()`
- A geometry collection: `st_geometrycollection()`

`sfg` objects can be created from three base R data types:

1. A numeric vector: a single point
2. A matrix: a set of points, where each row represents a point, a multipoint or linestring

3. A list: a collection of objects such as matrices, multilinestrings or geometry collections

The function `st_point()` creates single points from numeric vectors:

```
st_point(c(5, 2))                 # XY point
#> POINT (5 2)
st_point(c(5, 2, 3))              # XYZ point
#> POINT Z (5 2 3)
st_point(c(5, 2, 1), dim = "XYM") # XYM point
#> POINT M (5 2 1)
st_point(c(5, 2, 3, 1))           # XYZM point
#> POINT ZM (5 2 3 1)
```

The results show that XY (2D coordinates), XYZ (3D coordinates) and XYZM (3D with an additional variable, typically measurement accuracy) point types are created from vectors of length 2, 3, and 4, respectively. The XYM type must be specified using the `dim` argument (which is short for dimension).
By contrast, use matrices in the case of multipoint (`st_multipoint()`) and linestring (`st_linestring()`) objects:

```
# the rbind function simplifies the creation of matrices
## MULTIPOINT
multipoint_matrix = rbind(c(5, 2), c(1, 3), c(3, 4), c(3, 2))
st_multipoint(multipoint_matrix)
#> MULTIPOINT (5 2, 1 3, 3 4, 3 2)
## LINESTRING
linestring_matrix = rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2))
st_linestring(linestring_matrix)
#> LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

Finally, use lists for the creation of multilinestrings, (multi-)polygons and geometry collections:

```
## POLYGON
polygon_list = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
st_polygon(polygon_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
```

```
## POLYGON with a hole
polygon_border = rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))
polygon_hole = rbind(c(2, 4), c(3, 4), c(3, 3), c(2, 3), c(2, 4))
polygon_with_hole_list = list(polygon_border, polygon_hole)
st_polygon(polygon_with_hole_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))
```

```
## MULTILINESTRING
multilinestring_list = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                            rbind(c(1, 2), c(2, 4)))
st_multilinestring((multilinestring_list))
#> MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))
```

```
## MULTIPOLYGON
multipolygon_list = list(list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))),
                         list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2))))
st_multipolygon(multipolygon_list)
#> MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5)), ((0 2, 1 2, 1 3, 0 3, 0 2)))
```

```
## GEOMETRYCOLLECTION
gemetrycollection_list = list(st_multipoint(multipoint_matrix),
                              st_linestring(linestring_matrix))
st_geometrycollection(gemetrycollection_list)
#> GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2),
#>   LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))
```

0.2.2.7 Simple feature columns (sfc)

One `sfg` object contains only a single simple feature geometry. A simple feature geometry column (`sfc`) is a list of `sfg` objects, which is additionally able to contain information about the coordinate reference system in use. For instance, to combine two simple features into one object with two features, we can use the `st_sfc()` function. This is important since `sfc` represents the geometry column in **sf** data frames:

```
# sfc POINT
point1 = st_point(c(5, 2))
point2 = st_point(c(1, 3))
points_sfc = st_sfc(point1, point2)
points_sfc
#> Geometry set for 2 features
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> epsg (SRID):    NA
#> proj4string:    NA
#> POINT (5 2)
#> POINT (1 3)
```

In most cases, an `sfc` object contains objects of the same geometry type. Therefore, when we convert `sfg` objects of type polygon into a simple feature geometry column, we would also end up with an `sfc` object of type polygon, which can be verified with `st_geometry_type()`. Equally, a geometry column of multilinestrings would result in an `sfc` object of type multilinestring:

```
# sfc POLYGON
polygon_list1 = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
polygon1 = st_polygon(polygon_list1)
polygon_list2 = list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2)))
polygon2 = st_polygon(polygon_list2)
polygon_sfc = st_sfc(polygon1, polygon2)
st_geometry_type(polygon_sfc)
#> [1] POLYGON POLYGON
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON ... TRIANGLE
```

```
# sfc MULTILINESTRING
multilinestring_list1 = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                             rbind(c(1, 2), c(2, 4)))
multilinestring1 = st_multilinestring((multilinestring_list1))
multilinestring_list2 = list(rbind(c(2, 9), c(7, 9), c(5, 6), c(4, 7), c(2, 7)),
                             rbind(c(1, 7), c(3, 8)))
multilinestring2 = st_multilinestring((multilinestring_list2))
multilinestring_sfc = st_sfc(multilinestring1, multilinestring2)
```

```
st_geometry_type(multilinestring_sfc)
#> [1] MULTILINESTRING MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON ... TRIANGLE
```

It is also possible to create an `sfc` object from `sfg` objects with different geometry types:

```
# sfc GEOMETRY
point_multilinestring_sfc = st_sfc(point1, multilinestring1)
st_geometry_type(point_multilinestring_sfc)
#> [1] POINT           MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON ... TRIANGLE
```

As mentioned before, `sfc` objects can additionally store information on the coordinate reference systems (CRS). To specify a certain CRS, we can use the `epsg (SRID)` or `proj4string` attributes of an `sfc` object. The default value of `epsg (SRID)` and `proj4string` is `NA` (*Not Available*), as can be verified with `st_crs()`:

```
st_crs(points_sfc)
#> Coordinate Reference System: NA
```

All geometries in an `sfc` object must have the same CRS. We can add coordinate reference system as a `crs` argument of `st_sfc()`. This argument accepts an integer with the `epsg` code such as `4326`, which automatically adds the 'proj4string' (see Section **??**):

```
# EPSG definition
points_sfc_wgs = st_sfc(point1, point2, crs = 4326)
st_crs(points_sfc_wgs)
#> Coordinate Reference System:
#>   EPSG: 4326
#>   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

It also accepts a raw proj4string (result not shown):

```
# PROJ4STRING definition
st_sfc(point1, point2, crs = "+proj=longlat +datum=WGS84 +no_defs")
```

Sometimes `st_crs()` will return a `proj4string` but not an `epsg` code. This is because there is no general method to convert from `proj4string` to `epsg` (see Chapter **??**).

0.2.2.8 The sf class

Sections **??** to **??** deal with purely geometric objects, 'sf geometry' and 'sf column' objects, respectively. These are geographic building blocks of geographic vector data represented as simple features. The final building block is non-geographic attributes, representing the name of the feature or other attributes such as measured values, groups, and other things.

To illustrate attributes, we will represent a temperature of 25°C in London on June 21st, 2017. This example contains a geometry (the coordinates), and three attributes with three different classes (place name, temperature and date).[118]

Objects of class `sf` represent such data by combining the attributes (`data.frame`) with the simple feature geometry column (`sfc`). They are created with `st_sf()` as illustrated below, which creates the London example described above:

```
lnd_point = st_point(c(0.1, 51.5))                 # sfg object
lnd_geom = st_sfc(lnd_point, crs = 4326)           # sfc object
lnd_attrib = data.frame(                           # data.frame object
  name = "London",
  temperature = 25,
  date = as.Date("2017-06-21")
  )
lnd_sf = st_sf(lnd_attrib, geometry = lnd_geom)    # sf object
```

What just happened? First, the coordinates were used to create the simple feature geometry (`sfg`). Second, the geometry was converted into a simple feature geometry column (`sfc`), with a CRS. Third, attributes were stored in a `data.frame`, which was combined with the `sfc` object with `st_sf()`. This results in an `sf` object, as demonstrated below (some output is ommited):

[118] Other attributes might include an urbanity category (city or village), or a remark if the measurement was made using an automatic station.

l

```
lnd_sf
#> Simple feature collection with 1 features and 3 fields
#> ...
#>     name temperature       date          geometry
#> 1 London          25 2017-06-21 POINT (0.1 51.5)
```

```
class(lnd_sf)
#> [1] "sf"          "data.frame"
```

The result shows that `sf` objects actually have two classes, `sf` and `data.frame`. Simple features are simply data frames (square tables), but with spatial attributes stored in a list column, usually called `geometry`, as described in Section **??**. This duality is central to the concept of simple features: most of the time a `sf` can be treated as and behaves like a `data.frame`. Simple features are, in essence, data frames with a spatial extension.

0.2.3 Raster data

The geographic raster data model usually consists of a raster header and a matrix (with rows and columns) representing equally spaced cells (often also called pixels; Figure **??**:A).[119] The raster header defines the coordinate reference system, the extent and the origin. The origin (or starting point) is frequently the coordinate of the lower-left corner of the matrix (the **raster** package, however, uses the upper left corner, by default (Figure **??**:B)). The header defines the extent via the number of columns, the number of rows and the cell size resolution. Hence, starting from the origin, we can easily access and modify each single cell by either using the ID of a cell (Figure **??**:B) or by explicitly specifying the rows and columns. This matrix representation avoids storing explicitly the coordinates for the four corner points (in fact it only stores one coordinate, namely the origin) of each cell corner as would be the case for rectangular vector polygons. This and map algebra makes raster processing much more efficient and faster than vector data processing. However, in contrast to vector data, the cell of one raster layer can only hold a single value. The value might be numeric or categorical (Figure **??**:C).

Raster maps usually represent continuous phenomena such as elevation, temperature, population density or spectral data (Figure **??**). Of course, we can

[119] Depending on the file format the header is part of the actual image data file, e.g., GeoTIFF, or stored in an extra header or world file, e.g., ASCII grid formats. There is also the headerless (flat) binary raster format which should facilitate the import into various software programs.

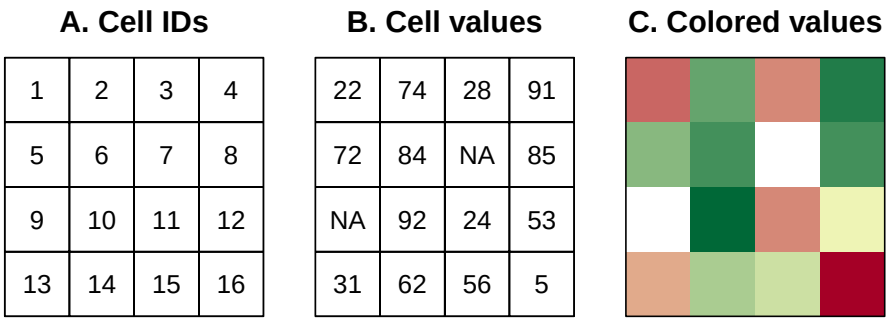


FIGURE 11: Raster data types: (A) cell IDs, (B) cell values, (C) a colored raster map.

represent discrete features such as soil or land-cover classes also with the help of a raster data model (Figure **??**). Consequently, the discrete borders of these features become blurred, and depending on the spatial task a vector representation might be more suitable.

```
#> Warning: Detecting old grouped_df format, replacing `vars` attribute by
#> `groups`
```

0.2.3.1 An introduction to raster

The **raster** package supports raster objects in R. It provides an extensive set of functions to create, read, export, manipulate and process raster datasets. Aside from general raster data manipulation, **raster** provides many low-level functions that can form the basis to develop more advanced raster functionality. **raster** also lets you work on large raster datasets that are too large to fit into the main memory. In this case, **raster** provides the possibility to divide the raster into smaller chunks (rows or blocks), and processes these iteratively instead of loading the whole raster file into RAM (for more information, please refer to `vignette("functions", package = "raster")`.

For the illustration of **raster** concepts, we will use datasets from the **spDataLarge** (note these packages were loaded at the beginning of the chapter). It consists of a few raster objects and one vector object covering an area of the Zion National Park (Utah, USA). For example, `srtm.tif` is a digital

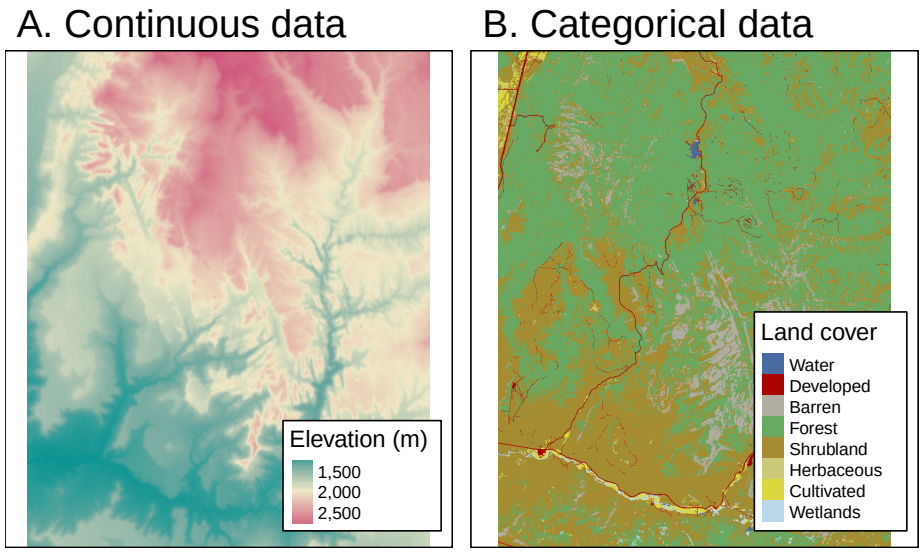


FIGURE 12: Examples of continuous and categorical rasters.

elevation model of this area (for more details, see its documentation `?srtm`). First, let's create a `RasterLayer` object named `new_raster`:

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
new_raster = raster(raster_filepath)
```

Typing the name of the raster into the console, will print out the raster header (extent, dimensions, resolution, CRS) and some additional information (class, data source name, summary of the raster values):

```
new_raster
#> class       : RasterLayer
#> dimensions  : 457, 465, 212505  (nrow, ncol, ncell)
#> resolution  : 0.000833, 0.000833  (x, y)
#> extent      : -113, -113, 37.1, 37.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
#> data source : /home/robin/R/x86_64-pc-linux../3.5/spDataLarge/raster/srtm.tif
#> names       : srtm
#> values      : 1024, 2892  (min, max)
```

Dedicated functions report each component: `dim(new_raster)` returns the

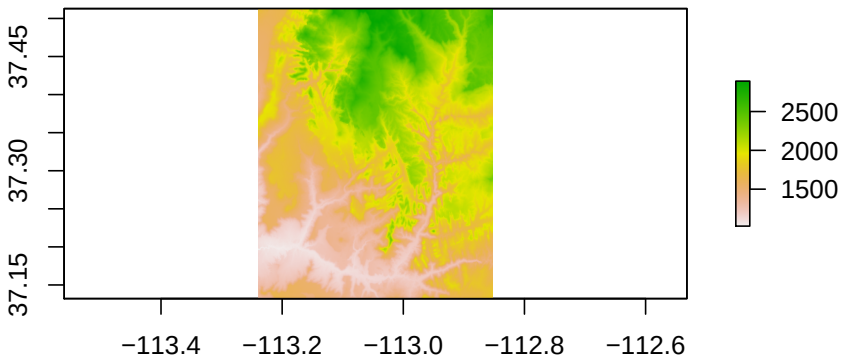


FIGURE 13: Basic raster plot.

number of rows, columns and layers; the `ncell()` function the number of cells (pixels); `res()` the raster's spatial resolution; `extent()` its spatial extent; and `crs()` its coordinate reference system (raster reprojection is covered in Section **??**). `inMemory()` reports whether the raster data is stored in memory (the default) or on disk.
`help("raster-package")` returns a full list of all available **raster** functions.

0.2.3.2 Basic map making

Similar to the **sf** package, **raster** also provides `plot()` methods for its own classes.

```
plot(new_raster)
```

There are several other approaches for plotting raster data in R that are outside the scope of this section, including:

- Functions such as `spplot()` and `levelplot()` (from the **sp** and **rasterVis** packages, respectively) to create facets, a common technique for visualizing change over time.
- Packages such as **tmap**, **mapview** and **leaflet** to create interactive maps of raster and vector objects (see Chapter **??**).

0.2.3.3 Raster classes

The `RasterLayer` class represents the simplest form of a raster object, and consists of only one layer. The easiest way to create a raster object in R is to read-in a raster file from disk or from a server.

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
new_raster = raster(raster_filepath)
```

The **raster** package supports numerous drivers with the help of **rgdal**. To find out which drivers are available on your system, run `raster::writeFormats()` and `rgdal::gdalDrivers()`.
Rasters can also be created from scratch using the `raster()` function. This is illustrated in the subsequent code chunk, which results in a new `RasterLayer` object. The resulting raster consists of 36 cells (6 columns and 6 rows specified by `nrows` and `ncols`) centered around the Prime Meridian and the Equator (see `xmn`, `xmx`, `ymn` and `ymx` parameters). The CRS is the default of raster objects: WGS84. This means the unit of the resolution is in degrees which we set to 0.5 (`res`). Values (`vals`) are assigned to each cell: 1 to cell 1, 2 to cell 2, and so on.
Remember: `raster()` fills cells row-wise (unlike `matrix()`) starting at the upper left corner, meaning the top row contains the values 1 to 6, the second 7 to 12, etc.

```
new_raster2 = raster(nrows = 6, ncols = 6, res = 0.5,
                     xmn = -1.5, xmx = 1.5, ymn = -1.5, ymx = 1.5,
                     vals = 1:36)
```

For other ways of creating raster objects, see `?raster`.
Aside from `RasterLayer`, there are two additional classes: `RasterBrick` and `RasterStack`. Both can handle multiple layers, but differ regarding the number of supported file formats, type of internal representation and processing speed.
A `RasterBrick` consists of multiple layers, which typically correspond to a single multispectral satellite file or a single multilayer object in memory. The `brick()` function creates a `RasterBrick` object. Usually, you provide it with a filename to a multilayer raster file but might also use another raster object and other spatial objects (see `?brick` for all supported formats).

```
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
r_brick = brick(multi_raster_file)
```

```
r_brick
#> class       : RasterBrick
#> resolution  : 30, 30  (x, y)
#> ...
#> names       : landsat.1, landsat.2, landsat.3, landsat.4
#> min values  :      7550,      6404,      5678,      5252
#> max values  :     19071,     22051,     25780,     31961
```

`nlayers()` retrieves the number of layers stored in a `Raster*` object:

```
nlayers(r_brick)
#> [1] 4
```

A `RasterStack` is similar to a `RasterBrick` in the sense that it consists also of multiple layers. However, in contrast to `RasterBrick`, `RasterStack` allows you to connect several raster objects stored in different files or multiply objects in memory. More specifically, a `RasterStack` is a list of `RasterLayer` objects with the same extent and resolution. Hence, one way to create it is with the help of spatial objects already existing in R's global environment. And again, one can simply specify a path to a file stored on disk.

```
raster_on_disk = raster(r_brick, layer = 1)
raster_in_memory = raster(xmn = 301905, xmx = 335745,
                          ymn = 4111245, ymx = 4154085,
                          res = 30)
values(raster_in_memory) = sample(seq_len(ncell(raster_in_memory)))
crs(raster_in_memory) = crs(raster_on_disk)
```

```
r_stack = stack(raster_in_memory, raster_on_disk)
r_stack
#> class : RasterStack
#> dimensions : 1428, 1128, 1610784, 2
#> resolution : 30, 30
#> ...
#> names       :   layer, landsat.1
#> min values  :       1,      7550
#> max values  : 1610784,     19071
```

Another difference is that the processing time for `RasterBrick` objects is usually shorter than for `RasterStack` objects.
Decision on which `Raster*` class should be used depends mostly on a character of input data. Processing of a single mulitilayer file or object is the most effective with `RasterBrick`, while `RasterStack` allows calculations based on many files, many `Raster*` objects, or both.

Operations on `RasterBrick` and `RasterStack` objects will typically return a `RasterBrick`.

0.2.4 Coordinate Reference Systems

Vector and raster spatial data types share concepts intrinsic to spatial data. Perhaps the most fundamental of these is the Coordinate Reference System (CRS), which defines how the spatial elements of the data relate to the surface of the Earth (or other bodies). CRSs are either geographic or projected, as introduced at the beginning of this chapter (see Figure **??**). This section will explain each type, laying the foundations for Section **??** on CRS transformations.

0.2.4.1 Geographic coordinate systems

Geographic coordinate systems identify any location on the Earth's surface using two values — longitude and latitude. *Longitude* is location in the East-West direction in angular distance from the Prime Meridian plane. *Latitude* is angular distance North or South of the equatorial plane. Distances in geographic CRSs are therefore not measured in meters. This has important consequences, as demonstrated in Section **??**.
The surface of the Earth in geographic coordinate systems is represented by a spherical or ellipsoidal surface. Spherical models assume that the Earth is a perfect sphere of a given radius. Spherical models have the advantage of simplicity but are rarely used because they are inaccurate: the Earth is not a sphere! Ellipsoidal models are defined by two parameters: the equatorial radius and the polar radius. These are suitable because the Earth is compressed: the equatorial radius is around 11.5 km longer than the polar radius (Maling 1992).[120]
Ellipsoids are part of a wider component of CRSs: the *datum*. This contains information on what ellipsoid to use (with the `ellps` parameter in the PROJ

[120]The degree of compression is often referred to as *flattening*, defined in terms of the equatorial radius (a) and polar radius (b) as follows: $f = (a - b)/a$. The terms *ellipticity* and *compression* can also be used (Maling 1992). Because f is a rather small value, digital ellipsoid models use the 'inverse flattening' ($rf = 1/f$) to define the Earth's compression. Values of a and rf in various ellipsoidal models can be seen by executing `st_proj_info(type = "ellps")`.

CRS library) and the precise relationship between the Cartesian coordinates and location on the Earth's surface. These additional details are stored in the `towgs84` argument of proj4string[121] notation (see proj4.org/parameters.html[122] for details). These allow local variations in Earth's surface, for example due to large mountain ranges, to be accounted for in a local CRS. There are two types of datum — local and geocentric. In a *local datum* such as `NAD83` the ellipsoidal surface is shifted to align with the surface at a particular location. In a *geocentric datum* such as `WGS84` the center is the Earth's center of gravity and the accuracy of projections is not optimized for a specific location. Available datum definitions can be seen by executing `st_proj_info(type = "datum")`.

0.2.4.2 Projected coordinate reference systems

Projected CRSs are based on Cartesian coordinates on an implicitly flat surface. They have an origin, x and y axes, and a linear unit of measurement such as meters. All projected CRSs are based on a geographic CRS, described in the previous section, and rely on map projections to convert the three-dimensional surface of the Earth into Easting and Northing (x and y) values in a projected CRS.

This transition cannot be done without adding some distortion. Therefore, some properties of the Earth's surface are distorted in this process, such as area, direction, distance, and shape. A projected coordinate system can preserve only one or two of those properties. Projections are often named based on a property they preserve: equal-area preserves area, azimuthal preserve direction, equidistant preserve distance, and conformal preserve local shape.

There are three main groups of projection types - conic, cylindrical, and planar. In a conic projection, the Earth's surface is projected onto a cone along a single line of tangency or two lines of tangency. Distortions are minimized along the tangency lines and rise with the distance from those lines in this projection. Therefore, it is the best suited for maps of mid-latitude areas. A cylindrical projection maps the surface onto a cylinder. This projection could also be created by touching the Earth's surface along a single line of tangency or two lines of tangency. Cylindrical projections are used most often when mapping the entire world. A planar projection projects data onto a flat surface touching the globe at a point or along a line of tangency. It is typically used in mapping polar regions. `st_proj_info(type = "proj")` gives a list of the available projections supported by the PROJ library.

[121] `https://proj4.org/operations/conversions/latlon.html?highlight=towgs#cmdoption-arg-towgs84`
[122] `https://proj4.org/usage/projections.html`

0.2.4.3 CRSs in R

Two main ways to describe CRS in R are an `epsg` code or a `proj4string` definition. Both of these approaches have advantages and disadvantages. An `epsg` code is usually shorter, and therefore easier to remember. The code also refers to only one, well-defined coordinate reference system. On the other hand, a `proj4string` definition allows you more flexibility when it comes to specifying different parameters such as the projection type, the datum and the ellipsoid.[123] This way you can specify many different projections, and modify existing ones. This also makes the `proj4string` approach more complicated. `epsg` points to exactly one particular CRS.

Spatial R packages support a wide range of CRSs and they use the long-established PROJ[124] library. Other than searching for EPSG codes online, another quick way to find out about available CRSs is via the `rgdal::make_EPSG()` function, which outputs a data frame of available projections. Before going into more detail, it's worth learning how to view and filter them inside R, as this could save time trawling the internet. The following code will show available CRSs interactively, allowing you to filter ones of interest (try filtering for the OSGB CRSs for example):

```
crs_data = rgdal::make_EPSG()
View(crs_data)
```

In **sf** the CRS of an object can be retrieved using `st_crs()`. For this, we need to read-in a vector dataset:

```
vector_filepath = system.file("vector/zion.gpkg", package = "spDataLarge")
new_vector = st_read(vector_filepath)
```

Our new object, `new_vector`, is a polygon representing the borders of Zion National Park (`?zion`).

```
st_crs(new_vector) # get CRS
#> Coordinate Reference System:
#> No EPSG code
#> proj4string: "+proj=utm +zone=12 +ellps=GRS80 ... +units=m +no_defs"
```

[123] A complete list of the `proj4string` parameters can be found at `https://proj4.org/`.
[124] `http://proj4.org/`

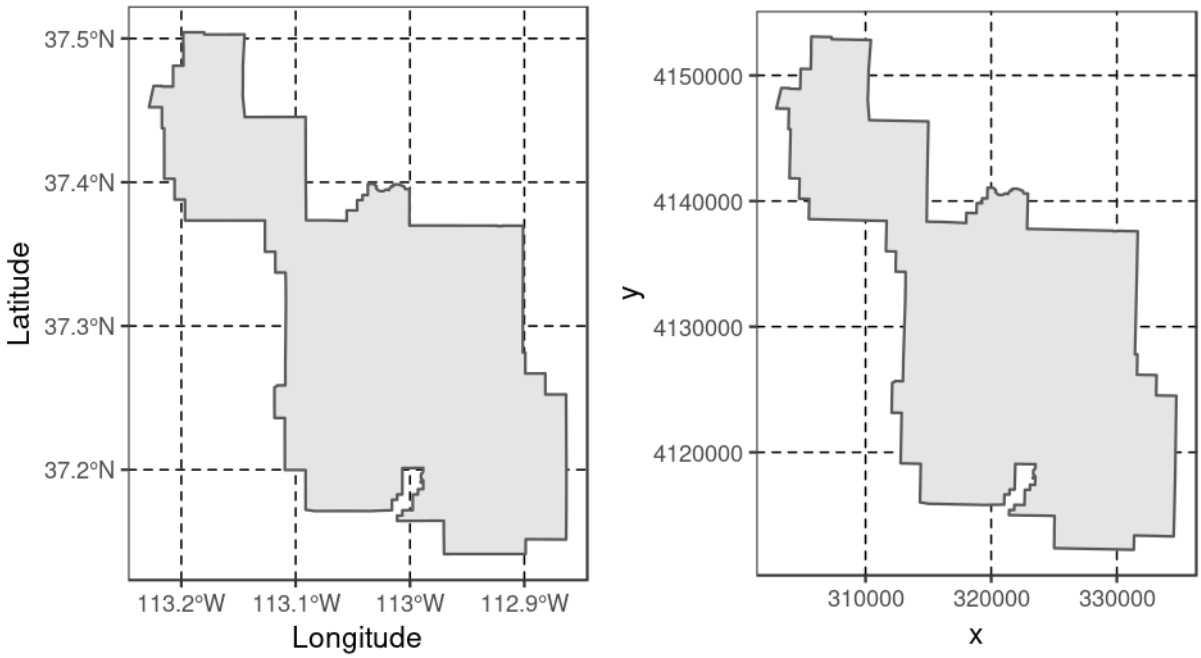


FIGURE 14: Examples of geographic (WGS 84; left) and projected (NAD83 / UTM zone 12N; right) coordinate systems for a vector data type.

In cases when a coordinate reference system (CRS) is missing or the wrong CRS is set, the `st_set_crs()` function can be used:

```
new_vector = st_set_crs(new_vector, 4326) # set CRS
#> Warning: st_crs<- : replacing crs does not reproject data; use st_transform
#> for that
```

The warning message informs us that the `st_set_crs()` function does not transform data from one CRS to another.
The `projection()` function can be used to access CRS information from a `Raster*` object:

```
projection(new_raster) # get CRS
#> [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

The same function, `projection()`, is used to set a CRS for raster objects. The main difference, compared to vector data, is that raster objects only accept `proj4` definitions:

lx

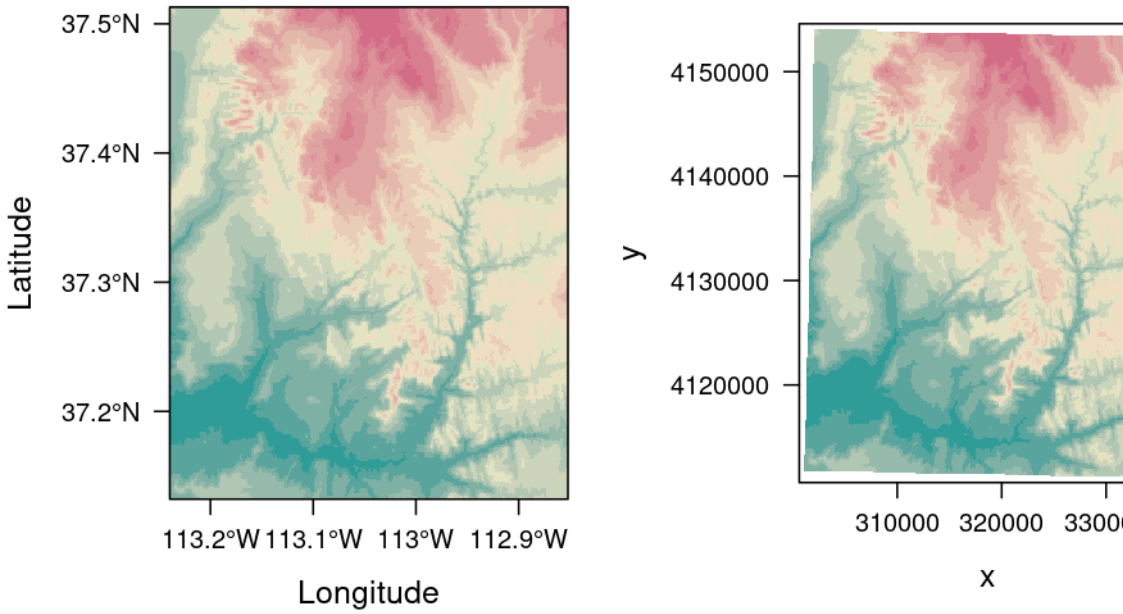


FIGURE 15: Examples of geographic (WGS 84; left) and projected (NAD83 / UTM zone 12N; right) coordinate systems for raster data.

```
projection(new_raster) = "+proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0
                          +units=m +no_defs" # set CRS
```

We will expand on CRSs and how to project from one CRS to another in much more detail in Chapter **??**.

0.2.5 Units

An important feature of CRSs is that they contain information about spatial units. Clearly, it is vital to know whether a house's measurements are in feet or meters, and the same applies to maps. It is good cartographic practice to add a *scale bar* onto maps to demonstrate the relationship between distances on the page or screen and distances on the ground. Likewise, it is important to formally specify the units in which the geometry data or pixels are measured to provide context, and ensure that subsequent calculations are done in context. A novel feature of geometry data in `sf` objects is that they have *native support* for units. This means that distance, area and other geometric calculations in **sf** return values that come with a `units` attribute, defined by the **units** package

(Pebesma, Mailund, and Hiebert 2016). This is advantageous, preventing confusion caused by different units (most CRSs use meters, some use feet) and providing information on dimensionality. This is demonstrated in the code chunk below, which calculates the area of Luxembourg:

```
luxembourg = world[world$name_long == "Luxembourg", ]
```

```
st_area(luxembourg)
#> 2.41e+09 [m^2]
```

The output is in units of square meters (m^2), showing that the result represents two-dimensional space. This information, stored as an attribute (which interested readers can discover with `attributes(st_area(luxembourg))`), can feed into subsequent calculations that use units, such as population density (which is measured in people per unit area, typically per km^2). Reporting units prevents confusion. To take the Luxembourg example, if the units remained unspecified, one could incorrectly assume that the units were in hectares. To translate the huge number into a more digestible size, it is tempting to divide the results by a million (the number of square meters in a square kilometer):

```
st_area(luxembourg) / 1000000
#> 2414 [m^2]
```

However, the result is incorrectly given again as square meters. The solution is to set the correct units with the **units** package:

```
units::set_units(st_area(luxembourg), km^2)
#> 2414 [km^2]
```

Units are of equal importance in the case of raster data. However, so far **sf** is the only spatial package that supports units, meaning that people working on raster data should approach changes in the units of analysis (for example, converting pixel widths from imperial to decimal units) with care. The `new_raster` object (see above) uses a WGS84 projection with decimal degrees as units. Consequently, its resolution is also given in decimal degrees but you have to know it, since the `res()` function simply returns a numeric vector.

```
res(new_raster)
#> [1] 0.000833 0.000833
```

If we used the UTM projection, the units would change.

```
repr = projectRaster(new_raster, crs = "+init=epsg:26912")
res(repr)
#> [1] 0.000833 0.000833
```

Again, the `res()` command gives back a numeric vector without any unit, forcing us to know that the unit of the UTM projection is meters.

0.2.6 Exercises

1. Use `summary()` on the geometry column of the `world` data object. What does the output tell us about:
    - Its geometry type?
    - The number of countries?
    - Its coordinate reference system (CRS)?
2. Run the code that 'generated' the map of the world in Figure **??** at the end of Section **??**. Find two similarities and two differences between the image on your computer and that in the book.
    - What does the `cex` argument do (see `?plot`)?
    - Why was `cex` set to the `sqrt(world$pop) / 10000`?
    - Bonus: experiment with different ways to visualize the global population.
3. Use `plot()` to create maps of Nigeria in context (see Section **??**).
    - Adjust the `lwd`, `col` and `expandBB` arguments of `plot()`.
    - Challenge: read the documentation of `text()` and annotate the map.
4. Create an empty `RasterLayer` object called `my_raster` with 10 columns and 10 rows. Assign random values between 0 and 10 to the new raster and plot it.
5. Read-in the `raster/nlcd2011.tif` file from the **spDataLarge** package. What kind of information can you get about the properties of this file?

Reminder: solutions can be found online at `https://geocompr.github.io`

0.3 Attribute data operations

Prerequisites

- This chapter requires the following packages to be installed and attached:

```
library(sf)
library(raster)
library(dplyr)
library(stringr) # for working with strings (pattern matching)
```

- It also relies on **spData**, which loads datasets used in the code examples of this chapter:

```
library(spData)
```

0.3.1 Introduction

Attribute data is non-spatial information associated with geographic (geometry) data. A bus stop provides a simple example: its position would typically be represented by latitude and longitude coordinates (geometry data), in addition to its name. The name is an *attribute* of the feature (to use Simple Features terminology) that bears no relation to its geometry.

Another example is the elevation value (attribute) for a specific grid cell in raster data. Unlike the vector data model, the raster data model stores the coordinate of the grid cell indirectly, meaning the distinction between attribute and spatial information is less clear. To illustrate the point, think of a pixel in the 3rd row and the 4th column of a raster matrix. Its spatial location is defined by its index in the matrix: move from the origin four cells in the x direction (typically east and right on maps) and three cells in the y direction (typically south and down). The raster's *resolution* defines the distance for each x- and y-step which is specified in a *header*. The header is a vital component of raster datasets which specifies how pixels relate to geographic coordinates (see also Chapter **??**).

The focus of this chapter is manipulating geographic objects based on attributes such as the name of a bus stop and elevation. For vector data, this means operations such as subsetting and aggregation (see Sections **??** and **??**). These

non-spatial operations have spatial equivalents: the `[` operator in base R, for example, works equally for subsetting objects based on their attribute and spatial objects, as we will see in Chapter **??**. This is good news: skills developed here are cross-transferable, meaning that this chapter lays the foundation for Chapter **??**, which extends the methods presented here to the spatial world. Sections **??** and **??** demonstrate how to join data onto simple feature objects using a shared ID and how to create new variables, respectively.
Raster attribute data operations are covered in Section **??**, which covers creating continuous and categorical raster layers and extracting cell values from one layer and multiple layers (raster subsetting). Section **??** provides an overview of 'global' raster operations which can be used to characterize entire raster datasets.

0.3.2 Vector attribute manipulation

Geographic vector data in R are well supported by **sf**, a class which extends the `data.frame`. Thus `sf` objects have one column per attribute variable (such as 'name') and one row per observation, or *feature* (e.g., per bus station). `sf` objects also have a special column to contain geometry data, usually named `geometry`. The `geometry` column is special because it is a *list column*, which can contain multiple geographic entities (points, lines, polygons) per row. This was described in Chapter **??**, which demonstrated how *generic methods* such as `plot()` and `summary()` work on `sf` objects. **sf** also provides methods that allow `sf` objects to behave like regular data frames, as illustrated by other `sf`-specific methods that were originally developed for data frames:

```
methods(class = "sf") # methods for sf objects, first 12 shown
```

```
#>  [1] aggregate            cbind                coerce
#>  [4] initialize           merge                plot
#>  [7] print                rbind                [
#> [10] [[<-                 $<-                  show
```

Many of these functions, including `rbind()` (for binding rows of data together) and `$<-` (for creating new columns) were developed for data frames. A key feature of `sf` objects is that they store spatial and non-spatial data in the same way, as columns in a `data.frame`.

The geometry column of `sf` objects is typically called `geometry` but any name can be used. The following command, for example, creates a geometry column named g:

```
st_sf(data.frame(n = world$name_long), g = world$geom)
```

This enables geometries imported from spatial databases to have a variety of names such as `wkb_geometry` and `the_geom`.

`sf` objects also support `tibble` and `tbl` classes used in the tidyverse, allowing 'tidy' data analysis workflows for spatial data. Thus **sf** enables the full power of R's data analysis capabilities to be unleashed on geographic data. Before using these capabilities it is worth re-capping how to discover the basic properties of vector data objects. Let's start by using base R functions to get a measure of the `world` dataset:

```
dim(world) # it is a 2 dimensional object, with rows and columns
#> [1] 177  11
nrow(world) # how many rows?
#> [1] 177
ncol(world) # how many columns?
#> [1] 11
```

Our dataset contains ten non-geographic columns (and one geometry list column) with almost 200 rows representing the world's countries. Extracting the attribute data of an `sf` object is the same as removing its geometry:

```
world_df = st_drop_geometry(world)
class(world_df)
#> [1] "data.frame"
```

This can be useful if the geometry column causes problems, e.g., by occupying large amounts of RAM, or to focus the attention on the attribute data. For most cases, however, there is no harm in keeping the geometry column because non-spatial data operations on `sf` objects only change an object's geometry when appropriate (e.g., by dissolving borders between adjacent polygons following aggregation). This means that proficiency with attribute data in `sf` objects equates to proficiency with data frames in R.

For many applications, the tidyverse package **dplyr** offers the most effective and intuitive approach for working with data frames. Tidyverse compatibility is

an advantage of **sf** over its predecessor **sp**, but there are some pitfalls to avoid (see the supplementary `tidyverse-pitfalls` vignette at geocompr.github.io[125] for details).

0.3.2.1 Vector attribute subsetting

Base R subsetting functions include `[`, `subset()` and `$`. **dplyr** subsetting functions include `select()`, `filter()`, and `pull()`. Both sets of functions preserve the spatial components of attribute data in `sf` objects.
The `[` operator can subset both rows and columns. You use indices to specify the elements you wish to extract from an object, e.g., `object[i, j]`, with `i` and `j` typically being numbers or logical vectors — `TRUE`s and `FALSE`s — representing rows and columns (they can also be character strings, indicating row or column names). Leaving `i` or `j` empty returns all rows or columns, so `world[1:5, ]` returns the first five rows and all columns. The examples below demonstrate subsetting with base R. The results are not shown; check the results on your own computer:

```
world[1:6, ] # subset rows by position
world[, 1:3] # subset columns by position
world[, c("name_long", "lifeExp")] # subset columns by name
```

A demonstration of the utility of using `logical` vectors for subsetting is shown in the code chunk below. This creates a new object, `small_countries`, containing nations whose surface area is smaller than 10,000 km^2:

```
sel_area = world$area_km2 < 10000
summary(sel_area) # a logical vector
#>    Mode   FALSE    TRUE
#> logical     170       7
small_countries = world[sel_area, ]
```

The intermediary `sel_area` is a logical vector that shows that only seven countries match the query. A more concise command, which omits the intermediary object, generates the same result:

```
small_countries = world[world$area_km2 < 10000, ]
```

[125] `https://geocompr.github.io/geocompkg/articles/tidyverse-pitfalls.html`

The base R function `subset()` provides yet another way to achieve the same result:

```
small_countries = subset(world, area_km2 < 10000)
```

Base R functions are mature and widely used. However, the more recent **dplyr** approach has several advantages. It enables intuitive workflows. It is fast, due to its C++ backend. This is especially useful when working with big data as well as **dplyr**'s database integration. The main **dplyr** subsetting functions are `select()`, `slice()`, `filter()` and `pull()`.

raster and **dplyr** packages have a function called `select()`. When using both packages, the function in the most recently attached package will be used, 'masking' the incumbent function. This can generate error messages containing text like: `unable to find an inherited method for function 'select' for signature '"sf"'`. To avoid this error message, and prevent ambiguity, we use the long-form function name, prefixed by the package name and two colons (usually omitted from R scripts for concise code): `dplyr::select()`.

`select()` selects columns by name or position. For example, you could select only two columns, `name_long` and `pop`, with the following command (note the sticky `geom` column remains):

```
world1 = dplyr::select(world, name_long, pop)
names(world1)
#> [1] "name_long" "pop"       "geom"
```

`select()` also allows subsetting of a range of columns with the help of the `:` operator:

```
# all columns between name_long and pop (inclusive)
world2 = dplyr::select(world, name_long:pop)
```

Omit specific columns with the `-` operator:

```
# all columns except subregion and area_km2 (inclusive)
world3 = dplyr::select(world, -subregion, -area_km2)
```

Conveniently, `select()` lets you subset and rename columns at the same time, for example:

```
world4 = dplyr::select(world, name_long, population = pop)
names(world4)
#> [1] "name_long"  "population" "geom"
```

This is more concise than the base R equivalent:

```
world5 = world[, c("name_long", "pop")] # subset columns by name
names(world5)[names(world5) == "pop"] = "population" # rename column manually
```

`select()` also works with 'helper functions' for advanced subsetting operations, including `contains()`, `starts_with()` and `num_range()` (see the help page with `?select` for details).

Most **dplyr** verbs return a data frame. To extract a single vector, one has to explicitly use the `pull()` command. The subsetting operator in base R (see `?[`), by contrast, tries to return objects in the lowest possible dimension. This means selecting a single column returns a vector in base R. To turn off this behavior, set the `drop` argument to `FALSE`.

```
# create throw-away data frame
d = data.frame(pop = 1:10, area = 1:10)
# return data frame object when selecting a single column
d[, "pop", drop = FALSE] # equivalent to d["pop"]
select(d, pop)
# return a vector when selecting a single column
d[, "pop"]
pull(d, pop)
```

Due to the sticky geometry column, selecting a single attribute from an sf-object with the help of `[()` returns also a data frame. Contrastingly, `pull()` and `$` will give back a vector.

TABLE 0.3: Comparison operators that return Booleans (TRUE/FALSE).

Symbol	Name
'=='	Equal to
'!='	Not equal to
'>', '<'	Greater/Less than
'>=', '<='	Greater/Less than or equal
'&', '|', '¡	Logical operators: And, Or, Not

```
# data frame object
world[, "pop"]
# vector objects
world$pop
pull(world, pop)
```

`slice()` is the row-equivalent of `select()`. The following code chunk, for example, selects the 3rd to 5th rows:

```
slice(world, 3:5)
```

`filter()` is **dplyr**'s equivalent of base R's `subset()` function. It keeps only rows matching given criteria, e.g., only countries with a very high average of life expectancy:

```
# Countries with a life expectancy longer than 82 years
world6 = filter(world, lifeExp > 82)
```

The standard set of comparison operators can be used in the `filter()` function, as illustrated in Table **??**:

dplyr works well with the 'pipe'[126] operator `%>%`, which takes its name from the Unix pipe `|` (Grolemund and Wickham 2016). It enables expressive code: the output of a previous function becomes the first argument of the next function, enabling *chaining*. This is illustrated below, in which only countries from Asia are filtered from the `world` dataset, next the object is subset by columns (`name_long` and `continent`) and the first five rows (result not shown).

[126] `http://r4ds.had.co.nz/pipes.html`

```
world7 = world %>%
  filter(continent == "Asia") %>%
  dplyr::select(name_long, continent) %>%
  slice(1:5)
```

The above chunk shows how the pipe operator allows commands to be written in a clear order: the above run from top to bottom (line-by-line) and left to right. The alternative to `%>%` is nested function calls, which is harder to read:

```
world8 = slice(
  dplyr::select(
    filter(world, continent == "Asia"),
    name_long, continent),
  1:5)
```

0.3.2.2 Vector attribute aggregation

Aggregation operations summarize datasets by a 'grouping variable', typically an attribute column (spatial aggregation is covered in the next chapter). An example of attribute aggregation is calculating the number of people per continent based on country-level data (one row per country). The `world` dataset contains the necessary ingredients: the columns `pop` and `continent`, the population and the grouping variable, respectively. The aim is to find the `sum()` of country populations for each continent. This can be done with the base R function `aggregate()` as follows:

```
world_agg1 = aggregate(pop ~ continent, FUN = sum, data = world, na.rm = TRUE)
class(world_agg1)
#> [1] "data.frame"
```

The result is a non-spatial data frame with six rows, one per continent, and two columns reporting the name and population of each continent (see Table **??** with results for the top 3 most populous continents).

`aggregate()` is a generic function which means that it behaves differently depending on its inputs. **sf** provides a function that can be called directly with `sf:::aggregate()` that is activated when a `by` argument is provided, rather than using the `~` to refer to the grouping variable:

```
world_agg2 = aggregate(world["pop"], by = list(world$continent),
                       FUN = sum, na.rm = TRUE)
class(world_agg2)
#> [1] "sf"          "data.frame"
```

As illustrated above, an object of class `sf` is returned this time. `world_agg2` which is a spatial object containing 6 polygons representing the columns of the world.

`summarize()` is the **dplyr** equivalent of `aggregate()`. It usually follows `group_by()`, which specifies the grouping variable, as illustrated below:

```
world_agg3 = world %>%
  group_by(continent) %>%
  summarize(pop = sum(pop, na.rm = TRUE))
```

This approach is flexible and gives control over the new column names. This is illustrated below: the command calculates the Earth's population (~7 billion) and number of countries (result not shown):

```
world %>%
  summarize(pop = sum(pop, na.rm = TRUE), n = n())
```

In the previous code chunk `pop` and `n` are column names in the result. `sum()` and `n()` were the aggregating functions. The result is an `sf` object with a single row representing the world (this works thanks to the geometric operation 'union', as explained in Section **??**).

Let's combine what we have learned so far about **dplyr** by chaining together functions to find the world's 3 most populous continents (with `dplyr::top_n()`) and the number of countries they contain (the result of this command is presented in Table **??**):

```
world %>%
  dplyr::select(pop, continent) %>%
  group_by(continent) %>%
  summarize(pop = sum(pop, na.rm = TRUE), n_countries = n()) %>%
  top_n(n = 3, wt = pop) %>%
  st_drop_geometry()
```

TABLE 0.4: The top 3 most populous continents, and the number of countries in each.

continent	pop	n_countries
Africa	1154946633	51
Asia	4311408059	47
Europe	669036256	39

More details are provided in the help pages (which can be accessed via `?summarize` and `vignette(package = "dplyr")` and Chapter 5 of R for Data Science[127].

0.3.2.3 Vector attribute joining

Combining data from different sources is a common task in data preparation. Joins do this by combining tables based on a shared 'key' variable. **dplyr** has multiple join functions including `left_join()` and `inner_join()` — see `vignette("two-table")` for a full list. These function names follow conventions used in the database language SQL[128] (Grolemund and Wickham 2016, Chapter 13); using them to join non-spatial datasets to `sf` objects is the focus of this section. **dplyr** join functions work the same on data frames and `sf` objects, the only important difference being the `geometry` list column. The result of data joins can be either an `sf` or `data.frame` object. The most common type of attribute join on spatial data takes an `sf` object as the first argument and adds columns to it from a `data.frame` specified as the second argument.

To demonstrate joins, we will combine data on coffee production with the `world` dataset. The coffee data is in a data frame called `coffee_data` from the **spData** package (see `?coffee_data` for details). It has 3 columns: `name_long` names major coffee-producing nations and `coffee_production_2016` and `coffee_production_2017` contain estimated values for coffee production in units of 60-kg bags in each year. A 'left join', which preserves the first dataset, merges `world` with `coffee_data`:

```
world_coffee = left_join(world, coffee_data)
#> Joining, by = "name_long"
class(world_coffee)
#> [1] "sf"         "data.frame"
```

Because the input datasets share a 'key variable' (`name_long`) the join worked

[128] `http://r4ds.had.co.nz/relational-data.html`

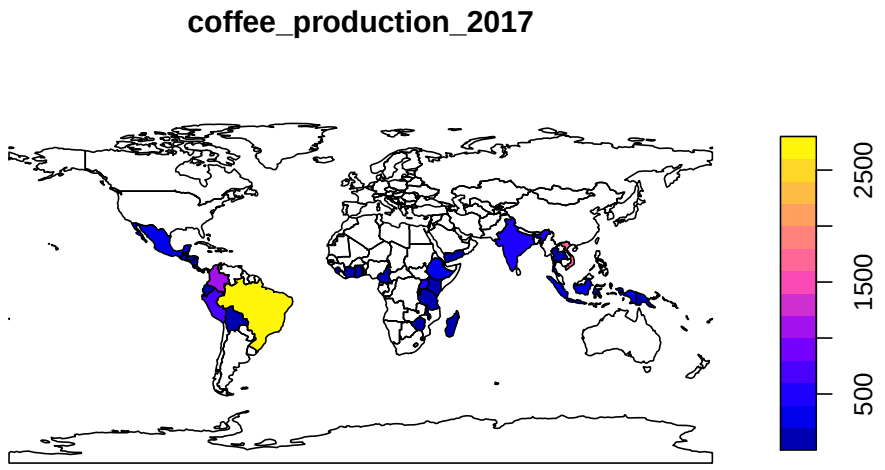


FIGURE 16: World coffee production (thousand 60-kg bags) by country, 2017. Source: International Coffee Organization.

without using the `by` argument (see `?left_join` for details). The result is an `sf` object identical to the original `world` object but with two new variables (with column indices 11 and 12) on coffee production. This can be plotted as a map, as illustrated in Figure **??**, generated with the `plot()` function below:

```
names(world_coffee)
#>  [1] "iso_a2"                  "name_long"
#>  [3] "continent"               "region_un"
#>  [5] "subregion"               "type"
#>  [7] "area_km2"                "pop"
#>  [9] "lifeExp"                 "gdpPercap"
#> [11] "coffee_production_2016" "coffee_production_2017"
#> [13] "geom"
plot(world_coffee["coffee_production_2017"])
```

For joining to work, a 'key variable' must be supplied in both datasets. By default **dplyr** uses all variables with matching names. In this case, both `world_coffee` and `world` objects contained a variable called `name_long`, explaining the message `Joining, by = "name_long"`. In the majority of cases where variable names are not the same, you have two options:

1. Rename the key variable in one of the objects so they match.

2. Use the `by` argument to specify the joining variables.

The latter approach is demonstrated below on a renamed version of `coffee_data`:

```
coffee_renamed = rename(coffee_data, nm = name_long)
world_coffee2 = left_join(world, coffee_renamed, by = c(name_long = "nm"))
```

Note that the name in the original object is kept, meaning that `world_coffee` and the new object `world_coffee2` are identical. Another feature of the result is that it has the same number of rows as the original dataset. Although there are only 47 rows of data in `coffee_data`, all 177 the country records are kept intact in `world_coffee` and `world_coffee2`: rows in the original dataset with no match are assigned `NA` values for the new coffee production variables. What if we only want to keep countries that have a match in the key variable? In that case an inner join can be used:

```
world_coffee_inner = inner_join(world, coffee_data)
#> Joining, by = "name_long"
nrow(world_coffee_inner)
#> [1] 45
```

Note that the result of `inner_join()` has only 45 rows compared with 47 in `coffee_data`. What happened to the remaining rows? We can identify the rows that did not match using the `setdiff()` function as follows:

```
setdiff(coffee_data$name_long, world$name_long)
#> [1] "Congo, Dem. Rep. of" "Others"
```

The result shows that `Others` accounts for one row not present in the `world` dataset and that the name of the `Democratic Republic of the Congo` accounts for the other: it has been abbreviated, causing the join to miss it. The following command uses a string matching (regex) function from the **stringr** package to confirm what `Congo, Dem. Rep. of` should be:

```
str_subset(world$name_long, "Dem*.+Congo")
#> [1] "Democratic Republic of the Congo"
```

To fix this issue, we will create a new version of `coffee_data` and update the name. `inner_join()`ing the updated data frame returns a result with all 46 coffee-producing nations:

```
coffee_data$name_long[grepl("Congo,", coffee_data$name_long)] =
  str_subset(world$name_long, "Dem*.+Congo")
world_coffee_match = inner_join(world, coffee_data)
#> Joining, by = "name_long"
nrow(world_coffee_match)
#> [1] 46
```

It is also possible to join in the other direction: starting with a non-spatial dataset and adding variables from a simple features object. This is demonstrated below, which starts with the `coffee_data` object and adds variables from the original `world` dataset. In contrast with the previous joins, the result is *not* another simple feature object, but a data frame in the form of a **tidyverse** tibble: the output of a join tends to match its first argument:

```
coffee_world = left_join(coffee_data, world)
#> Joining, by = "name_long"
class(coffee_world)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

In most cases, the geometry column is only useful in an `sf` object. The geometry column can only be used for creating maps and spatial operations if R 'knows' it is a spatial object, defined by a spatial package such as **sf**. Fortunately, non-spatial data frames with a geometry list column (like `coffee_world`) can be coerced into an `sf` object as follows: `st_as_sf(coffee_world)`.

This section covers the majority joining use cases. For more information, we recommend Grolemund and Wickham (2016), the join vignette[129] in the **geocompkg** package that accompanies this book, and documentation of the **data.table** package.[130] Another type of join is a spatial join, covered in the next chapter (Section **??**).

[129] `https://geocompr.github.io/geocompkg/articles/join.html`

[130] **data.table** is a high-performance data processing package. Its application to geographic data is covered in a blog post hosted at r-spatial.org/r/2017/11/13/perp-performance.html.

0.3.2.4 Creating attributes and removing spatial information

Often, we would like to create a new column based on already existing columns. For example, we want to calculate population density for each country. For this we need to divide a population column, here `pop`, by an area column, here `area_km2` with unit area in square kilometers. Using base R, we can type:

```
world_new = world # do not overwrite our original data
world_new$pop_dens = world_new$pop / world_new$area_km2
```

Alternatively, we can use one of **dplyr** functions - `mutate()` or `transmute()`. `mutate()` adds new columns at the penultimate position in the `sf` object (the last one is reserved for the geometry):

```
world %>%
  mutate(pop_dens = pop / area_km2)
```

The difference between `mutate()` and `transmute()` is that the latter skips all other existing columns (except for the sticky geometry column):

```
world %>%
  transmute(pop_dens = pop / area_km2)
```

`unite()` pastes together existing columns. For example, we want to combine the `continent` and `region_un` columns into a new column named `con_reg`. Additionally, we can define a separator (here: a colon `:`) which defines how the values of the input columns should be joined, and if the original columns should be removed (here: `TRUE`):

```
world_unite = world %>%
  unite("con_reg", continent:region_un, sep = ":", remove = TRUE)
```

The `separate()` function does the opposite of `unite()`: it splits one column into multiple columns using either a regular expression or character positions.

```
world_separate = world_unite %>%
  separate(con_reg, c("continent", "region_un"), sep = ":")
```

The two functions `rename()` and `set_names()` are useful for renaming columns. The first replaces an old name with a new one. The following command, for example, renames the lengthy `name_long` column to simply `name`:

```
world %>%
  rename(name = name_long)
```

`set_names()` changes all column names at once, and requires a character vector with a name matching each column. This is illustrated below, which outputs the same `world` object, but with very short names:

```
new_names = c("i", "n", "c", "r", "s", "t", "a", "p", "l", "gP", "geom")
world %>%
  set_names(new_names)
```

It is important to note that attribute data operations preserve the geometry of the simple features. As mentioned at the outset of the chapter, it can be useful to remove the geometry. To do this, you have to explicitly remove it because `sf` explicitly makes the geometry column sticky. This behavior ensures that data frame operations do not accidentally remove the geometry column. Hence, an approach such as `select(world, -geom)` will be unsuccessful and you should instead use `st_drop_geometry()`.[131]

```
world_data = world %>% st_drop_geometry()
class(world_data)
#> [1] "data.frame"
```

0.3.3 Manipulating raster objects

In contrast to the vector data model underlying simple features (which represents points, lines and polygons as discrete entities in space), raster data represent continuous surfaces. This section shows how raster objects work by creating them *from scratch*, building on Section **??**. Because of their unique structure, subsetting and other operations on raster datasets work in a different way, as demonstrated in Section **??**.

The following code recreates the raster dataset used in Section **??**, the result of

[131] `st_geometry(world_st) = NULL` also works to remove the geometry from `world`, but overwrites the original object.

which is illustrated in Figure **??**. This demonstrates how the `raster()` function works to create an example raster named `elev` (representing elevations).

```
elev = raster(nrows = 6, ncols = 6, res = 0.5,
              xmn = -1.5, xmx = 1.5, ymn = -1.5, ymx = 1.5,
              vals = 1:36)
```

The result is a raster object with 6 rows and 6 columns (specified by the `nrow` and `ncol` arguments), and a minimum and maximum spatial extent in x and y direction (`xmn`, `xmx`, `ymn`, `ymax`). The `vals` argument sets the values that each cell contains: numeric data ranging from 1 to 36 in this case. Raster objects can also contain categorical values of class `logical` or `factor` variables in R. The following code creates a raster representing grain sizes (Figure **??**):

```
grain_order = c("clay", "silt", "sand")
grain_char = sample(grain_order, 36, replace = TRUE)
grain_fact = factor(grain_char, levels = grain_order)
grain = raster(nrows = 6, ncols = 6, res = 0.5,
               xmn = -1.5, xmx = 1.5, ymn = -1.5, ymx = 1.5,
               vals = grain_fact)
```

`raster` objects can contain values of class `numeric`, `integer`, `logical` or `factor`, but not `character`. To use character values, they must first be converted into an appropriate class, for example using the function `factor()`. The `levels` argument was used in the preceding code chunk to create an ordered factor: clay < silt < sand in terms of grain size. See the Data structures chapter of Wickham (2014a) for further details on classes.

`raster` objects represent categorical variables as integers, so `grain[1, 1]` returns a number that represents a unique identifier, rather than "clay", "silt" or "sand". The raster object stores the corresponding look-up table or "Raster Attribute Table" (RAT) as a data frame in a new slot named `attributes`, which can be viewed with `ratify(grain)` (see `?ratify()` for more information). Use the function `levels()` for retrieving and adding new factor levels to the attribute table:

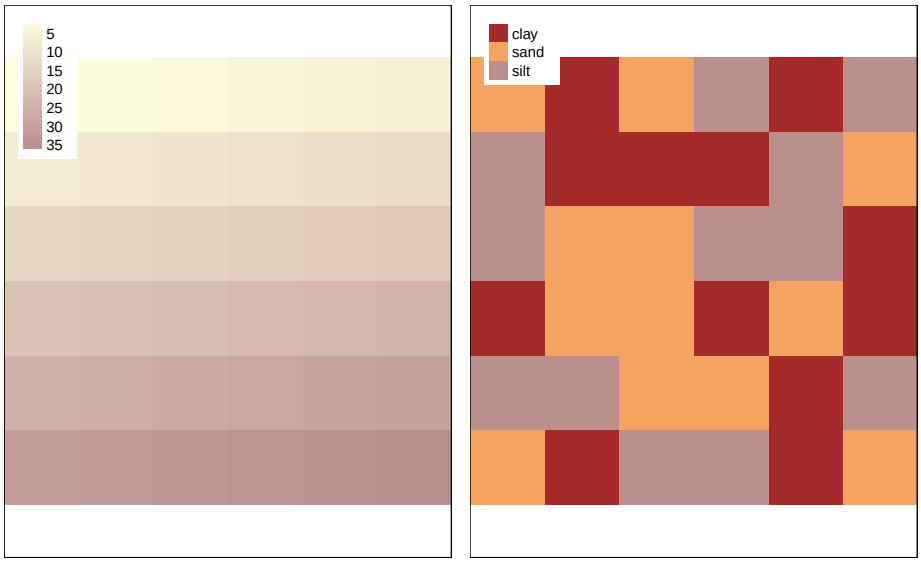


FIGURE 17: Raster datasets with numeric (left) and categorical values (right).

```
levels(grain)[[1]] = cbind(levels(grain)[[1]], wetness = c("wet", "moist", "dry"))
levels(grain)
#> [[1]]
#>   ID VALUE wetness
#> 1  1  clay     wet
#> 2  2  silt   moist
#> 3  3  sand     dry
```

This behavior demonstrates that raster cells can only possess one value, an identifier which can be used to look up the attributes in the corresponding attribute table (stored in a slot named `attributes`). This is illustrated in command below, which returns the grain size and wetness of cell IDs 1, 11 and 35:

```
factorValues(grain, grain[c(1, 11, 35)])
#>   VALUE wetness
#> 1  sand     dry
#> 2  silt   moist
#> 3  clay     wet
```

0.3.3.1 Raster subsetting

Raster subsetting is done with the base R operator `[`, which accepts a variety of inputs:

- Row-column indexing
- Cell IDs
- Coordinates
- Another raster object

Here, we only show the first two options since these can be considered non-spatial operations. If we need a spatial object to subset another or the output is a spatial object, we refer to this as spatial subsetting. Therefore, the latter two options will be shown in the next chapter (see Section **??** in the next chapter).
The first two subsetting options are demonstrated in the commands below — both return the value of the top left pixel in the raster object `elev` (results not shown):

```
# row 1, column 1
elev[1, 1]
# cell ID 1
elev[1]
```

To extract all values or complete rows, you can use `values()` and `getValues()`. For multi-layered raster objects `stack` or `brick`, this will return the cell value(s) for each layer. For example, `stack(elev, grain)[1]` returns a matrix with one row and two columns — one for each layer. For multi-layer raster objects another way to subset is with `raster::subset()`, which extracts layers from a raster stack or brick. The `[[` and `$` operators can also be used:

```
r_stack = stack(elev, grain)
names(r_stack) = c("elev", "grain")
# three ways to extract a layer of a stack
raster::subset(r_stack, "elev")
r_stack[["elev"]]
r_stack$elev
```

Cell values can be modified by overwriting existing values in conjunction with a subsetting operation. The following code chunk, for example, sets the upper left cell of `elev` to 0:

```
elev[1, 1] = 0
elev[]
#>  [1]  0  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
#> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36
```

Leaving the square brackets empty is a shortcut version of `values()` for retrieving all values of a raster. Multiple cells can also be modified in this way:

```
elev[1, 1:2] = 0
```

0.3.3.2 Summarizing raster objects

raster contains functions for extracting descriptive statistics for entire rasters. Printing a raster object to the console by typing its name returns minimum and maximum values of a raster. `summary()` provides common descriptive statistics (minimum, maximum, interquartile range and number of `NA`s). Further summary operations such as the standard deviation (see below) or custom summary statistics can be calculated with `cellStats()`.

```
cellStats(elev, sd)
```

If you provide the `summary()` and `cellStats()` functions with a raster stack or brick object, they will summarize each layer separately, as can be illustrated by running: `summary(brick(elev, grain))`.

Raster value statistics can be visualized in a variety of ways. Specific functions such as `boxplot()`, `density()`, `hist()` and `pairs()` work also with raster objects, as demonstrated in the histogram created with the command below (not shown):

```
hist(elev)
```

In case a visualization function does not work with raster objects, one can extract the raster data to be plotted with the help of `values()` or `getValues()`. Descriptive raster statistics belong to the so-called global raster operations. These and other typical raster processing operations are part of the map algebra scheme, which are covered in the next chapter (Section **??**).

Some function names clash between packages (e.g., `select()`, as discussed in a previous note). In addition to not loading packages by referring to functions verbosely (e.g., `dplyr::select()`), another way to prevent function names clashes is by unloading the offending package with `detach()`. The following command, for example, unloads the **raster** package (this can also be done in the *package* tab which resides by default in the right-bottom pane in RStudio): `detach("package:raster", unload = TRUE, force = TRUE)`. The `force` argument makes sure that the package will be detached even if other packages depend on it. This, however, may lead to a restricted usability of packages depending on the detached package, and is therefore not recommended.

0.3.4 Exercises

For these exercises we will use the `us_states` and `us_states_df` datasets from the **spData** package:

```
library(spData)
data(us_states)
data(us_states_df)
```

`us_states` is a spatial object (of class `sf`), containing geometry and a few attributes (including name, region, area, and population) of states within the contiguous United States. `us_states_df` is a data frame (of class `data.frame`) containing the name and additional variables (including median income and poverty level, for the years 2010 and 2015) of US states, including Alaska, Hawaii and Puerto Rico. The data comes from the United States Census Bureau, and is documented in `?us_states` and `?us_states_df`.

1. Create a new object called `us_states_name` that contains only the `NAME` column from the `us_states` object. What is the class of the new object and what makes it geographic?
2. Select columns from the `us_states` object which contain population data. Obtain the same result using a different command (bonus: try to find three ways of obtaining the same result). Hint: try to use helper functions, such as `contains` or `starts_with` from **dplyr** (see `?contains`).
3. Find all states with the following characteristics (bonus find *and* plot them):
    - Belong to the Midwest region.

- Belong to the West region, have an area below 250,000 km^2 *and* in 2015 a population greater than 5,000,000 residents (hint: you may need to use the function `units::set_units()` or `as.numeric()`).
- Belong to the South region, had an area larger than 150,000 km^2 or a total population in 2015 larger than 7,000,000 residents.

4. What was the total population in 2015 in the `us_states` dataset? What was the minimum and maximum total population in 2015?
5. How many states are there in each region?
6. What was the minimum and maximum total population in 2015 in each region? What was the total population in 2015 in each region?
7. Add variables from `us_states_df` to `us_states`, and create a new object called `us_states_stats`. What function did you use and why? Which variable is the key in both datasets? What is the class of the new object?
8. `us_states_df` has two more rows than `us_states`. How can you find them? (hint: try to use the `dplyr::anti_join()` function)
9. What was the population density in 2015 in each state? What was the population density in 2010 in each state?
10. How much has population density changed between 2010 and 2015 in each state? Calculate the change in percentages and map them.
11. Change the columns' names in `us_states` to lowercase. (Hint: helper functions - `tolower()` and `colnames()` may help.)
12. Using `us_states` and `us_states_df` create a new object called `us_states_sel`. The new object should have only two variables - `median_income_15` and `geometry`. Change the name of the `median_income_15` column to `Income`.
13. Calculate the change in median income between 2010 and 2015 for each state. Bonus: What was the minimum, average and maximum median income in 2015 for each region? What is the region with the largest increase of the median income?
14. Create a raster from scratch with nine rows and columns and a resolution of 0.5 decimal degrees (WGS84). Fill it with random numbers. Extract the values of the four corner cells.
15. What is the most common class of our example raster `grain` (hint: `modal()`)?
16. Plot the histogram and the boxplot of the `data(dem, package = "RQGIS")` raster.

0.4 Spatial data operations

Prerequisites

- This chapter requires the same packages used in Chapter **??**:

```
library(sf)
library(raster)
library(dplyr)
library(spData)
```

0.4.1 Introduction

Spatial operations are a vital part of geocomputation. This chapter shows how spatial objects can be modified in a multitude of ways based on their location and shape. The content builds on the previous chapter because many spatial operations have a non-spatial (attribute) equivalent. This is especially true for *vector* operations: Section **??** on vector attribute manipulation provides the basis for understanding its spatial counterpart, namely spatial subsetting (covered in Section **??**). Spatial joining (Section **??**) and aggregation (Section **??**) also have non-spatial counterparts, covered in the previous chapter.

Spatial operations differ from non-spatial operations in some ways, however. To illustrate the point, imagine you are researching road safety. Spatial joins can be used to find road speed limits related with administrative zones, even when no zone ID is provided. But this raises the question: should the road completely fall inside a zone for its values to be joined? Or is simply crossing or being within a certain distance sufficient? When posing such questions, it becomes apparent that spatial operations differ substantially from attribute operations on data frames: the *type* of spatial relationship between objects must be considered. These are covered in Section **??**, on topological relations.

Another unique aspect of spatial objects is distance. All spatial objects are related through space and distance calculations, covered in Section **??**, can be used to explore the strength of this relationship.

Spatial operations also apply to raster objects. Spatial subsetting of raster objects is covered in Section **??**; merging several raster 'tiles' into a single object is covered in Section **??**. For many applications, the most important spatial operation on raster objects is *map algebra*, as we will see in Sections **??** to **??**. Map algebra is also the prerequisite for distance calculations on rasters, a technique which is covered in Section **??**.

It is important to note that spatial operations that use two spatial objects rely on both objects having the same coordinate reference system, a topic that was introduced in Section **??** and which will be covered in more depth in Chapter **??**.

0.4.2 Spatial operations on vector data

This section provides an overview of spatial operations on vector geographic data represented as simple features in the **sf** package before Section **??**, which presents spatial methods using the **raster** package.

0.4.2.1 Spatial subsetting

Spatial subsetting is the process of selecting features of a spatial object based on whether or not they in some way *relate* in space to another object. It is analogous to *attribute subsetting* (covered in Section **??**) and can be done with the base R square bracket (`[`) operator or with the `filter()` function from the **tidyverse**.

An example of spatial subsetting is provided by the `nz` and `nz_height` datasets in **spData**. These contain projected data on the 16 main regions and 101 highest points in New Zealand, respectively (Figure **??**). The following code chunk first creates an object representing Canterbury, then uses spatial subsetting to return all high points in the region:

```
canterbury = nz %>% filter(Name == "Canterbury")
canterbury_height = nz_height[canterbury, ]
```

Like attribute subsetting `x[y, ]` subsets features of a *target* `x` using the contents of a *source* object `y`. Instead of `y` being of class `logical` or `integer` — a vector of `TRUE` and `FALSE` values or whole numbers — for spatial subsetting it is another spatial (`sf`) object.

Various *topological relations* can be used for spatial subsetting. These determine the type of spatial relationship that features in the target object must have with the subsetting object to be selected, including *touches*, *crosses* or *within* (see Section **??**). *Intersects* is the default spatial subsetting operator, a default that returns `TRUE` for many types of spatial relations, including *touches*, *crosses* and *is within*. These alternative spatial operators can be specified with the `op =` argument, a third argument that can be passed to the `[` operator for `sf` objects.

This is demonstrated in the following command which returns the opposite of `st_intersect()`, points that do not intersect with Canterbury (see Section **??**):

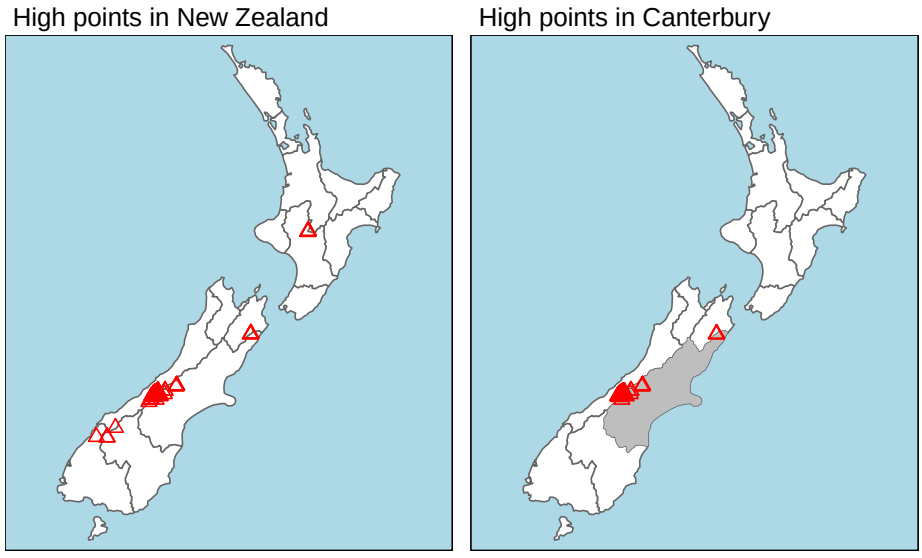


FIGURE 18: Illustration of spatial subsetting with red triangles representing 101 high points in New Zealand, clustered near the central Canterbuy region (left). The points in Canterbury were created with the '[' subsetting operator (highlighted in gray, right).

```
nz_height[canterbury, , op = st_disjoint]
```

Note the empty argument — denoted with `, ,` — in the preceding code chunk is included to highlight `op`, the third argument in `[` for `sf` objects. One can use this to change the subsetting operation in many ways. `nz_height[canterbury, 2, op = st_disjoint]`, for example, returns the same rows but only includes the second attribute column (see `sf:::`[.sf`` and the `?sf` for details).

For many applications, this is all you'll need to know about spatial subsetting for vector data. In this case, you can safely skip to Section **??**.
If you're interested in the details, including other ways of subsetting, read on.
Another way of doing spatial subsetting uses objects returned by *topological operators*. This is demonstrated in the first command below:

```
sel_sgbp = st_intersects(x = nz_height, y = canterbury)
class(sel_sgbp)
```

```
#> [1] "sgbp"
sel_logical = lengths(sel_sgbp) > 0
canterbury_height2 = nz_height[sel_logical, ]
```

In the above code chunk, an object of class `sgbp` (a sparse geometry binary predicate, a list of length `x` in the spatial operation) is created and then converted into a logical vector `sel_logical` (containing only `TRUE` and `FALSE` values). The function `lengths()` identifies which features in `nz_height` intersect with *any* objects in `y`. In this case 1 is the greatest possible value but for more complex operations one could use the method to subset only features that intersect with, for example, 2 or more features from the source object.

Note: another way to return a logical output is by setting `sparse = FALSE` (meaning 'return a dense matrix not a sparse one') in operators such as `st_intersects()`. The command `st_intersects(x = nz_height, y = canterbury, sparse = FALSE)[, 1]`, for example, would return an output identical `sel_logical`. Note: the solution involving `sgbp` objects is more generalisable though, as it works for many-to-many operations and has lower memory requirements.

It should be noted that a logical can also be used with `filter()` as follows (`sparse = FALSE` is explained in Section **??**):

```
canterbury_height3 = nz_height %>%
  filter(st_intersects(x = ., y = canterbury, sparse = FALSE))
```

At this point, there are three versions of `canterbury_height`, one created with spatial subsetting directly and the other two via intermediary selection objects. To explore these objects and spatial subsetting in more detail, see the supplementary vignettes on `subsetting` and `tidverse-pitfalls`[132].

0.4.2.2 Topological relations

Topological relations describe the spatial relationships between objects. To understand them, it helps to have some simple test data to work with. Figure **??** contains a polygon (`a`), a line (`l`) and some points (`p`), which are created in the code below.

[132] `https://geocompr.github.io/geocompkg/articles/`

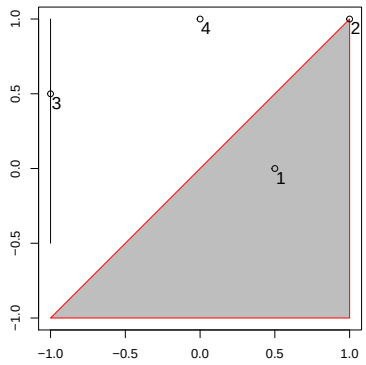


FIGURE 19: Points (p 1 to 4), line and polygon objects arranged to illustrate topological relations.

```
# create a polygon
a_poly = st_polygon(list(rbind(c(-1, -1), c(1, -1), c(1, 1), c(-1, -1))))
a = st_sfc(a_poly)
# create a line
l_line = st_linestring(x = matrix(c(-1, -1, -0.5, 1), ncol = 2))
l = st_sfc(l_line)
# create points
p_matrix = matrix(c(0.5, 1, -1, 0, 0, 1, 0.5, 1), ncol = 2)
p_multi = st_multipoint(x = p_matrix)
p = st_cast(st_sfc(p_multi), "POINT")
```

A simple query is: which of the points in `p` intersect in some way with polygon `a`? The question can be answered by inspection (points 1 and 2 are over or touch the triangle). It can also be answered by using a *spatial predicate* such as *do the objects intersect*? This is implemented in **sf** as follows:

```
st_intersects(p, a)
#> Sparse geometry binary ..., where the predicate was `intersects'
#> 1: 1
#> 2: 1
#> 3: (empty)
#> 4: (empty)
```

The contents of the result should be as you expected: the function returns a

positive (1) result for the first two points, and a negative result (represented by an empty vector) for the last two. What may be unexpected is that the result comes in the form of a list of vectors. This *sparse matrix* output only registers a relation if one exists, reducing the memory requirements of topological operations on multi-feature objects. As we saw in the previous section, a *dense matrix* consisting of `TRUE` or `FALSE` values for each combination of features can also be returned when `sparse = FALSE`:

```
st_intersects(p, a, sparse = FALSE)
#>       [,1]
#> [1,]  TRUE
#> [2,]  TRUE
#> [3,] FALSE
#> [4,] FALSE
```

The output is a matrix in which each row represents a feature in the target object and each column represents a feature in the selecting object. In this case, only the first two features in `p` intersect with `a` and there is only one feature in `a` so the result has only one column. The result can be used for subsetting as we saw in Section **??**.

Note that `st_intersects()` returns `TRUE` for the second feature in the object `p` even though it just touches the polygon `a`: *intersects* is a 'catch-all' topological operation which identifies many types of spatial relation.

The opposite of `st_intersects()` is `st_disjoint()`, which returns only objects that do not spatially relate in any way to the selecting object (note `[, 1]` converts the result into a vector):

```
st_disjoint(p, a, sparse = FALSE)[, 1]
#> [1] FALSE FALSE  TRUE  TRUE
```

`st_within()` returns `TRUE` only for objects that are completely within the selecting object. This applies only to the first object, which is inside the triangular polygon, as illustrated below:

```
st_within(p, a, sparse = FALSE)[, 1]
#> [1]  TRUE FALSE FALSE FALSE
```

Note that although the first point is *within* the triangle, it does not *touch* any part of its border. For this reason `st_touches()` only returns `TRUE` for the second point:

xc

```
st_touches(p, a, sparse = FALSE)[, 1]
#> [1] FALSE  TRUE FALSE FALSE
```

What about features that do not touch, but *almost touch* the selection object? These can be selected using `st_is_within_distance()`, which has an additional `dist` argument. It can be used to set how close target objects need to be before they are selected. Note that although point 4 is one unit of distance from the nearest node of `a` (at point 2 in Figure **??**), it is still selected when the distance is set to 0.9. This is illustrated in the code chunk below, the second line of which converts the lengthy list output into a `logical` object:

```
sel = st_is_within_distance(p, a, dist = 0.9) # can only return a sparse matrix
lengths(sel) > 0
#> [1]  TRUE  TRUE FALSE  TRUE
```

Functions for calculating topological relations use spatial indices to largely speed up spatial query performance. They achieve that using the Sort-Tile-Recursive (STR) algorithm. The `st_join` function, mentioned in the next section, also uses the spatial indexing. You can learn more at `https://www.r-spatial.org/r/2017/06/22/spatial-index.html`.

0.4.2.3 Spatial joining

Joining two non-spatial datasets relies on a shared 'key' variable, as described in Section **??**. Spatial data joining applies the same concept, but instead relies on shared areas of geographic space (it is also know as spatial overlay). As with attribute data, joining adds a new column to the target object (the argument `x` in joining functions), from a source object (`y`).

The process can be illustrated by an example. Imagine you have ten points randomly distributed across the Earth's surface. Of the points that are on land, which countries are they in? Random points to demonstrate spatial joining are created as follows:

```
set.seed(2018) # set seed for reproducibility
(bb_world = st_bbox(world)) # the world's bounds
#>    xmin    ymin    xmax    ymax
#> -180.0  -90.0  180.0   83.6
```

```
random_df = tibble(
  x = runif(n = 10, min = bb_world[1], max = bb_world[3]),
  y = runif(n = 10, min = bb_world[2], max = bb_world[4])
)
random_points = random_df %>%
  st_as_sf(coords = c("x", "y")) %>% # set coordinates
  st_set_crs(4326) # set geographic CRS
```

The scenario is illustrated in Figure **??**. The `random_points` object (top left) has no attribute data, while the `world` (top right) does. The spatial join operation is done by `st_join()`, which adds the `name_long` variable to the points, resulting in `random_joined` which is illustrated in Figure **??** (bottom left — see `04-spatial-join.R`[133]). Before creating the joined dataset, we use spatial subsetting to create `world_random`, which contains only countries that contain random points, to verify the number of country names returned in the joined dataset should be four (see the top right panel of Figure **??**).

```
world_random = world[random_points, ]
nrow(world_random)
#> [1] 4
random_joined = st_join(random_points, world["name_long"])
```

By default, `st_join()` performs a left join (see Section **??**), but it can also do inner joins by setting the argument `left = FALSE`. Like spatial subsetting, the default topological operator used by `st_join()` is `st_intersects()`. This can be changed with the `join` argument (see `?st_join` for details). In the example above, we have added features of a polygon layer to a point layer. In other cases, we might want to join point attributes to a polygon layer. There might be occasions where more than one point falls inside one polygon. In such a case `st_join()` duplicates the polygon feature: it creates a new row for each match.

0.4.2.4 Non-overlapping joins

Sometimes two geographic datasets do not touch but still have a strong geographic relationship enabling joins. The datasets `cycle_hire` and `cycle_hire_osm`, already attached in the **spData** package, provide a good example. Plotting them shows that they are often closely related but they do not touch, as shown in Figure **??**, a base version of which is created with the following code below:

[133] `https://github.com/Robinlovelace/geocompr/blob/master/code/04-spatial-join.R`

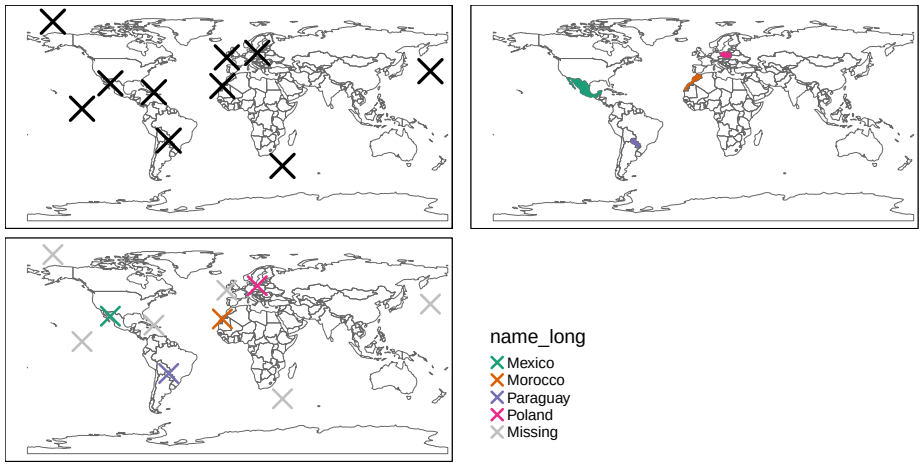


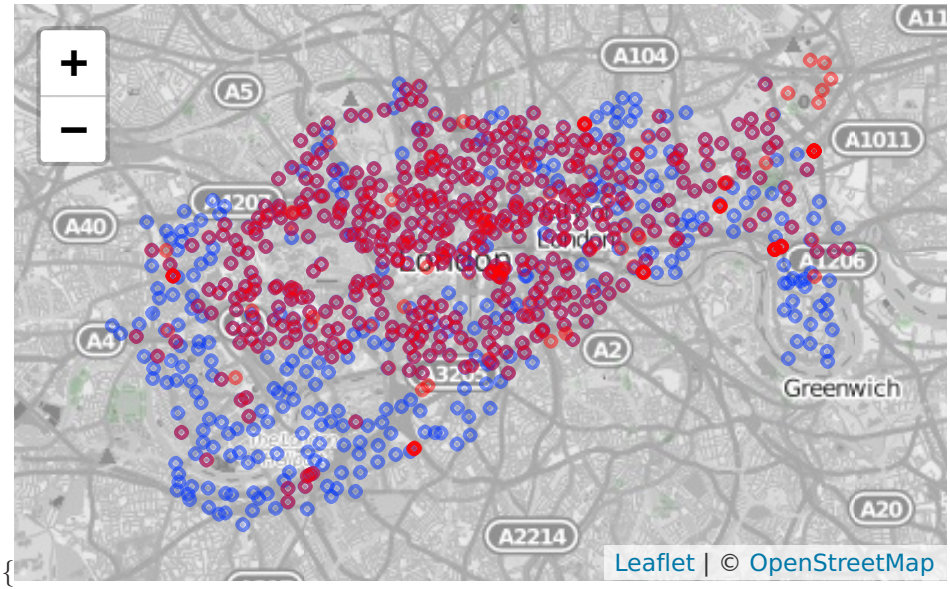
FIGURE 20: Illustration of a spatial join. A new attribute variable is added to random points (top left) from source world object (top right) resulting in the data represented in the final panel.

```
plot(st_geometry(cycle_hire), col = "blue")
plot(st_geometry(cycle_hire_osm), add = TRUE, pch = 3, col = "red")
```

We can check if any points are the same `st_intersects()` as shown below:

```
any(st_touches(cycle_hire, cycle_hire_osm, sparse = FALSE))
#> [1] FALSE
```

\begin{figure}[t]



{

Imagine that we need to join the `capacity` variable in `cycle_hire_osm` onto the official 'target' data contained in `cycle_hire`. This is when a non-overlapping join is needed. The simplest method is to use the topological operator `st_is_within_distance()` shown in Section **??**, using a threshold distance of 20 m. Note that, before performing the relation, both objects are transformed into a projected CRS. These projected objects are created below (note the affix `_P`, short for projected):

```
cycle_hire_P = st_transform(cycle_hire, 27700)
cycle_hire_osm_P = st_transform(cycle_hire_osm, 27700)
sel = st_is_within_distance(cycle_hire_P, cycle_hire_osm_P, dist = 20)
summary(lengths(sel) > 0)
#>    Mode   FALSE    TRUE
#> logical     304     438
```

This shows that there are 438 points in the target object `cycle_hire_P` within the threshold distance of `cycle_hire_osm_P`. How to retrieve the *values* associated with the respective `cycle_hire_osm_P` points? The solution is again with `st_join()`, but with an addition `dist` argument (set to 20 m below):

```
z = st_join(cycle_hire_P, cycle_hire_osm_P, st_is_within_distance, dist = 20)
nrow(cycle_hire)
#> [1] 742
```

```
nrow(z)
#> [1] 762
```

Note that the number of rows in the joined result is greater than the target. This is because some cycle hire stations in `cycle_hire_P` have multiple matches in `cycle_hire_osm_P`. To aggregate the values for the overlapping points and return the mean, we can use the aggregation methods learned in Chapter **??**, resulting in an object with the same number of rows as the target:

```
z = z %>%
  group_by(id) %>%
  summarize(capacity = mean(capacity))
nrow(z) == nrow(cycle_hire)
#> [1] TRUE
```

The capacity of nearby stations can be verified by comparing a plot of the capacity of the source `cycle_hire_osm` data with the results in this new object (plots not shown):

```
plot(cycle_hire_osm["capacity"])
plot(z["capacity"])
```

The result of this join has used a spatial operation to change the attribute data associated with simple features; the geometry associated with each feature has remained unchanged.

0.4.2.5 Spatial data aggregation

Like attribute data aggregation, covered in Section **??**, spatial data aggregation can be a way of *condensing* data. Aggregated data show some statistics about a variable (typically average or total) in relation to some kind of *grouping variable*. Section **??** demonstrated how `aggregate()` and `group_by() %>% summarize()` condense data based on attribute variables. This section demonstrates how the same functions work using spatial grouping variables. Returning to the example of New Zealand, imagine you want to find out the average height of high points in each region. This is a good example of spatial aggregation: it is the geometry of the source (`y` or `nz` in this case) that defines how values in the target object (`x` or `nz_height`) are grouped. This is illustrated using the base `aggregate()` function below:

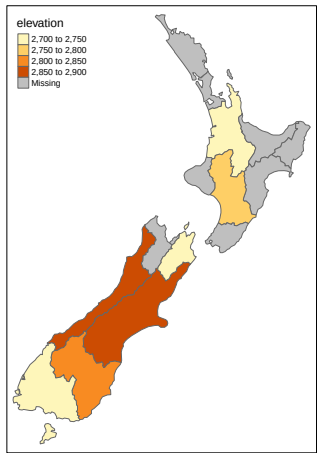


FIGURE 21: Average height of the top 101 high points across the regions of New Zealand.

```
nz_avheight = aggregate(x = nz_height, by = nz, FUN = mean)
```

The result of the previous command is an `sf` object with the same geometry as the (spatial) aggregating object (`nz`).[134] The result of the previous operation is illustrated in Figure **??**. The same result can also be generated using the 'tidy' functions `group_by()` and `summarize()` (used in combination with `st_join()`):

```
nz_avheight2 = nz %>%
  st_join(nz_height) %>%
  group_by(Name) %>%
  summarize(elevation = mean(elevation, na.rm = TRUE))
```

The resulting `nz_avheight` objects have the same geometry as the aggregating object `nz` but with a new column representing the mean average height of points within each region of New Zealand (other summary functions such as `median()` and `sd()` can be used in place of `mean()`). Note that regions containing no points have an associated `elevation` value of `NA`. For aggregating operations which also create new geometries, see Section **??**.

Spatial congruence is an important concept related to spatial aggregation. An *aggregating object* (which we will refer to as `y`) is *congruent* with the target object (`x`) if the two objects have shared borders. Often this is the case for administrative boundary data, whereby larger units — such as Middle Layer

[134]This can be verified with `identical(st_geometry(nz), st_geometry(nz_avheight))`.

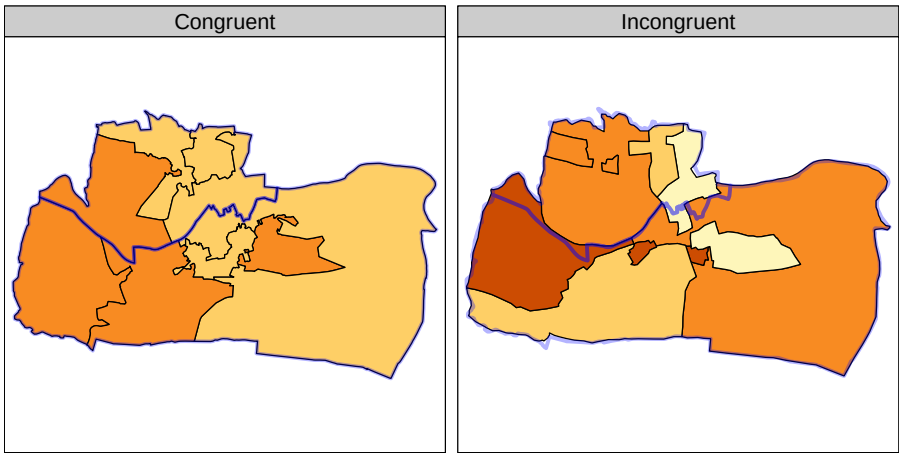


FIGURE 22: Illustration of congruent (left) and incongruent (right) areal units with respect to larger aggregating zones (translucent blue borders).

Super Output Areas (MSOAs[135]) in the UK or districts in many other European countries — are composed of many smaller units.

Incongruent aggregating objects, by contrast, do not share common borders with the target (Qiu, Zhang, and Zhou 2012). This is problematic for spatial aggregation (and other spatial operations) illustrated in Figure **??**. Areal interpolation overcomes this issue by transferring values from one set of areal units to another. Algorithms developed for this task include area weighted and 'pycnophylactic' areal interpolation methods (Tobler 1979).

The **spData** package contains a dataset named `incongruent` (colored polygons with black borders in the right panel of Figure **??**) and a dataset named `aggregating_zones` (the two polygons with the translucent blue border in the right panel of Figure **??**). Let us assume that the `value` column of `incongruent` refers to the total regional income in million Euros. How can we transfer the values of the underlying nine spatial polygons into the two polygons of `aggregating_zones`?

The simplest useful method for this is *area weighted* spatial interpolation. In this case values from the `incongruent` object are allocated to the `aggregating_zones` in proportion to area; the larger the spatial intersection between input and output features, the larger the corresponding value. For instance, if one intersection of `incongruent` and `aggregating_zones` is 1.5 km^2 but the whole incongruent polygon in question has 2 km^2 and a total income of 4 million Euros, then the target aggregating zone will obtain three quarters of

[135] `https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography`

the income, in this case 3 million Euros. This is implemented in `st_interpolate_aw()`, as demonstrated in the code chunk below.

```
agg_aw = st_interpolate_aw(incongruent[, "value"], aggregating_zones,
                           extensive = TRUE)
#> Warning in st_interpolate_aw.sf(incongruent[, "value"],
#> aggregating_zones, : st_interpolate_aw assumes attributes are constant over
#> areas of x
# show the aggregated result
agg_aw$value
#> [1] 19.6 25.7
```

In our case it is meaningful to sum up the values of the intersections falling into the aggregating zones since total income is a so-called spatially extensive variable. This would be different for spatially intensive variables, which are independent of the spatial units used, such as income per head or percentages[136]. In this case it is more meaningful to apply an average function when doing the aggregation instead of a sum function. To do so, one would only have to set the `extensive` parameter to `FALSE`.

0.4.2.6 Distance relations

While topological relations are binary — a feature either intersects with another or does not — distance relations are continuous. The distance between two objects is calculated with the `st_distance()` function. This is illustrated in the code chunk below, which finds the distance between the highest point in New Zealand and the geographic centroid of the Canterbury region, created in Section **??**:

```
nz_heighest = nz_height %>% top_n(n = 1, wt = elevation)
canterbury_centroid = st_centroid(canterbury)
st_distance(nz_heighest, canterbury_centroid)
#> Units: [m]
#>        [,1]
#> [1,] 115540
```

There are two potentially surprising things about the result:

- It has `units`, telling us the distance is 100,000 meters, not 100,000 inches, or any other measure of distance.

[136] `http://ibis.geog.ubc.ca/courses/geob370/notes/intensive_extensive.htm`

- It is returned as a matrix, even though the result only contains a single value.

This second feature hints at another useful feature of `st_distance()`, its ability to return *distance matrices* between all combinations of features in objects `x` and `y`. This is illustrated in the command below, which finds the distances between the first three features in `nz_height` and the Otago and Canterbury regions of New Zealand represented by the object `co`.

```
co = filter(nz, grepl("Canter|Otag", Name))
st_distance(nz_height[1:3, ], co)
#> Units: [m]
#>        [,1]  [,2]
#> [1,] 123537 15498
#> [2,]  94283     0
#> [3,]  93019     0
```

Note that the distance between the second and third features in `nz_height` and the second feature in `co` is zero. This demonstrates the fact that distances between points and polygons refer to the distance to *any part of the polygon*: The second and third points in `nz_height` are *in* Otago, which can be verified by plotting them (result not shown):

```
plot(st_geometry(co)[2])
plot(st_geometry(nz_height)[2:3], add = TRUE)
```

0.4.3 Spatial operations on raster data

This section builds on Section **??**, which highlights various basic methods for manipulating raster datasets, to demonstrate more advanced and explicitly spatial raster operations, and uses the objects `elev` and `grain` manually created in Section **??**. For the reader's convenience, these datasets can be also found in the **spData** package.

0.4.3.1 Spatial subsetting

The previous chapter (Section **??**) demonstrated how to retrieve values associated with specific cell IDs or row and column combinations. Raster objects can also be extracted by location (coordinates) and other spatial objects. To use coordinates for subsetting, one can 'translate' the coordinates into a cell ID with the **raster** function `cellFromXY()`. An alternative is to use `raster::extract()` (be careful, there is also a function called `extract()` in

the **tidyverse**) to extract values. Both methods are demonstrated below to find the value of the cell that covers a point located 0.1 units from the origin.

```
id = cellFromXY(elev, xy = c(0.1, 0.1))
elev[id]
# the same as
raster::extract(elev, data.frame(x = 0.1, y = 0.1))
```

It is convenient that both functions also accept objects of class `Spatial*Objects`. Raster objects can also be subset with another raster object, as illustrated in Figure **??** (left panel) and demonstrated in the code chunk below:

```
clip = raster(xmn = 0.9, xmx = 1.8, ymn = -0.45, ymx = 0.45,
              res = 0.3, vals = rep(1, 9))
elev[clip]
#> [1] 18 24
# we can also use extract
# extract(elev, extent(clip))
```

Basically, this amounts to retrieving the values of the first raster (here: `elev`) falling within the extent of a second raster (here: `clip`).
So far, the subsetting returned the values of specific cells, however, when doing spatial subsetting, one often also expects a spatial object as an output. To do this, we can use again the `[` when we additionally set the `drop` parameter to `FALSE`. To illustrate this, we retrieve the first two cells of `elev` as an individual raster object. As mentioned in Section **??**, the `[` operator accepts various inputs to subset rasters and returns a raster object when `drop = FALSE`. The code chunk below subsets the `elev` raster by cell ID and row-column index with identical results: the first two cells on the top row (only the first 2 lines of the output is shown):

```
elev[1:2, drop = FALSE]    # spatial subsetting with cell IDs
elev[1, 1:2, drop = FALSE] # spatial subsetting by row,column indices
#> class       : RasterLayer
#> dimensions  : 1, 2, 2  (nrow, ncol, ncell)
#> ...
```

Another common use case of spatial subsetting is when a raster with `logical` (or `NA`) values is used to mask another raster with the same extent and

c

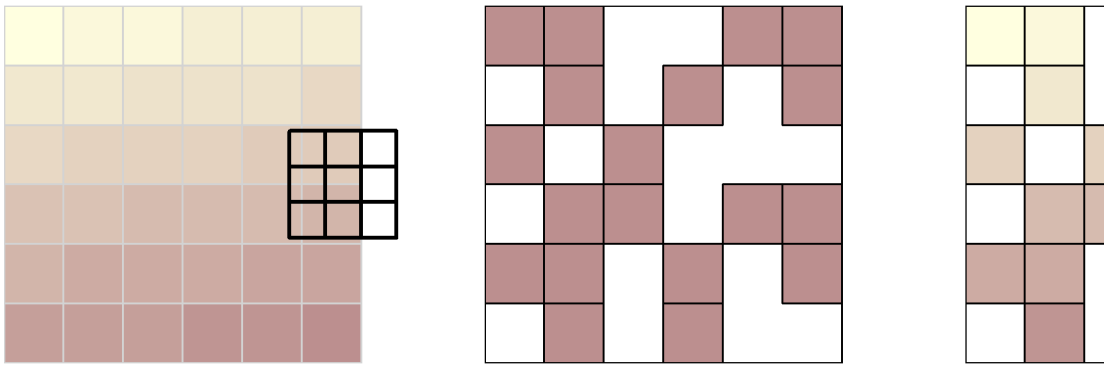


FIGURE 23: Subsetting raster values with the help of another raster (left). Raster mask (middle). Output of masking a raster (right).

resolution, as illustrated in Figure **??**, middle and right panel. In this case, the `[`, `mask()` and `overlay()` functions can be used (results not shown):

```
# create raster mask
rmask = elev
values(rmask) = sample(c(NA, TRUE), 36, replace = TRUE)

# spatial subsetting
elev[rmask, drop = FALSE]           # with [ operator
mask(elev, rmask)                   # with mask()
overlay(elev, rmask, fun = "max")   # with overlay
```

In the code chunk above, we have created a mask object called `rmask` with values randomly assigned to `NA` and `TRUE`. Next, we want to keep those values of `elev` which are `TRUE` in `rmask`. In other words, we want to mask `elev` with `rmask`. These operations are in fact Boolean local operations since we compare cell-wise two rasters. The next subsection explores these and related operations in more detail.

0.4.3.2 Map algebra

Map algebra makes raster processing really fast. This is because raster datasets only implicitly store coordinates. To derive the coordinate of a specific cell, we have to calculate it using its matrix position and the raster resolution and origin. For the processing, however, the geographic position of a cell is barely relevant as long as we make sure that the cell position is still the same after the processing (one-to-one locational correspondence). Additionally, if two or more raster datasets share the same extent, projection and resolution, one could treat them as matrices for the processing. This is exactly what map algebra is doing in R. First, the **raster** package checks the headers of the rasters on which to perform any algebraic operation, and only if they are correspondent to each other, the processing goes on.[137] And secondly, map algebra retains the so-called one-to-one locational correspondence. This is where it substantially differs from matrix algebra which changes positions when for example multiplying or dividing matrices.

Map algebra (or cartographic modeling) divides raster operations into four subclasses (Tomlin 1990), with each working on one or several grids simultaneously:

1. *Local* or per-cell operations.
2. *Focal* or neighborhood operations. Most often the output cell value is the result of a 3 x 3 input cell block.
3. *Zonal* operations are similar to focal operations, but the surrounding pixel grid on which new values are computed can have irregular sizes and shapes.
4. *Global* or per-raster operations; that means the output cell derives its value potentially from one or several entire rasters.

This typology classifies map algebra operations by the number/shape of cells used for each pixel processing step. For the sake of completeness, we should mention that raster operations can also be classified by discipline such as terrain, hydrological analysis or image classification. The following sections explain how each type of map algebra operations can be used, with reference to worked examples (also see `vignette("Raster")` for a technical description of map algebra).

[137] Map algebra operations are also possible with headerless rasters; in this case the user has to make sure that in fact there exists a one-to-one locational correspondence. An example showing how to import a headerless raster into R is provided in a post at `https://stat.ethz.ch/pipermail/r-sig-geo/2013-May/018278.html`.

0.4.3.3 Local operations

Local operations comprise all cell-by-cell operations in one or several layers. A good example is the classification of intervals of numeric values into groups such as grouping a digital elevation model into low (class 1), middle (class 2) and high elevations (class 3). Using the `reclassify()` command, we need first to construct a reclassification matrix, where the first column corresponds to the lower and the second column to the upper end of the class. The third column represents the new value for the specified ranges in column one and two. Here, we assign the raster values in the ranges 0–12, 12–24 and 24–36 are *reclassified* to take values 1, 2 and 3, respectively.

```
rcl = matrix(c(0, 12, 1, 12, 24, 2, 24, 36, 3), ncol = 3, byrow = TRUE)
recl = reclassify(elev, rcl = rcl)
```

We will perform several reclassifactions in Chapter **??**.

Raster algebra is another classical use case of local operations. This includes adding, subtracting and squaring two rasters. Raster algebra also allows logical operations such as finding all raster cells that are greater than a specific value (5 in our example below). The **raster** package supports all these operations and more, as described in `vignette("Raster")` and demonstrated below (results not show):

```
elev + elev
elev^2
log(elev)
elev > 5
```

Instead of arithmetic operators, one can also use the `calc()` and `overlay()` functions. These functions are more efficient, hence, they are preferable in the presence of large raster datasets. Additionally, they allow you to directly store an output file.

The calculation of the normalized difference vegetation index (NDVI) is a well-known local (pixel-by-pixel) raster operation. It returns a raster with values between -1 and 1; positive values indicate the presence of living plants (mostly > 0.2). NDVI is calculated from red and near-infrared (NIR) bands of remotely sensed imagery, typically from satellite systems such as Landsat or Sentinel. Vegetation absorbs light heavily in the visible light spectrum, and especially in the red channel, while reflecting NIR light, explaining the NVDI formula:

$$NDVI = \frac{\text{NIR} - \text{Red}}{\text{NIR} + \text{Red}}$$

Predictive mapping is another interesting application of local raster operations. The response variable corresponds to measured or observed points in space, for example, species richness, the presence of landslides, tree disease or crop yield. Consequently, we can easily retrieve space- or airborne predictor variables from various rasters (elevation, pH, precipitation, temperature, landcover, soil class, etc.). Subsequently, we model our response as a function of our predictors using `lm`, `glm`, `gam` or a machine-learning technique. Spatial predictions on raster objects can therefore be made by applying estimated coefficients to the predictor raster vaules, and summing the output raster values (see Chapter **??**).

0.4.3.4 Focal operations

While local functions operate on one cell, though possibly from multiple layers, **focal** operations take into account a central cell and its neighbors. The neighborhood (also named kernel, filter or moving window) under consideration is typically of size 3-by-3 cells (that is the central cell and its eight surrounding neighbors), but can take on any other (not necessarily rectangular) shape as defined by the user. A focal operation applies an aggregation function to all cells within the specified neighborhood, uses the corresponding output as the new value for the the central cell, and moves on to the next central cell (Figure **??**). Other names for this operation are spatial filtering and convolution (Burrough, McDonnell, and Lloyd 2015).

In R, we can use the `focal()` function to perform spatial filtering. We define the shape of the moving window with a `matrix` whose values correspond to weights (see `w` parameter in the code chunk below). Secondly, the `fun` parameter lets us specify the function we wish to apply to this neighborhood. Here, we choose the minimum, but any other summary function, including `sum()`, `mean()`, or `var()` can be used.

```
r_focal = focal(elev, w = matrix(1, nrow = 3, ncol = 3), fun = min)
```

We can quickly check if the output meets our expectations. In our example, the minimum value has to be always the upper left corner of the moving window (remember we have created the input raster by row-wise incrementing the cell values by one starting at the upper left corner). In this example, the weighting matrix consists only of 1s, meaning each cell has the same weight on the output, but this can be changed.

Focal functions or filters play a dominant role in image processing. Low-pass or

smoothing filters use the mean function to remove extremes. In the case of categorical data, we can replace the mean with the mode, which is the most common value. By contrast, high-pass filters accentuate features. The line detection Laplace and Sobel filters might serve as an example here. Check the `focal()` help page for how to use them in R (this will also be used in the excercises at the end of this chapter).

Terrain processing, the calculation of topographic characteristics such as slope, aspect and flow directions, relies on focal functions. `terrain()` can be used to calculate these metrics, although some terrain algorithms, including the Zevenbergen and Thorne method to compute slope, are not implemented in this **raster** function. Many other algorithms — including curvatures, contributing areas and wetness indices — are implemented in open source desktop geographic information system (GIS) software. Chapter **??** shows how to access such GIS functionality from within R.

0.4.3.5 Zonal operations

Zonal operations are similar to focal operations. The difference is that zonal filters can take on any shape instead of a predefined rectangular window. Our grain size raster is a good example (Figure **??**) because the different grain sizes are spread in an irregular fashion throughout the raster.

To find the mean elevation for each grain size class, we can use the `zonal()` command. This kind of operation is also known as *zonal statistics* in the GIS world.

```
z = zonal(elev, grain, fun = "mean") %>%
  as.data.frame()
z
#>   zone mean
#> 1    1 17.8
#> 2    2 18.5
#> 3    3 19.2
```

This returns the statistics for each category, here the mean altitude for each grain size class, and can be added to the attribute table of the ratified raster (see previous chapter).

0.4.3.6 Global operations and distances

Global operations are a special case of zonal operations with the entire raster dataset representing a single zone. The most common global operations are descriptive statistics for the entire raster dataset such as the minimum or

maximum (see Section **??**). Aside from that, global operations are also useful for the computation of distance and weight rasters. In the first case, one can calculate the distance from each cell to a specific target cell. For example, one might want to compute the distance to the nearest coast (see also `raster::distance()`). We might also want to consider topography, that means, we are not only interested in the pure distance but would like also to avoid the crossing of mountain ranges when going to the coast. To do so, we can weight the distance with elevation so that each additional altitudinal meter 'prolongs' the Euclidean distance. Visibility and viewshed computations also belong to the family of global operations (in the exercises of Chapter **??**, you will compute a viewshed raster).

Many map algebra operations have a counterpart in vector processing (Liu and Mason 2009). Computing a distance raster (zonal operation) while only considering a maximum distance (logical focal operation) is the equivalent to a vector buffer operation (Section **??**). Reclassifying raster data (either local or zonal function depending on the input) is equivalent to dissolving vector data (Section **??**). Overlaying two rasters (local operation), where one contains `NULL` or `NA` values representing a mask, is similar to vector clipping (Section **??**). Quite similar to spatial clipping is intersecting two layers (Section **??**). The difference is that these two layers (vector or raster) simply share an overlapping area (see Figure **??** for an example). However, be careful with the wording. Sometimes the same words have slightly different meanings for raster and vector data models. Aggregating in the case of vector data refers to dissolving polygons, while it means increasing the resolution in the case of raster data. In fact, one could see dissolving or aggregating polygons as decreasing the resolution. However, zonal operations might be the better raster equivalent compared to changing the cell resolution. Zonal operations can dissolve the cells of one raster in accordance with the zones (categories) of another raster using an aggregation function (see above).

0.4.3.7 Merging rasters

Suppose we would like to compute the NDVI (see Section **??**), and additionally want to compute terrain attributes from elevation data for observations within a study area. Such computations rely on remotely sensed information. The corresponding imagery is often divided into scenes covering a specific spatial extent. Frequently, a study area covers more than one scene. In these cases we would like to merge the scenes covered by our study area. In the easiest case, we can just merge these scenes, that is put them side by side. This is possible with digital elevation data (SRTM, ASTER). In the following code chunk we first download the SRTM elevation data for Austria and Switzerland (for the

country codes, see the **raster** function `ccodes()`). In a second step, we merge the two rasters into one.

```
aut = getData("alt", country = "AUT", mask = TRUE)
ch = getData("alt", country = "CHE", mask = TRUE)
aut_ch = merge(aut, ch)
```

Raster's `merge()` command combines two images, and in case they overlap, it uses the value of the first raster. You can do exactly the same with `gdalUtils::mosaic_rasters()` which is faster, and therefore recommended if you have to merge a multitude of large rasters stored on disk.

The merging approach is of little use when the overlapping values do not correspond to each other. This is frequently the case when you want to combine spectral imagery from scenes that were taken on different dates. The `merge()` command will still work but you will see a clear border in the resulting image. The `mosaic()` command lets you define a function for the overlapping area. For instance, we could compute the mean value. This might smooth the clear border in the merged result but it will most likely not make it disappear. To do so, we need a more advanced approach. Remote sensing scientists frequently apply histogram matching or use regression techniques to align the values of the first image with those of the second image. The packages **landsat** (`histmatch()`, `relnorm()`, `PIF()`), **satellite** (`calcHistMatch()`) and **RStoolbox** (`histMatch()`, `pifMatch()`) provide the corresponding functions. For a more detailed introduction on how to use R for remote sensing, we refer the reader to Wegmann, Leutner, and Dech (2016).

0.4.4 Exercises

1. It was established in Section **??** that Canterbury was the region of New Zealand containing most of the 100 highest points in the country. How many of these high points does the Canterbury region contain?
2. Which region has the second highest number of `nz_height` points in, and how many does it have?
3. Generalizing the question to all regions: how many of New Zealand's 16 regions contain points which belong to the top 100 highest points in the country? Which regions?
    - Bonus: create a table listing these regions in order of the number of points and their name.
4. Use `data(dem, package = "RQGIS")`, and reclassify the elevation in

three classes: low, medium and high. Secondly, attach the NDVI raster (`data(ndvi, package = "RQGIS")`) and compute the mean NDVI and the mean elevation for each altitudinal class.

5. Apply a line detection filter to `raster(system.file("external/rlogo.grd", package = "raster"))`. Plot the result. Hint: Read `?raster::focal()`.
6. Calculate the NDVI of a Landsat image. Use the Landsat image provided by the **spDataLarge** package (`system.file("raster/landsat.tif", package="spDataLarge")`).
7. A StackOverflow post[138] shows how to compute distances to the nearest coastline using `raster::distance()`. Retrieve a digital elevation model of Spain, and compute a raster which represents distances to the coast across the country (hint: use `getData()`). Second, use a simple approach to weight the distance raster with elevation (other weighting approaches are possible, include flow direction and steepness); every 100 altitudinal meters should increase the distance to the coast by 10 km. Finally, compute the difference between the raster using the Euclidean distance and the raster weighted by elevation. Note: it may be wise to increase the cell size of the input raster to reduce compute time during this operation.

0.5 Geometry operations

Prerequisites

- This chapter uses the same packages as Chapter **??** but with the addition of **spDataLarge**, which was installed in Chapter **??**:

```
library(sf)
library(raster)
library(dplyr)
library(spData)
library(spDataLarge)
```

[138] `https://stackoverflow.com/questions/35555709/global-raster-of-geographic-distances`

0.5.1 Introduction

The previous three chapters have demonstrated how geographic datasets are structured in R (Chapter **??**) and how to manipulate them based on their non-geographic attributes (Chapter **??**) and spatial properties (Chapter **??**). This chapter extends these skills. After reading it — and attempting the exercises at the end — you should understand and have control over the geometry column in `sf` objects and the geographic location of pixels represented in rasters.

Section **??** covers transforming vector geometries with 'unary' and 'binary' operations. Unary operations work on a single geometry in isolation. This includes simplification (of lines and polygons), the creation of buffers and centroids, and shifting/scaling/rotating single geometries using 'affine transformations' (Sections **??** to **??**). Binary transformations modify one geometry based on the shape of another. This includes clipping and geometry unions, covered in Sections **??** and **??**, respectively. Type transformations (from a polygon to a line, for example) are demonstrated in Section **??**.

Section **??** covers geometric transformations on raster objects. This involves changing the size and number of the underlying pixels, and assigning them new values. It teaches how to change the resolution (also called raster aggregation and disaggregation), the extent and the origin of a raster. These operations are especially useful if one would like to align raster datasets from diverse sources.

Aligned raster objects share a one-to-one correspondence between pixels, allowing them to be processed using map algebra operations, described in Section **??**. The final Section **??** connects vector and raster objects. It shows how raster values can be 'masked' and 'extracted' by vector geometries. Importantly it shows how to 'polygonize' rasters and 'rasterize' vector datasets, making the two data models more interchangeable.

0.5.2 Geometric operations on vector data

This section is about operations that in some way change the geometry of vector (`sf`) objects. It is more advanced than the spatial data operations presented in the previous chapter (in Section **??**), because here we drill down into the geometry: the functions discussed in this section work on objects of class `sfc` in addition to objects of class `sf`.

0.5.2.1 Simplification

Simplification is a process for generalization of vector objects (lines and polygons) usually for use in smaller scale maps. Another reason for simplifying objects is to reduce the amount of memory, disk space and network bandwidth they consume: it may be wise to simplify complex geometries before publishing

them as interactive maps. The **sf** package provides `st_simplify()`, which uses the GEOS implementation of the Douglas-Peucker algorithm to reduce the vertex count. `st_simplify()` uses the `dTolerance` to control the level of generalization in map units (see Douglas and Peucker 1973 for details). Figure **??** illustrates simplification of a `LINESTRING` geometry representing the river Seine and tributaries. The simplified geometry was created by the following command:

```
seine_simp = st_simplify(seine, dTolerance = 2000)  # 2000 m
```

The resulting `seine_simp` object is a copy of the original `seine` but with fewer vertices. This is apparent, with the result being visually simpler (Figure **??**, right) and consuming less memory than the original object, as verified below:

```
object.size(seine)
#> 17304 bytes
object.size(seine_simp)
#> 8320 bytes
```

Simplification is also applicable for polygons. This is illustrated using `us_states`, representing the contiguous United States. As we show in Chapter **??**, GEOS assumes that the data is in a projected CRS and this could lead to unexpected results when using a geographic CRS. Therefore, the first step is to project the data into some adequate projected CRS, such as US National Atlas Equal Area (epsg = 2163) (on the left in Figure **??**):

```
us_states2163 = st_transform(us_states, 2163)
```

`st_simplify()` works equally well with projected polygons:

```
us_states_simp1 = st_simplify(us_states2163, dTolerance = 100000)  # 100 km
```

A limitation with `st_simplify()` is that it simplifies objects on a per-geometry basis. This means the 'topology' is lost, resulting in overlapping and 'holy' areal units illustrated in Figure **??** (middle panel). `ms_simplify()` from **rmapshaper** provides an alternative that overcomes this issue. By default it uses the Visvalingam algorithm, which overcomes some limitations of the Douglas-Peucker algorithm (Visvalingam and Whyatt 1993). The following code

cx

chunk uses this function to simplify `us_states2163`. The result has only 1% of the vertices of the input (set using the argument `keep`) but its number of objects remains intact because we set `keep_shapes = TRUE`:[139]

```
# proportion of points to retain (0-1; default 0.05)
us_states2163$AREA = as.numeric(us_states2163$AREA)
us_states_simp2 = rmapshaper::ms_simplify(us_states2163, keep = 0.01,
                                          keep_shapes = TRUE)
```

Finally, the visual comparison of the original dataset and the two simplified versions shows differences between the Douglas-Peucker (`st_simplify`) and Visvalingam (`ms_simplify`) algorithm outputs (Figure **??**):

0.5.2.2 Centroids

Centroid operations identify the center of geographic objects. Like statistical measures of central tendency (including mean and median definitions of 'average'), there are many ways to define the geographic center of an object. All of them create single point representations of more complex vector objects. The most commonly used centroid operation is the *geographic centroid*. This type of centroid operation (often referred to as 'the centroid') represents the center of mass in a spatial object (think of balancing a plate on your finger). Geographic centroids have many uses, for example to create a simple point representation of complex geometries, or to estimate distances between polygons. They can be calculated with the **sf** function `st_centroid()` as demonstrated in the code below, which generates the geographic centroids of regions in New Zealand and tributaries to the River Seine, illustrated with black points in Figure **??**.

```
nz_centroid = st_centroid(nz)
seine_centroid = st_centroid(seine)
```

Sometimes the geographic centroid falls outside the boundaries of their parent objects (think of a doughnut). In such cases *point on surface* operations can be used to guarantee the point will be in the parent object (e.g., for labeling irregular multipolygon objects such as island states), as illustrated by the red

[139] Simplification of multipolygon objects can remove small internal polygons, even if the `keep_shapes` argument is set to TRUE. To prevent this, you need to set `explode = TRUE`. This option converts all mutlipolygons into separate polygons before its simplification.

points in Figure **??**. Notice that these red points always lie on their parent objects. They were created with `st_point_on_surface()` as follows:[140]

```
nz_pos = st_point_on_surface(nz)
seine_pos = st_point_on_surface(seine)
```

Other types of centroids exist, including the *Chebyshev center* and the *visual center*. We will not explore these here but it is possible to calculate them using R, as we'll see in Chapter **??**.

0.5.2.3 Buffers

Buffers are polygons representing the area within a given distance of a geometric feature: regardless of whether the input is a point, line or polygon, the output is a polygon. Unlike simplification (which is often used for visualization and reducing file size) buffering tends to be used for geographic data analysis. How many points are within a given distance of this line? Which demographic groups are within travel distance of this new shop? These kinds of questions can be answered and visualized by creating buffers around the geographic entities of interest.

Figure **??** illustrates buffers of different sizes (5 and 50 km) surrounding the river Seine and tributaries. These buffers were created with commands below, which show that the command `st_buffer()` requires at least two arguments: an input geometry and a distance, provided in the units of the CRS (in this case meters):

```
seine_buff_5km = st_buffer(seine, dist = 5000)
seine_buff_50km = st_buffer(seine, dist = 50000)
```

The third and final argument of `st_buffer()` is `nQuadSegs`, which means 'number of segments per quadrant' and is set by default to 30 (meaning circles created by buffers are composed of $4 \times 30 = 120$ lines). This argument rarely needs to be set. Unusual cases where it may be useful include when the memory consumed by the output of a buffer operation is a major concern (in which case it should be reduced) or when very high precision is needed (in which case it should be increased).

[140] A description of how `st_point_on_surface()` works is provided at `https://gis.stackexchange.com/q/76498`.

0.5.2.4 Affine transformations

Affine transformation is any transformation that preserves lines and parallelism. However, angles or length are not necessarily preserved. Affine transformations include, among others, shifting (translation), scaling and rotation. Additionally, it is possible to use any combination of these. Affine transformations are an essential part of geocomputation. For example, shifting is needed for labels placement, scaling is used in non-contiguous area cartograms (see Section **??**), and many affine transformations are applied when reprojecting or improving the geometry that was created based on a distorted or wrongly projected map. The **sf** package implements affine transformation for objects of classes `sfg` and `sfc`.

```
nz_sfc = st_geometry(nz)
```

Shifting moves every point by the same distance in map units. It could be done by adding a numerical vector to a vector object. For example, the code below shifts all y-coordinates by 100,000 meters to the north, but leaves the x-coordinates untouched (left panel of Figure **??**).

```
nz_shift = nz_sfc + c(0, 100000)
```

Scaling enlarges or shrinks objects by a factor. It can be applied either globally or locally. Global scaling increases or decreases all coordinates values in relation to the origin coordinates, while keeping all geometries topological relations intact. It can be done by subtraction or multiplication of a`sfg` or `sfc` object. Local scaling treats geometries independently and requires points around which geometries are going to be scaled, e.g., centroids. In the example below, each geometry is shrunk by a factor of two around the centroids (middle panel in Figure **??**). To achieve that, each object is firstly shifted in a way that its center has coordinates of `0, 0` (`(nz_sfc - nz_centroid_sfc)`). Next, the sizes of the geometries are reduced by half (`* 0.5`). Finally, each object's centroid is moved back to the input data coordinates (`+ nz_centroid_sfc`).

```
nz_centroid_sfc = st_centroid(nz_sfc)
nz_scale = (nz_sfc - nz_centroid_sfc) * 0.5 + nz_centroid_sfc
```

Rotation of two-dimensional coordinates requires a rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

It rotates points in a counterclockwise direction. The rotation matrix can be implemented in R as:

```
rotation = function(a){
  r = a * pi / 180 #degrees to radians
  matrix(c(cos(r), sin(r), -sin(r), cos(r)), nrow = 2, ncol = 2)
}
```

The `rotation` function accepts one argument `a` - a rotation angle in degrees. Rotation could be done around selected points, such as centroids (right panel of Figure **??**). See `vignette("sf3")` for more examples.

```
nz_rotate = (nz_sfc - nz_centroid_sfc) * rotation(30) + nz_centroid_sfc
```

Finally, the newly created geometries can replace the old ones with the `st_set_geometry()` function:

```
nz_scale_sf = st_set_geometry(nz, nz_scale)
```

0.5.2.5 Clipping

Spatial clipping is a form of spatial subsetting that involves changes to the `geometry` columns of at least some of the affected features.
Clipping can only apply to features more complex than points: lines, polygons and their 'multi' equivalents. To illustrate the concept we will start with a simple example: two overlapping circles with a center point one unit away from each other and a radius of one (Figure **??**).

```
b = st_sfc(st_point(c(0, 1)), st_point(c(1, 1))) # create 2 points
b = st_buffer(b, dist = 1) # convert points to circles
plot(b)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y")) # add text
```

Imagine you want to select not one circle or the other, but the space covered by both `x` *and* `y`. This can be done using the function `st_intersection()`, illustrated using objects named `x` and `y` which represent the left- and right-hand circles (Figure **??**).

```
x = b[1]
y = b[2]
x_and_y = st_intersection(x, y)
plot(b)
plot(x_and_y, col = "lightgrey", add = TRUE) # color intersecting area
```

The subsequent code chunk demonstrates how this works for all combinations of the 'Venn' diagram representing `x` and `y`, inspired by Figure 5.1[141] of the book *R for Data Science* (Grolemund and Wickham 2016).
To illustrate the relationship between subsetting and clipping spatial data, we will subset points that cover the bounding box of the circles `x` and `y` in Figure **??**. Some points will be inside just one circle, some will be inside both and some will be inside neither. `st_sample()` is used below to generate a *simple random* distribution of points within the extent of circles `x` and `y`, resulting in output illustrated in Figure **??**.

```
bb = st_bbox(st_union(x, y))
box = st_as_sfc(bb)
set.seed(2017)
p = st_sample(x = box, size = 10)
plot(box)
plot(x, add = TRUE)
plot(y, add = TRUE)
plot(p, add = TRUE)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"))
```

The logical operator way would find the points inside both `x` and `y` using a spatial predicate such as `st_intersects()`, whereas the intersection method simply finds the points inside the intersecting region created above as `x_and_y`. As demonstrated below the results are identical, but the method that uses the clipped polygon is more concise:

```
sel_p_xy = st_intersects(p, x, sparse = FALSE)[, 1] &
  st_intersects(p, y, sparse = FALSE)[, 1]
p_xy1 = p[sel_p_xy]
p_xy2 = p[x_and_y]
identical(p_xy1, p_xy2)
#> [1] TRUE
```

[141] http://r4ds.had.co.nz/transform.html#logical-operators

0.5.2.6 Geometry unions

As we saw in Section **??**, spatial aggregation can silently dissolve the geometries of touching polygons in the same group. This is demonstrated in the code chunk below in which 49 `us_states` are aggregated into 4 regions using base and **tidyverse** functions (see results in Figure **??**):

```
regions = aggregate(x = us_states[, "total_pop_15"], by = list(us_states$REGION),
                    FUN = sum, na.rm = TRUE)
regions2 = us_states %>% group_by(REGION) %>%
  summarize(pop = sum(total_pop_15, na.rm = TRUE))
```

What is going on in terms of the geometries? Behind the scenes, both `aggregate()` and `summarize()` combine the geometries and dissolve the boundaries between them using `st_union()`. This is demonstrated in the code chunk below which creates a united western US:

```
us_west = us_states[us_states$REGION == "West", ]
us_west_union = st_union(us_west)
```

The function can take two geometries and unite them, as demonstrated in the code chunk below which creates a united western block incorporating Texas (challenge: reproduce and plot the result):

```
texas = us_states[us_states$NAME == "Texas", ]
texas_union = st_union(us_west_union, texas)
```

0.5.2.7 Type transformations

Geometry casting is a powerful operation that enables transformation of the geometry type. It is implemented in the `st_cast` function from the **sf** package. Importantly, `st_cast` behaves differently on single simple feature geometry (`sfg`) objects, simple feature geometry column (`sfc`) and simple features objects. Let's create a multipoint to illustrate how geometry casting works on simple feature geometry (`sfg`) objects:

```
multipoint = st_multipoint(matrix(c(1, 3, 5, 1, 3, 1), ncol = 2))
```

In this case, `st_cast` can be useful to transform the new object into linestring or polygon (Figure **??**):

```
linestring = st_cast(multipoint, "LINESTRING")
polyg = st_cast(multipoint, "POLYGON")
```

Conversion from multipoint to linestring is a common operation that creates a line object from ordered point observations, such as GPS measurements or geotagged media. This allows spatial operations such as the length of the path traveled. Conversion from multipoint or linestring to polygon is often used to calculate an area, for example from the set of GPS measurements taken around a lake or from the corners of a building lot.

The transformation process can be also reversed using `st_cast`:

```
multipoint_2 = st_cast(linestring, "MULTIPOINT")
multipoint_3 = st_cast(polyg, "MULTIPOINT")
all.equal(multipoint, multipoint_2, multipoint_3)
#> [1] TRUE
```

For single simple feature geometries (`sfg`), `st_cast` also provides geometry casting from non-multi-types to multi-types (e.g., `POINT` to `MULTIPOINT`) and from multi-types to non-multi-types. However, only the first element of the old object would remain in the second group of cases.

Geometry casting of simple features geometry column (`sfc`) and simple features objects works the same as for single geometries in most of the cases. One important difference is the conversion between multi-types to non-multi-types. As a result of this process, multi-objects are split into many non-multi-objects. Table **??** shows possible geometry type transformations on simple feature objects. Each input simple feature object with only one element (first column) is transformed directly into another geometry type. Several of the transformations are not possible, for example, you cannot convert a single point into a multilinestring or a polygon (so the cells `[1, 4:5]` in the table are NA). On the other hand, some of the transformations are splitting the single element input object into a multi-element object. You can see that, for example, when you cast a multipoint consisting of five pairs of coordinates into a point.

Let's try to apply geometry type transformations on a new object, `multilinestring_sf`, as an example (on the left in Figure **??**):

TABLE 0.5: Geometry casting on simple feature geometries (see Section 2.1) with input type by row and output type by column. Values like (1) represent the number of features; NA means the operation is not possible. Abbreviations: POI, LIN, POL and GC refer to POINT, LINESTRING, POLYGON and GEOMETRYCOLLECTION. The MULTI version of these geometry types is indicated by a preceding M, e.g., MPOI is the acronym for MULTIPOINT.

	POI	MPOI	LIN	MLIN	POL	MPOL	GC
POI(1)	1	1	1	NA	NA	NA	NA
MPOI(1)	4	1	1	1	1	NA	NA
LIN(1)	5	1	1	1	1	NA	NA
MLIN(1)	7	2	2	1	NA	NA	NA
POL(1)	5	1	1	1	1	1	NA
MPOL(1)	10	1	NA	1	2	1	1
GC(1)	9	1	NA	NA	NA	NA	1

```
multilinestring_list = list(matrix(c(1, 4, 5, 3), ncol = 2),
                            matrix(c(4, 4, 4, 1), ncol = 2),
                            matrix(c(2, 4, 2, 2), ncol = 2))
multilinestring = st_multilinestring((multilinestring_list))
multilinestring_sf = st_sf(geom = st_sfc(multilinestring))
multilinestring_sf
#> Simple feature collection with 1 feature and 0 fields
#> geometry type:  MULTILINESTRING
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> epsg (SRID):    NA
#> proj4string:    NA
#>                              geom
#> 1 MULTILINESTRING ((1 5, 4 3)...
```

You can imagine it as a road or river network. The new object has only one row that defines all the lines. This restricts the number of operations that can be done, for example it prevents adding names to each line segment or calculating lengths of single lines. The `st_cast` function can be used in this situation, as it separates one mutlilinestring into three linestrings:

```
linestring_sf2 = st_cast(multilinestring_sf, "LINESTRING")
linestring_sf2
#> Simple feature collection with 3 features and 0 fields
```

```
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> epsg (SRID):    NA
#> proj4string:    NA
#>                    geom
#> 1 LINESTRING (1 5, 4 3)
#> 2 LINESTRING (4 4, 4 1)
#> 3 LINESTRING (2 2, 4 2)
```

The newly created object allows for attributes creation (see more in Section **??**) and length measurements:

```
linestring_sf2$name = c("Riddle Rd", "Marshall Ave", "Foulke St")
linestring_sf2$length = st_length(linestring_sf2)
linestring_sf2
#> Simple feature collection with 3 features and 2 fields
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:           xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> epsg (SRID):    NA
#> proj4string:    NA
#>                    geom         name length
#> 1 LINESTRING (1 5, 4 3)    Riddle Rd   3.61
#> 2 LINESTRING (4 4, 4 1) Marshall Ave   3.00
#> 3 LINESTRING (2 2, 4 2)    Foulke St   2.00
```

0.5.3 Geometric operations on raster data

Geometric raster operations include the shift, flipping, mirroring, scaling, rotation or warping of images. These operations are necessary for a variety of applications including georeferencing, used to allow images to be overlaid on an accurate map with a known CRS (Liu and Mason 2009). A variety of georeferencing techniques exist, including:

- Georeferencing based on known ground control points[142].
- Orthorectification also georeferences an image, but additionally takes into account local topography.

[142] http://www.qgistutorials.com/en/docs/georeferencing_basics.html

- Image (co-)registration is the process of aligning one image with another (in terms of coordinate reference system, origin and resolution). Registration becomes necessary for images from the same scene but shot from different sensors or from different angles or at different points in time.

R is unsuitable for the first two points since these often require manual intervention which is why they are usually done with the help of dedicated GIS software (see also Chapter **??**). On the other hand, aligning several images is possible in R and this section shows among others how to do so. This often includes changing the extent, the resolution and the origin of an image. A matching projection is of course also required but is already covered in Section **??**. In any case, there are other reasons to perform a geometric operation on a single raster image. For instance, in Chapter **??** we define metropolitan areas in Germany as 20 km^2 pixels with more than 500,000 inhabitants. The original inhabitant raster, however, has a resolution of 1 km^2 which is why we will decrease (aggregate) the resolution by a factor of 20 (see Section **??**). Another reason for aggregating a raster is simply to decrease run-time or save disk space. Of course, this is only possible if the task at hand allows a coarser resolution. Sometimes a coarser resolution is sufficient for the task at hand.

0.5.3.1 Geometric intersections

In Section **??** we have shown how to extract values from a raster overlaid by other spatial objects. To retrieve a spatial output, we can use almost the same subsetting syntax. The only difference is that we have to make clear that we would like to keep the matrix structure by setting the `drop`-parameter to `FALSE`. This will return a raster object containing the cells whose midpoints overlap with `clip`.

```
data("elev", package = "spData")
clip = raster(xmn = 0.9, xmx = 1.8, ymn = -0.45, ymx = 0.45,
              res = 0.3, vals = rep(1, 9))
elev[clip, drop = FALSE]
#> class      : RasterLayer
#> dimensions : 2, 1, 2  (nrow, ncol, ncell)
#> resolution : 0.5, 0.5  (x, y)
#> extent     : 1, 1.5, -0.5, 0.5  (xmin, xmax, ymin, ymax)
#> crs        : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
#> source     : memory
#> names      : layer
#> values     : 18, 24  (min, max)
```

cxx

For the same operation we can also use the `intersect()` and `crop()` command.

0.5.3.2 Extent and origin

When merging or performing map algebra on rasters, their resolution, projection, origin and/or extent have to match. Otherwise, how should we add the values of one raster with a resolution of 0.2 decimal degrees to a second with a resolution of 1 decimal degree? The same problem arises when we would like to merge satellite imagery from different sensors with different projections and resolutions. We can deal with such mismatches by aligning the rasters. In the simplest case, two images only differ with regard to their extent. Following code adds one row and two columns to each side of the raster while setting all new values to an elevation of 1000 meters (Figure **??**).

```
data(elev, package = "spData")
elev_2 = extend(elev, c(1, 2), value = 1000)
plot(elev_2)
```

Performing an algebraic operation on two objects with differing extents in R, the **raster** package returns the result for the intersection, and says so in a warning.

```
elev_3 = elev + elev_2
#> Warning in elev + elev_2: Raster objects have different extents. Result for
#> their intersection is returned
```

However, we can also align the extent of two rasters with `extend()`. Instead of telling the function how many rows or columns should be added (as done before), we allow it to figure it out by using another raster object. Here, we extend the `elev` object to the extent of `elev_2`. The newly added rows and column receive the default value of the `value` parameter, i.e., `NA`.

```
elev_4 = extend(elev, elev_2)
```

The origin is the point closest to (0, 0) if you moved towards it (starting from any given cell corner of the raster) in steps of x and y resolution.

```
origin(elev_4)
#> [1] 0 0
```

If two rasters have different origins, their cells do not overlap completely which would make map algebra impossible. To change the origin , use `origin()`.[143] Looking at Figure **??** reveals the effect of changing the origin.

```
# change the origin
origin(elev_4) = c(0.25, 0.25)
plot(elev_4)
# and add the original raster
plot(elev, add = TRUE)
```

Note that changing the resolution frequently (next section) also changes the origin.

0.5.3.3 Aggregation and disaggregation

Raster datasets can also differ with regard to their resolution. To match resolutions, one can either decrease (`aggregate()`) or increase (`disaggregate()`) the resolution of one raster.[144] As an example, we here change the spatial resolution of `dem` (found in the **RQGIS** package) by a factor of 5 (Figure **??**). Additionally, the output cell value should correspond to the mean of the input cells (note that one could use other functions as well, such as `median()`, `sum()`, etc.):

```
data("dem", package = "RQGIS")
dem_agg = aggregate(dem, fact = 5, fun = mean)
```

By contrast, the`disaggregate()` function increases the resolution. However, we have to specify a method on how to fill the new cells. The `disaggregate()` function provides two methods. The first (nearest neighbor, `method = ""`) simply gives all output cells the value of the nearest input cell, and hence duplicates values which leads to a blocky output image.
The `bilinear` method, in turn, is an interpolation technique that uses the four nearest pixel centers of the input image (salmon colored points in Figure **??**) to compute an average weighted by distance (arrows in Figure **??** as the value of the output cell - square in the upper left corner in Figure **??**).

[143] If the origins of two raster datasets are just marginally apart, it sometimes is sufficient to simply increase the `tolerance` argument of `raster::rasterOptions()`.
[144] Here we refer to spatial resolution. In remote sensing the spectral (spectral bands), temporal (observations through time of the same area) and radiometric (color depth) resolution are also important. Check out the `stackApply()` example in the documentation for getting an idea on how to do temporal raster aggregation.

```
dem_disagg = disaggregate(dem_agg, fact = 5, method = "bilinear")
identical(dem, dem_disagg)
#> [1] FALSE
```

Comparing the values of `dem` and `dem_disagg` tells us that they are not identical (you can also use `compareRaster()` or `all.equal()`). However, this was hardly to be expected, since disaggregating is a simple interpolation technique. It is important to keep in mind that disaggregating results in a finer resolution; the corresponding values, however, are only as accurate as their lower resolution source.

The process of computing values for new pixel locations is also called resampling. In fact, the **raster** package provides a `resample()` function. It lets you align several raster properties in one go, namely origin, extent and resolution. By default, it uses the `bilinear`-interpolation.

```
# add 2 rows and columns, i.e. change the extent
dem_agg = extend(dem_agg, 2)
dem_disagg_2 = resample(dem_agg, dem)
```

Finally, in order to align many (possibly hundreds or thousands of) images stored on disk, you could use the `gdalUtils::align_rasters()` function. However, you may also use **raster** with very large datasets. This is because **raster**:

1. Lets you work with raster datasets that are too large to fit into the main memory (RAM) by only processing chunks of it.
2. Tries to facilitate parallel processing. For more information, see the help pages of `beginCluster()` and `clusteR()`. Additionally, check out the *Multi-core functions* section in `vignette("functions", package = "raster")`.

0.5.4 Raster-vector interactions

This section focuses on interactions between raster and vector geographic data models, introduced in Chapter **??**. It includes four main techniques: raster cropping and masking using vector objects (Section **??**); extracting raster values using different types of vector data (Section **??**); and raster-vector conversion (Sections **??** and **??**). The above concepts are demonstrated using data used in previous chapters to understand their potential real-world applications.

0.5.4.1 Raster cropping

Many geographic data projects involve integrating data from many different sources, such as remote sensing images (rasters) and administrative boundaries (vectors). Often the extent of input raster datasets is larger than the area of interest. In this case raster **cropping** and **masking** are useful for unifying the spatial extent of input data. Both operations reduce object memory use and associated computational resources for subsequent analysis steps, and may be a necessary preprocessing step before creating attractive maps involving raster data.

We will use two objects to illustrate raster cropping:

- A `raster` object `srtm` representing elevation (meters above sea level) in south-western Utah.
- A vector (`sf`) object `zion` representing Zion National Park.

Both target and cropping objects must have the same projection. The following code chunk therefore not only loads the datasets, from the **spDataLarge** package installed in Chapter **??**, it also reprojects `zion` (see Section **??** for more on reprojection):

```
srtm = raster(system.file("raster/srtm.tif", package = "spDataLarge"))
zion = st_read(system.file("vector/zion.gpkg", package = "spDataLarge"))
zion = st_transform(zion, projection(srtm))
```

We will use `crop()` from the **raster** package to crop the `srtm` raster. `crop()` reduces the rectangular extent of the object passed to its first argument based on the extent of the object passed to its second argument, as demonstrated in the command below (which generates Figure **??**(B) — note the smaller extent of the raster background):

```
srtm_cropped = crop(srtm, zion)
```

Related to `crop()` is the **raster** function `mask()`, which sets values outside of the bounds of the object passed to its second argument to `NA`. The following command therefore masks every cell outside of the Zion National Park boundaries (Figure **??**(C)):

```
srtm_masked = mask(srtm, zion)
```

Changing the settings of `mask()` yields different results. Setting `maskvalue = 0`, for example, will set all pixels outside the national park to 0. Setting `inverse = TRUE` will mask everything *inside* the bounds of the park (see `?mask` for details) (Figure **??**(D)).

```
srtm_inv_masked = mask(srtm, zion, inverse = TRUE)
```

```
#> Warning: Detecting old grouped_df format, replacing `vars` attribute by
#> `groups`
```

0.5.4.2 Raster extraction

Raster extraction is the process of identifying and returning the values associated with a 'target' raster at specific locations, based on a (typically vector) geographic 'selector' object. The results depend on the type of selector used (points, lines or polygons) and arguments passed to the `raster::extract()` function, which we use to demonstrate raster extraction. The reverse of raster extraction — assigning raster cell values based on vector objects — is rasterization, described in Section **??**.

The simplest example is extracting the value of a raster cell at specific **points**. For this purpose, we will use `zion_points`, which contain a sample of 30 locations within the Zion National Park (Figure **??**). The following command extracts elevation values from `srtm` and assigns the resulting vector to a new column (`elevation`) in the `zion_points` dataset:

```
data("zion_points", package = "spDataLarge")
zion_points$elevation = raster::extract(srtm, zion_points)
```

The `buffer` argument can be used to specify a buffer radius (in meters) around each point. The result of `raster::extract(srtm, zion_points, buffer = 1000)`, for example, is a list of vectors, each of which representing the values of cells inside the buffer associated with each point. In practice, this example is a special case of extraction with a polygon selector, described below.

Raster extraction also works with **line** selectors. To demonstrate this, the code below creates `zion_transect`, a straight line going from northwest to southeast of the Zion National Park, illustrated in Figure **??**(A) (see Section **??** for a recap on the vector data model):

```
zion_transect = cbind(c(-113.2, -112.9), c(37.45, 37.2)) %>%
  st_linestring() %>%
  st_sfc(crs = projection(srtm)) %>%
  st_sf()
```

The utility of extracting heights from a linear selector is illustrated by imagining that you are planning a hike. The method demonstrated below provides an 'elevation profile' of the route (the line does not need to be straight), useful for estimating how long it will take due to long climbs:

```
transect = raster::extract(srtm, zion_transect,
                           along = TRUE, cellnumbers = TRUE)
```

Note the use of `along = TRUE` and `cellnumbers = TRUE` arguments to return cell IDs *along* the path. The result is a list containing a matrix of cell IDs in the first column and elevation values in the second. The number of list elements is equal to the number of lines or polygons from which we are extracting values. The subsequent code chunk first converts this tricky matrix-in-a-list object into a simple data frame, returns the coordinates associated with each extracted cell, and finds the associated distances along the transect (see `?geosphere::distGeo()` for details):

```
transect_df = purrr::map_dfr(transect, as_data_frame, .id = "ID")
transect_coords = xyFromCell(srtm, transect_df$cell)
transect_df$dist = c(0, cumsum(geosphere::distGeo(transect_coords)))
```

The resulting `transect_df` can be used to create elevation profiles, as illustrated in Figure **??**(B).

The final type of geographic vector object for raster extraction is **polygons**. Like lines and buffers, polygons tend to return many raster values per polygon. This is demonstrated in the command below, which results in a data frame with column names `ID` (the row number of the polygon) and `srtm` (associated elevation values):

```
zion_srtm_values = raster::extract(x = srtm, y = zion, df = TRUE)
```

Such results can be used to generate summary statistics for raster values per polygon, for example to characterize a single region or to compare many regions.

The generation of summary statistics is demonstrated in the code below, which creates the object `zion_srtm_df` containing summary statistics for elevation values in Zion National Park (see Figure **??**(A)):

```
group_by(zion_srtm_values, ID) %>%
  summarize_at(vars(srtm), funs(min, mean, max))
#> Warning: funs() is soft deprecated as of dplyr 0.8.0
#> please use list() instead
#>
#> # Before:
#> funs(name = f(.)
#>
#> # After:
#> list(name = ~f(.))
#> This warning is displayed once per session.
#> # A tibble: 1 x 4
#>      ID   min  mean   max
#>   <dbl> <dbl> <dbl> <dbl>
#> 1     1  1122 1818.  2661
```

The preceding code chunk used the **tidyverse** to provide summary statistics for cell values per polygon ID, as described in Chapter **??**. The results provide useful summaries, for example that the maximum height in the park is around 2,661 meters (other summary statistics, such as standard deviation, can also be calculated in this way). Because there is only one polygon in the example a data frame with a single row is returned; however, the method works when multiple selector polygons are used.

The same approach works for counting occurrences of categorical raster values within polygons. This is illustrated with a land cover dataset (`nlcd`) from the **spDataLarge** package in Figure **??**(B), and demonstrated in the code below:

```
zion_nlcd = raster::extract(nlcd, zion, df = TRUE, factors = TRUE)
dplyr::select(zion_nlcd, ID, levels) %>%
  tidyr::gather(key, value, -ID) %>%
  group_by(ID, key, value) %>%
  tally() %>%
  tidyr::spread(value, n, fill = 0)
#> # A tibble: 1 x 9
#> # Groups:   ID, key [1]
#>      ID key   Barren Cultivated Developed Forest Herbaceous Shrubland
```

```
#>   <dbl> <chr>  <dbl>      <dbl>     <dbl>  <dbl>      <dbl>     <dbl>
#> 1     1 leve~  98285         62      4205 298299        235    203701
#> # ... with 1 more variable: Wetlands <dbl>
```

So far, we have seen how `raster::extract()` is a flexible way of extracting raster cell values from a range of input geographic objects. An issue with the function, however, is that it is relatively slow. If this is a problem, it is useful to know about alternatives and work-arounds, three of which are presented below.

- **Parallelization**: this approach works when using many geographic vector selector objects by splitting them into groups and extracting cell values independently for each group (see `?raster::clusterR()` for details of this approach).
- Use the **velox** package (Hunziker 2017), which provides a fast method for extracting raster data that fits in memory (see the packages `extract`[145] vignette for details).
- Using **R-GIS bridges** (see Chapter **??**): efficient calculation of raster statistics from polygons can be found in the SAGA function `saga:gridstatisticsforpolygons`, for example, which can be accessed via **RQGIS**.

0.5.4.3 Rasterization

Rasterization is the conversion of vector objects into their representation in raster objects. Usually, the output raster is used for quantitative analysis (e.g., analysis of terrain) or modeling. As we saw in Chapter **??** the raster data model has some characteristics that make it conducive to certain methods. Furthermore, the process of rasterization can help simplify datasets because the resulting values all have the same spatial resolution: rasterization can be seen as a special type of geographic data aggregation.

The **raster** package contains the function `rasterize()` for doing this work. Its first two arguments are, `x`, vector object to be rasterized and, `y`, a 'template raster' object defining the extent, resolution and CRS of the output. The geographic resolution of the input raster has a major impact on the results: if it is too low (cell size is too large), the result may miss the full geographic variability of the vector data; if it is too high, computational times may be excessive. There are no simple rules to follow when deciding an appropriate geographic resolution, which is heavily dependent on the intended use of the results. Often the target resolution is imposed on the user, for example when the output of rasterization needs to be aligned to the existing raster.

To demonstrate rasterization in action, we will use a template raster that has

[145] `https://hunzikp.github.io/velox/extract.html`

the same extent and CRS as the input vector data `cycle_hire_osm_projected` (a dataset on cycle hire points in London is illustrated in Figure **??**(A)) and spatial resolution of 1000 meters:

```
cycle_hire_osm_projected = st_transform(cycle_hire_osm, 27700)
raster_template = raster(extent(cycle_hire_osm_projected), resolution = 1000,
                         crs = st_crs(cycle_hire_osm_projected)$proj4string)
```

Rasterization is a very flexible operation: the results depend not only on the nature of the template raster, but also on the type of input vector (e.g., points, polygons) and a variety of arguments taken by the `rasterize()` function. To illustrate this flexibility we will try three different approaches to rasterization. First, we create a raster representing the presence or absence of cycle hire points (known as presence/absence rasters). In this case `rasterize()` requires only one argument in addition to `x` and `y` (the aforementioned vector and raster objects): a value to be transferred to all non-empty cells specified by `field` (results illustrated Figure **??**(B)).

```
ch_raster1 = rasterize(cycle_hire_osm_projected, raster_template, field = 1)
```

The `fun` argument specifies summary statistics used to convert multiple observations in close proximity into associate cells in the raster object. By default `fun = "last"` is used but other options such as `fun = "count"` can be used, in this case to count the number of cycle hire points in each grid cell (the results of this operation are illustrated in Figure **??**(C)).

```
ch_raster2 = rasterize(cycle_hire_osm_projected, raster_template,
                       field = 1, fun = "count")
```

The new output, `ch_raster2`, shows the number of cycle hire points in each grid cell. The cycle hire locations have different numbers of bicycles described by the `capacity` variable, raising the question, what's the capacity in each grid cell? To calculate that we must `sum` the field (`"capacity"`), resulting in output illustrated in Figure **??**(D), calculated with the following command (other summary functions such as `mean` could be used):

```
ch_raster3 = rasterize(cycle_hire_osm_projected, raster_template,
                       field = "capacity", fun = sum)
```

Another dataset based on California's polygons and borders (created below) illustrates rasterization of lines. After casting the polygon objects into a multilinestring, a template raster is created with a resolution of a 0.5 degree:

```
california = dplyr::filter(us_states, NAME == "California")
california_borders = st_cast(california, "MULTILINESTRING")
raster_template2 = raster(extent(california), resolution = 0.5,
                          crs = st_crs(california)$proj4string)
```

Line rasterization is demonstrated in the code below. In the resulting raster, all cells that are touched by a line get a value, as illustrated in Figure **??**(A).

```
california_raster1 = rasterize(california_borders, raster_template2)
```

Polygon rasterization, by contrast, selects only cells whose centroids are inside the selector polygon, as illustrated in Figure **??**(B).

```
california_raster2 = rasterize(california, raster_template2)
```

As with `raster::extract()`, `raster::rasterize()` works well for most cases but is not performance optimized. Fortunately, there are several alternatives, including the `fasterize::fasterize()` and `gdalUtils::gdal_rasterize()`. The former is much (100 times+) faster than `rasterize()`, but is currently limited to polygon rasterization. The latter is part of GDAL and therefore requires a vector file (instead of an `sf` object) and rasterization parameters (instead of a `Raster*` template object) as inputs.[146]

0.5.4.4 Spatial vectorization

Spatial vectorization is the counterpart of rasterization (Section **??**), but in the opposite direction. It involves converting spatially continuous raster data into spatially discrete vector data such as points, lines or polygons.

Be careful with the wording! In R, vectorization refers to the possibility of replacing `for`-loops and alike by doing things like `1:10 / 2` (see also Wickham (2014a)).

[146] See more at `http://gdal.org/gdal_rasterize.html`.

cxxx

The simplest form of vectorization is to convert the centroids of raster cells into points. `rasterToPoints()` does exactly this for all non-`NA` raster grid cells (Figure **??**). Setting the `spatial` parameter to `TRUE` ensures the output is a spatial object, not a matrix.

```
elev_point = rasterToPoints(elev, spatial = TRUE) %>%
  st_as_sf()
```

Another common type of spatial vectorization is the creation of contour lines representing lines of continuous height or temperatures (isotherms) for example. We will use a real-world digital elevation model (DEM) because the artificial raster `elev` produces parallel lines (task: verify this and explain why this happens). Contour lines can be created with the **raster** function `rasterToContour()`, which is itself a wrapper around `contourLines()`, as demonstrated below (not shown):

```
data(dem, package = "RQGIS")
cl = rasterToContour(dem)
plot(dem, axes = FALSE)
plot(cl, add = TRUE)
```

Contours can also be added to existing plots with functions such as `contour()`, `rasterVis::contourplot()` or `tmap::tm_iso()`. As illustrated in Figure **??**, isolines can be labelled.

```
# create hillshade
hs = hillShade(slope = terrain(dem, "slope"), aspect = terrain(dem, "aspect"))
plot(hs, col = gray(0:100 / 100), legend = FALSE)
# overlay with DEM
plot(dem, col = terrain.colors(25), alpha = 0.5, legend = FALSE, add = TRUE)
# add contour lines
contour(dem, col = "white", add = TRUE)
```

The final type of vectorization involves conversion of rasters to polygons. This can be done with `raster::rasterToPolygons()`, which converts each raster cell into a polygon consisting of five coordinates, all of which are stored in memory (explaining why rasters are often fast compared with vectors!). This is illustrated below by converting the `grain` object into polygons and subsequently dissolving borders between polygons with the same attribute

values (also see the `dissolve` argument in `rasterToPolygons()`). Attributes in this case are stored in a column called `layer` (see Section **??** and Figure **??**). (Note: a convenient alternative for converting rasters into polygons is `spex::polygonize()` which by default returns an `sf` object.)

```
grain_poly = rasterToPolygons(grain) %>%
  st_as_sf()
grain_poly2 = grain_poly %>%
  group_by(layer) %>%
  summarize()
```

0.5.5 Exercises

Some of the exercises use a vector (`random_points`) and raster dataset (`ndvi`) from the **RQGIS** package. They also use a polygonal 'convex hull' derived from the vector dataset (`ch`) to represent the area of interest:

```
library(RQGIS)
data(random_points)
data(ndvi)
ch = st_combine(random_points) %>%
  st_convex_hull()
```

1. Generate and plot simplified versions of the `nz` dataset. Experiment with different values of `keep` (ranging from 0.5 to 0.00005) for `ms_simplify()` and `dTolerance` (from 100 to 100,000) `st_simplify()` .
    - At what value does the form of the result start to break down for each method, making New Zealand unrecognizable?
    - Advanced: What is different about the geometry type of the results from `st_simplify()` compared with the geometry type of `ms_simplify()`? What problems does this create and how can this be resolved?
2. In the first exercise in Chapter **??** it was established that Canterbury region had 70 of the 101 highest points in New Zealand. Using `st_buffer()`, how many points in `nz_height` are within 100 km of Canterbury?

3. Find the geographic centroid of New Zealand. How far is it from the geographic centroid of Canterbury?
4. Most world maps have a north-up orientation. A world map with a south-up orientation could be created by a reflection (one of the affine transformations not mentioned in Section **??**) of the `world` object's geometry. Write code to do so. Hint: you need to use a two-element vector for this transformation.

    - Bonus: create an upside-down map of your country.

5. Subset the point in `p` that is contained within `x` *and* `y` (see Section **??** and Figure **??**).

    - Using base subsetting operators.
    - Using an intermediary object created with `st_intersection()`.

6. Calculate the length of the boundary lines of US states in meters. Which state has the longest border and which has the shortest? Hint: The `st_length` function computes the length of a `LINESTRING` or `MULTILINESTRING` geometry.
7. Crop the `ndvi` raster using (1) the `random_points` dataset and (2) the `ch` dataset. Are there any differences in the output maps? Next, mask `ndvi` using these two datasets. Can you see any difference now? How can you explain that?
8. Firstly, extract values from `ndvi` at the points represented in `random_points`. Next, extract average values of `ndvi` using a 90 buffer around each point from `random_points` and compare these two sets of values. When would extracting values by buffers be more suitable than by points alone?
9. Subset points higher than 3100 meters in New Zealand (the `nz_height` object) and create a template raster with a resolution of 3 km. Using these objects:

    - Count numbers of the highest points in each grid cell.
    - Find the maximum elevation in each grid cell.

10. Aggregate the raster counting high points in New Zealand (created in the previous exercise), reduce its geographic resolution by half (so cells are 6 by 6 km) and plot the result.

    - Resample the lower resolution raster back to a resolution of 3 km. How have the results changed?

- Name two advantages and disadvantages of reducing raster resolution.

11. Polygonize the `grain` dataset and filter all squares representing clay.

- Name two advantages and disadvantages of vector data over raster data.
- At which points would it be useful to convert rasters to vectors in your work?

0.6 Reprojecting geographic data

Prerequisites

- This chapter requires the following packages (**lwgeom** is also used, but does not need to be attached):

```
library(sf)
library(raster)
library(dplyr)
library(spData)
library(spDataLarge)
```

0.6.1 Introduction

Section **??** introduced coordinate reference systems (CRSs) and demonstrated their importance. This chapter goes further. It highlights issues that can arise when using inappropriate CRSs and how to *transform* data from one CRS to another.

As illustrated in Figure **??**, there are two types of CRSs: *geographic* ('lon/lat', with units in degrees longitude and latitude) and *projected* (typically with units of meters from a datum). This has consequences. Many geometry operations in **sf**, for example, assume their inputs have a projected CRS, because the GEOS functions they are based on assume projected data. To deal with this issue **sf** provides the function `st_is_longlat()` to check. In some cases the CRS is unknown, as shown below using the example of London introduced in Section **??**:

```
london = data.frame(lon = -0.1, lat = 51.5) %>%
  st_as_sf(coords = c("lon", "lat"))
st_is_longlat(london)
#> [1] NA
```

This shows that unless a CRS is manually specified or is loaded from a source that has CRS metadata, the CRS is `NA`. A CRS can be added to `sf` objects with `st_set_crs()` as follows:[147]

```
london_geo = st_set_crs(london, 4326)
st_is_longlat(london_geo)
#> [1] TRUE
```

Datasets without a specified CRS can cause problems. An example is provided below, which creates a buffer of one unit around `london` and `london_geo` objects:

```
london_buff_no_crs = st_buffer(london, dist = 1)
london_buff = st_buffer(london_geo, dist = 1)
#> Warning in st_buffer.sfc(st_geometry(x), dist, nQuadSegs, endCapStyle =
#> endCapStyle, : st_buffer does not correctly buffer longitude/latitude data
#> dist is assumed to be in decimal degrees (arc_degrees).
```

Only the second operation generates a warning. The warning message is useful, telling us that the result may be of limited use because it is in units of latitude and longitude, rather than meters or some other suitable measure of distance assumed by `st_buffer()`. The consequences of a failure to work on projected data are illustrated in Figure **??** (left panel): the buffer is elongated in the north-south direction because lines of longitude converge towards the Earth's poles.

The distance between two lines of longitude, called meridians, is around 111 km at the equator (execute `geosphere::distGeo(c(0, 0), c(1, 0))` to find the precise distance). This shrinks to zero at the poles. At the latitude of London,

[147] The CRS can also be added when creating `sf` objects with the `crs` argument (e.g., `st_sf(geometry = st_sfc(st_point(c(-0.1, 51.5))), crs = 4326)`). The same argument can also be used to set the CRS when creating raster datasets (e.g., `raster(crs = "+proj=longlat")`).

for example, meridians are less than 70 km apart (challenge: execute code that verifies this). Lines of latitude, by contrast, have constant distance from each other irrespective of latitude: they are always around 111 km apart, including at the equator and near the poles. This is illustrated in Figures **??** and **??**.

Do not interpret the warning about the geographic (`longitude/latitude`) CRS as "the CRS should not be set": it almost always should be! It is better understood as a suggestion to *reproject* the data onto a projected CRS. This suggestion does not always need to be heeded: performing spatial and geometric operations makes little or no difference in some cases (e.g., spatial subsetting). But for operations involving distances such as buffering, the only way to ensure a good result is to create a projected copy of the data and run the operation on that. This is done in the code chunk below:

```
london_proj = data.frame(x = 530000, y = 180000) %>%
  st_as_sf(coords = 1:2, crs = 27700)
```

The result is a new object that is identical to `london`, but reprojected onto a suitable CRS (the British National Grid, which has an EPSG code of 27700 in this case) that has units of meters. We can verify that the CRS has changed using `st_crs()` as follows (some of the output has been replaced by `...`):

```
st_crs(london_proj)
#> Coordinate Reference System:
#>   EPSG: 27700
#>   proj4string: "+proj=tmerc +lat_0=49 +lon_0=-2 ... +units=m +no_defs"
```

Notable components of this CRS description include the EPSG code (`EPSG: 27700`), the projection (transverse Mercator[148], `+proj=tmerc`), the origin (`+lat_0=49 +lon_0=-2`) and units (`+units=m`).[149] The fact that the units of the CRS are meters (rather than degrees) tells us that this is a projected CRS: `st_is_longlat(london_proj)` now returns `FALSE` and geometry operations on `london_proj` will work without a warning, meaning buffers can be produced from it using proper units of distance. As pointed out above, moving one degree

[148] `https://en.wikipedia.org/wiki/Transverse_Mercator_projection`

[149] For a short description of the most relevant projection parameters and related concepts, see the fourth lecture by Jochen Albrecht hosted at `http://www.geography.hunter.cuny.edu/~jochen/GTECH361/lectures/` and information at `https://proj4.org/parameters.html`. Other great resources on projections are spatialreference.org and progonos.com/furuti/MapProj.

means moving a bit more than 111 km at the equator (to be precise: 111,320 meters). This is used as the new buffer distance:

```
london_proj_buff = st_buffer(london_proj, 111320)
```

The result in Figure **??** (right panel) shows that buffers based on a projected CRS are not distorted: every part of the buffer's border is equidistant to London. The importance of CRSs (primarily whether they are projected or geographic) has been demonstrated using the example of London. The subsequent sections go into more depth, exploring which CRS to use and the details of reprojecting vector and raster objects.

0.6.2 When to reproject?

The previous section showed how to set the CRS manually, with `st_set_crs(london, 4326)`. In real world applications, however, CRSs are usually set automatically when data is read-in. The main task involving CRSs is often to *transform* objects, from one CRS into another. But when should data be transformed? And into which CRS? There are no clear-cut answers to these questions and CRS selection always involves trade-offs (Maling 1992). However, there are some general principles provided in this section that can help you decide.

First it's worth considering *when to transform*. In some cases transformation to a projected CRS is essential, such as when using geometric functions such as `st_buffer()`, as Figure **??** shows. Conversely, publishing data online with the **leaflet** package may require a geographic CRS. Another case is when two objects with different CRSs must be compared or combined, as shown when we try to find the distance between two objects with different CRSs:

```
st_distance(london_geo, london_proj)
# > Error: st_crs(x) == st_crs(y) is not TRUE
```

To make the `london` and `london_proj` objects geographically comparable one of them must be transformed into the CRS of the other. But which CRS to use? The answer is usually 'to the projected CRS', which in this case is the British National Grid (EPSG:27700):

```
london2 = st_transform(london_geo, 27700)
```

Now that a transformed version of `london` has been created, using the **sf** function `st_transform()`, the distance between the two representations of London can be found. It may come as a surprise that `london` and `london2` are just over 2 km apart![150]

```
st_distance(london2, london_proj)
#> Units: [m]
#>      [,1]
#> [1,] 2018
```

0.6.3 Which CRS to use?

The question of *which CRS* is tricky, and there is rarely a 'right' answer: "There exist no all-purpose projections, all involve distortion when far from the center of the specified frame" (Bivand, Pebesma, and Gómez-Rubio 2013). For geographic CRSs, the answer is often WGS84[151], not only for web mapping (covered in the previous paragraph) but also because GPS datasets and thousands of raster and vector datasets are provided in this CRS by default. WGS84 is the most common CRS in the world, so it is worth knowing its EPSG code: 4326. This 'magic number' can be used to convert objects with unusual projected CRSs into something that is widely understood.

What about when a projected CRS is required? In some cases, it is not something that we are free to decide: "often the choice of projection is made by a public mapping agency" (Bivand, Pebesma, and Gómez-Rubio 2013). This means that when working with local data sources, it is likely preferable to work with the CRS in which the data was provided, to ensure compatibility, even if the official CRS is not the most accurate. The example of London was easy to answer because (a) the British National Grid (with its associated EPSG code 27700) is well known and (b) the original dataset (`london`) already had that CRS.

In cases where an appropriate CRS is not immediately clear, the choice of CRS should depend on the properties that are most important to preserve in the subsequent maps and analysis. All CRSs are either equal-area, equidistant, conformal (with shapes remaining unchanged), or some combination of

[150] The difference in location between the two points is not due to imperfections in the transforming operation (which is in fact very accurate) but the low precision of the manually-created coordinates that created `london` and `london_proj`. Also surprising may be that the result is provided in a matrix with units of meters. This is because `st_distance()` can provide distances between many features and because the CRS has units of meters. Use `as.numeric()` to coerce the result into a regular number.

[151] `https://en.wikipedia.org/wiki/World_Geodetic_System#A_new_World_Geodetic_System:_WGS_84`

compromises of those. Custom CRSs with local parameters can be created for a region of interest and multiple CRSs can be used in projects when no single CRS suits all tasks. 'Geodesic calculations' can provide a fall-back if no CRSs are appropriate (see proj4.org/geodesic.html[152]). For any projected CRS the results may not be accurate when used on geometries covering hundreds of kilometers. When deciding a custom CRS, we recommend the following:[153]

- A Lambert azimuthal equal-area (LAEA[154]) projection for a custom local projection (set `lon_0` and `lat_0` to the center of the study area), which is an equal-area projection at all locations but distorts shapes beyond thousands of kilometres.
- Azimuthal equidistant (AEQD[155]) projections for a specifically accurate straight-line distance between a point and the centre point of the local projection.
- Lambert conformal conic (LCC[156]) projections for regions covering thousands of kilometres, with the cone set to keep distance and area properties reasonable between the secant lines.
- Stereographic (STERE[157]) projections for polar regions, but taking care not to rely on area and distance calculations thousands of kilometres from the center.

A commonly used default is Universal Transverse Mercator (UTM[158]), a set of CRSs that divides the Earth into 60 longitudinal wedges and 20 latitudinal segments. The transverse Mercator projection used by UTM CRSs is conformal but distorts areas and distances with increasing severity with distance from the center of the UTM zone. Documentation from the GIS software Manifold therefore suggests restricting the longitudinal extent of projects using UTM zones to 6 degrees from the central meridian (source: manifold.net[159]). Almost every place on Earth has a UTM code, such as "60H" which refers to northern New Zealand where R was invented. All UTM projections have the same datum (WGS84) and their EPSG codes run sequentially from 32601 to 32660 (for northern hemisphere locations) and 32701 to 32760 (southern hemisphere locations).

To show how the system works, let's create a function, `lonlat2UTM()` to calculate the EPSG code associated with any point on the planet as follows[160]:

[152] `https://proj4.org/geodesic.html`
[153] Many thanks to an anonymous reviewer whose comments formed the basis of this advice.
[154] `https://en.wikipedia.org/wiki/Lambert_azimuthal_equal-area_projection`
[155] `https://en.wikipedia.org/wiki/Azimuthal_equidistant_projection`
[156] `https://en.wikipedia.org/wiki/Lambert_conformal_conic_projection`
[157] `https://en.wikipedia.org/wiki/Stereographic_projection`
[158] `https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system`
[159] `http://www.manifold.net/doc/mfd9/universal_transverse_mercator_projection.htm`
[160] `https://stackoverflow.com/a/9188972/`

```
lonlat2UTM = function(lonlat) {
  utm = (floor((lonlat[1] + 180) / 6) %% 60) + 1
  if(lonlat[2] > 0) {
    utm + 32600
  } else{
    utm + 32700
  }
}
```

The following command uses this function to identify the UTM zone and associated EPSG code for Auckland and London:

```
epsg_utm_auk = lonlat2UTM(c(174.7, -36.9))
epsg_utm_lnd = lonlat2UTM(st_coordinates(london))
st_crs(epsg_utm_auk)$proj4string
#> [1] "+proj=utm +zone=60 +south +datum=WGS84 +units=m +no_defs"
st_crs(epsg_utm_lnd)$proj4string
#> [1] "+proj=utm +zone=30 +datum=WGS84 +units=m +no_defs"
```

Maps of UTM zones such as that provided by dmap.co.uk[161] confirm that London is in UTM zone 30U.

Another approach to automatically selecting a projected CRS specific to a local dataset is to create an azimuthal equidistant (AEQD[162]) projection for the center-point of the study area. This involves creating a custom CRS (with no EPSG code) with units of meters based on the centerpoint of a dataset. This approach should be used with caution: no other datasets will be compatible with the custom CRS created and results may not be accurate when used on extensive datasets covering hundreds of kilometers.

Although we used vector datasets to illustrate the points outlined in this section, the principles apply equally to raster datasets. The subsequent sections explain features of CRS transformation that are unique to each geographic data model, continuing with vector data in the next section (Section **??**) and moving on to explain how raster transformation is different, in Section **??**.

0.6.4 Reprojecting vector geometries

Chapter **??** demonstrated how vector geometries are made-up of points, and how points form the basis of more complex objects such as lines and polygons.

[161] `http://www.dmap.co.uk/utmworld.htm`
[162] `https://en.wikipedia.org/wiki/Azimuthal_equidistant_projection`

cxl

Reprojecting vectors thus consists of transforming the coordinates of these points. This is illustrated by `cycle_hire_osm`, an `sf` object from **spData** that represents cycle hire locations across London. The previous section showed how the CRS of vector data can be queried with `st_crs()`. Although the output of this function is printed as a single entity, the result is in fact a named list of class `crs`, with names `proj4string` (which contains full details of the CRS) and `epsg` for its code. This is demonstrated below:

```
crs_lnd = st_crs(cycle_hire_osm)
class(crs_lnd)
#> [1] "crs"
crs_lnd$epsg
#> [1] 4326
```

This duality of CRS objects means that they can be set either using an EPSG code or a `proj4string`. This means that `st_crs("+proj=longlat +datum=WGS84 +no_defs")` is equivalent to `st_crs(4326)`, although not all `proj4string`s have an associated EPSG code. Both elements of the CRS are changed by transforming the object to a projected CRS:

```
cycle_hire_osm_projected = st_transform(cycle_hire_osm, 27700)
```

The resulting object has a new CRS with an EPSG code 27700. But how to find out more details about this EPSG code, or any code? One option is to search for it online. Another option is to use a function from the **rgdal** package to find the name of the CRS:

```
crs_codes = rgdal::make_EPSG()[1:2]
dplyr::filter(crs_codes, code == 27700)
#>    code                                  note
#> 1 27700 # OSGB 1936 / British National Grid
```

The result shows that the EPSG code 27700 represents the British National Grid, a result that could have been found by searching online for "EPSG 27700[163]". But what about the `proj4string` element? `proj4string`s are text strings in a particular format the describe the CRS. They can be seen as formulas for converting a projected point into a point on the surface of the

[163] `https://www.google.com/search?q=CRS+27700`

Earth and can be accessed from `crs` objects as follows (see proj4.org[164] for further details of what the output means):

```
st_crs(27700)$proj4string
#> [1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 ...
```

Printing a spatial object in the console, automatically returns its coordinate reference system. To access and modify it explicitly, use the `st_crs` function, for example, `st_crs(cycle_hire_osm)`.

0.6.5 Modifying map projections

Established CRSs captured by EPSG codes are well-suited for many applications. However in some cases it is desirable to create a new CRS, using a custom `proj4string`. This system allows a very wide range of projections to be created, as we'll see in some of the custom map projections in this section. A long and growing list of projections has been developed and many of these these can be set with the `+proj=` element of `proj4string`s.[165] When mapping the world while preserving area relationships, the Mollweide projection is a good choice (Jenny et al. 2017) (Figure **??**). To use this projection, we need to specify it using the `proj4string` element, `"+proj=moll"`, in the `st_transform` function:

```
world_mollweide = st_transform(world, crs = "+proj=moll")
```

On the other hand, when mapping the world, it is often desirable to have as little distortion as possible for all spatial properties (area, direction, distance). One of the most popular projections to achieve this is the Winkel tripel projection (Figure **??**).[166] `st_transform_proj()` from the **lwgeom** package which allows for coordinate transformations to a wide range of CRSs, including the Winkel tripel projection:

```
world_wintri = lwgeom::st_transform_proj(world, crs = "+proj=wintri")
```

[164] `http://proj4.org/`
[165] The Wikipedia page 'List of map projections' has 70+ projections and illustrations.
[166] This projection is used, among others, by the National Geographic Society.

The three main functions for transformation of simple features coordinates are `sf::st_transform()`, `sf::sf_project()`, and `lwgeom::st_transform_proj()`. The `st_transform` function uses the GDAL interface to PROJ, while `sf_project()` (which works with two-column numeric matrices, representing points) and `lwgeom::st_transform_proj()` use the PROJ API directly. The first one is appropriate for most situations, and provides a set of the most often used parameters and well-defined transformations. The next one allows for greater customization of a projection, which includes cases when some of the PROJ parameters (e.g., `+over`) or projection (`+proj=wintri`) is not available in `st_transform()`.

Moreover, PROJ parameters can be modified in most CRS definitions. The below code transforms the coordinates to the Lambert azimuthal equal-area projection centered on longitude and latitude of `0` (Figure **??**).

```
world_laea1 = st_transform(world,
                           crs = "+proj=laea +x_0=0 +y_0=0 +lon_0=0 +lat_0=0")
```

We can change the PROJ parameters, for example the center of the projection, using the `+lon_0` and `+lat_0` parameters. The code below gives the map centered on New York City (Figure **??**).

```
world_laea2 = st_transform(world,
                           crs = "+proj=laea +x_0=0 +y_0=0 +lon_0=-74 +lat_0=40")
```

More information on CRS modifications can be found in the Using PROJ[167] documentation.

0.6.6 Reprojecting raster geometries

The projection concepts described in the previous section apply equally to rasters. However, there are important differences in reprojection of vectors and rasters: transforming a vector object involves changing the coordinates of every vertex but this does not apply to raster data. Rasters are composed of rectangular cells of the same size (expressed by map units, such as degrees or meters), so it is impossible to transform coordinates of pixels separately. Raster reprojection involves creating a new raster object, often with a different number

[167] `http://proj4.org/usage/index.html`

of columns and rows than the original. The attributes must subsequently be re-estimated, allowing the new pixels to be 'filled' with appropriate values. In other words, raster reprojection can be thought of as two separate spatial operations: a vector reprojection of cell centroids to another CRS (Section **??**), and computation of new pixel values through resampling (Section **??**). Thus in most cases when both raster and vector data are used, it is better to avoid reprojecting rasters and reproject vectors instead.

The raster reprojection process is done with `projectRaster()` from the **raster** package. Like the `st_transform()` function demonstrated in the previous section, `projectRaster()` takes a geographic object (a raster dataset in this case) and a `crs` argument. However, `projectRaster()` only accepts the lengthy `proj4string` definitions of a CRS rather than concise EPSG codes.

It is possible to use a EPSG code in a `proj4string` definition with `"+init=epsg:MY_NUMBER"`. For example, one can use the `"+init=epsg:4326"` definition to set CRS to WGS84 (EPSG code of 4326). The PROJ library automatically adds the rest of the parameters and converts them into `"+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"`.

Let's take a look at two examples of raster transformation: using categorical and continuous data. Land cover data are usually represented by categorical maps. The `nlcd2011.tif` file provides information for a small area in Utah, USA obtained from National Land Cover Database 2011[168] in the NAD83 / UTM zone 12N CRS.

```
cat_raster = raster(system.file("raster/nlcd2011.tif", package = "spDataLarge"))
crs(cat_raster)
#> CRS arguments:
#>  +proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m
#> +no_defs
```

In this region, 14 land cover classes were distinguished (a full list of NLCD2011 land cover classes can be found at mrlc.gov[169]):

[168] `https://www.mrlc.gov/nlcd2011.php`
[169] `https://www.mrlc.gov/nlcd11_leg.php`

```
unique(cat_raster)
#>  [1] 11 21 22 23 31 41 42 43 52 71 81 82 90 95
```

When reprojecting categorical rasters, the estimated values must be the same as those of the original. This could be done using the nearest neighbor method (`ngb`). This method assigns new cell values to the nearest cell center of the input raster. An example is reprojecting `cat_raster` to WGS84, a geographic CRS well suited for web mapping. The first step is to obtain the PROJ definition of this CRS, which can be done using the http://spatialreference.org[170] webpage. The final step is to reproject the raster with the `projectRaster()` function which, in the case of categorical data, uses the nearest neighbor method (`ngb`):

```
wgs84 = "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"
cat_raster_wgs84 = projectRaster(cat_raster, crs = wgs84, method = "ngb")
```

Many properties of the new object differ from the previous one, including the number of columns and rows (and therefore number of cells), resolution (transformed from meters into degrees), and extent, as illustrated in Table **??** (note that the number of categories increases from 14 to 15 because of the addition of `NA` values, not because a new category has been created — the land cover classes are preserved).

```
#> Warning: `data_frame()` is deprecated, use `tibble()`.
#> This warning is displayed once per session.
```

\begin{table}[t]
\caption[Key attributes in the original and projected raster datasets]{Key attributes in the original ('cat_raster') and projected ('cat_raster_wgs84') categorical raster datasets.}

CRS	nrow	ncol	ncell	resolution	unique_categories
NAD83	1359	1073	1458207	31.5275	14
WGS84	1394	1111	1548734	0.0003	15

\end{table}

Reprojecting numeric rasters (with `numeric` or in this case `integer` values) follows an almost identical procedure. This is demonstrated below with `srtm.tif` in **spDataLarge** from the Shuttle Radar Topography Mission (SRTM)[171], which represents height in meters above sea level (elevation) with the WGS84 CRS:

[170] `http://spatialreference.org/ref/epsg/wgs-84/`
[171] `https://www2.jpl.nasa.gov/srtm/`

```
con_raster = raster(system.file("raster/srtm.tif", package = "spDataLarge"))
crs(con_raster)
#> CRS arguments:
#>  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

We will reproject this dataset into a projected CRS, but *not* with the nearest neighbor method which is appropriate for categorical data. Instead, we will use the bilinear method which computes the output cell value based on the four nearest cells in the original raster. The values in the projected dataset are the distance-weighted average of the values from these four cells: the closer the input cell is to the center of the output cell, the greater its weight. The following commands create a text string representing the Oblique Lambert azimuthal equal-area projection, and reproject the raster into this CRS, using the `bilinear` method:

```
equalarea = "+proj=laea +lat_0=37.32 +lon_0=-113.04"
con_raster_ea = projectRaster(con_raster, crs = equalarea, method = "bilinear")
crs(con_raster_ea)
#> CRS arguments:
#>  +proj=laea +lat_0=37.32 +lon_0=-113.04 +ellps=WGS84
```

Raster reprojection on numeric variables also leads to small changes to values and spatial properties, such as the number of cells, resolution, and extent. These changes are demonstrated in Table **??**[172]:

\begin{table}[t]

\caption[Key attributes in the original and projected raster datasets]{Key attributes in the original ('con_raster') and projected ('con_raster') continuous raster datasets.}

CRS	nrow	ncol	ncell	resolution	mean
WGS84	457	465	212505	31.5275	1843
Equal-area	467	478	223226	0.0003	1842

\end{table}

[172]Another minor change, that is not represented in Table **??**, is that the class of the values in the new projected raster dataset is `numeric`. This is because the `bilinear` method works with continuous data and the results are rarely coerced into whole integer values. This can have implications for file sizes when raster datasets are saved.

Of course, the limitations of 2D Earth projections apply as much to vector as to raster data. At best we can comply with two out of three spatial properties (distance, area, direction). Therefore, the task at hand determines which projection to choose. For instance, if we are interested in a density (points per grid cell or inhabitants per grid cell) we should use an equal-area projection (see also Chapter **??**).

There is more to learn about CRSs. An excellent resource in this area, also implemented in R, is the website R Spatial. Chapter 6 for this free online book is recommended reading — see: rspatial.org/spatial/rst/6-crs.html[173]

0.6.7 Exercises

1. Create a new object called `nz_wgs` by transforming `nz` object into the WGS84 CRS.
    - Create an object of class `crs` for both and use this to query their CRSs.
    - With reference to the bounding box of each object, what units does each CRS use?
    - Remove the CRS from `nz_wgs` and plot the result: what is wrong with this map of New Zealand and why?
2. Transform the `world` dataset to the transverse Mercator projection (`"+proj=tmerc"`) and plot the result. What has changed and why? Try to transform it back into WGS 84 and plot the new object. Why does the new object differ from the original one?
3. Transform the continuous raster (`cat_raster`) into WGS 84 using the nearest neighbor interpolation method. What has changed? How does it influence the results?
4. Transform the categorical raster (`cat_raster`) into WGS 84 using the bilinear interpolation method. What has changed? How does it influence the results?
5. Create your own `proj4string`. It should have the Lambert Azimuthal Equal Area (`laea`) projection, the WGS84 ellipsoid, the longitude of projection center of 95 degrees west, the latitude of projection center of 60 degrees north, and its units should be in meters. Next, subset Canada from the `world` object and transform it into the new projection. Plot and compare a map before and after the transformation.

[173] `http://rspatial.org/spatial/rst/6-crs.html`

0.7 Geographic data I/O

Prerequisites

This chapter requires the following packages:

```
library(sf)
library(raster)
library(dplyr)
library(spData)
```

0.7.1 Introduction

This chapter is about reading and writing geographic data. Geographic data *import* is essential for geocomputation: real-world applications are impossible without data. For others to benefit from the results of your work, data *output* is also vital. Taken together, we refer to these processes as I/O, short for input/output.

Geographic data I/O is almost always part of a wider process. It depends on knowing which datasets are *available*, where they can be *found* and how to *retrieve* them. These topics are covered in Section **??**, which describes various *geoportals*, which collectively contain many terabytes of data, and how to use them. To further ease data access, a number of packages for downloading geographic data have been developed. These are described in Section **??**.

There are many geographic file formats, each of which has pros and cons. These are described in Section **??**. The process of actually reading and writing such file formats efficiently is not covered until Sections **??** and **??**, respectively. The final Section **??** demonstrates methods for saving visual outputs (maps), in preparation for Chapter **??** on visualization.

0.7.2 Retrieving open data

A vast and ever-increasing amount of geographic data is available on the internet, much of which is free to access and use (with appropriate credit given to its providers). In some ways there is now *too much* data, in the sense that there are often multiple places to access the same dataset. Some datasets are of poor quality. In this context, it is vital to know where to look, so the first section covers some of the most important sources. Various 'geoportals' (web

services providing geospatial datasets such as Data.gov[174]) are a good place to start, providing a wide range of data but often only for specific locations (as illustrated in the updated Wikipedia page[175] on the topic).

Some global geoportals overcome this issue. The GEOSS portal[176] and the Copernicus Open Access Hub[177], for example, contain many raster datasets with global coverage. A wealth of vector datasets can be accessed from the National Aeronautics and Space Administration agency (NASA), SEDAC[178] portal and the European Union's INSPIRE geoportal[179], with global and regional coverage.

Most geoportals provide a graphical interface allowing datasets to be queried based on characteristics such spatial and temporal extent, the United States Geological Services' EarthExplorer[180] being a prime example. *Exploring* datasets interactively on a browser is an effective way of understanding available layers. *Downloading* data is best done with code, however, from reproducibility and efficiency perspectives. Downloads can be initiated from the command line using a variety of techniques, primarily via URLs and APIs (see the Sentinel API[181] for example). Files hosted on static URLs can be downloaded with `download.file()`, as illustrated in the code chunk below which accesses US National Parks data from: catalog.data.gov/dataset/national-parks[182]:

```
download.file(url = "http://nrdata.nps.gov/programs/lands/nps_boundary.zip",
              destfile = "nps_boundary.zip")
unzip(zipfile = "nps_boundary.zip")
usa_parks = st_read(dsn = "nps_boundary.shp")
```

0.7.3 Geographic data packages

A multitude of R packages have been developed for accessing geographic data, some of which are presented in Table **??**. These provide interfaces to one or more spatial libraries or geoportals and aim to make data access even quicker from the command line.

It should be emphasised that Table **??** represents only a small number of available geographic data packages. Other notable packages include **GSODR**,

[174] https://catalog.data.gov/dataset?metadata_type=geospatial
[175] https://en.wikipedia.org/wiki/Geoportal
[176] http://www.geoportal.org/
[177] https://scihub.copernicus.eu/
[178] http://sedac.ciesin.columbia.edu/
[179] http://inspire-geoportal.ec.europa.eu/
[180] https://earthexplorer.usgs.gov/
[181] https://scihub.copernicus.eu/twiki/do/view/SciHubWebPortal/APIHubDescription
[182] https://catalog.data.gov/dataset/national-parks

TABLE 0.6: Selected R packages for geographic data retrieval.

Package	Description
getlandsat	Provides access to Landsat 8 data.
osmdata	Download and import of OpenStreetMap data.
raster	getData() imports administrative, elevation, WorldClim data.
rnaturalearth	Access to Natural Earth vector and raster data.
rnoaa	Imports National Oceanic and Atmospheric Administration (NOAA) climate data.
rWBclimate	Access World Bank climate data.

which provides Global Summary Daily Weather Data in R (see the package's README[183] for an overview of weather data sources); **tidycensus** and **tigris**, which provide socio-demographic vector data for the USA; and **hddtools**, which provides access to a range of hydrological datasets.

Each data package has its own syntax for accessing data. This diversity is demonstrated in the subsequent code chunks, which show how to get data using three packages from Table **??**. Country borders are often useful and these can be accessed with the `ne_countries()` function from the **rnaturalearth** package as follows:

```
library(rnaturalearth)
usa = ne_countries(country = "United States of America") # United States borders
class(usa)
#> [1] "SpatialPolygonsDataFrame"
#> attr(,"package")
#> [1] "sp"
# alternative way of accessing the data, with raster::getData()
# getData("GADM", country = "USA", level = 0)
```

By default **rnaturalearth** returns objects of class `Spatial`. The result can be converted into an `sf` objects with `st_as_sf()` as follows:

```
usa_sf = st_as_sf(usa)
```

A second example downloads a series of rasters containing global monthly precipitation sums with spatial resolution of ten minutes. The result is a multilayer object of class `RasterStack`.

[183] `https://github.com/ropensci/GSODR`

cl

```
library(raster)
worldclim_prec = getData(name = "worldclim", var = "prec", res = 10)
class(worldclim_prec)
#> [1] "RasterStack"
#> attr(,"package")
#> [1] "raster"
```

A third example uses the **osmdata** package (Padgham et al. 2018) to find parks from the OpenStreetMap (OSM) database. As illustrated in the code-chunk below, queries begin with the function `opq()` (short for OpenStreetMap query), the first argument of which is bounding box, or text string representing a bounding box (the city of Leeds in this case). The result is passed to a function for selecting which OSM elements we're interested in (parks in this case), represented by *key-value pairs*. Next, they are passed to the function `osmdata_sf()` which does the work of downloading the data and converting it into a list of `sf` objects (see `vignette('osmdata')` for further details):

```
library(osmdata)
parks = opq(bbox = "leeds uk") %>%
  add_osm_feature(key = "leisure", value = "park") %>%
  osmdata_sf()
```

OpenStreetMap is a vast global database of crowd-sourced data and it is growing daily. Although the quality is not as spatially consistent as many official datasets, OSM data have many advantages: they are globally available free of charge and using crowd-source data can encourage 'citizen science' and contributions back to the digital commons. Further examples of **osmdata** in action are provided in Chapters **??**, **??** and **??**.

Sometimes, packages come with inbuilt datasets. These can be accessed in four ways: by attaching the package (if the package uses 'lazy loading' as **spData** does), with `data(dataset)`, by referring to the dataset with `pkg::dataset` or with `system.file()` to access raw data files. The following code chunk illustrates the latter two options using the `world` (already loaded by attaching its parent package with `library(spData)`):[184]

```
world2 = spData::world
world3 = st_read(system.file("shapes/world.gpkg", package = "spData"))
```

[184]For more information on data import with R packages, see Sections 5.5 and 5.6 of Gillespie and Lovelace (2016).

0.7.4 Geographic web services

In an effort to standardize web APIs for accessing spatial data, the Open Geospatial Consortium (OGC) has created a number of specifications for web services (collectively known as OWS, which is short for OGC Web Services). These specifications include the Web Feature Service (WFS), Web Map Service (WMS), Web Map Tile Service (WMTS), the Web Coverage Service (WCS) and even a Wep Processing Service (WPS). Map servers such as PostGIS have adopted these protocols, leading to standardization of queries: Like other web APIs, OWS APIs use a 'base URL' and a 'query string' proceding a `?` to request data.

There are many requests that can be made to a OWS service. One of the most fundamental is `getCapabilities`, demonstrated with the **httr** package to show how API queries can be constructed and dispatched, in this case to discover the capabilities of a service providing run by the Food and Agriculture Organization of the United Nations (FAO):

```
base_url = "http://www.fao.org/figis/geoserver/wfs"
q = list(request = "GetCapabilities")
res = httr::GET(url = base_url, query = q)
res$url
#> [1] "http://www.fao.org/figis/geoserver/wfs?request=GetCapabilities"
```

The above code chunk demonstrates how API requests can be constructed programmatically with the `GET()` function, which takes a base URL and a list of query parameters which can easily be extended. The result of the request is saved in `res`, an object of class `response` defined in the **httr** package, which is a list containing information of the request, including the URL. As can be seen by executing `browseURL(res$url)`, the results can also be read directly in a browser. One way of extracting the contents of the request is as follows:

```
txt = httr::content(res, "text")
xml = xml2::read_xml(txt)
```

```
#> {xml_document} ...
#> [1] <ows:ServiceIdentification>\n  <ows:Title>GeoServer WFS...
#> [2] <ows:ServiceProvider>\n  <ows:ProviderName>Food and Agr...
#> ...
```

clii

Data can be downloaded from WFS services with the `GetFeature` request and a specific `typeName` (as illustrated in the code chunk below).
Available names differ depending on the accessed web feature service. One can extract them programmatically using web technologies (Nolan and Lang 2014) or scrolling manually through the contents of the `GetCapabilities` output in a browser.

```
qf = list(request = "GetFeature", typeName = "area:FAO_AREAS")
file = tempfile(fileext = ".gml")
httr::GET(url = base_url, query = qf, httr::write_disk(file))
fao_areas = sf::read_sf(file)
```

Note the use of `write_disc()` to ensure that the results are written to disk rather than loaded into memory, allowing them to be imported with **sf**. This example shows how to gain low-level access to web services using **httr**, which can be useful for understanding how web services work. For many everyday tasks, however, a higher-level interface may be more appropriate, and a number of R packages, and tutorials, have been developed precisely for this purpose. Packages **ows4R**, **rwfs** and **sos4R** have been developed for working with OWS services in general, WFS and the sensor observation service (SOS) respectively. As of October 2018, only **ows4R** is on CRAN. The package's basic functionality is demonstrated below, in commands that get all `FAO_AREAS` as we did in the previous code chunk:[185]

```
library(ows4R)
wfs = WFSClient$new("http://www.fao.org/figis/geoserver/wfs",
                    serviceVersion = "1.0.0", logger = "INFO")
fao_areas = wfs$getFeatures("area:FAO_AREAS")
```

There is much more to learn about web services and much potential for development of R-OWS interfaces, an active area of development. For further information on the topic, we recommend examples from European Centre for Medium-Range Weather Forecasts (ECMWF) services at github.com/OpenDataHack[186] and reading-up on OCG Web Services at opengeospatial.org[187].

[185] To filter features on the server before downloading them, the argument `cql_filter` can be used. Adding `cql_filter = URLencode("F_CODE= '27'")` to the command, for example, would instruct the server to only return the feature with values in the `F_CODE` column equal to 27.
[186] `https://github.com/OpenDataHack/data_service_catalogue`
[187] `http://www.opengeospatial.org/standards`

0.7.5 File formats

Geographic datasets are usually stored as files or in spatial databases. File formats can either store vector or raster data, while spatial databases such as PostGIS[188] can store both (see also Section **??**). Today the variety of file formats may seem bewildering but there has been much consolidation and standardization since the beginnings of GIS software in the 1960s when the first widely distributed program (SYMAP[189]) for spatial analysis was created at Harvard University (Coppock and Rhind 1991).

GDAL (which should be pronounced "goo-dal", with the double "o" making a reference to object-orientation), the Geospatial Data Abstraction Library, has resolved many issues associated with incompatibility between geographic file formats since its release in 2000. GDAL provides a unified and high-performance interface for reading and writing of many raster and vector data formats. Many open and proprietary GIS programs, including GRASS, ArcGIS and QGIS, use GDAL behind their GUIs for doing the legwork of ingesting and spitting out geographic data in appropriate formats.

GDAL provides access to more than 200 vector and raster data formats. Table **??** presents some basic information about selected and often used spatial file formats.

An important development ensuring the standardization and open-sourcing of file formats was the founding of the Open Geospatial Consortium (OGC[190]) in 1994. Beyond defining the simple features data model (see Section **??**), the OGC also coordinates the development of open standards, for example as used in file formats such as KML and GeoPackage. Open file formats of the kind endorsed by the OGC have several advantages over proprietary formats: the standards are published, ensure transparency and open up the possibility for users to further develop and adjust the file formats to their specific needs.

ESRI Shapefile is the most popular vector data exchange format. However, it is not an open format (though its specification is open). It was developed in the early 1990s and has a number of limitations. First of all, it is a multi-file format, which consists of at least three files. It only supports 255 columns, column names are restricted to ten characters and the file size limit is 2 GB. Furthermore, ESRI Shapefile does not support all possible geometry types, for example, it is unable to distinguish between a polygon and a multipolygon.[191] Despite these limitations, a viable alternative had been missing for a long time.

In the meantime, GeoPackage[192] emerged, and seems to be a more than

[188] `https://trac.osgeo.org/postgis/wiki/WKTRaster`
[189] `https://news.harvard.edu/gazette/story/2011/10/the-invention-of-gis/`
[190] `http://www.opengeospatial.org/`
[191] To learn more about ESRI Shapefile limitations and possible alternative file formats, visit `http://switchfromshapefile.org/`.
[192] `https://www.geopackage.org/`

cliv

suitable replacement candidate for ESRI Shapefile. Geopackage is a format for exchanging geospatial information and an OGC standard. The GeoPackage standard describes the rules on how to store geospatial information in a tiny SQLite container. Hence, GeoPackage is a lightweight spatial database container, which allows the storage of vector and raster data but also of non-spatial data and extensions. Aside from GeoPackage, there are other geospatial data exchange formats worth checking out (Table **??**).

0.7.6 Data input (I)

Executing commands such as `sf::st_read()` (the main function we use for loading vector data) or `raster::raster()` (the main function used for loading raster data) silently sets off a chain of events that reads data from files. Moreover, there are many R packages containing a wide range of geographic data or providing simple access to different data sources. All of them load the data into R or, more precisely, assign objects to your workspace, stored in RAM accessible from the `.GlobalEnv`[193] of the R session.

0.7.6.1 Vector data

Spatial vector data comes in a wide variety of file formats, most of which can be read-in via the **sf** function `st_read()`. Behind the scenes this calls GDAL. To find out which data formats **sf** supports, run `st_drivers()`. Here, we show only the first five drivers (see Table **??**):

```
sf_drivers = st_drivers()
head(sf_drivers, n = 5)
```

The first argument of `st_read()` is `dsn`, which should be a text string or an object containing a single text string. The content of a text string could vary between different drivers. In most cases, as with the ESRI Shapefile (`.shp`) or the `GeoPackage` format (`.gpkg`), the `dsn` would be a file name. `st_read()` guesses the driver based on the file extension, as illustrated for a `.gpkg` file below:

```
vector_filepath = system.file("shapes/world.gpkg", package = "spData")
world = st_read(vector_filepath)
#> Reading layer `world' from data source `.../world.gpkg' using driver `GPKG'
#> Simple feature collection with 177 features and 10 fields
```

[193] `http://adv-r.had.co.nz/Environments.html`

```
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -180 ymin: -90 xmax: 180 ymax: 83.64513
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

For some drivers, `dsn` could be provided as a folder name, access credentials for a database, or a GeoJSON string representation (see the examples of the `st_read()` help page for more details).
Some vector driver formats can store multiple data layers. By default, `st_read()` automatically reads the first layer of the file specified in `dsn`; however, using the `layer` argument you can specify any other layer.
Naturally, some options are specific to certain drivers.[194] For example, think of coordinates stored in a spreadsheet format (`.csv`). To read in such files as spatial objects, we naturally have to specify the names of the columns (`X` and `Y` in our example below) representing the coordinates. We can do this with the help of the `options` parameter. To find out about possible options, please refer to the 'Open Options' section of the corresponding GDAL driver description.
For the comma-separated value (csv) format, visit `http://www.gdal.org/drv_csv.html`.

```
cycle_hire_txt = system.file("misc/cycle_hire_xy.csv", package = "spData")
cycle_hire_xy = st_read(cycle_hire_txt, options = c("X_POSSIBLE_NAMES=X",
                                                    "Y_POSSIBLE_NAMES=Y"))
```

Instead of columns describing xy-coordinates, a single column can also contain the geometry information. Well-known text (WKT), well-known binary (WKB), and the GeoJSON formats are examples of this. For instance, the `world_wkt.csv` file has a column named `WKT` representing polygons of the world's countries. We will again use the `options` parameter to indicate this. Here, we will use `read_sf()` which does exactly the same as `st_read()` except it does not print the driver name to the console and stores strings as characters instead of factors.

```
world_txt = system.file("misc/world_wkt.csv", package = "spData")
world_wkt = read_sf(world_txt, options = "GEOM_POSSIBLE_NAMES=WKT")
```

[194] A list of supported vector formats and options can be found at `http://gdal.org/ogr_formats.html`.

clvi

```
# the same as
world_wkt = st_read(world_txt, options = "GEOM_POSSIBLE_NAMES=WKT",
                    quiet = TRUE, stringsAsFactors = FALSE)
```

Not all of the supported vector file formats store information about their coordinate reference system. In these situations, it is possible to add the missing information using the `st_set_crs()` function. Please refer also to Section **??** for more information.

As a final example, we will show how `st_read()` also reads KML files. A KML file stores geographic information in XML format - a data format for the creation of web pages and the transfer of data in an application-independent way (Nolan and Lang 2014). Here, we access a KML file from the web. This file contains more than one layer. `st_layers()` lists all available layers. We choose the first layer `Placemarks` and say so with the help of the `layer` parameter in `read_sf()`.

```
u = "https://developers.google.com/kml/documentation/KML_Samples.kml"
download.file(u, "KML_Samples.kml")
st_layers("KML_Samples.kml")
#> Driver: LIBKML
#> Available layers:
#>                layer_name geometry_type features fields
#> 1              Placemarks                        3     11
#> 2      Styles and Markup                         1     11
#> 3       Highlighted Icon                         1     11
#> 4        Ground Overlays                         1     11
#> 5        Screen Overlays                         0     11
#> 6                  Paths                         6     11
#> 7               Polygons                         0     11
#> 8          Google Campus                         4     11
#> 9       Extruded Polygon                         1     11
#> 10 Absolute and Relative                         4     11
kml = read_sf("KML_Samples.kml", layer = "Placemarks")
```

All the examples presented in this section so far have used the **sf** package for geographic data import. It is fast and flexible but it may be worth looking at other packages for specific file formats. An example is the **geojsonsf** package.

A benchmark[195] suggests it is around 10 times faster than the **sf** package for reading `.geojson`.

0.7.6.2 Raster data

Similar to vector data, raster data comes in many file formats with some of them supporting even multilayer files. **raster**'s `raster()` command reads in a single layer.

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
single_layer = raster(raster_filepath)
```

In case you want to read in a single band from a multilayer file, use the `band` parameter to indicate a specific layer.

```
multilayer_filepath = system.file("raster/landsat.tif", package = "spDataLarge")
band3 = raster(multilayer_filepath, band = 3)
```

If you want to read in all bands, use `brick()` or `stack()`.

```
multilayer_brick = brick(multilayer_filepath)
multilayer_stack = stack(multilayer_filepath)
```

Please refer to Section **??** for information on the difference between raster stacks and bricks.

0.7.7 Data output (O)

Writing geographic data allows you to convert from one format to another and to save newly created objects. Depending on the data type (vector or raster), object class (e.g., `multipoint` or `RasterLayer`), and type and amount of stored information (e.g., object size, range of values), it is important to know how to store spatial files in the most efficient way. The next two sections will demonstrate how to do this.

0.7.7.1 Vector data

The counterpart of `st_read()` is `st_write()`. It allows you to write **sf** objects to a wide range of geographic vector file formats, including the most common

[195] `https://github.com/ATFutures/geobench`

such as `.geojson`, `.shp` and `.gpkg`. Based on the file name, `st_write()` decides automatically which driver to use. The speed of the writing process depends also on the driver.

```
st_write(obj = world, dsn = "world.gpkg")
#> Writing layer `world' to data source `world.gpkg' using driver `GPKG'
#> features:       177
#> fields:         10
#> geometry type:  Multi Polygon
```

Note: if you try to write to the same data source again, the function will fail:

```
st_write(obj = world, dsn = "world.gpkg")
#> Updating layer `world' to data source `world.gpkg' using driver `GPKG'
#> Creating layer world failed.
#> Error in CPL_write_ogr(obj, dsn, layer, driver, ...),  :
#>   Layer creation failed.
#> In addition: Warning message:
#> In CPL_write_ogr(obj, dsn, layer, driver, ...),  :
#>   GDAL Error 1: Layer world already exists, CreateLayer failed.
#> Use the layer creation option OVERWRITE=YES to replace it.
```

The error message provides some information as to why the function failed. The `GDAL Error 1` statement makes clear that the failure occurred at the GDAL level. Additionally, the suggestion to use `OVERWRITE=YES` provides a clue about how to fix the problem. However, this is not a `st_write()` argument, it is a GDAL option. Luckily, `st_write` provides a `layer_options` argument through which we can pass driver-dependent options:

```
st_write(obj = world, dsn = "world.gpkg", layer_options = "OVERWRITE=YES")
```

Another solution is to use the `st_write()` argument `delete_layer`. Setting it to `TRUE` deletes already existing layers in the data source before the function attempts to write (note there is also a `delete_dsn` argument):

```
st_write(obj = world, dsn = "world.gpkg", delete_layer = TRUE)
```

You can achieve the same with `write_sf()` since it is equivalent to (technically

an *alias* for) `st_write()`, except that its defaults for `delete_layer` and `quiet` is `TRUE`.

```
write_sf(obj = world, dsn = "world.gpkg")
```

The `layer_options` argument could be also used for many different purposes. One of them is to write spatial data to a text file. This can be done by specifying `GEOMETRY` inside of `layer_options`. It could be either `AS_XY` for simple point datasets (it creates two new columns for coordinates) or `AS_WKT` for more complex spatial data (one new column is created which contains the well-known text representation of spatial objects).

```
st_write(cycle_hire_xy, "cycle_hire_xy.csv", layer_options = "GEOMETRY=AS_XY")
st_write(world_wkt, "world_wkt.csv", layer_options = "GEOMETRY=AS_WKT")
```

0.7.7.2 Raster data

The `writeRaster()` function saves `Raster*` objects to files on disk. The function expects input regarding output data type and file format, but also accepts GDAL options specific to a selected file format (see `?writeRaster` for more details).

The **raster** package offers nine data types when saving a raster: LOG1S, INT1S, INT1U, INT2S, INT2U, INT4S, INT4U, FLT4S, and FLT8S.[196] The data type determines the bit representation of the raster object written to disk (Table **??**). Which data type to use depends on the range of the values of your raster object. The more values a data type can represent, the larger the file will get on disk. Commonly, one would use LOG1S for bitmap (binary) rasters. Unsigned integers (INT1U, INT2U, INT4U) are suitable for categorical data, while float numbers (FLT4S and FLT8S) usually represent continuous data. `writeRaster()` uses FLT4S as the default. While this works in most cases, the size of the output file will be unnecessarily large if you save binary or categorical data. Therefore, we would recommend to use the data type that needs the least storage space, but is still able to represent all values (check the range of values with the `summary()` function).

The file extension determines the output file when saving a `Raster*` object to disk. For example, the `.tif` extension will create a GeoTIFF file:

[196] Using INT4U is not recommended as R does not support 32-bit unsigned integers.

clx

```
writeRaster(x = single_layer,
            filename = "my_raster.tif",
            datatype = "INT2U")
```

The `raster` file format (native to the `raster` package) is used when a file extension is invalid or missing. Some raster file formats come with additional options. You can use them with the `options` parameter[197]. GeoTIFF files, for example, can be compressed using `COMPRESS`:

```
writeRaster(x = single_layer,
            filename = "my_raster.tif",
            datatype = "INT2U",
            options = c("COMPRESS=DEFLATE"),
            overwrite = TRUE)
```

Note that `writeFormats()` returns a list with all supported file formats on your computer.

0.7.8 Visual outputs

R supports many different static and interactive graphics formats. The most general method to save a static plot is to open a graphic device, create a plot, and close it, for example:

```
png(filename = "lifeExp.png", width = 500, height = 350)
plot(world["lifeExp"])
dev.off()
```

Other available graphic devices include `pdf()`, `bmp()`, `jpeg()`, `png()`, and `tiff()`. You can specify several properties of the output plot, including width, height and resolution.
Additionally, several graphic packages provide their own functions to save a graphical output. For example, the **tmap** package has the `tmap_save()` function. You can save a `tmap` object to different graphic formats by specifying the object name and a file path to a new graphic file.

[197] `http://www.gdal.org/formats_list.html`

```
library(tmap)
tmap_obj = tm_shape(world) +
  tm_polygons(col = "lifeExp")
tmap_save(tm  = tmap_obj, filename = "lifeExp_tmap.png")
```

On the other hand, you can save interactive maps created in the `mapview` package as an HTML file or image using the `mapshot()` function:

```
library(mapview)
mapview_obj = mapview(world, zcol = "lifeExp", legend = TRUE)
mapshot(mapview_obj, file = "my_interactive_map.html")
```

0.7.9 Exercises

1. List and describe three types of vector, raster, and geodatabase formats.
2. Name at least two differences between `read_sf()` and the more well-known function `st_read()`.
3. Read the `cycle_hire_xy.csv` file from the **spData** package as a spatial object (Hint: it is located in the `misc\` folder). What is a geometry type of the loaded object?
4. Download the borders of Germany using **rnaturalearth**, and create a new object called `germany_borders`. Write this new object to a file of the GeoPackage format.
5. Download the global monthly minimum temperature with a spatial resolution of five minutes using the **raster** package. Extract the June values, and save them to a file named `tmin_june.tif` file (hint: use `raster::subset()`).
6. Create a static map of Germany's borders, and save it to a PNG file.
7. Create an interactive map using data from the `cycle_hire_xy.csv` file. Export this map to a file called `cycle_hire.html`.

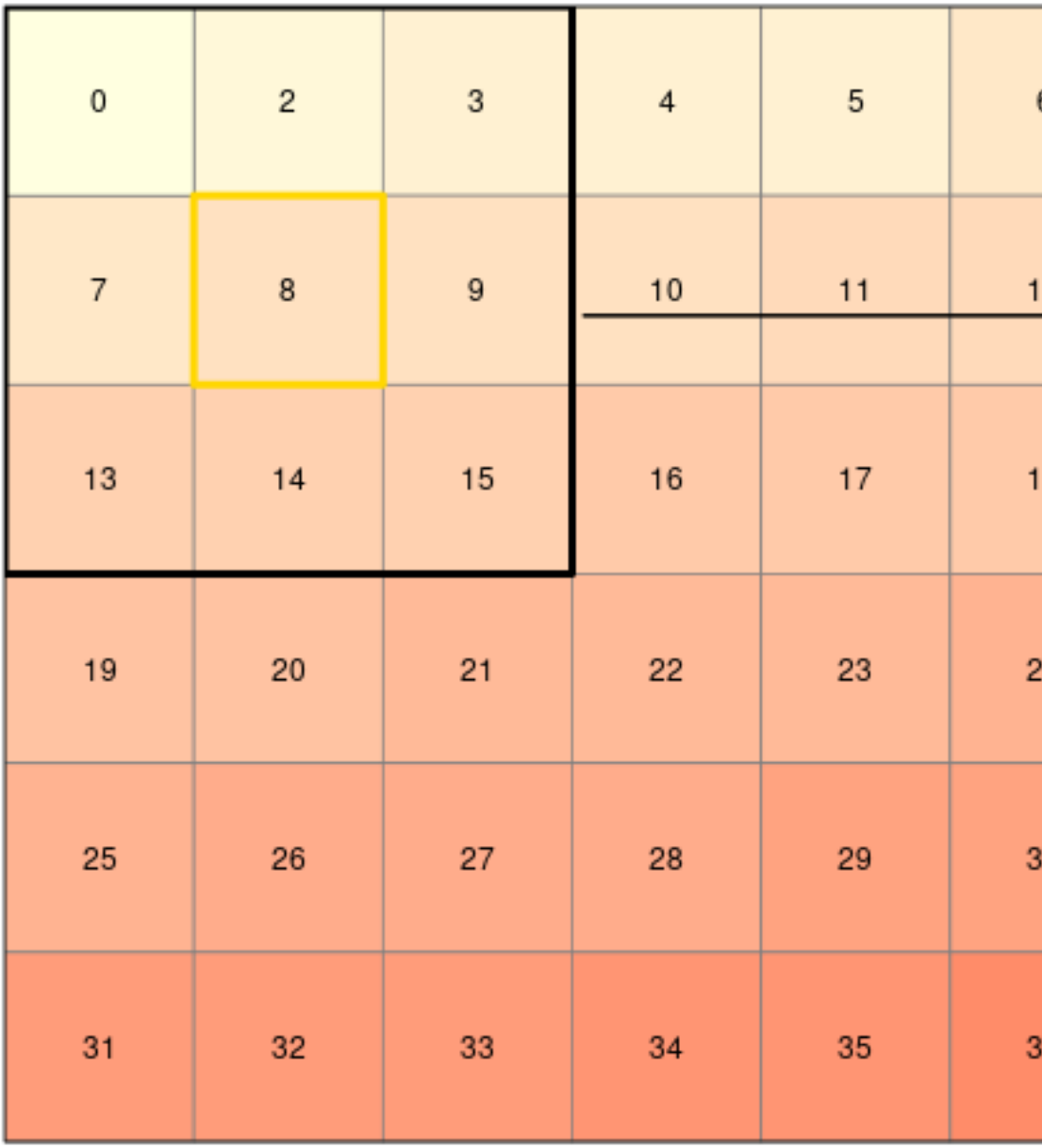


FIGURE 24: Input raster (left) and resulting output raster (right) due to a focal operation - summing up 3-by-3 windows.

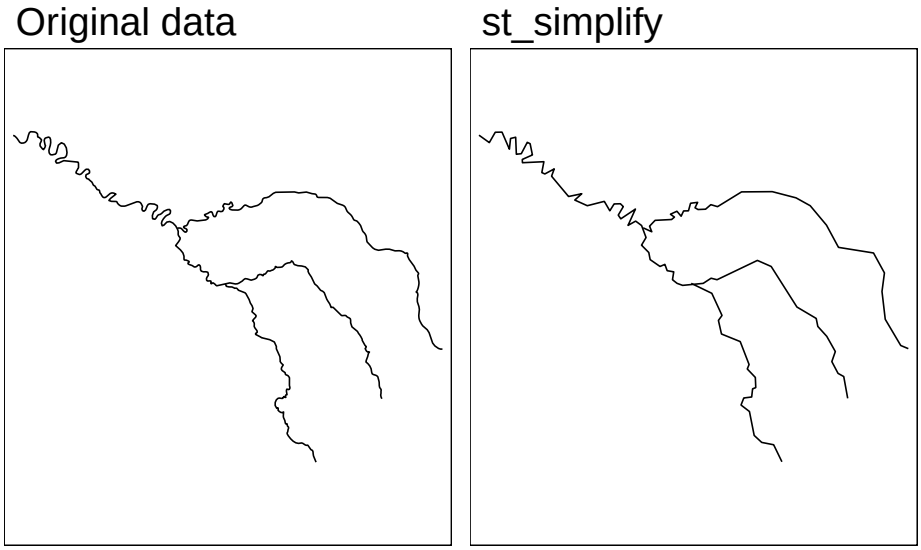


FIGURE 25: Comparison of the original and simplified geometry of the seine object.

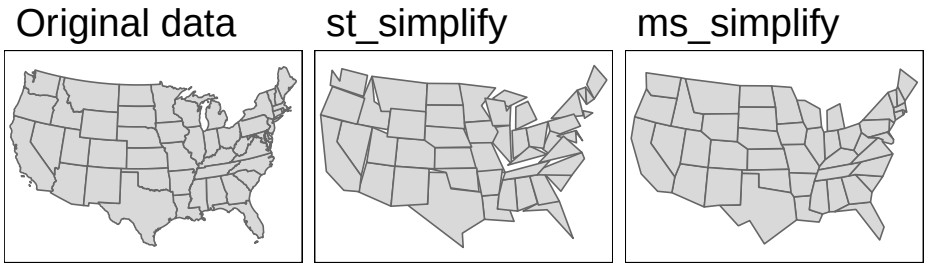


FIGURE 26: Polygon simplification in action, comparing the original geometry of the contiguous United States with simplified versions, generated with functions from sf (center) and rmapshaper (right) packages.

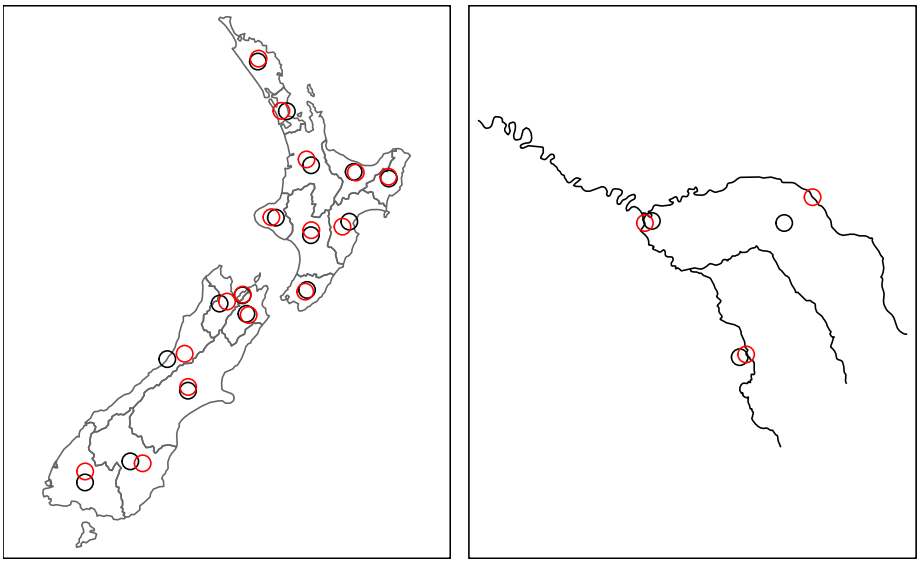


FIGURE 27: Centroids (black points) and 'points on surface' (red points) of New Zealand's regions (left) and the Seine (right) datasets.

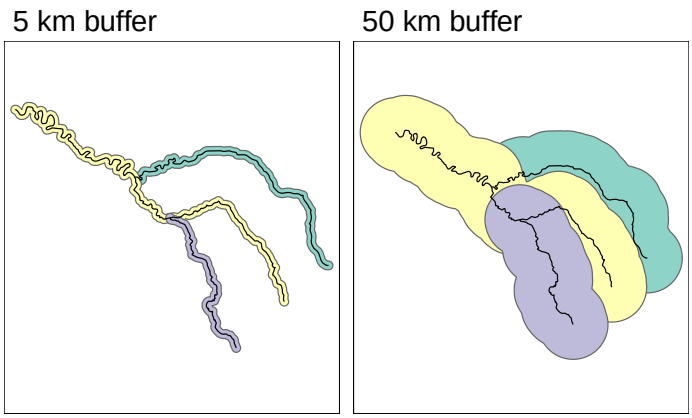


FIGURE 28: Buffers around the Seine dataset of 5 km (left) and 50 km (right). Note the colors, which reflect the fact that one buffer is created per geometry feature.

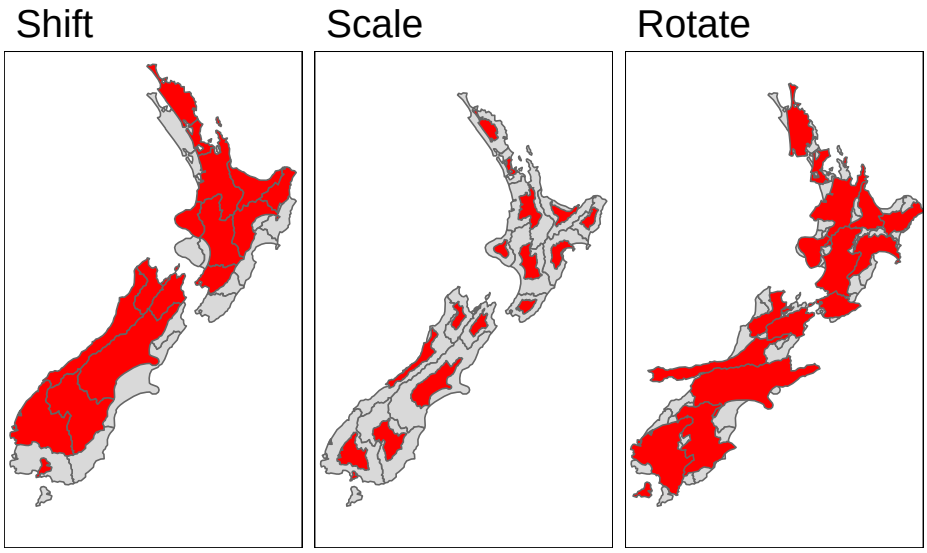


FIGURE 29: Illustrations of affine transformations: shift, scale and rotate.

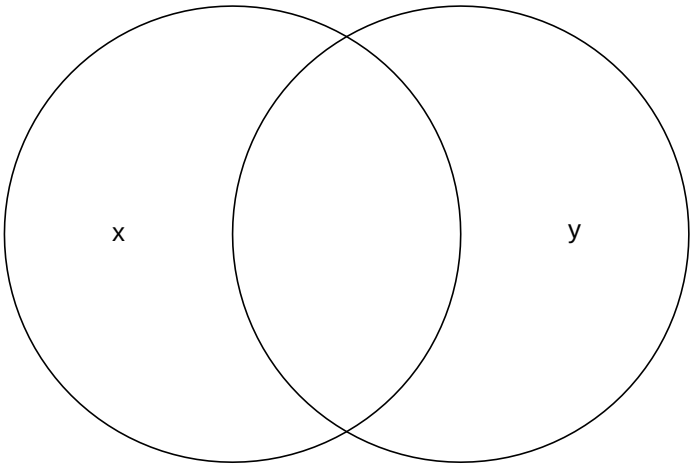


FIGURE 30: Overlapping circles.

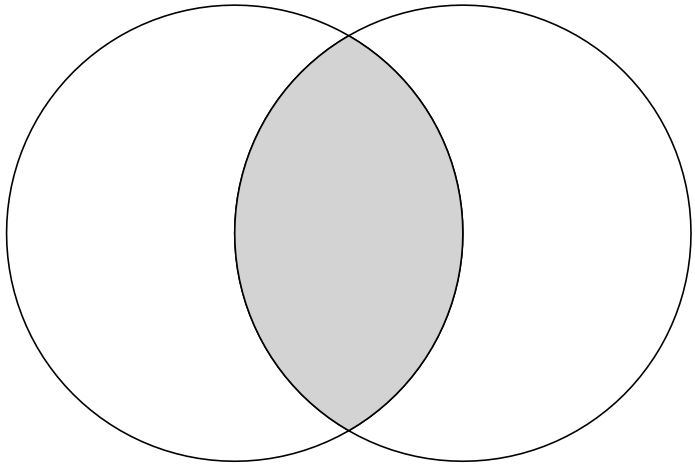


FIGURE 31: Overlapping circles with a gray color indicating intersection between them.

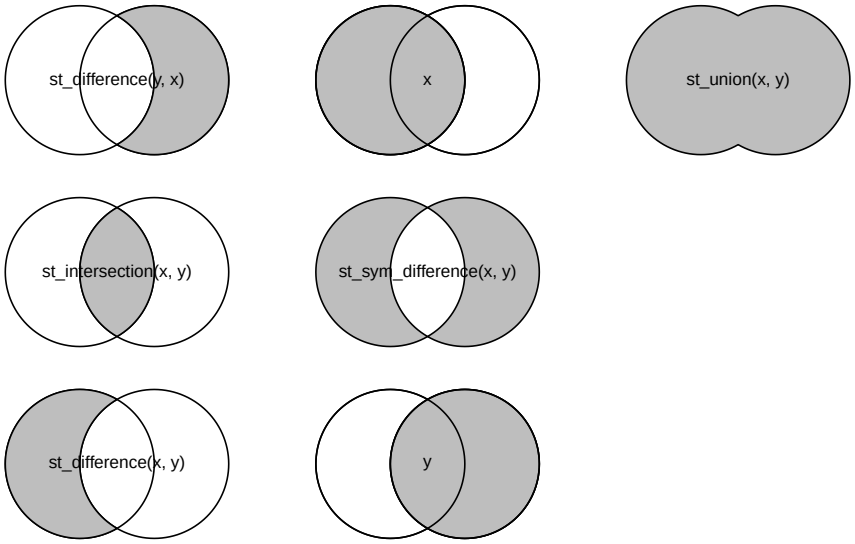


FIGURE 32: Spatial equivalents of logical operators.

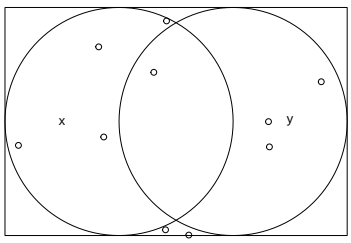


FIGURE 33: Randomly distributed points within the bounding box enclosing circles x and y.

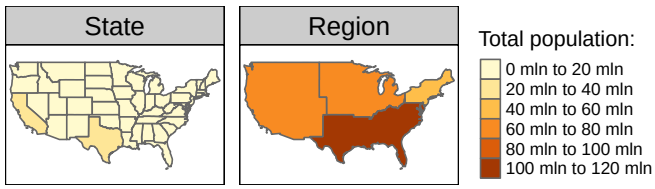


FIGURE 34: Spatial aggregation on contiguous polygons, illustrated by aggregating the population of US states into regions, with population represented by color. Note the operation automatically dissolves boundaries between states.

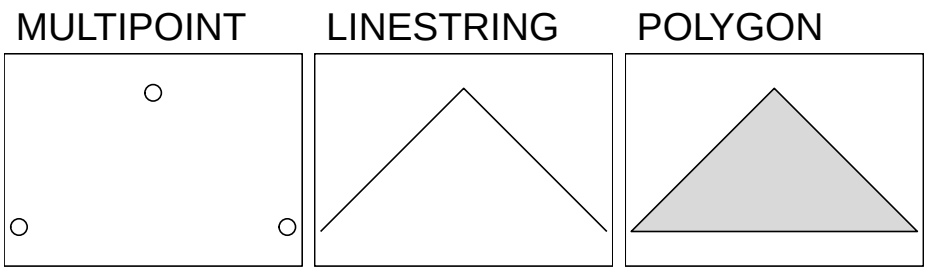


FIGURE 35: Examples of linestring and polygon casted from a multipoint geometry.

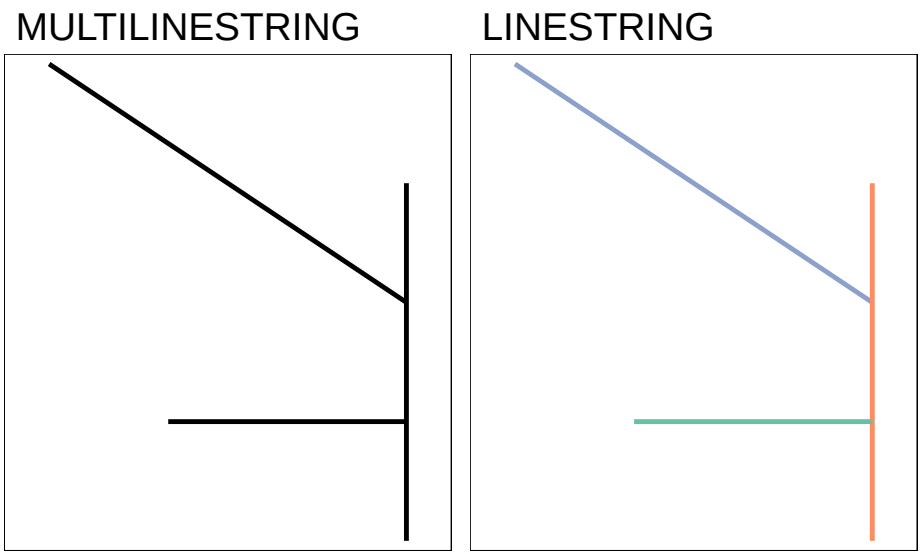


FIGURE 36: Examples of type casting between MULTILINESTRING (left) and LINESTRING (right).

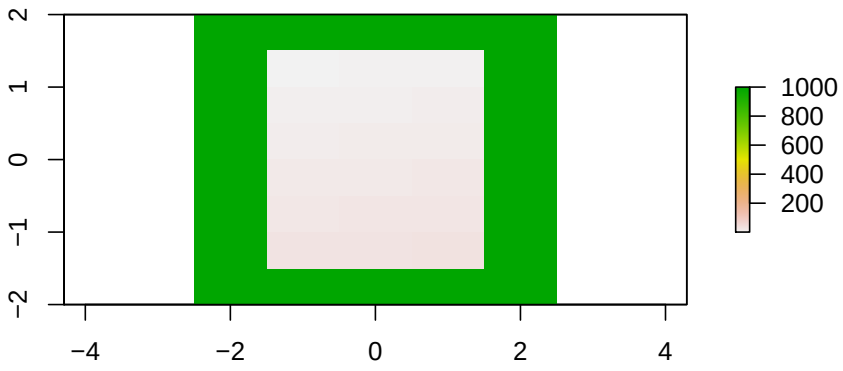


FIGURE 37: Original raster extended by one row on each side (top, bottom) and two columns on each side (right, left).

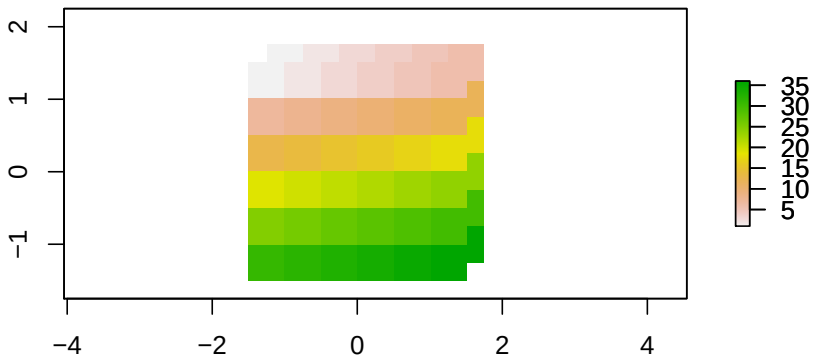


FIGURE 38: Rasters with identical values but different origins.

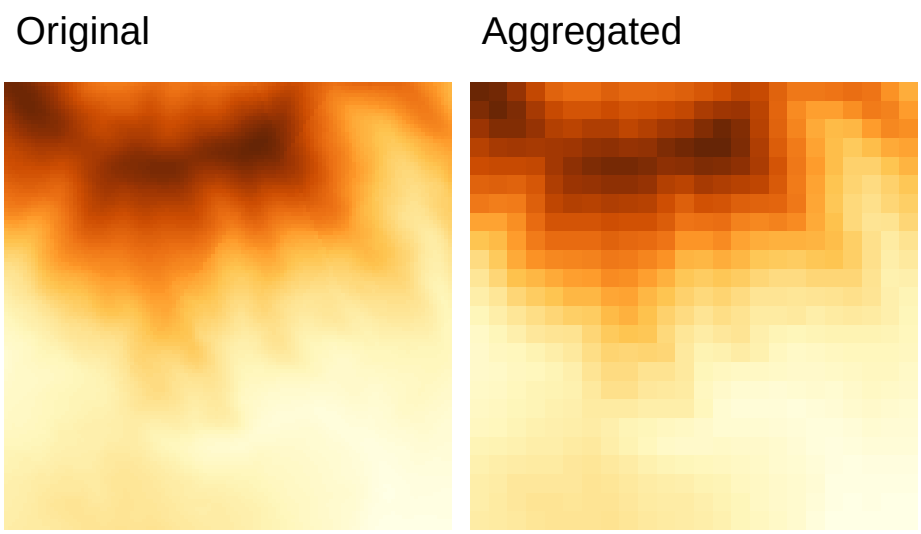


FIGURE 39: Original raster (left). Aggregated raster (right).

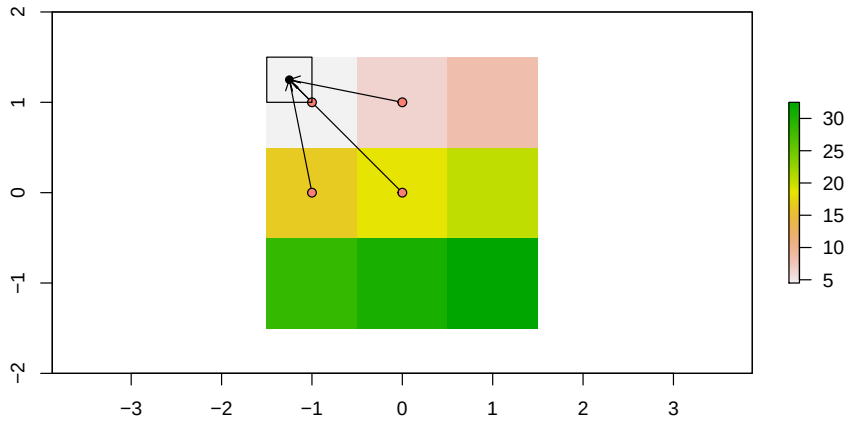


FIGURE 40: Bilinear disaggregation in action.

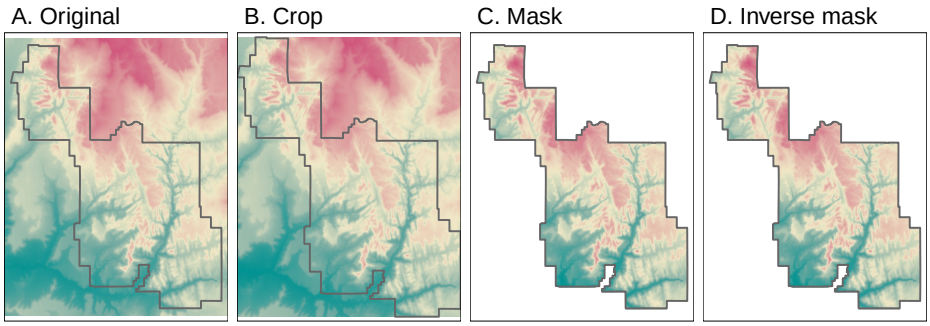


FIGURE 41: Illustration of raster cropping and raster masking.

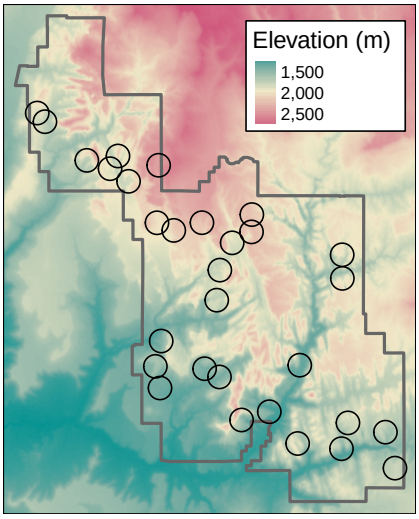


FIGURE 42: Locations of points used for raster extraction.

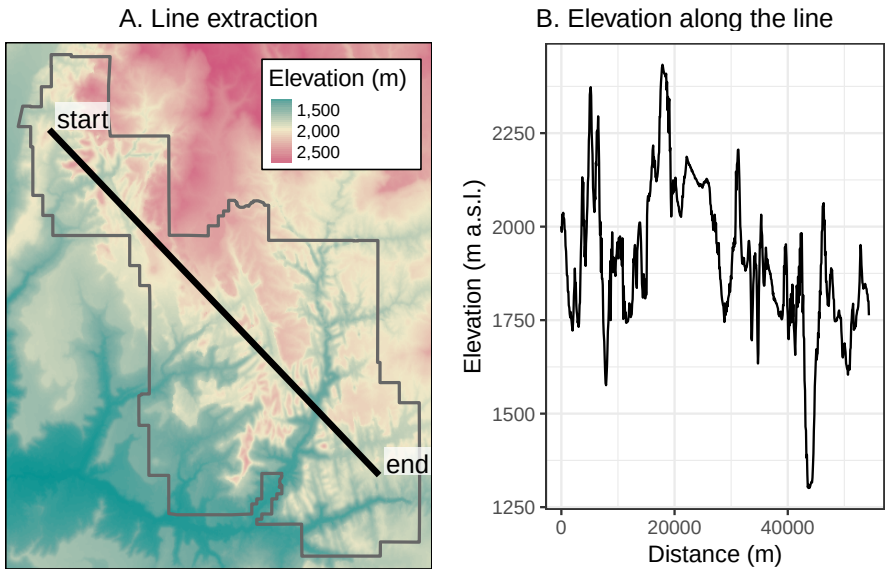


FIGURE 43: Location of a line used for raster extraction (left) and the elevation along this line (right).

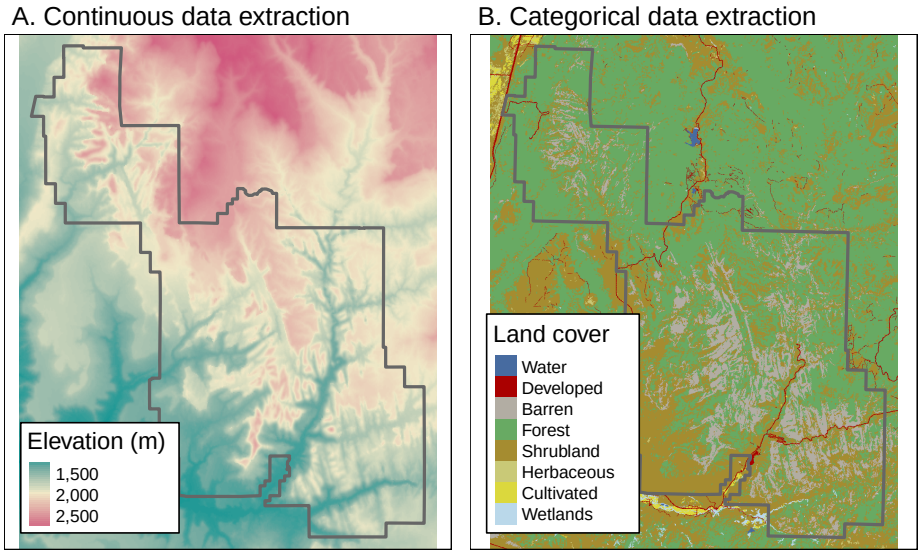


FIGURE 44: Area used for continuous (left) and categorical (right) raster extraction.

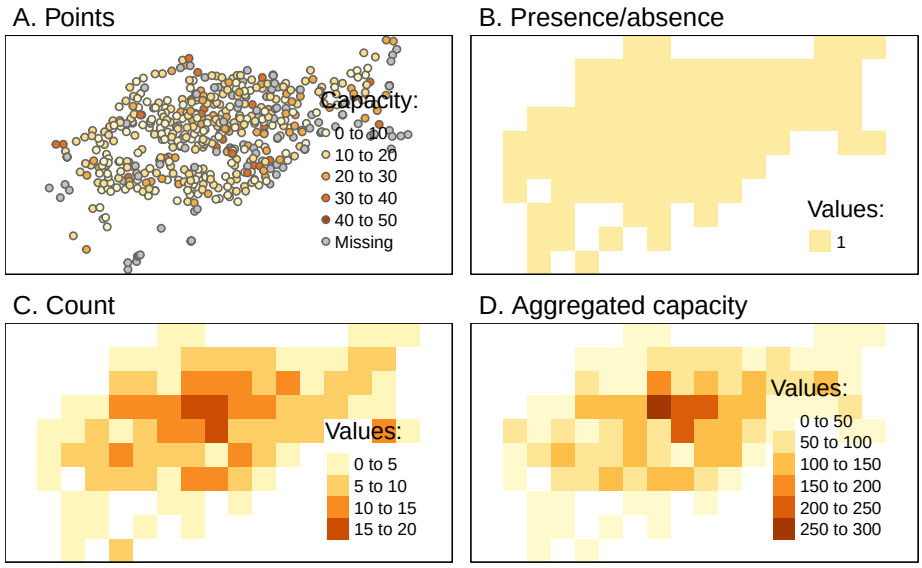


FIGURE 45: Examples of point rasterization.

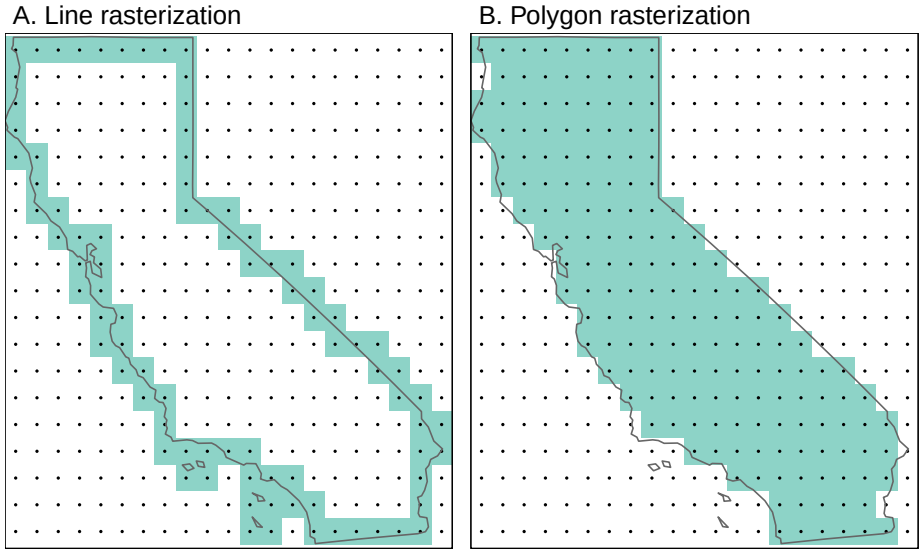


FIGURE 46: Examples of line and polygon rasterizations.

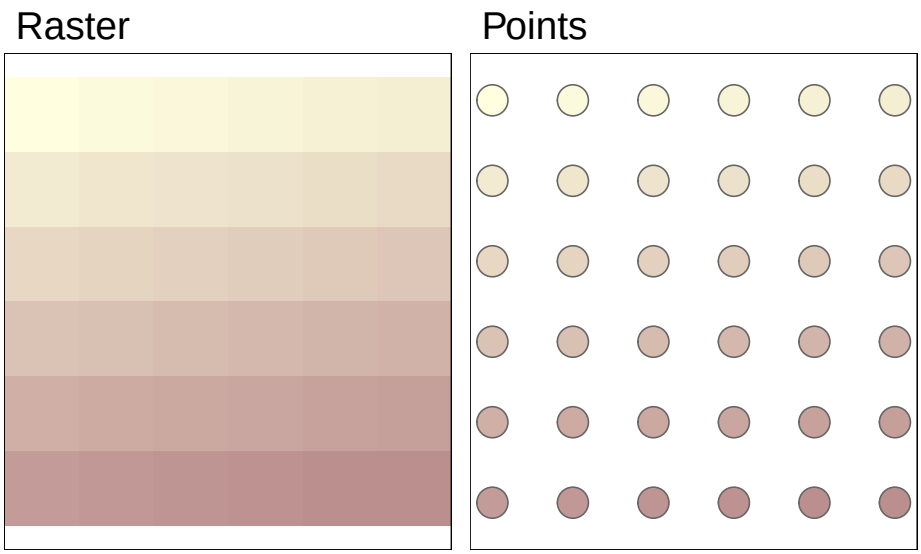


FIGURE 47: Raster and point representation of the elev object.

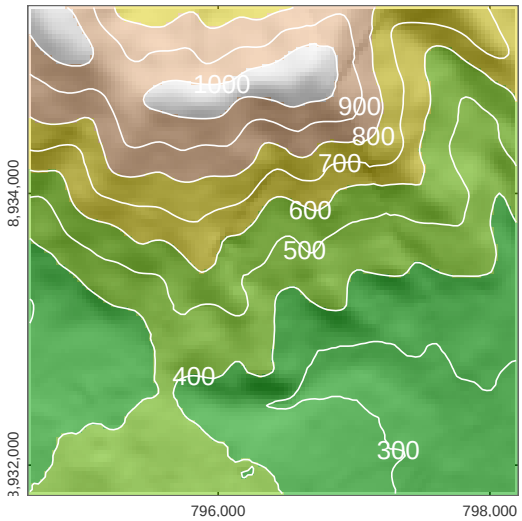


FIGURE 48: DEM hillshade of the southern flank of Mt. Mongón overlaid by contour lines.

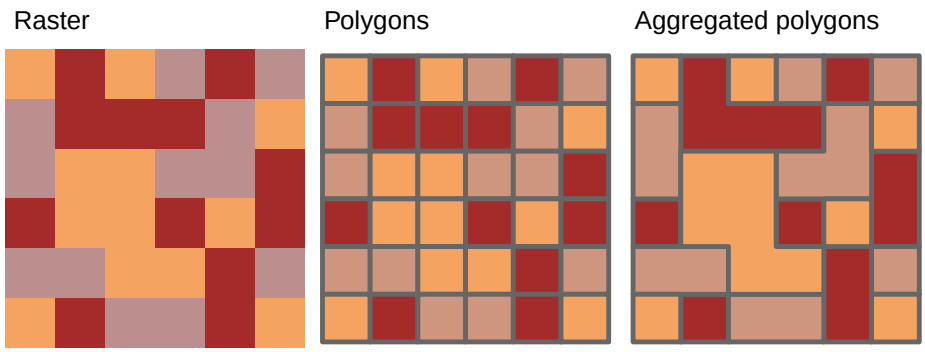


FIGURE 49: Illustration of vectorization of raster (left) into polygon (center) and polygon aggregation (right).

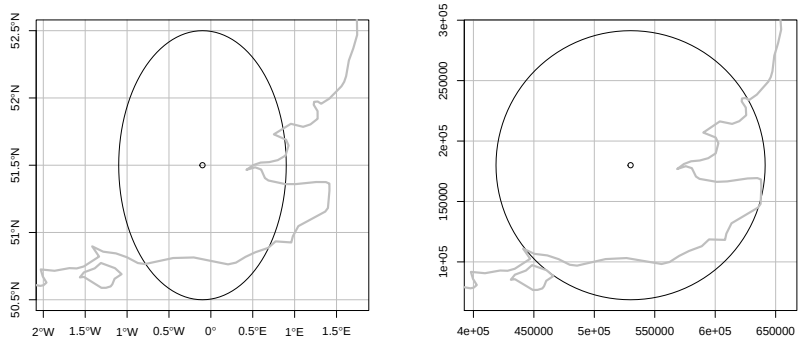


FIGURE 50: Buffers around London with a geographic (left) and projected (right) CRS. The gray outline represents the UK coastline.

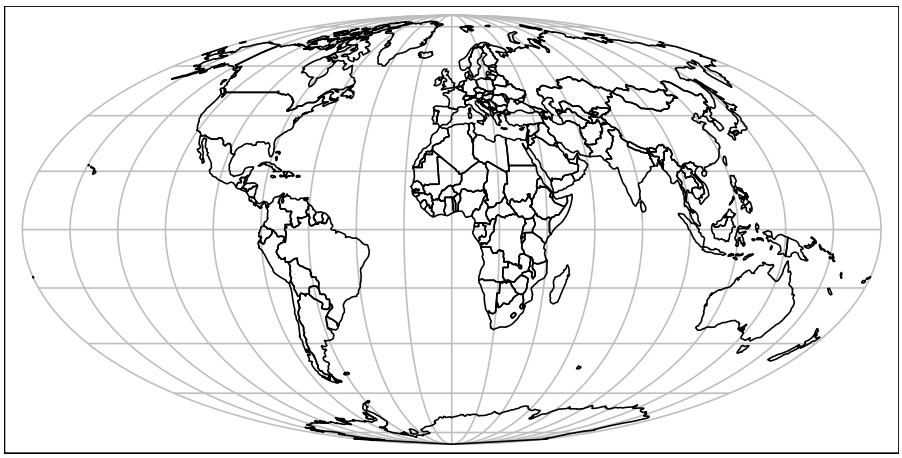


FIGURE 51: Mollweide projection of the world.

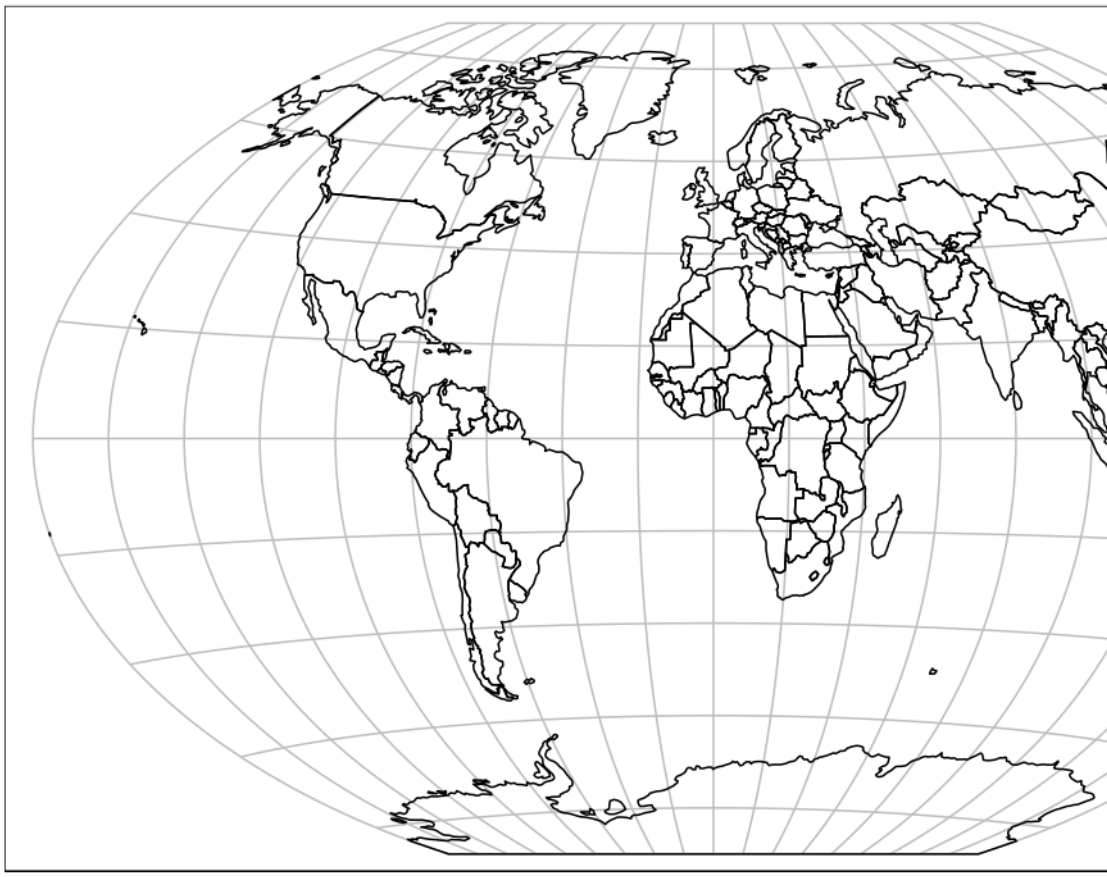


FIGURE 52: Winkel tripel projection of the world.

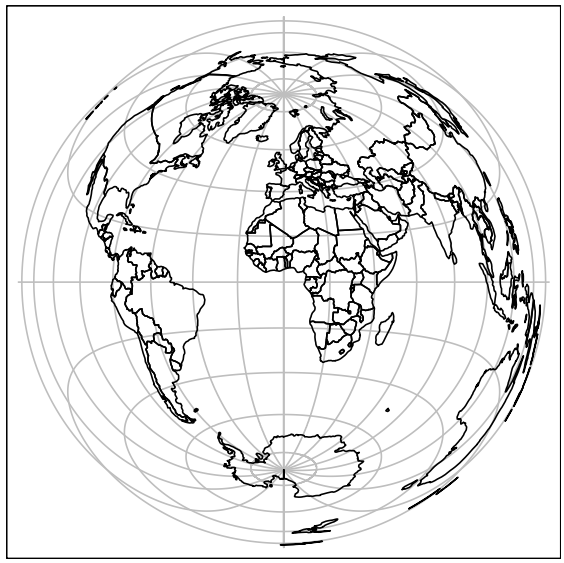


FIGURE 53: Lambert azimuthal equal-area projection of the world centered on longitude and latitude of 0.

FIGURE 54: Lambert azimuthal equal-area projection of the world centered on New York City.

TABLE 0.7: Selected spatial file formats.

Name	Extension	Info	Type	Mo
ESRI Shapefile	.shp (the main file)	Popular format consisting of at least three files. No support for: files > 2GB; mixed types; names > 10 chars; cols > 255.	Vector	Par
GeoJSON	.geojson	Extends the JSON exchange format by including a subset of the simple feature representation.	Vector	Op
KML	.kml	XML-based format for spatial visualization, developed for use with Google Earth. Zipped KML file forms the KMZ format.	Vector	Op
GPX	.gpx	XML schema created for exchange of GPS data.	Vector	Op
GeoTIFF	.tiff	Popular raster format similar to '.tif' format but stores raster header.	Raster	Op
Arc ASCII	.asc	Text format where the first six lines represent the raster header, followed by the raster cell values arranged in rows and columns.	Raster	Op
R-raster	.gri, .grd	Native raster format of the R-package raster.	Raster	Op
SQLite/SpatiaLite	.sqlite	Standalone relational database, SpatiaLite is the spatial extension of SQLite.	Vector and raster	Op
ESRI FileGDB	.gdb	Spatial and nonspatial objects created by ArcGIS. Allows: multiple feature classes; topology. Limited support from GDAL.	Vector and raster	Pro
GeoPackage	.gpkg	Lightweight database container based on SQLite allowing an easy and platform-independent exchange of geodata	Vector and raster	Op

TABLE 0.8: Sample of available drivers for reading/writing vector data (it could vary between different GDAL versions).

name	long_name	write	copy	is_raster	is_vector	vsi
ESRI Shapefile	ESRI Shapefile	TRUE	FALSE	FALSE	TRUE	TRUE
GPX	GPX	TRUE	FALSE	FALSE	TRUE	TRUE
KML	Keyhole Markup Language (KML)	TRUE	FALSE	FALSE	TRUE	TRUE
GeoJSON	GeoJSON	TRUE	FALSE	FALSE	TRUE	TRUE
GPKG	GeoPackage	TRUE	TRUE	TRUE	TRUE	TRUE

TABLE 0.9: Data types supported by the raster package.

Data type	Minimum value	Maximum value
LOG1S	FALSE (0)	TRUE (1)
INT1S	-127	127
INT1U	0	255
INT2S	-32,767	32,767
INT2U	0	65,534
INT4S	-2,147,483,647	2,147,483,647
INT4U	0	4,294,967,296
FLT4S	-3.4e+38	3.4e+38
FLT8S	-1.7e+308	1.7e+308

Part II

Extensions

0.8 Making maps with R

Prerequisites

- This chapter requires the following packages that we have already been using:

```
library(sf)
library(raster)
library(dplyr)
library(spData)
library(spDataLarge)
```

- In addition, it uses the following visualization packages:

```
library(tmap)    # for static and interactive maps
library(leaflet) # for interactive maps
library(mapview) # for interactive maps
library(ggplot2) # tidyverse vis package
library(shiny)   # for web applications
```

0.8.1 Introduction

A satisfying and important aspect of geographic research is communicating the results. Map making — the art of cartography — is an ancient skill that involves communication, intuition, and an element of creativity. Static mapping is straightforward with `plot()`, as we saw in Section **??**. It is possible to create advanced maps using base R methods (Murrell 2016), but this chapter focuses on dedicated map-making packages. When learning a new skill, it makes sense to gain depth-of-knowledge in one area branching out. Map making is no exception, hence this chapter's coverage of one package (**tmap**) in depth rather than many superficially. In addition to being fun and creative, cartography also has important practical applications. A carefully crafted map is vital for effectively communicating the results of your work (Brewer 2015):

Amateur-looking maps can undermine your audience's

ability to understand important information and weaken the presentation of a professional data investigation.

Maps have been used for several thousand years for a wide variety of purposes. Historic examples include maps of buildings and land ownership in the Old Babylonian dynasty more than 3000 years ago and Ptolemy's world map in his masterpiece *Geography* nearly 2000 years ago (Talbert 2014).

Map making has historically been an activity undertaken only by, or on behalf of, the elite. This has changed with the emergence of open source mapping software such as the R package **tmap** and the 'print composer' in QGIS which enable anyone to make high-quality maps, enabling 'citizen science'. Maps are also often the best way to present the findings of geocomputational research in a way that is accessible. Map making is therefore a critical part of geocomputation and its emphasis not only on describing, but also *changing* the world.

This chapter shows how to make a wide range of maps. The next section covers a range of static maps, including aesthetic considerations, facets and inset maps. Sections **??** to **??** cover animated and interactive maps (including web maps and mapping applications). Finally, Section **??** covers a range of alternative map-making packages including **ggplot2** and **cartogram**.

0.8.2 Static maps

Static maps are the most common type of visual output from geocomputation. Fixed images for printed outputs, common formats for static maps include `.png` and `.pdf`, for raster and vector outputs, respectively (interactive maps are covered in Section **??**). Initially static maps were the *only* type of map that R could produce. Things have advanced greatly since **sp** was released (see Pebesma and Bivand 2005). Many new techniques for map making have been developed since then. However, a decade later static plotting was still the emphasis of geographic data visualisation in R (Cheshire and Lovelace 2015).

Despite the innovation of interactive mapping in R, static maps are still the foundation of mapping in R. The generic `plot()` function is often the fastest way to create static maps from vector and raster spatial objects, as shown in Sections **??** and **??**. Sometimes simplicity and speed are priorities, especially during the development phase of a project, and this is where `plot()` excels. The base R approach is also extensible, with `plot()` offering dozens of arguments. Another low-level approach is the **grid** package, which provides functions for low-level control of graphical outputs, — see *R Graphics* (Murrell 2016),

especially Chapter 14[198]. The focus of this section, however, is making static maps with **tmap**.

Why **tmap**? It is a powerful and flexible map-making package with sensible defaults. It has a concise syntax that allows for the creation of attractive maps with minimal code, which will be familiar to **ggplot2** users. Furthermore, **tmap** has a unique capability to generate static and interactive maps using the same code via `tmap_mode()`. It accepts a wider range of spatial classes (including `raster` objects) than alternatives such as **ggplot2**, as documented in vignettes `tmap-getstarted`[199] and `tmap-changes-v2`[200] and an academic paper on the subject (Tennekes 2018). This section teaches how to make static maps with **tmap**, emphasizing the important aesthetic and layout options.

0.8.2.1 tmap basics

Like **ggplot2**, **tmap** is based on the idea of a 'grammar of graphics' (Wilkinson and Wills 2005). This involves a separation between the input data and the aesthetics (how data are visualised): each input dataset can be 'mapped' in a range of different ways including location on the map (defined by data's `geometry`), color, and other visual variables. The basic building block is `tm_shape()` (which defines input data, raster and vector objects), followed by one or more layer elements such as `tm_fill()` and `tm_dots()`. This layering is demonstrated in the chunk below, which generates the maps presented in Figure **??**:

```
# Add fill layer to nz shape
tm_shape(nz) +
  tm_fill()
# Add border layer to nz shape
tm_shape(nz) +
  tm_borders()
# Add fill and border layers to nz shape
tm_shape(nz) +
  tm_fill() +
  tm_borders()
```

The object passed to `tm_shape()` in this case is `nz`, an `sf` object representing the regions of New Zealand (see Section **??** for more on `sf` objects). Layers are added to represent `nz` visually, with `tm_fill()` and `tm_borders()` creating

[198] `https://www.stat.auckland.ac.nz/~paul/RG2e/chapter14.html`
[199] `https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html`
[200] `https://cran.r-project.org/web/packages/tmap/vignettes/tmap-changes-v2.html`

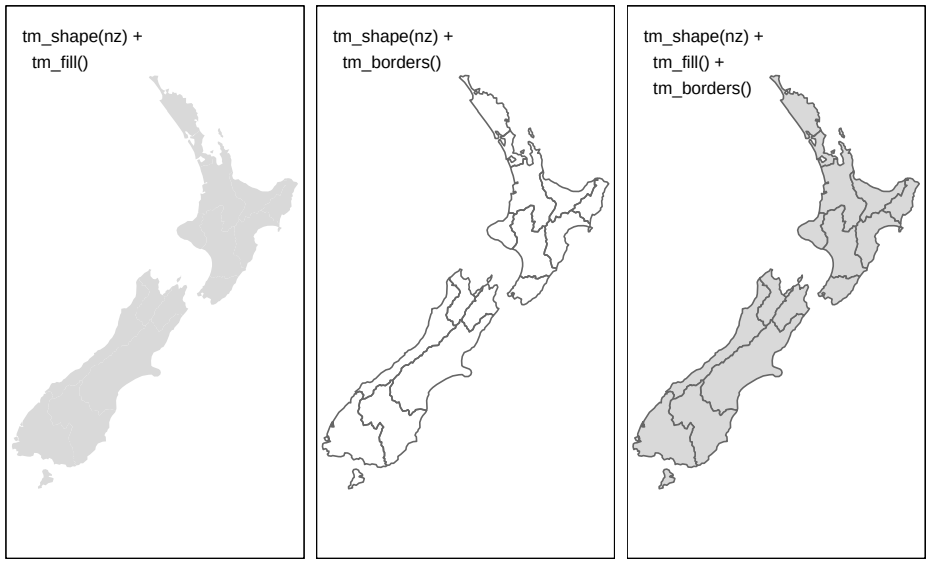


FIGURE 55: New Zealand's shape plotted with fill (left), border (middle) and fill and border (right) layers added using tmap functions.

shaded areas (left panel) and border outlines (middle panel) in Figure **??**, respectively.

This is an intuitive approach to map making: the common task of *adding* new layers is undertaken by the addition operator `+`, followed by `tm_*()`. The asterisk (*) refers to a wide range of layer types which have self-explanatory names including `fill`, `borders` (demonstrated above), `bubbles`, `text` and `raster` (see `help("tmap-element"` for a full list). This layering is illustrated in the right panel of Figure **??**, the result of adding a border *on top of* the fill layer.

`qtm()` is a handy function for **q**uickly creating **t**map **m**aps (hence the snappy name). It is concise and provides a good default visualization in many cases: `qtm(nz)`, for example, is equivalent to `tm_shape(nz) + tm_fill() + tm_borders()`. Further, layers can be added concisely using multiple `qtm()` calls, such as `qtm(nz) + qtm(nz_height)`. The disadvantage is that it makes aesthetics of individual layers harder to control, explaining why we avoid teaching it in this chapter.

0.8.2.2 Map objects

A useful feature of **tmap** is its ability to store *objects* representing maps. The code chunk below demonstrates this by saving the last plot in Figure **??** as an

object of class `tmap` (note the use of `tm_polygons()` which condenses `tm_fill() + tm_borders()` into a single function):

```
map_nz = tm_shape(nz) + tm_polygons()
class(map_nz)
#> [1] "tmap"
```

`map_nz` can be plotted later, for example by adding additional layers (as shown below) or simply running `map_nz` in the console, which is equivalent to `print(map_nz)`.

New *shapes* can be added with `+ tm_shape(new_obj)`. In this case `new_obj` represents a new spatial object to be plotted on top of preceding layers. When a new shape is added in this way, all subsequent aesthetic functions refer to it, until another new shape is added. This syntax allows the creation of maps with multiple shapes and layers, as illustrated in the next code chunk which uses the function `tm_raster()` to plot a raster layer (with `alpha` set to make the layer semi-transparent):

```
map_nz1 = map_nz +
  tm_shape(nz_elev) + tm_raster(alpha = 0.7)
```

Building on the previously created `map_nz` object, the preceding code creates a new map object `map_nz1` that contains another shape (`nz_elev`) representing average elevation across New Zealand (see Figure **??**, left). More shapes and layers can be added, as illustrated in the code chunk below which creates `nz_water`, representing New Zealand's territorial waters[201], and adds the resulting lines to an existing map object.

```
nz_water = st_union(nz) %>% st_buffer(22200) %>%
  st_cast(to = "LINESTRING")
map_nz2 = map_nz1 +
  tm_shape(nz_water) + tm_lines()
```

There is no limit to the number of layers or shapes that can be added to `tmap` objects. The same shape can even be used multiple times. The final map illustrated in Figure **??** is created by adding a layer representing high points (stored in the object `nz_height`) onto the previously created `map_nz2` object

[201] `https://en.wikipedia.org/wiki/Territorial_waters`

cxc

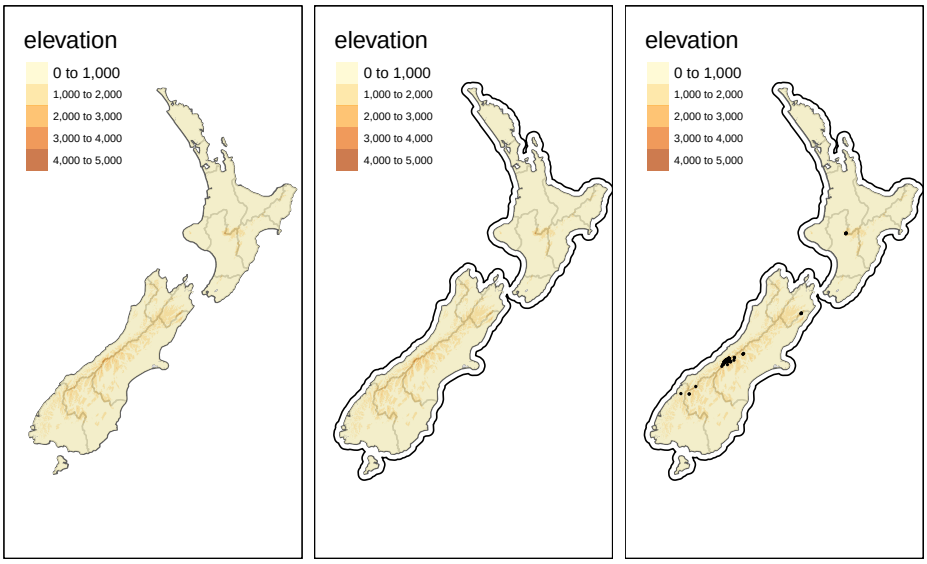


FIGURE 56: Maps with additional layers added to the final map of Figure 8.1.

with `tm_dots()` (see `?tm_dots` and `?tm_bubbles` for details on **tmap**'s point plotting functions). The resulting map, which has four layers, is illustrated in the right-hand panel of Figure **??**:

```
map_nz3 = map_nz2 +
  tm_shape(nz_height) + tm_dots()
```

A useful and little known feature of **tmap** is that multiple map objects can be arranged in a single 'metaplot' with `tmap_arrange()`. This is demonstrated in the code chunk below which plots `map_nz1` to `map_nz3`, resulting in Figure **??**.

```
tmap_arrange(map_nz1, map_nz2, map_nz3)
```

More elements can also be added with the `+` operator. Aesthetic settings, however, are controlled by arguments to layer functions.

0.8.2.3 Aesthetics

The plots in the previous section demonstrate **tmap**'s default aesthetic settings. Gray shades are used for `tm_fill()` and `tm_bubbles()` layers and a continuous black line is used to represent lines created with `tm_lines()`. Of course, these default values and other aesthetics can be overridden. The purpose of this section is to show how.

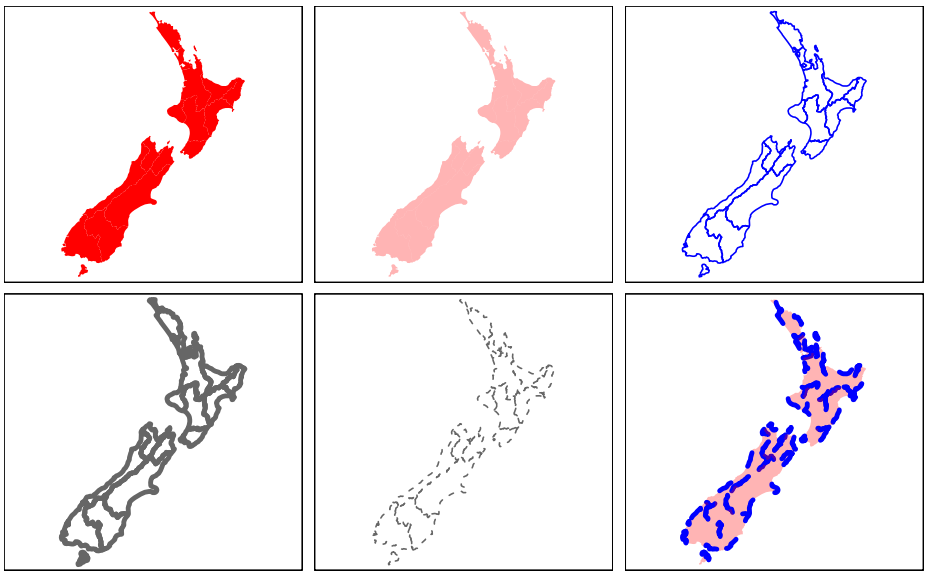


FIGURE 57: The impact of changing commonly used fill and border aesthetics to fixed values.

There are two main types of map aesthetics: those that change with the data and those that are constant. Unlike **ggplot2**, which uses the helper function `aes()` to represent variable aesthetics, **tmap** accepts aesthetic arguments that are either variable fields (based on column names) or constant values.[202] The most commonly used aesthetics for fill and border layers include color, transparency, line width and line type, set with `col`, `alpha`, `lwd`, and `lty` arguments, respectively. The impact of setting these with fixed values is illustrated in Figure **??**.

```
ma1 = tm_shape(nz) + tm_fill(col = "red")
ma2 = tm_shape(nz) + tm_fill(col = "red", alpha = 0.3)
ma3 = tm_shape(nz) + tm_borders(col = "blue")
ma4 = tm_shape(nz) + tm_borders(lwd = 3)
ma5 = tm_shape(nz) + tm_borders(lty = 2)
ma6 = tm_shape(nz) + tm_fill(col = "red", alpha = 0.3) +
  tm_borders(col = "blue", lwd = 3, lty = 2)
tmap_arrange(ma1, ma2, ma3, ma4, ma5, ma6)
```

Like base R plots, arguments defining aesthetics can also receive values that

[202] If there is a clash between a fixed value and a column name, the column name takes precedence. This can be verified by running the next code chunk after running `nz$red = 1:nrow(nz)`.

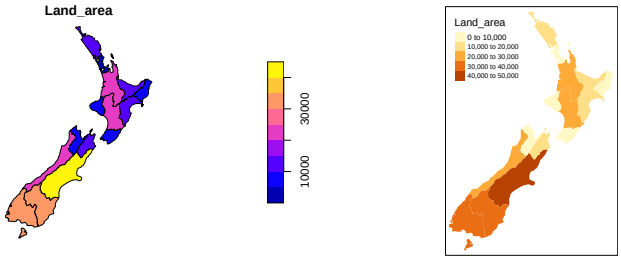


FIGURE 58: Comparison of base (left) and tmap (right) handling of a numeric color field.

vary. Unlike the base R code below (which generates the left panel in Figure **??**), **tmap** aesthetic arguments will not accept a numeric vector:

```
plot(st_geometry(nz), col = nz$Land_area)  # works
tm_shape(nz) + tm_fill(col = nz$Land_area) # fails
#> Error: Fill argument neither colors nor valid variable name(s)
```

Instead `col` (and other aesthetics that can vary such as `lwd` for line layers and `size` for point layers) requires a character string naming an attribute associated with the geometry to be plotted. Thus, one would achieve the desired result as follows (plotted in the right-hand panel of Figure **??**):

```
tm_shape(nz) + tm_fill(col = "Land_area")
```

An important argument in functions defining aesthetic layers such as `tm_fill()` is `title`, which sets the title of the associated legend. The following code chunk demonstrates this functionality by providing a more attractive name than the variable name `Land_area` (note the use of `expression()` to create superscript text):

```
legend_title = expression("Area (km"^2*")")
map_nza = tm_shape(nz) +
  tm_fill(col = "Land_area", title = legend_title) + tm_borders()
```

0.8.2.4 Color settings

Color settings are an important part of map design. They can have a major impact on how spatial variability is portrayed as illustrated in Figure **??**. This

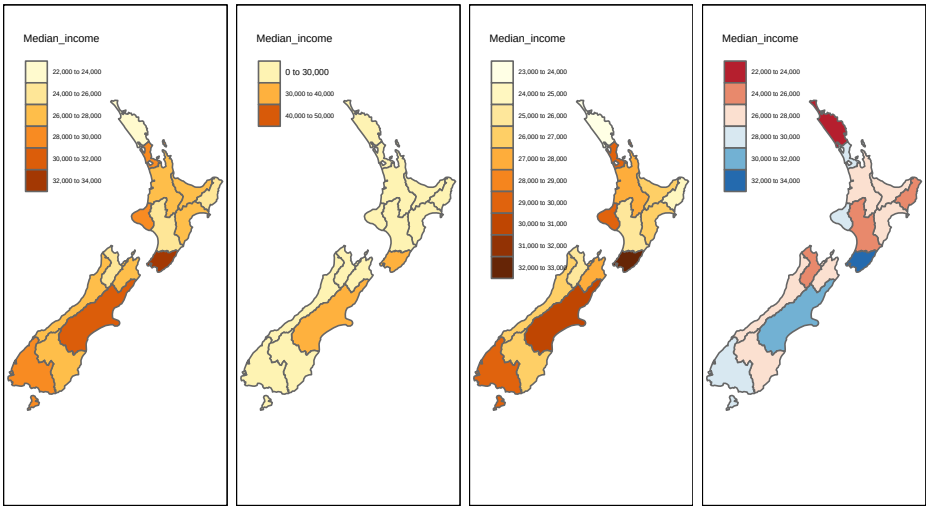


FIGURE 59: Illustration of settings that affect color settings. The results show (from left to right): default settings, manual breaks, n breaks, and the impact of changing the palette.

shows four ways of coloring regions in New Zealand depending on median income, from left to right (and demonstrated in the code chunk below):

- The default setting uses 'pretty' breaks, described in the next paragraph.
- `breaks` allows you to manually set the breaks.
- `n` sets the number of bins into which numeric variables are categorized.
- `palette` defines the color scheme, for example `BuGn`.

```
tm_shape(nz) + tm_polygons(col = "Median_income")
breaks = c(0, 3, 4, 5) * 10000
tm_shape(nz) + tm_polygons(col = "Median_income", breaks = breaks)
tm_shape(nz) + tm_polygons(col = "Median_income", n = 10)
tm_shape(nz) + tm_polygons(col = "Median_income", palette = "BuGn")
```

Another way to change color settings is by altering color break (or bin) settings. In addition to manually setting `breaks` **tmap** allows users to specify algorithms to automatically create breaks with the `style` argument. Six of the most useful break styles are illustrated in Figure **??** and described in the bullet points below:

- `style = pretty`, the default setting, rounds breaks into whole numbers where possible and spaces them evenly.

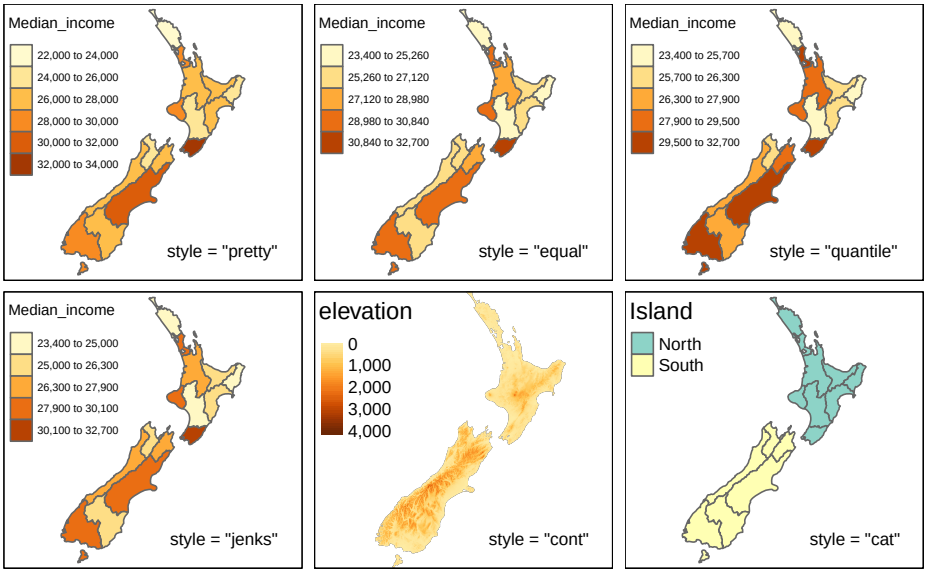


FIGURE 60: Illustration of different binning methods set using the style argument in tmap.

- `style = equal` divides input values into bins of equal range, and is appropriate for variables with a uniform distribution (not recommended for variables with a skewed distribution as the resulting map may end-up having little color diversity).
- `style = quantile` ensures the same number of observations fall into each category (with the potential down side that bin ranges can vary widely).
- `style = jenks` identifies groups of similar values in the data and maximizes the differences between categories.
- `style = cont` (and `order`) present a large number of colors over continuous color field, and are particularly suited for continuous rasters (`order` can help visualize skewed distributions).
- `style = cat` was designed to represent categorical values and assures that each category receives a unique color.

Although `style` is an argument of **tmap** functions, in fact it originates as an argument in `classInt::classIntervals()` — see the help page of this function for details.

Palettes define the color ranges associated with the bins and determined by the `breaks`, `n`, and `style` arguments described above. The default color palette is specified in `tm_layout()` (see Section **??** to learn more); however, it could be

quickly changed using the `palette` argument. It expects a vector of colors or a new color palette name, which can be selected interactively with `tmaptools::palette_explorer()`. You can add a `-` as prefix to reverse the palette order.

There are three main groups of color palettes: categorical, sequential and diverging (Figure **??**), and each of them serves a different purpose. Categorical palettes consist of easily distinguishable colors and are most appropriate for categorical data without any particular order such as state names or land cover classes. Colors should be intuitive: rivers should be blue, for example, and pastures green. Avoid too many categories: maps with large legends and many colors can be uninterpretable.[203]

The second group is sequential palettes. These follow a gradient, for example from light to dark colors (light colors tend to represent lower values), and are appropriate for continuous (numeric) variables. Sequential palettes can be single (`Blues` go from light to dark blue, for example) or multi-color/hue (`YlOrBr` is gradient from light yellow to brown via orange, for example), as demonstrated in the code chunk below — output not shown, run the code yourself to see the results!

```
tm_shape(nz) + tm_polygons("Population", palette = "Blues")
tm_shape(nz) + tm_polygons("Population", palette = "YlOrBr")
```

The last group, diverging palettes, typically range between three distinct colors (purple-white-green in Figure **??**) and are usually created by joining two single-color sequential palettes with the darker colors at each end. Their main purpose is to visualize the difference from an important reference point, e.g., a certain temperature, the median household income or the mean probability for a drought event. The reference point's value can be adjusted in **tmap** using the `midpoint` argument.

There are two important principles for consideration when working with colors: perceptibility and accessibility. Firstly, colors on maps should match our perception. This means that certain colors are viewed through our experience and also cultural lenses. For example, green colors usually represent vegetation or lowlands and blue is connected with water or cool. Color palettes should also be easy to understand to effectively convey information. It should be clear which values are lower and which are higher, and colors should change gradually. This property is not preserved in the rainbow color palette; therefore, we suggest avoiding it in geographic data visualization (Borland and Taylor II

[203] `col = "MAP_COLORS"` can be used in maps with a large number of individual polygons (for example, a map of individual countries) to create unique colors for adjacent polygons.

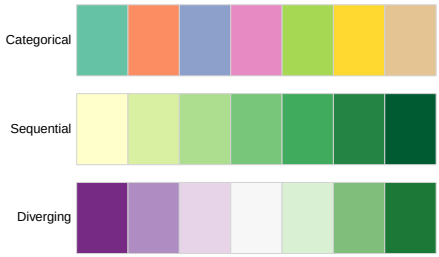


FIGURE 61: Examples of categorical, sequential and diverging palettes.

2007). Instead, the viridis color palettes[204], also available in **tmap**, can be used. Secondly, changes in colors should be accessible to the largest number of people. Therefore, it is important to use colorblind friendly palettes as often as possible.[205]

0.8.2.5 Layouts

The map layout refers to the combination of all map elements into a cohesive map. Map elements include among others the objects to be mapped, the title, the scale bar, margins and aspect ratios, while the color settings covered in the previous section relate to the palette and break-points used to affect how the map looks. Both may result in subtle changes that can have an equally large impact on the impression left by your maps.
Additional elements such as north arrows and scale bars have their own functions - `tm_compass()` and `tm_scale_bar()` (Figure **??**).

```
map_nz +
  tm_compass(type = "8star", position = c("left", "top")) +
  tm_scale_bar(breaks = c(0, 100, 200), size = 1)
```

tmap also allows a wide variety of layout settings to be changed, some of which are illustrated in Figure **??**, produced using the following code (see `args(tm_layout)` or `?tm_layout` for a full list):

```
map_nz + tm_layout(title = "New Zealand")
map_nz + tm_layout(scale = 5)
map_nz + tm_layout(bg.color = "lightblue")
map_nz + tm_layout(frame = FALSE)
```

[204] `https://cran.r-project.org/web/packages/viridis/`
[205] See the "Color blindness simulator" options in `tmaptools::palette_explorer()`.

FIGURE 62: Map with additional elements - a north arrow and scale bar.

The other arguments in `tm_layout()` provide control over many more aspects of the map in relation to the canvas on which it is placed. Some useful layout settings are listed below (see Figure **??** for illustrations of a selection of these):

- Frame width (`frame.lwd`) and an option to allow double lines (`frame.double.line`).
- Margin settings including `outer.margin` and `inner.margin`.
- Font settings controlled by `fontface` and `fontfamily`.
- Legend settings including binary options such as `legend.show` (whether or not to show the legend) `legend.only` (omit the map) and `legend.outside` (should the legend go outside the map?), as well as multiple choice settings such as `legend.position`.
- Default colors of aesthetic layers (`aes.color`), map attributes such as the frame (`attr.color`).

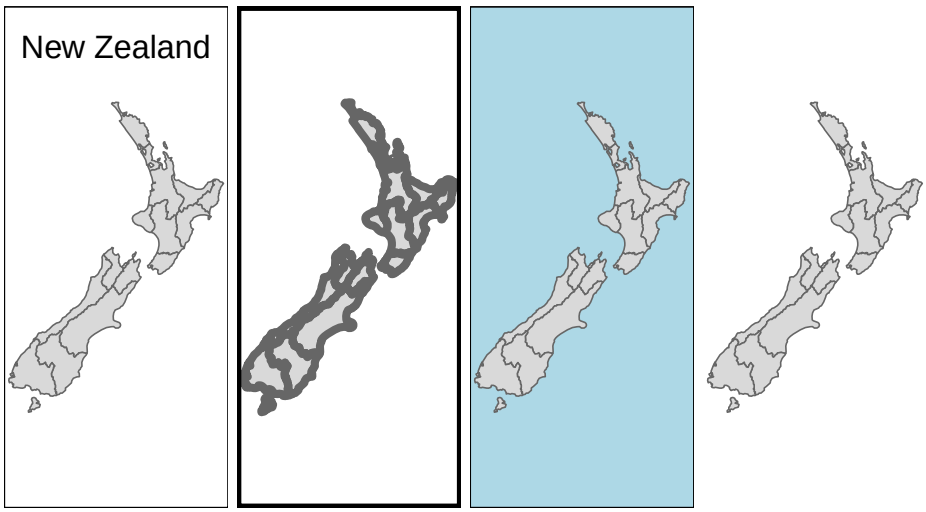


FIGURE 63: Layout options specified by (from left to right) title, scale, bg.color and frame arguments.

- Color settings controlling `sepia.intensity` (how yellowy the map looks) and `saturation` (a color-grayscale).

The impact of changing the color settings listed above is illustrated in Figure **??** (see `?tm_layout` for a full list).

Beyond the low-level control over layouts and colors, **tmap** also offers high-level styles, using the `tm_style()` function (representing the second meaning of 'style' in the package). Some styles such as `tm_style("cobalt")` result in stylized maps, while others such as `tm_style("gray")` make more subtle changes, as illustrated in Figure **??**, created using code below (see `08-tmstyles.R`):

```
map_nza + tm_style("bw")
map_nza + tm_style("classic")
map_nza + tm_style("cobalt")
map_nza + tm_style("col_blind")
```

\begin{figure}[t]

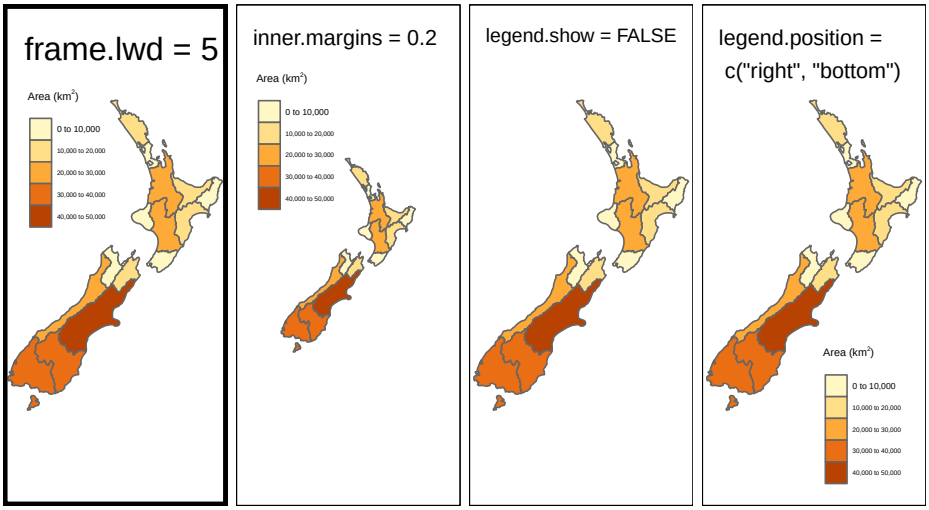


FIGURE 64: Illustration of selected layout options.

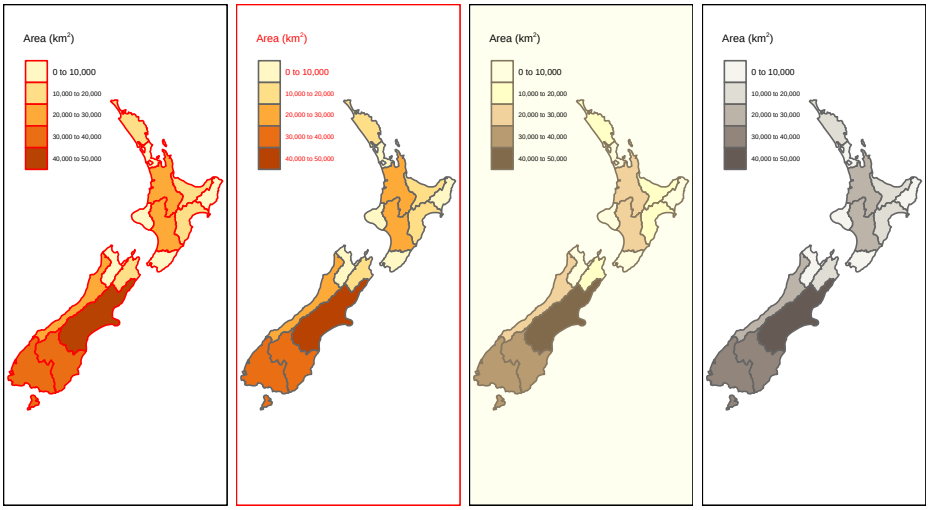
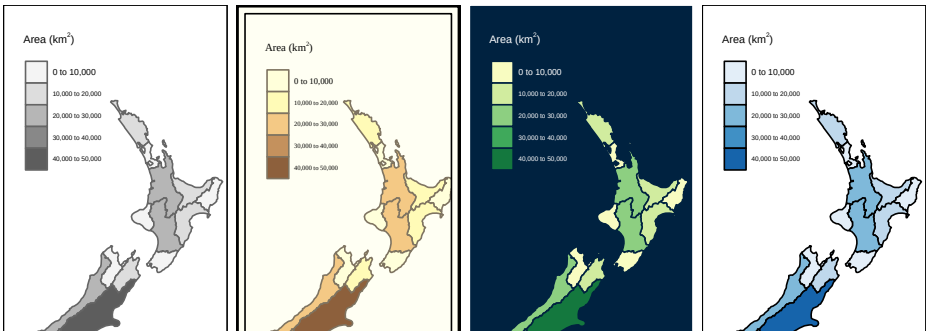


FIGURE 65: Illustration of selected color-related layout options.



cc

}
\caption[Selected tmap styles: bw, classic, cobalt and col_blind.]{Selected tmap styles: bw, classic, cobalt and col_blind (from left to right).} \end{figure}

A preview of predefined styles can be generated by executing `tmap_style_catalogue()`. This creates a folder called `tmap_style_previews` containing nine images. Each image, from `tm_style_albatross.png` to `tm_style_white.png`, shows a faceted map of the world in the corresponding style. Note: `tmap_style_catalogue()` takes some time to run.

0.8.2.6 Faceted maps

Faceted maps, also referred to as 'small multiples', are composed of many maps arranged side-by-side, and sometimes stacked vertically (Meulemans et al. 2017). Facets enable the visualization of how spatial relationships change with respect to another variable, such as time. The changing populations of settlements, for example, can be represented in a faceted map with each panel representing the population at a particular moment in time. The time dimension could be represented via another *aesthetic* such as color. However, this risks cluttering the map because it will involve multiple overlapping points (cities do not tend to move over time!).

Typically all individual facets in a faceted map contain the same geometry data repeated multiple times, once for each column in the attribute data (this is the default plotting method for `sf` objects, see Chapter **??**). However, facets can also represent shifting geometries such as the evolution of a point pattern over time. This use case of faceted plot is illustrated in Figure **??**.

```
urb_1970_2030 = urban_agglomerations %>%
  filter(year %in% c(1970, 1990, 2010, 2030))
tm_shape(world) + tm_polygons() +
  tm_shape(urb_1970_2030) + tm_symbols(col = "black", border.col = "white",
                                       size = "population_millions") +
  tm_facets(by = "year", nrow = 2, free.coords = FALSE)
```

The preceding code chunk demonstrates key features of faceted maps created with **tmap**:

- Shapes that do not have a facet variable are repeated (the countries in `world` in this case).
- The `by` argument which varies depending on a variable (`year` in this case).

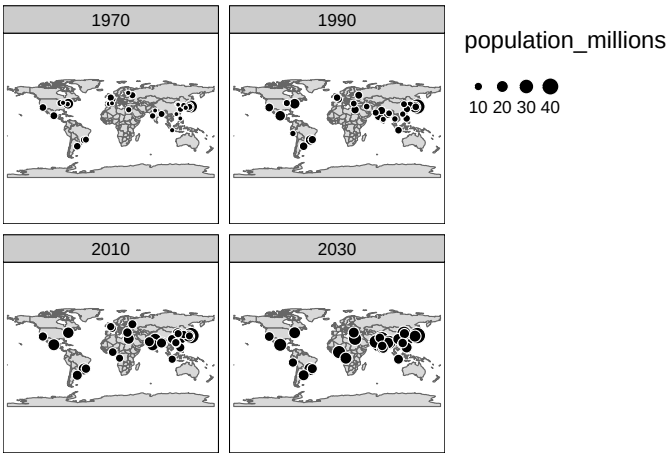


FIGURE 66: Faceted map showing the top 30 largest urban agglomerations from 1970 to 2030 based on population projects by the United Nations.

- `nrow`/`ncol` setting specifying the number of rows and columns that facets should be arranged into.
- The `free.coords`-parameter specifying if each map has its own bounding box.

In addition to their utility for showing changing spatial relationships, faceted maps are also useful as the foundation for animated maps (see Section **??**).

0.8.2.7 Inset maps

An inset map is a smaller map rendered within or next to the main map. It could serve many different purposes, including providing a context (Figure **??**) or bringing some non-contiguous regions closer to ease their comparison (Figure **??**). They could be also used to focus on a smaller area in more detail or to cover the same area as the map, but representing a different topic.
In the example below, we create a map of the central part of New Zealand's Southern Alps. Our inset map will show where the main map is in relation to the whole New Zealand. The first step is to define the area of interest, which can be done by creating a new spatial object, `nz_region`.

```
nz_region = st_bbox(c(xmin = 1340000, xmax = 1450000,
                      ymin = 5130000, ymax = 5210000),
                    crs = st_crs(nz_height)) %>%
  st_as_sfc()
```

In the second step, we create a base map showing the New Zealand's Southern Alps area. This is a place where the most important message is stated.

ccii

```
nz_height_map = tm_shape(nz_elev, bbox = nz_region) +
  tm_raster(style = "cont", palette = "YlGn", legend.show = TRUE) +
  tm_shape(nz_height) + tm_symbols(shape = 2, col = "red", size = 1) +
  tm_scale_bar(position = c("left", "bottom"))
```

The third step consists of the inset map creation. It gives a context and helps to locate the area of interest. Importantly, this map needs to clearly indicate the location of the main map, for example by stating its borders.

```
nz_map = tm_shape(nz) + tm_polygons() +
  tm_shape(nz_height) + tm_symbols(shape = 2, col = "red", size = 0.1) +
  tm_shape(nz_region) + tm_borders(lwd = 3)
```

Finally, we combine the two maps using the function `viewport()` from the **grid** package, the first arguments of which specify the center location (`x` and `y`) and a size (`width` and `height`) of the inset map.

```
library(grid)
nz_height_map
print(nz_map, vp = viewport(0.8, 0.27, width = 0.5, height = 0.5))
```

Inset map can be saved to file either by using a graphic device (see Section **??**) or the `tmap_save()` function and its arguments - `insets_tm` and `insets_vp`. Inset maps are also used to create one map of non-contiguous areas. Probably, the most often used example is a map of the United States, which consists of the contiguous United States, Hawaii and Alaska. It is very important to find the best projection for each individual inset in these types of cases (see Chapter **??** to learn more). We can use US National Atlas Equal Area for the map of the contiguous United States by putting its EPSG code in the `projection` argument of `tm_shape()`.

```
us_states_map = tm_shape(us_states, projection = 2163) + tm_polygons() +
  tm_layout(frame = FALSE)
```

The rest of our objects, `hawaii` and `alaska`, already have proper projections; therefore, we just need to create two separate maps:

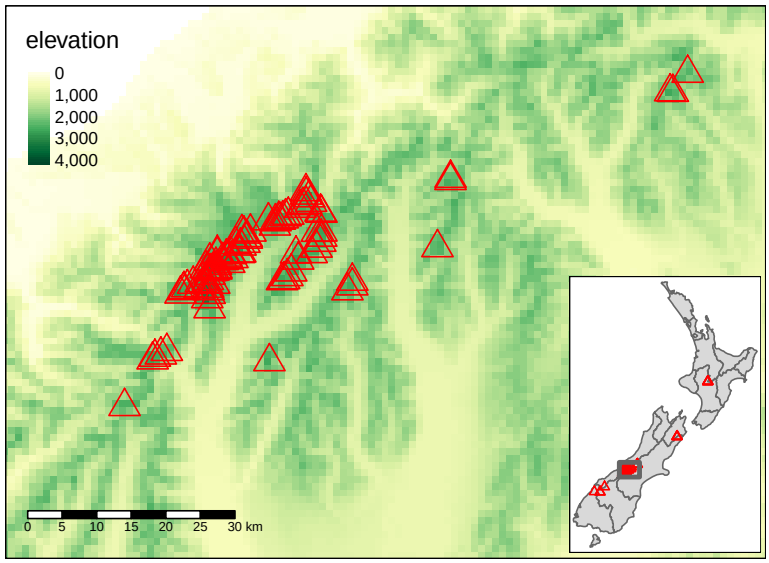


FIGURE 67: Inset map providing a context - location of the central part of the Southern Alps in New Zealand.

```
hawaii_map = tm_shape(hawaii) + tm_polygons() +
  tm_layout(title = "Hawaii", frame = FALSE, bg.color = NA,
            title.position = c("LEFT", "BOTTOM"))
alaska_map = tm_shape(alaska) + tm_polygons() +
  tm_layout(title = "Alaska", frame = FALSE, bg.color = NA)
```

The final map is created by combining and arranging these three maps:

```
us_states_map
print(hawaii_map, vp = grid::viewport(0.35, 0.1, width = 0.2, height = 0.1))
print(alaska_map, vp = grid::viewport(0.15, 0.15, width = 0.3, height = 0.3))
```

The code presented above is compact and can be used as the basis for other inset maps but the results, in Figure **??**, provide a poor representation of the locations of Hawaii and Alaska. For a more in-depth approach, see the `us-map`[206] vignette from the **geocompkg**.

[206] `https://geocompr.github.io/geocompkg/articles/us-map.html`

FIGURE 68: Map of the United States.

0.8.3 Animated maps

Faceted maps, described in Section **??**, can show how spatial distributions of variables change (e.g., over time), but the approach has disadvantages. Facets become tiny when there are many of them. Furthermore, the fact that each facet is physically separated on the screen or page means that subtle differences between facets can be hard to detect.

Animated maps solve these issues. Although they depend on digital publication, this is becoming less of an issue as more and more content moves online. Animated maps can still enhance paper reports: you can always link readers to a web-page containing an animated (or interactive) version of a printed map to help make it come alive. There are several ways to generate animations in R, including with animation packages such as **gganimate**, which builds on **ggplot2** (see Section **??**). This section focusses on creating animated maps with **tmap** because its syntax will be familiar from previous sections and the flexibility of the approach.

Figure **??** is a simple example of an animated map. Unlike the faceted plot, it does not squeeze multiple maps into a single screen and allows the reader to see how the spatial distribution of the world's most populous agglomerations evolve over time (see the book's website for the animated version).

. . . .

The animated map illustrated in Figure **??** can be created using the same **tmap**

techniques that generate faceted maps, demonstrated in Section **??**. There are two differences, however, related to arguments in `tm_facets()`:

- `along = "year"` is used instead of `by = "year"`.
- `free.coords = FALSE`, which maintains the map extent for each map iteration.

These additional arguments are demonstrated in the subsequent code chunk:

```
urb_anim = tm_shape(world) + tm_polygons() +
  tm_shape(urban_agglomerations) + tm_dots(size = "population_millions") +
  tm_facets(along = "year", free.coords = FALSE)
```

The resulting `urb_anim` represents a set of separate maps for each year. The final stage is to combine them and save the result as a `.gif` file with `tmap_animation()`. The following command creates the animation illustrated in Figure **??**, with a few elements missing, that we will add in during the exercises:

```
tmap_animation(urb_anim, filename = "urb_anim.gif", delay = 25)
```

Another illustration of the power of animated maps is provided in Figure **??**. This shows the development of states in the United States, which first formed in the east and then incrementally to the west and finally into the interior. Code to reproduce this map can be found in the script `08-usboundaries.R`.

. . . .

0.8.4 Interactive maps

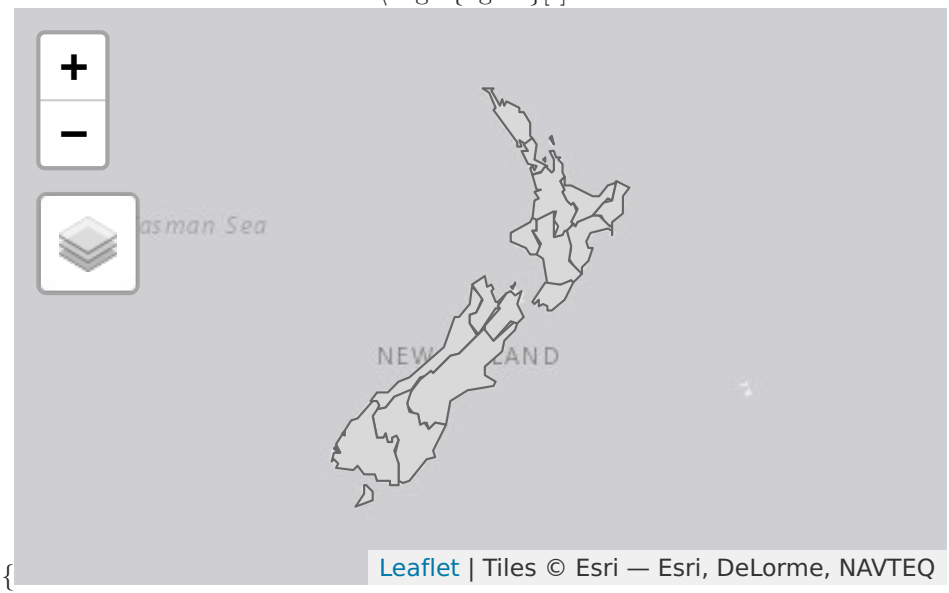
While static and animated maps can enliven geographic datasets, interactive maps can take them to a new level. Interactivity can take many forms, the most common and useful of which is the ability to pan around and zoom into any part of a geographic dataset overlaid on a 'web map' to show context. Less advanced interactivity levels include popups which appear when you click on different features, a kind of interactive label. More advanced levels of interactivity include the ability to tilt and rotate maps, as demonstrated in the **mapdeck** example below, and the provision of "dynamically linked" sub-plots which automatically update when the user pans and zooms (Pezanowski et al. 2018). The most important type of interactivity, however, is the display of geographic data on interactive or 'slippy' web maps. The release of the **leaflet** package in 2015 revolutionized interactive web map creation from within R and a number of packages have built on these foundations adding new features (e.g., **leaflet.extras**) and making the creation of web maps as simple as creating

static maps (e.g., **mapview** and **tmap**). This section illustrates each approach in the opposite order. We will explore how to make slippy maps with **tmap** (the syntax of which we have already learned), **mapview** and finally **leaflet** (which provides low-level control over interactive maps).

A unique feature of **tmap** mentioned in Section **??** is its ability to create static and interactive maps using the same code. Maps can be viewed interactively at any point by switching to view mode, using the command `tmap_mode("view")`. This is demonstrated in the code below, which creates an interactive map of New Zealand based on the `tmap` object `map_nz`, created in Section **??**, and illustrated in Figure **??**:

```
tmap_mode("view")
map_nz
```

\begin{figure}[t]



{

Now that the interactive mode has been 'turned on', all maps produced with **tmap** will launch (another way to create interactive maps is with the `tmap_leaflet` function). Notable features of this interactive mode include the ability to specify the basemap with `tm_basemap()` (or `tmap_options()`) as demonstrated below (result not shown):

```
map_nz + tm_basemap(server = "OpenTopoMap")
```

An impressive and little-known feature of **tmap**'s view mode is that it also