

Spatial Data Science

with applications in R

Edzer Pebesma, Roger Bivand

2021-07-13

Contents

Preface	9
0.1 Acknowledgements	10
I Spatial Data	11
1 Getting Started	15
1.1 A first map	15
1.2 Coordinate reference systems	18
1.3 Raster and vector data	19
1.4 Raster types	21
1.5 Time series, arrays, data cubes	23
1.6 Support	23
1.7 Spatial data science software	24
1.8 Exercises	26
2 Coordinates	27
2.1 Quantities, units, datum	27
2.2 Ellipsoidal coordinates	28
2.3 Coordinate Reference Systems	32
2.4 PROJ and mapping accuracy	33
2.5 WKT-2	37
2.6 Exercises	38

3 Geometries	39
3.1 Simple feature geometries	39
3.2 Operations on geometries	44
3.3 Precision	51
3.4 Coverages: tessellations and rasters	51
3.5 Networks	54
3.6 Exercises	54
4 Spherical Geometries	57
4.1 Straight lines	57
4.2 Ring direction	58
4.3 Full polygon	58
4.4 Bounding box, rectangle, and cap	58
4.5 Validity on the sphere	59
4.6 Exercises	59
5 Attributes and Support	63
5.1 Attribute-geometry relationships and support	64
5.2 Aggregating and summarising	66
5.3 Area-weighted interpolation	69
5.4 Up- and Downscaling	71
5.5 Exercises	72
6 Data Cubes	75
6.1 A four-dimensional data cube	76
6.2 Dimensions, attributes, and support	76
6.3 Operations on data cubes	80
6.4 Aggregating raster to vector cubes	82
6.5 Switching dimension with attributes	85
6.6 Other dynamic spatial data	86
6.7 Exercises	86

CONTENTS	5
----------	---

II R for Spatial Data Science	89
--------------------------------------	-----------

7 Introduction to sf and stars	91
---------------------------------------	-----------

7.1 Package sf	91
7.2 Spatial joins	97
7.3 Package stars	99
7.4 Vector data cube examples	114
7.5 raster-to-vector, vector-to-raster	126
7.6 Coordinate transformations and conversions	128
7.7 Transforming and warping rasters	133
7.8 Exercises	135

8 Large data sets	137
--------------------------	------------

8.1 Vector data: sf	137
8.2 Raster data: stars	142
8.3 Very large data cubes	146
8.4 Exercises	148

9 Plotting spatial data	149
--------------------------------	------------

9.1 Every plot is a projection	149
9.2 Plotting points, lines, polygons, grid cells	152
9.3 Base plot	154
9.4 Maps with ggplot2	156
9.5 Maps with tmap	158
9.6 Interactive maps: leaflet , mapview , tmap	161
9.7 Exercises	161

III Models for Spatial Data	163
------------------------------------	------------

10 Statistical modelling of spatial data	165
---	------------

10.1 Design-based and model-based inference	165
10.2 Predictive models with coordinates	167
10.3 Further reading	168

11 Point Pattern Analysis	169
11.1 Observation window	170
11.2 Coordinate reference systems	175
11.3 Marked point patterns, points on linear networks	175
11.4 Spatial sampling and simulating a point process	177
11.5 Simulating points on the globe	178
11.6 Exercises	180
12 Spatial Interpolation	181
12.1 A first dataset	181
12.2 Sample variogram	184
12.3 Fitting variogram models	187
12.4 Kriging interpolation	188
12.5 Areal means: block kriging	188
12.6 Conditional simulation	191
12.7 Trend models	193
12.8 Exercises	198
13 Multivariate and Spatiotemporal Geostatistics	199
13.1 Preparing the air quality dataset	199
13.2 Multivariable geostatistics	201
13.3 Spatiotemporal geostatistics	202
13.4 Exercises	207
14 Proximity and Areal Data	211
14.1 Representing proximity in spdep	212
14.2 Contiguous neighbours	215
14.3 Graph-based neighbours	217
14.4 Distance-based neighbours	219
14.5 Weights specification	224
14.6 Higher order neighbours	226
14.7 Exercises	228

15 Measures of spatial autocorrelation	231
15.1 Measures and process mis-specification	231
15.2 Global measures	234
15.3 Local measures	238
15.4 Exercises	252
16 Spatial Regression	253
16.1 Markov random field and multilevel models with spatial weights	253
16.2 Multilevel models of the Boston data set	258
17 Spatial econometrics models	267
17.1 Spatial econometric models: definitions	268
17.2 Maximum likelihood estimation in spatialreg	271
17.3 Impacts	275
17.4 Predictions	276
18 Older R Spatial Packages	279
18.1 Retiring rgdal and rgeos	279
18.2 links and differences between sf and sp	280
18.3 migration code and packages	280
18.4 Package raster and terra	281
R basics	283
Pipes	283
Data structures	284
dissecting a MULTIPOLYGON	292
# Warning in utils::citation(..., lib.loc = lib.loc): no date field # in DESCRIPTION file of package 'gstat' # Warning in utils::citation(..., lib.loc = lib.loc): could not # determine year for 'gstat' from package DESCRIPTION file # Warning in utils::citation(..., lib.loc = lib.loc): could not # determine year for 'INLA' from package DESCRIPTION file # Warning in utils::citation(..., lib.loc = lib.loc): no date field # in DESCRIPTION file of package 'sf' # Warning in utils::citation(..., lib.loc = lib.loc): could not	

```
# determine year for 'sf' from package DESCRIPTION file
# Warning in utils::citation(..., lib.loc = lib.loc): no date field
# in DESCRIPTION file of package 'stars'
# Warning in utils::citation(..., lib.loc = lib.loc): could not
# determine year for 'stars' from package DESCRIPTION file
# Warning in utils::citation(..., lib.loc = lib.loc): no date field
# in DESCRIPTION file of package 'units'
```

Preface

Data science is concerned with finding answers to questions on the basis of available data, and communicating that effort. Besides showing the results, this communication involves sharing the data used, but also exposing the path that led to the answers in a comprehensive and reproducible way. It also acknowledges the fact that available data may not be sufficient to answer questions, and that any answers are conditional on the data collection or sampling protocols employed.

This book introduces and explains the concepts underlying *spatial* data: points, lines, polygons, rasters, coverages, geometry attributes, data cubes, reference systems, as well as higher-level concepts including how attributes relate to geometries and how this affects analysis. The relationship of attributes to geometries is known as support, and changing support also changes the characteristics of attributes. Some data generation processes are continuous in space, and may be observed everywhere. Others are discrete, observed in tesselated containers. In modern spatial data analysis, tesellated methods are often used for all data, extending across the legacy partition into point process, geostatistical and lattice models. It is support (and the understanding of support) that underlies the importance of spatial representation. The book aims at data scientists who want to get a grip on using spatial data in their analysis. To exemplify how to do things, it uses R.

It is often thought that spatial data boils down to having observations' longitude and latitude in a dataset, and treating these just like any other variable. This carries the risk of missed opportunities and meaningless analyses. For instance,

- coordinate pairs really are pairs, and lose much of their meaning when treated independently
- rather than having point locations, observations are often associated with spatial lines, areas, or grid cells
- spatial distances between observations are often not well represented by straight-line distances, but by great circle distances, distances through networks, or by measuring the effort it takes getting from A to B

We introduce the concepts behind spatial data, coordinate reference systems,

spatial analysis, and introduce a number of packages, including **sf** (Pebesma, 2018, 2021c), **stars** (Pebesma, 2021d), **s2** (Dunnington et al., 2021) and **lwgeom** (Pebesma, 2021b), as well as a number of **tidyverse** (Wickham, 2021) extensions, and a number of spatial analysis and visualisation packages that can be used with these packages, including **gstat** (Pebesma and Graeler, 2021), **spdep** (Bivand, 2021b), **spatialreg** (Bivand and Piras, 2021), **spatstat** (Baddeley et al., 2021), **tmap** (Tennekes, 2021) and **mapview** (Appelhans et al., 2021).

The first part of this book introduces concepts of spatial data science, and uses R only to generate text output or figures. The R code used for this is not shown, as it would distract from the message. The online version of this book contains the R sections, which can be unfolded on demand and copied into the clipboard for execution and experimenting. The second part of this book explains how the concepts introduced in part I are dealt with using R, and deals with basic handling and plotting of spatial and spatiotemporal data. Part III is dedicated to statistical modelling of spatial data.

0.1 Acknowledgements

all GitHub contributors (t.b.d.), Claus Wilke, Jakub Nowosad, SDSWR class summer 2021, all sf and stars authors,

Part I

Spatial Data

Text introducing Part I

Chapter 1

Getting Started

This chapter introduces a number of concepts associated with handling spatial data, and points forward to later sections where they are discussed in more detail.

1.1 A first map

The typical way to graph spatial data is by creating a map. Let us consider a simple map, shown in figure 1.1.

A number of graphical elements are present here, in this case:

- polygons are drawn with a black outline and filled with colors chosen according to a variable `BIR74`, whose name is in the title
- a legend key explains the meaning of the colors, and has a certain *color palette* and *color breaks*, values at which color changes
- the background of the map shows curved lines with constant latitude or longitude (graticule)
- the axis ticks show the latitude and longitude values

Polygons are a particular form of *geometry*; spatial geometries (points, lines, polygons, pixels) are discussed in detail in chapter 3. Polygons consist of sequences of points, connected by straight lines. How point locations of spatial data are expressed, or measured, is discussed in chapter 2. As can be seen from figure 1.1, lines of equal latitude and longitude do not form straight lines, indicating that some form of projection took place before plotting; projections are also discussed in chapter 2 and section 9.1.

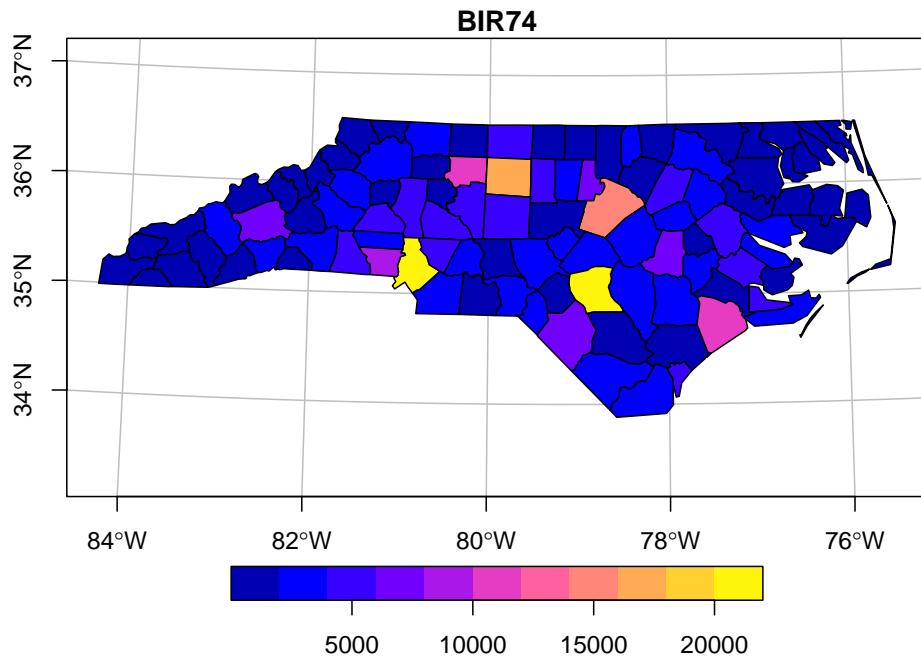


Figure 1.1: a first map

The color values in figure 1.1 are derived from numeric values of a variable, BIR74, which has a single value associated with each geometry or *feature*. Chapter 5 discusses such feature attributes, and the way they can relate to feature geometries. In this case, BIR74 refers to birth counts, meaning counts *over the region*. This implies that the count does not refer to a value associated with every point inside the polygon, which the continuous color might suggest, but rather measures an integral (sum) over the polygon.

Before plotting figure 1.1 we had to read the data, in this case from a file (section 7.1). Printing a data summary for the first three records of three attribute variables shows:

```
# Simple feature collection with 100 features and 3 fields
# Geometry type: MULTIPOLYGON
# Dimension: XY
# Bounding box: xmin: -84.3 ymin: 33.9 xmax: -75.5 ymax: 36.6
# Geodetic CRS: NAD27
# # A tibble: 100 x 4
#   AREA BIR74 SID74                                     geom
#   <dbl> <dbl> <dbl> <MULTIPOLYGON [°]>
# 1 0.114    1091      1 (((-81.5 36.2, -81.5 36.3, -81.6 36.3, -81.6 ~
# 2 0.061     487      0 (((-81.2 36.4, -81.2 36.4, -81.3 36.4, -81.3 ~
```

```
# 3 0.143 3188      5 (((-80.5 36.2, -80.5 36.3, -80.5 36.3, -80.5 ~
# # ... with 97 more rows
```

The printed output shows:

- the (selected) dataset has 100 features (records) and 3 fields (attributes)
- the geometry type is MULTIPOLYGON (chapter 3)
- it has dimension XY, indicating that each point will consist of 2 coordinate values
- the range of x and y values of the geometry
- the coordinate reference system (CRS) is geodetic, with coordinates in degrees longitude and latitude associated to the NAD27 datum (chapter 2)
- the three selected attribute variables are followed by a variable `geom` of type MULTIPOLYGON with unit degrees that contains the polygon information

More complicated plots can involve facet plots with a map in each facet, as shown in figure 1.2.

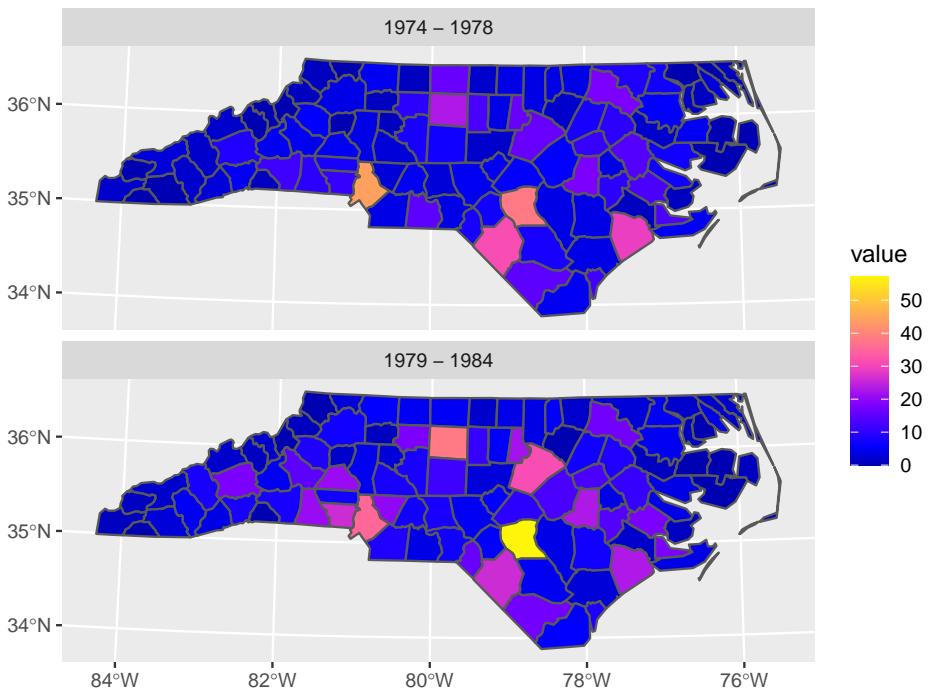


Figure 1.2: ggplot with facet maps

An interactive, leaflet-based map is obtained in figure 1.3.

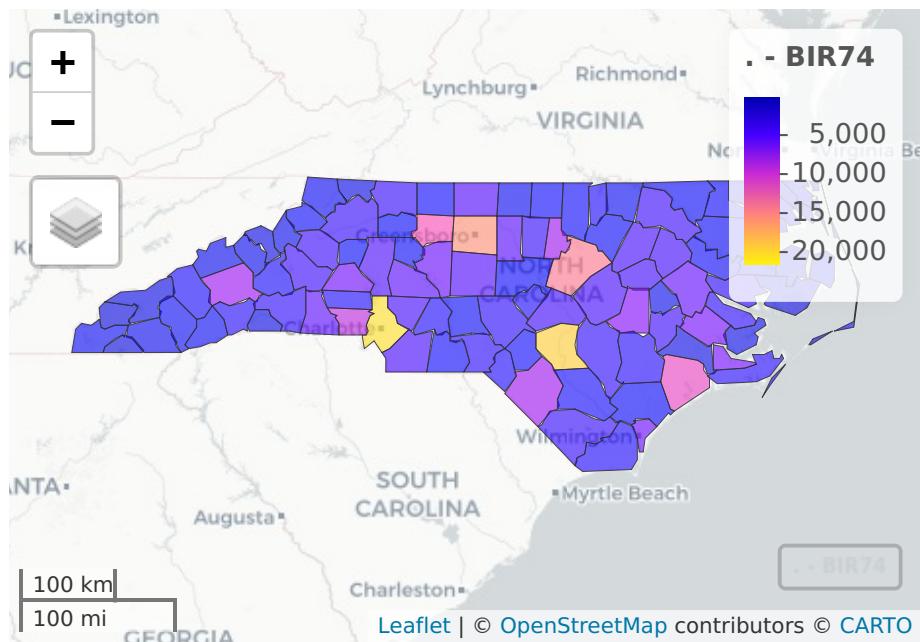


Figure 1.3: Interactive map created with mapview.

1.2 Coordinate reference systems

In figure 1.1, the grey lines denote the *graticule*, a grid with lines along constant latitude or longitude. Clearly, these lines are not straight, which indicates that a *projection* of the data was used for which the x and y axis do not align with longitude and latitude. In figure 1.3 we see that the north boundary of North Carolina is plotted as a straight line again, indicating that another projection was used.

The ellipsoidal coordinates of the graticule of figure 1.1 are associated with a particular *datum* (here: NAD27), which implicates a set of rules what the shape of the Earth is and how it is attached to the Earth (to which point of the Earth is the origin associated, and how is it directed). If one would measure coordinates with a GPS device (e.g. a mobile phone) it would typically report coordinates associated with the WGS84 datum, which can be around 30 m different from the identical coordinate values when associated with NAD27.

Projections describe how we go back and forth between

- **ellipsoidal coordinates** which are expressed as degrees latitude and longitude, pointing to locations on a shape approximating the Earth's shape (an ellipsoid or spheroid), and

- **projected coordinates** which are coordinates on a flat, two-dimensional coordinate system, used when plotting maps.

Datums transformations are associated with moving from one datum to another. Both topics are covered by *spatial reference systems* are described in more detail in chapter 2.

1.3 Raster and vector data

Polygon, point and line geometries are examples of *vector* data: point coordinates describe the “exact” locations that can be anywhere. Raster data on the other hand describe data where values are aligned on a *raster*, meaning on a regularly laid out lattice of usually square pixels. An example is shown in figure 1.4.

Vector and raster data can be combined in different ways; for instance we can query the raster at the three points of figure 1.4(c),

```
# Simple feature collection with 3 features and 1 field
# Geometry type: POINT
# Dimension: XY
# Bounding box: xmin: 290000 ymin: 9110000 xmax: 292000 ymax: 9120000
# Projected CRS: UTM Zone 25, Southern Hemisphere
# L7_ETMs.tif           geometry
# 1          80 POINT (290830 9114499)
# 2          58 POINT (290019 9119219)
# 3          63 POINT (291693 9116038)
```

or compute an aggregate, such as the average, over arbitrary regions such as the circles shown in figure 1.4(d):

```
# Simple feature collection with 3 features and 1 field
# Geometry type: POLYGON
# Dimension: XY
# Bounding box: xmin: 290000 ymin: 9110000 xmax: 292000 ymax: 9120000
# Projected CRS: UTM Zone 25, Southern Hemisphere
# V1           geometry
# 1 77.2 POLYGON ((291330 9114499, 2...
# 2 60.1 POLYGON ((290519 9119219, 2...
# 3 71.6 POLYGON ((292193 9116038, 2...
```

Other raster-to-vector conversions are discussed in 7.5 and include:

- converting raster pixels into point values

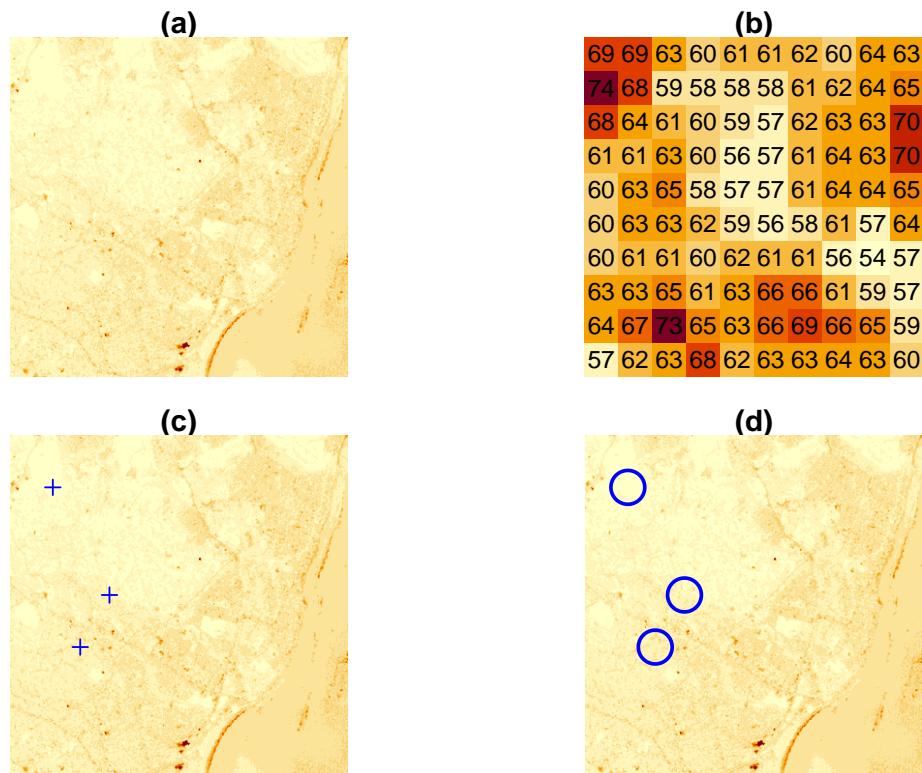


Figure 1.4: raster maps: Landsat-7 blue band, with color values derived from data values (a), the top-left 10x10 sub-image from (a) with numeric values shown (b), and overlayed by two different types of vector data: three sample points (c), and a 500m radius around the points represented as polygons (d)

- converting raster pixels into small polygons, possibly merging polygons with identical values (“polygonize”)
- generating lines or polygons that delineate continuous pixel areas with a certain value *range* (“contour”)

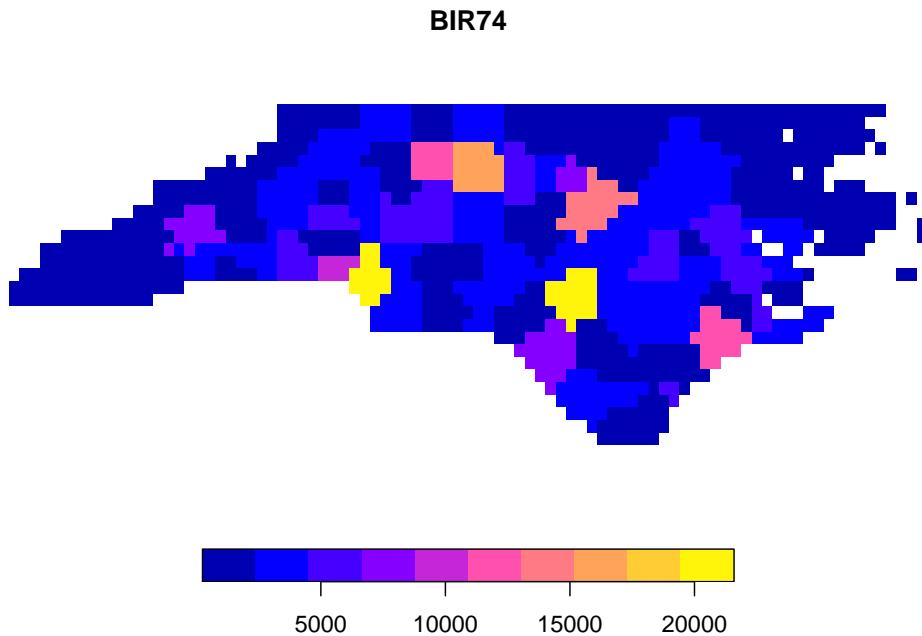


Figure 1.5: The map obtained by rasterizing county total number of births for the period 1974-1979 shown in figure 1.1

Vector-to-raster conversions can be as simple as rasterizing polygons, as shown in figure 1.5. Other, more general vector-to-raster conversions that may involve statistical modelling include:

- interpolation of point values to points on a regular grid (chapter 12)
- estimating densities of points over a regular grid (chapter 11)
- area-weighted interpolation of polygon values to grid cells (section 5.3)
- direct rasterization of points, lines or polygons (section 7.5)

1.4 Raster types

Raster dimensions describe how the rows and columns relate to spatial coordinates. Figure 1.6 shows a number of different possibilities.

Regular rasters like shown in figure 1.6 have a constant, not necessarily square cell size and axes aligned with the x and y (Easting and Northing) axes. Other

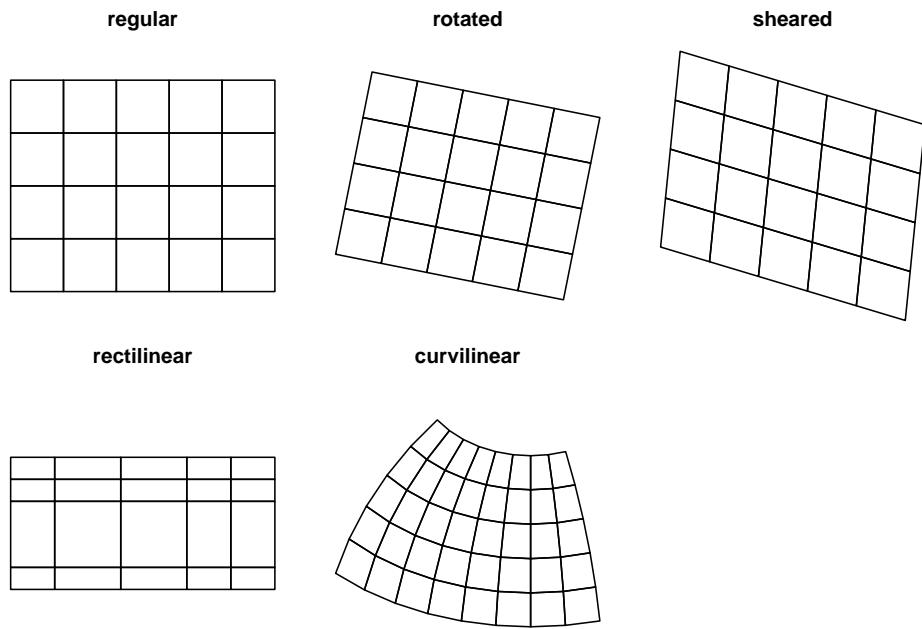


Figure 1.6: various raster types

raster types include those where the axes are no longer aligned with x and y (*rotated*), where axes are no longer perpendicular (*sheared*), or where cell size varies along a dimension (*rectilinear*). Finally, *curvilinear* rasters have cell size and/or direction properties that are no longer independent from the other raster dimension.

When a raster that is regular in a given coordinate reference system is projected to another raster while keeping each raster cell in tact, it changes shape and may become rectilinear (e.g. when going from ellipsoidal coordinates to Mercator, as in figure 1.3) or curvilinear (e.g. when going from ellipsoidal coordinates to Lambert Conic Conformal, as in figure 1.1). When reverting this procedure, one can recover the exact original raster.

Creating a new, regular grid in the new projection is called raster (or image) *reprojection* or *warping* (section 7.7). This process is lossy, irreversible, and may need to be informed whether raster cells should be interpolated, averaged or summed, whether they denote categorical variables, or whether resampling using nearest neighbours should be used; see also section 1.6.

1.5 Time series, arrays, data cubes

A lot of spatial data is not *just* spatial, but in addition temporal. Just like any observation is associated with an observation location, it is associated with an observation time or period. The dataset on the North Carolina counties shown above contains disease cases counted over two time periods, shown in figure 1.2. Although the original dataset has these variables in two different columns, for plotting them these columns had to be stacked first, while repeating the associated geometries - a form called *tidy* by (Wickham, 2014b). When we have longer time series associated with geometries, neither option - distributing time over multiple columns, or stacking columns while repeating geometries - works well, and a more effective way of storing such data would be a matrix or array, where one dimension refers to time, and the other(s) to space. The natural way for image or raster data is already to store them in matrices; time series of rasters then lead to a three-dimensional array. The general term for such data is a (spatiotemporal) **data cube**, where cube refers to arrays with any number of dimensions. Data cubes can refer to both raster and vector data, examples are given in chapter 6.

1.6 Support

When we have spatial data with geometries that are not points but collections of points (multi-points, lines, polygons, pixels), then the attributes associated with these geometries has one of several different relationships to them. Attributes can have:

- a **constant** value for every point of the geometry
- a value that is unique to only this geometry, describing its **identity**
- a single value that is an **aggregate** over all points of the geometry

An example of a constant is land use or bedrock type of a polygon. An example of an identity is a county name. An example of an aggregate is the number of births over a given period of time, of a county.

The area with to which an attribute value refers to is called its **support**: aggregate properties have “block” (or polygon, or line) support, constant properties have “point” support (they apply to every point). Support matters when we manipulate the data. For instance, figure 1.5 was derived from a variable that has polygon support: the number of births per county. Rasterizing these values gives pixels with values that are associated to counties. The result of the rasterization is a meaningless map: the numeric values (“birth totals”) are not associated with the raster cells, and the county boundaries are no longer present. Totals of birth for the whole state can no longer be recovered from the pixel values. Ignoring support can easily lead to meaningless results. Chapter 5 discusses this further.

Raster cell values may have point support, e.g. when the cell records the elevation of the point at the cell centre in a digital elevation model, or cell support, e.g. when a satellite image pixel gives the color value averaged over (an area similar to the) pixel. Most file formats do not provide this information, yet it may be important to know when aggregating, regridding or warping rasters (section 7.7).

1.7 Spatial data science software

Although this book largely uses R and R packages for spatial data science, a number of these packages use software libraries that were not developed for R specifically. As an example, the dependency of R package `sf` on other R packages and system libraries is shown in figure 1.7.

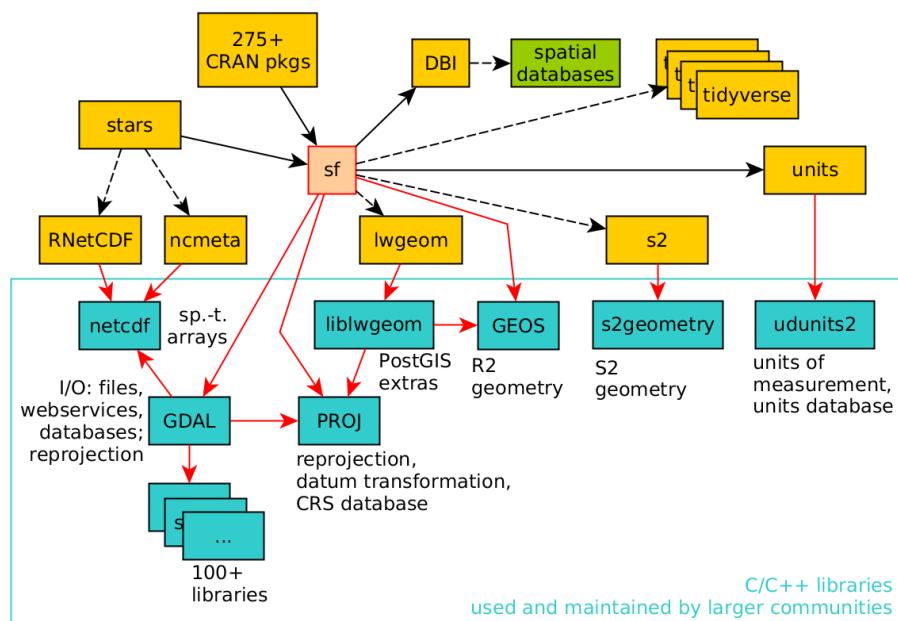


Figure 1.7: `sf` and its dependencies; arrows indicate strong dependency, dashed arrows weak dependency

The C or C++ libraries used (GDAL, GEOS, PROJ, liblwgeom, s2geometry, NetCDF, udunits2) are all developed, maintained and used by (spatial) data science communities that are large and mostly different from the R community. By using these libraries, R users share how we understand what we are doing with these other communities. Because R, Python and Julia provide interactive interfaces to this software, many users get closer to these libraries than do users

of other software based on these libraries. The first part of this book describes many of the concepts implemented in these libraries, which is relevant to spatial data science in general.

1.7.1 GDAL

GDAL (“Geospatial Data Abstraction Library”) can be seen as the Swiss army knife of spatial data; besides for R it is being used in Python, QGIS, PostGIS, and more than 100 other software projects.

GDAL is a “library of libraries” – in order to read all these data sources it needs a large number of other libraries. It typically links to over 100 other libraries, each of which provides access to e.g. a particular data file format, database access, or web service.

Binary R packages distributed by CRAN contain only statically linked code: CRAN does not want to make any assumptions about presence of third-party libraries on the host system. As a consequence, when the `sf` package is installed in binary form from CRAN, it includes a copy of all the required external libraries as well as their dependencies, which may amount to 100 Mb.

1.7.2 PROJ

PROJ (or PR ϕ J) is a library for cartographic projections and datum transformations: it converts spatial coordinates from one coordinate reference system to another. It comes with a large database of known projections and access to datum grids (high-precision pre-calculated values for datum transformations). It aligns with an international standard for coordinate reference systems (Lott, 2015). Chapter 2 deals with coordinate systems, and PROJ.

1.7.3 GEOS and s2geometry

GEOS (“Geometry Engine Open Source”) and s2geometry are two libraries for geometric operations. They are used to find measures (length, area, distance), and calculate predicates (do two geometries have any points in common?) or new geometries (which points do these two geometries have in common?). GEOS does this for flat, two-dimensional space (indicated by R^2), s2geometry does this for geometries on the sphere (indicated by S^2). Chapter 2 introduces coordinate reference systems, and chapter 4 discusses more about the differences between working with these two spaces.

1.7.4 NetCDF, udunits2, liblwgeom

NetCDF (UCAR, 2020) refers to a file format as well as a C library for reading and writing NetCDF files. It allows the definition of arrays of any dimensionality, and is widely used for spatial and spatiotemporal information, especially in the (climate) modelling communities. Udunits2 (UCAR, 2014; Pebesma et al., 2021) is a database and software library for units of measurement that allows the conversion of units, handles derived units, and supports user-defined units. The liblwgeom “library” is a software component of PostGIS (Obe and Hsu, 2015) that contains several routines missing from GDAL or GEOS, including convenient access to GeographicLib routines (Karney, 2013) that ship with PROJ.

1.8 Exercises

1. List five differences between raster and vector data.
2. In addition to those listed below figure 1.1, list five further graphical components that are often found on a map.
3. In your own words, why is the numeric information shown in figure 1.5 misleading (or meaningless)?
4. Under which conditions would you expect strong differences when doing geometrical operations on S^2 , compared to doing them on R^2 ?

Chapter 2

Coordinates

“*Data are not just numbers, they are numbers with a context*”; “*In data analysis, context provides meaning*” (Cobb and Moore, 1997)

Before we can try to understand geometries like points, lines, polygons, coverage and grids, it is useful to review coordinate systems so that we have an idea what exactly coordinates of a point reflect. For spatial data, the location of observations are characterized by coordinates, and coordinates are defined in a coordinate system. Different coordinate systems can be used for this, and the most important difference is whether coordinates are defined over a 2-dimensional or 3-dimensional space referenced to orthogonal axes (Cartesian coordinates), or using distance and directions (polar coordinates, spherical and ellipsoidal coordinates). Besides a location of observation, all observations are associated with time of observation, and so time coordinate systems are also briefly discussed. First we will briefly review *quantities*, to learn what units and datum are.

2.1 Quantities, units, datum

The VIM (“International Vocabulary of Metrology”, BIPM et al. (2012)) defines a *quantity* as a “property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference”, where “[a] reference can be a measurement unit, a measurement procedure, a reference material, or a combination of such.”

Although one could argue whether all data is constituted of quantities, there is no need to argue that proper data handling requires that numbers (or symbols) are accompanied by information on what they mean, in particular what they refer to.

A measurement system consist of *base units* for base quantities, and *derived units* for derived quantities. For instance, the SI system of units (Bureau In-

ternational des Poids et Mesures, 2006) consist of the seven base units length (metre, m), mass (kilogram, kg), time (second, s), electric current (ampere, A), thermodynamic temperature (Kelvin, K), amount of substance (mole, mol), and luminous intensity (candela, cd). Derived units are composed of products of integer powers of base units; examples are speed (m s^{-1}), density (kg m^{-3}) and area (m^2).

The special case of unitless measures can refer to either cases where units cancel out (e.g. mass fraction: kg/kg , or angle measured in rad: m/m) or to cases where objects or events were counted (e.g. 5 apples). Adding an angle to a count of apples would not make sense; adding 5 apples to 3 oranges may make sense if the result is reinterpreted as a superclass, e.g. as *pieces of fruit*. Many data variables have units that are not expressible as SI base units or derived units. Hand (2004) discusses many such measurement scales, e.g. those used to measure intelligence in social sciences, in the context of measurement units.

For many quantities, the natural origin of values is zero. This works for amounts, where differences between amounts result in meaningful negative values. For locations and times, differences have a natural zero interpretation: distance and duration. Absolute location (position) and time need a fixed origin, from which we can meaningfully measure other absolute space-time points: we call this **a datum**. For space, a datum involves more than one dimension. The combination of a datum and a measurement unit (scale) is a *reference system*.

We will now elaborate how spatial locations can be expressed as either ellipsoidal or Cartesian coordinates. The next sections will deal with temporal and spatial reference systems, and how they are handled in R.

2.2 Ellipsoidal coordinates

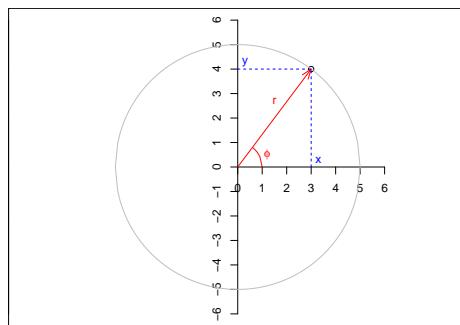


Figure 2.1: Two-dimensional polar (red) and Cartesian (blue) coordinates

Figure 2.1 shows both polar and Cartesian coordinates for a two-dimensional situation. In Cartesian coordinates, the point shown is $(x, y) = (3, 4)$, for polar

coordinates it is $(r, \phi) = (5, \arctan(4/3))$, where $\arctan(4/3)$ is approximately 0.93 radians, or 53° . Note that x , y and r all have length units, where ϕ is an angle (a unitless length/length ratio). Converting back and forth between Cartesian and polar coordinates is trivial, as

$$x = r \cos\phi,$$

$$y = r \sin\phi,$$

$$r = \sqrt{x^2 + y^2}, \text{ and}$$

$$\phi = \text{atan2}(y, x)$$

where atan2 is used in favor of $\text{atan}(y/x)$ to take care of the right quadrant.

2.2.1 Ellipsoidal coordinates

In three dimensions, where Cartesian coordinates are expressed as (x, y, z) , spherical coordinates are the three-dimensional equivalent of polar coordinates and can be expressed as (r, λ, ϕ) , where:

- r is the radius of the sphere,
- λ is the longitude, measured in the (x, y) plane counter-clockwise from positive x , and
- ϕ is the latitude, the angle between the vector and the (x, y) plane.

Figure 2.2 illustrates Cartesian geocentric and ellipsoidal coordinates.

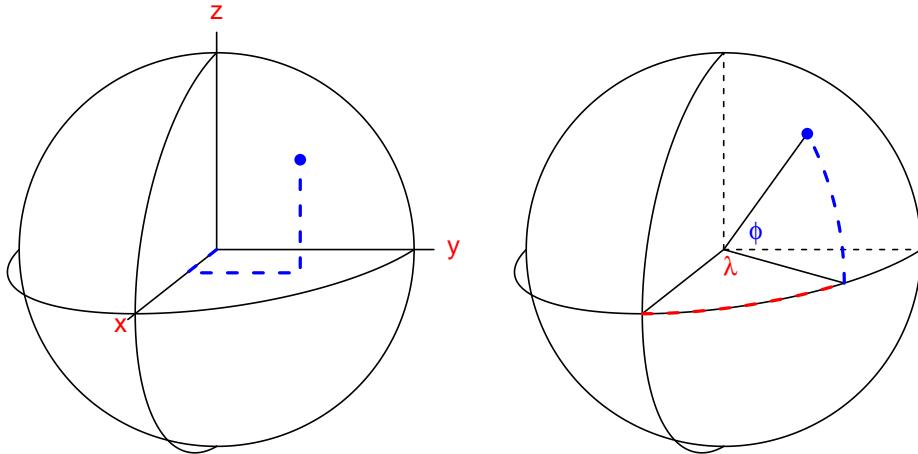


Figure 2.2: Cartesian geocentric coordinates (left) measure three distances, ellipsoidal coordinates (right) measure two angles, and possibly an ellipsoidal height

λ typically varies between -180° and 180° (or alternatively from 0° to 360°), ϕ from -90° to 90° . When we are only interested in points *on* a sphere with given radius, we can drop r : (λ, ϕ) now suffice to identify any point.

It should be noted that this is just *a* definition, one could for instance also choose to measure polar angle, the angle between the vector and z , instead of latitude. There is also a long tradition of specifying points as (ϕ, λ) but throughout this book we will stick to longitude-latitude, (λ, ϕ) . The point denoted in figure 2.2 has (λ, ϕ) or ellipsoidal coordinates with values

```
# POINT (60 47)
```

with angles measured in degrees, and geocentric coordinates

```
# POINT Z (2178844 3773868 4641765)
```

with unit metres.

For points on an ellipse, there are two ways in which angle can be expressed (figure 2.3): measured from the center of the ellipse (ψ), or measured perpendicular to the tangent on the ellipse at the target point (ϕ).

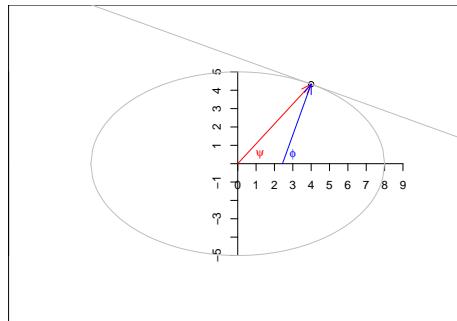


Figure 2.3: Angles on an ellipse: geodetic (blue) and geocentric (red) latitude

The most commonly used parametric model for the Earth is *an ellipsoid of revolution*, an ellipsoid with two equal semi-axes (Iliffe and Lott, 2008). In effect, this is a flattened sphere (or spheroid): the distance between the poles is (slightly: about 0.33%) smaller than the distance between two opposite points on the equator. Under this model, longitude is always measured along a circle (as in figure 2.2), and latitude along an ellipse (as in figure 2.3). If we think of figure 2.3 as a cross section of the Earth passing through the poles, the *geodetic* latitude measure ϕ is the one used when no further specification is given. The latitude measure ψ is called the *geocentric latitude*.

In addition to longitude and latitude we can add *altitude* or elevation to define points that are not on the ellipsoid, and obtain a three dimensional space again. When defining altitude, we need to choose:

- where zero altitude is: on the ellipsoid, or relative to the surface approximating mean sea level (the geoid)?
- which direction is positive, and
- which direction is “straight up”: perpendicular to the ellipsoid surface, or in the direction perpendicular to the surface of the geoid?

All these choices may matter, depending on the application area and required measurement accuracies.

The shape of the Earth is not a perfect ellipsoid. As a consequence, several ellipsoids with different shape parameters and bound to the Earth in different ways are being used. Such ellipsoids are called *datums*, and are briefly discussed in section 2.3, along with *coordinate reference systems*.

2.2.2 Projected coordinates, distances

Because paper maps and computer screens are much more abundant and practical than globes, most of the time we look at spatial data we see it *projected*: drawn on a flat, two-dimensional surface. Computing the locations in a two-dimensional space means that we work with *projected* coordinates. Projecting ellipsoidal coordinates means that shapes, directions, areas, or even all three, are distorted (Iliffe and Lott, 2008).

Distances between two points p_i and p_j in Cartesian coordinates are computed as Euclidean distances, in two dimensions by

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

with $p_i = (x_i, y_i)$ and in three dimensions by

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

with $p_i = (x_i, y_i, z_i)$. These distances represent the length of a *straight* line between two points i and j .

For two points on a circle, the length of the arc of two points $c_1 = (r, \phi_1)$ and $c_2 = (r, \phi_2)$ is

$$s_{ij} = r |\phi_1 - \phi_2| = r \theta$$

with θ the angle between ϕ_1 and ϕ_2 in radians. For very small values of θ , we will have $s_{ij} \approx d_{ij}$, because a small arc segment is nearly straight.

For two points $p_1 = (\lambda_1, \phi_1)$ and $p_2 = (\lambda_2, \phi_2)$ on a sphere with radius r' , the *great circle distance* is the arc length between p_1 and p_2 on the circle that passes through p_1 and p_2 and has the center of the sphere as its center, and is given by $s_{12} = r \theta_{12}$ with

$$\theta_{12} = \arccos(\sin \phi_1 \cdot \sin \phi_2 + \cos \phi_1 \cdot \cos \phi_2 \cdot \cos(|\lambda_1 - \lambda_2|))$$

the angle between p_1 and p_2 , in radians.

Arc distances between two points on a spheroid are more complicated to compute; a good discussion on the topic and an explanation of the method implemented in GeographicLib (part of PROJ) is given in Karney (2013).

To show that these distance measures actually give different values, we computed them for the distance Berlin - Paris. Here, `gc_` refers to ellipsoidal and spherical great circle distances, `straight_` refers to straight line, Euclidean distances between Cartesian geocentric coordinates associated on the WGS84 ellipse and sphere:

```
# Spherical geometry (s2) switched off
# Spherical geometry (s2) switched on
# Units: [km]
#      gc_ellipse straight_ellipse      gc_sphere
#          879.70            879.00        877.46
# straight_sphere
#          876.77
```

2.2.3 Bounded and unbounded spaces

Two-dimensional and three-dimensional Euclidean spaces (R^2 and R^3) are unbounded: every line in this space has infinite length, distances, areas or volumes are unbounded. In contrast, spaces defined on a circle (S^1) or sphere (S^2) define a bounded set: there may be infinitely many points but the length and area of the circle and the radius, area and volume of a sphere are bound.

This may sound trivial, but leads to some interesting findings when handling spatial data. A polygon on R^2 has unambiguously an inside and an outside. On a sphere, S^2 , any polygon divides the sphere in two parts, and which of these two is to be considered inside and which outside is ambiguous and needs to be defined e.g. by the traversal direction. Chapter 4 will further discuss consequences when working with geometries on S^2 .

2.3 Coordinate Reference Systems

We follow Lott (2015) when defining the following concepts (italics indicate literal quoting):

- a **coordinate system** is a *set of mathematical rules for specifying how coordinates are to be assigned to points*,
- a **datum** is a *parameter or set of parameters that define the position of the origin, the scale, and the orientation of a coordinate system*,

- a **geodetic datum** is a *datum describing the relationship of a two- or three-dimensional coordinate system to the Earth*, and
- a **coordinate reference system** is a *coordinate system that is related to an object by a datum; for geodetic and vertical datums, the object will be the Earth*.

A readable text that further explains these concepts is Iliffe and Lott (2008).

The Earth does not follow a regular shape. The topography of the Earth is of course known to vary strongly, but also the surface formed by constant gravity at mean sea level, the geoid, is irregular. A commonly used model that is fit to the geoid is an ellipsoid of revolution, which is an ellipse with two identical minor axes. Fitting such an ellipsoid to the Earth gives a datum. However, fitting it to different areas, or based on different sets of reference points gives different fits, and hence different datums: a datum can for instance be fixed to a particular tectonic plate (like ETRS89), others can be globally fit (like WGS84). More local fits lead to smaller approximation errors.

The definitions above imply that coordinates in degrees longitude and latitude only have a meaning, i.e. can only be interpreted unambiguously as Earth coordinates, when the datum they are associated with is given.

Note that for projected data, the data that *were* projected are associated with a reference ellipsoid (datum). Going from one projection to another *without* changing datum is called *coordinate conversion*, and passes through the ellipsoidal coordinates associated with the datum involved. This process is lossless and invertible: the parameters and equations associated with a *conversion* are not empirical. Recomputing coordinates in a new datum is called *coordinate transformation*, and is approximate: because datums are a result of model fitting, transformations between datums are models too that have been fit; the equations involved are empirical, and multiple transformation paths, based on different model fits and associated with different accuracies, are possible.

Plate tectonics imply that within a global datum, fixed objects may have coordinates that change over time, and that transformations from one datum to another may be time-dependent. Earthquakes are a cause of more local and sudden changes in coordinates.

2.4 PROJ and mapping accuracy

Very few living people active in open source geospatial software can remember the time before PROJ. PROJ (Evenden, 1990) started in the 1970s as a Fortran project, and was released in 1985 as a C library for cartographic projections. It came with command line tools for direct and inverse projections, and could be linked to software to let it support (re)projection directly. Originally, datums were considered implicit, and no datum transformations were allowed.

In the early 2000s, PROJ was known as PROJ.4, after its never changing major version number. Amongst others motivated by the rise of GPS, the need for datum transformations increased and PROJ.4 was extended with rudimentary datum support. PROJ definitions for coordinate reference systems would look like this:

```
+proj=utm +zone=33 +datum=WGS84 +units=m +no_defs
```

where *key=value* pairs are preceded by a `+` and separated by a space. This form came to be known as “PROJ.4 string”, since the PROJ project stayed at version 4.x for several decades. Other datums would come with fields like:

```
+ellps=bessel +towgs84=565.4,50.3,465.6,-0.399,0.344,-1.877,4.072
```

indicating another ellipse, as well as the seven (or three) parameters for transforming from this ellipse to WGS84 (the “World Geodetic System 1984” global datum once popularized by GPS), effectively defining the datum in terms of a transformation to WGS84.

Along with PROJ.4 came a set of databases with known (registered) projections, from which the best known is the EPSG registry. National mapping agencies would provide (and update over time) their best guesses of `+towgs84=` parameters for national coordinate reference systems, and distribute it through the EPSG registry, which was part of PROJ distributions. For some transformations, *datum grids* were available and distributed as part of PROJ.4: such grids are raster maps that provide for every location pre-computed values for the shift in longitude and latitude, or elevation, for a particular datum transformation.

```
# downsample set to c(3,3,1)

# downsample set to c(2,2)
```

In PROJ.4, every coordinate transformation had to go through a conversion to and from WGS84; even reprojecting data associated with a datum different from WGS84 had to go through a transformation to and from WGS84. The associated errors of up to 100 m were acceptable for mapping purposes for not too small areas, but applications that need high accuracy transformations, e.g. precision agriculture, planning flights of UAV’s, or object tracking are often more demanding in terms of accuracy.

In 2018, after a successful “GDAL Coordinate System Barn Raising” initiative, a number of companies profiting from the open source geospatial software stack supported the development of a more modern, mature coordinate transformation system in PROJ. Over a few years, PROJ.4 evolved through versions 5, 6, 7 and 8 and was hence renamed into PROJ (or PR ϕ J).

The most notable changes include:

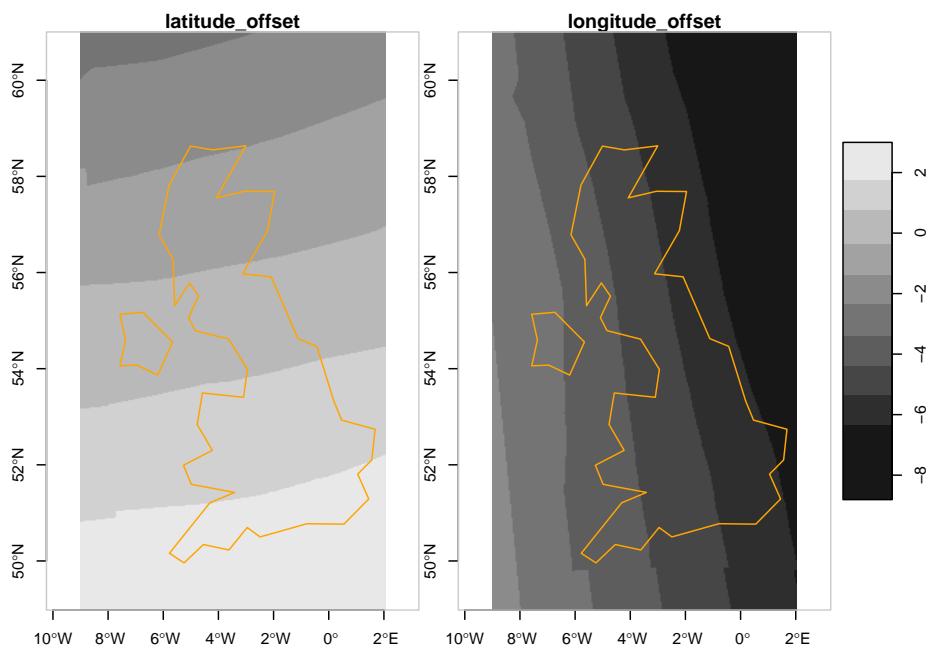


Figure 2.4: UK horizontal datum grid, from datum OSGB 1936 (EPSG:4277) to datum ETRS89 (EPSG:4258); units arc-seconds

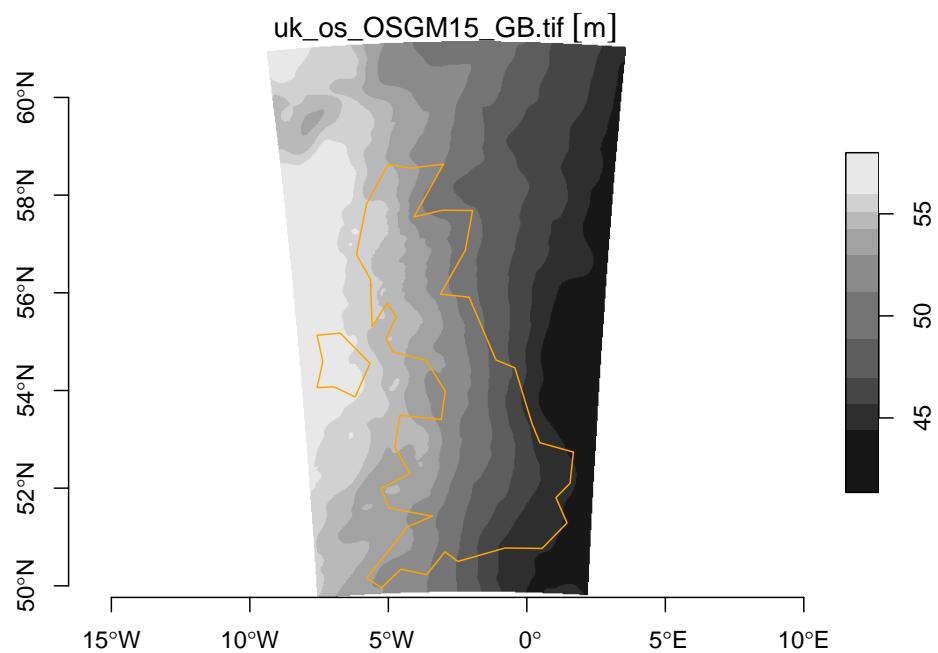


Figure 2.5: UK vertical datum grid, from ETRS89 (EPSG:4937) to ODN height (EPSG:5701), units m

- although PROJ.4 strings can still be used to initialize certain coordinate reference systems, they are no longer sufficient to represent all of them; a new format, WKT2 (described in next section) replaces it
- WGS84 as a hub datum is dropped: coordinate transformation no longer need to go through a particular datum
- multiple conversion or transformation paths (so-called pipelines) to go from CRS A to CRS B are possible, and can be reported along with the associated accuracy; PROJ will by default use the most accurate one but user control is possible
- transformation pipelines can chain an arbitrary number of elementary transformation operations, including swapping of axes and unit transformations
- datum grids, of which there are now *many* more, are no longer distributed with the library but are accessible from a content delivery network (CDN); PROJ allows to enabling and disabling network access to access these grids, and only downloads the section of the grid actually needed, storing it in a cache on the user's machine for future use
- coordinate transformations receive support for epochs, time-dependent transformations (and hence: four-dimensional coordinates, including the source and target time)
- the set of files with registered coordinate reference systems is handled in an SQLite database
- instead of always handling axis order (longitude, latitude), when the authority defines differently this is now obeyed (with the most notable example: EPSG:4326 defines axis order to be latitude, longitude.)

All these points sound like massive improvements, and accuracies of transformation can be below 1 metre. An interesting point is the last: Where we could safely assume for many decades that spatial data with ellipsoidal coordinates would have axis order (longitude, latitude), this is no longer the case. We will see in section 7.6.6 how to deal with this.

Examples of a horizontal datum grids, downloaded from cdn.proj.org, are shown in figure 2.4 and for a vertical datum grid in figure 2.5. Datum grids may carry per-pixel accuracy values.

2.5 WKT-2

Lott (2015) describes a standard for encoding coordinate reference systems, as well as transformations between them using *well known text*; the standard (and format) is referred to informally as WKT-2. As mentioned above, GDAL and PROJ fully support this encoding. An example of WKT2 for CRS OGC:CRS84 is:

```
GEOGCRS["WGS 84",
```

```

DATUM["World Geodetic System 1984",
    ELLIPSOID["WGS 84",6378137,298.257223563,
        LENGTHUNIT["metre",1]],
    ID["EPSG",6326]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433],
        ID["EPSG",8901]],
    CS[ellipsoidal,2,
        AXIS["longitude",east,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433,
                ID["EPSG",9122]]],
        AXIS["latitude",north,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433,
                ID["EPSG",9122]]]
]

```

This shows a WGS84 ellipsoid, and a coordinate system with the axis order (longitude, latitude) that can be used to replace EPSG:4326 when one wants unambiguously “traditional” (GIS) axis order.

A longer introduction on the history and recent changes in PROJ is given in Bivand (2020b), building upon the work of Knudsen and Evers (2017) and Evers and Knudsen (2017).

2.6 Exercises

Try to solve the following exercises with R (without loading packages); try to use functions where appropriate:

1. list three *geographic* measures that do not have a natural zero origin
2. convert the (x, y) points $(10, 2)$, $(-10, -2)$, $(10, -2)$ and $(0, 10)$ to polar coordinates
3. convert the polar (r, ϕ) points $(10, 45^\circ)$, $(0, 100^\circ)$ and $(5, 359^\circ)$ to Cartesian coordinates
4. assuming the Earth is a sphere with a radius of 6371 km, compute for (λ, ϕ) points the great circle distance between $(10, 10)$ and $(11, 10)$, between $(10, 80)$ and $(11, 80)$, between $(10, 10)$ and $(10, 11)$ and between $(10, 80)$ and $(10, 81)$ (units: degree). What are the distance units?

Chapter 3

Geometries

Having learned how we represent coordinates systems, we can define how geometries can be described using these coordinate systems. This chapter will explain:

- *simple features*, a standard that describes point, line and polygon geometries along with operations on them,
- operations on geometries
- coverages, subdivisions of larger regions into sub-regions
- networks

Geometries on the sphere are discussed in chapter 4, rasters and other rectangular subdivisions of space are discussed in chapter 6.

3.1 Simple feature geometries

Simple feature geometries are a way to describe the geometries of *features*. By *features* we mean *things* that have a geometry, potentially some time properties, and other attributes that could include a label describing the thing and quantitative measures of it. The main application of simple feature geometries is to describe geometries in two-dimensional space by points, lines, or polygons. The “simple” adjective refers to the fact that the line or polygon geometries are represented by sequences of points connected with straight lines that do not self-intersect.

Simple features access is a standard (Herring, 2011, 2010; ISO, 2004) for describing simple feature geometries that includes:

- a class hierarchy

- a set of operations
- binary and text encodings

We will first discuss the seven most common simple feature geometry types.

3.1.1 The big seven

The most commonly used simple features geometries, used to represent a *single* feature are:

type	description
POINT	single point geometry
MULTIPOINT	set of points
LINESTRING	single linestring (two or more points connected by straight lines)
MULTILINESTRING	set of linestrings
POLYGON	exterior ring with zero or more inner rings, denoting holes
MULTIPOLYGON	set of polygons
GEOMETRYCOLLECTION	set of the geometries above

Figure 3.1 shows examples of these basic geometry types. The human-readable, “well-known-text” (WKT) representation of the geometries plotted are:

```

POINT (0 1)
MULTIPOINT ((1 1), (2 2), (4 1), (2 3), (1 4))
LINESTRING (1 1, 5 5, 5 6, 4 6, 3 4, 2 3)
MULTILINESTRING ((1 1, 5 5, 5 6, 4 6, 3 4, 2 3), (3 0, 4 1, 2 1))
POLYGON ((2 1, 3 1, 5 2, 6 3, 5 3, 4 4, 3 4, 1 3, 2 1),
          (2 2, 3 3, 4 3, 4 2, 2 2))
MULTIPOLYGON (((2 1, 3 1, 5 2, 6 3, 5 3, 4 4, 3 4, 1 3, 2 1),
               (2 2, 3 3, 4 3, 4 2, 2 2)), ((3 7, 4 7, 5 8, 3 9, 2 8, 3 7)))
GEOMETRYCOLLECTION (
    POLYGON ((2 1, 3 1, 5 2, 6 3, 5 3, 4 4, 3 4, 1 3, 2 1),
              (2 2, 3 3, 4 3, 4 2, 2 2)),
    LINESTRING (1 6, 5 10, 5 11, 4 11, 3 9, 2 8),
    POINT (2 5),
    POINT (5 4)
)

```

In this representation, coordinates are separated by space, and points by commas. Sets are grouped by parentheses, and separated by commas.

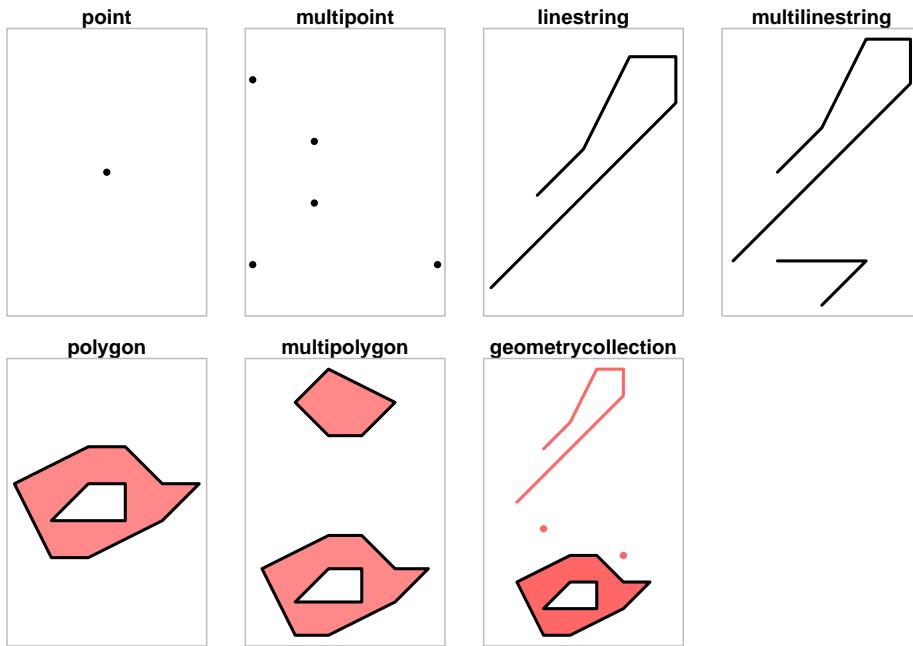


Figure 3.1: sketches of the main simple feature geometry types

Individual points in a geometry contain at least two coordinates: x and y, in that order. If these coordinates refer to ellipsoidal coordinates, x and y usually refer to longitude and latitude, respectively, although sometimes to latitude and longitude (see sections 2.4 and 7.6.6).

3.1.2 Simple and valid geometries, ring direction

Linestrings are called *simple* when they do not self-intersect:

```
# LINESTRING (0 0, 1 1, 2 2, 0 2, 1 1, 2 0)
# is_simple
#     FALSE
```

Valid polygons and multipolygons obey all of the following properties:

- polygon rings are closed (the last point equals the first)
- polygon holes (inner rings) are inside their exterior ring
- polygon inner rings maximally touch the exterior ring in single points, not over a line
- a polygon ring does not repeat its own path

- in a multipolygon, an external ring maximally touches another exterior ring in single points, not over a line

If this is not the case, the geometry concerned is not valid. Invalid geometries typically cause errors when they are processed, but can usually be repaired to make them valid.

A further convention is that the outer ring of a polygon is wounded counter-clockwise, while the holes are wounded clockwise, but polygons for which this is not the case are still considered valid. For polygons on the sphere, the “clockwise” is not very useful: if for instance we take the equator as polygon, is the Northern hemisphere or the Southern hemisphere “inside”? The convention taken here is to consider the area on the left while traversing the polygon is considered the polygon’s inside.

3.1.3 Z and M coordinates

In addition to X and Y coordinates, Single points (vertices) of simple feature geometries may have:

- a Z coordinate, denoting altitude, and/or
- an M value, denoting some “measure”

The M attribute shall be a property of the vertex. It sounds attractive to encode a time stamp in it, e.g. to pack movement data (trajectories) in `LINESTRINGS`. These become however invalid (or “non-simple”) once the trajectory self-intersects, which easily happens when only X and Y are considered for self-intersections.

Both Z and M are not found often, and software support to do something useful with them is (still) rare. Their WKT representation are fairly easily understood:

```
# POINT Z (1 3 2)
# POINT M (1 3 2)
# LINestring ZM (3 1 2 4, 4 4 2 2)
```

3.1.4 Empty geometries

A very important concept in the feature geometry framework is that of the empty geometry. Empty geometries arise naturally when we do geometrical operations (section 3.2), for instance when we want to know the intersection of `POINT (0 0)` and `POINT (1 1)`:

```
# GEOMETRYCOLLECTION EMPTY
```

and it represents essentially the empty set: when combining (unioning) an empty point it with other non-empty geometries, it vanishes.

All geometry types have a special value representing the empty (typed) geometry:

```
# POINT EMPTY
# LINESTRING M EMPTY
```

and so on, but they all point to the empty set, differing only in their dimension (section 3.2.2).

3.1.5 Ten further geometry types

There are 10 more geometry types which are more rare, but increasingly find implementation:

type	description
CIRCULARSTRING	The CIRCULARSTRING is the basic curve type, similar to a LINESTRING in the linear world. A single segment requires three points, the start and end points (first and third) and any other point on the arc. The exception to this is for a closed circle, where the start and end points are the same. In this case the second point MUST be the center of the arc, i.e. the opposite side of the circle. To chain arcs together, the last point of the previous arc becomes the first point of the next arc, just like in LINESTRING. This means that a valid circular string must have an odd number of points greater than 1.
COMPOUNDCURVE	A compound curve is a single, continuous curve that has both curved (circular) segments and linear segments. That means that in addition to having well-formed components, the end point of every component (except the last) must be coincident with the start point of the following component.
CURVEPOLYGON	Example compound curve in a curve polygon: CURVEPOLY- GON(COMPOUNDCURVE(CIRCULARSTRING(0 0,2 0, 2 1, 2 3, 4 3),(4 3, 4 5, 1 4, 0 0)), CIRCULARSTRING(1.7 1, 1.4 0.4, 1.6 0.4, 1.6 0.5, 1.7 1))

type	description
MULTICURVE	A MultiCurve is a 1-dimensional GeometryCollection whose elements are Curves, it can include linear strings, circular strings or compound strings.
MULTISURFACE	A MultiSurface is a 2-dimensional GeometryCollection whose elements are Surfaces, all using coordinates from the same coordinate reference system.
CURVE	A Curve is a 1-dimensional geometric object usually stored as a sequence of Points, with the subtype of Curve specifying the form of the interpolation between Points
SURFACE	A Surface is a 2-dimensional geometric object
POLYHEDRALSURFACE	A PolyhedralSurface is a contiguous collection of polygons, which share common boundary segments
TIN	A TIN (triangulated irregular network) is a PolyhedralSurface consisting only of Triangle patches.
TRIANGLE	A Triangle is a polygon with 3 distinct, non-collinear vertices and no interior boundary

`CIRCULARSTRING`, `COMPOUNDCURVE` and `CURVEPOLYGON` are not described in the SFA standard, but in the SQL-MM part 3 standard. The descriptions above were copied from the PostGIS manual.

3.1.6 Text and binary encodings

Part of the simple feature standard are two encodings: a text and a binary encoding. The well-known text encoding, used above, is human-readable, the well-known binary encoding is machine-readable. Binary encodings are lossless and typically faster to work with than text encoding (and decoding), and are used for instance in all communications between R package `sf` and the GDAL, GEOS, liblwgeom and s2geometry libraries (figure 1.7).

3.2 Operations on geometries

Simple feature geometries can be queried for properties, transformed into new geometries, and combinations of geometries can be queried for properties. This section gives an overview of the operations entirely focusing on *geometrical* properties. Chapter 5 focuses on the analysis of non-geometrical feature properties, in relationship to their geometries. Some of the material in this section appeared in Pebesma (2018).

We can categorize operations on geometries in terms of what they take as input, and what they return as output. In terms of output we have operations that return:

- **predicates**: a logical asserting a certain property is TRUE
- **measures**: a quantity (e.g. a numeric value with measurement unit)
- **transformations**: newly generated geometries

and in terms of what they operate on, we distinguish operations that are:

- **unary** when they work on a single geometry
- **binary** when they work on pairs of geometries
- **n-ary** when they work on sets of geometries

3.2.1 Unary predicates

Unary predicates describe a certain property of a geometry. The predicates `is_simple`, `is_valid`, and `is_empty` return respectively whether a geometry is simple, valid or empty. Given a coordinate reference system, `is_longlat` returns whether the coordinates are geographic or projected. `is(geometry, class)` checks whether a geometry belongs to a particular class.

3.2.2 Binary predicates and DE-9IM

The Dimensionally Extended Nine-Intersection Model (DE-9IM, Clementini et al. (1993); Egenhofer and Franzosa (1991)) is a model that helps describing the qualitative relation between any two geometries in two-dimensional space (R^2). Any geometry has a *dimension* value that is:

- 0 for points,
- 1 for linear geometries,
- 2 for polygonal geometries, and
- F (false) for empty geometries

Any geometry also has an inside (I), a boundary (B) and an exterior (E); these roles are obvious for polygons but, e.g. for:

- **lines** the boundary is formed by the end points, and the interior by all non-end points on the line
- **points** have a zero-dimensional inside but no boundary

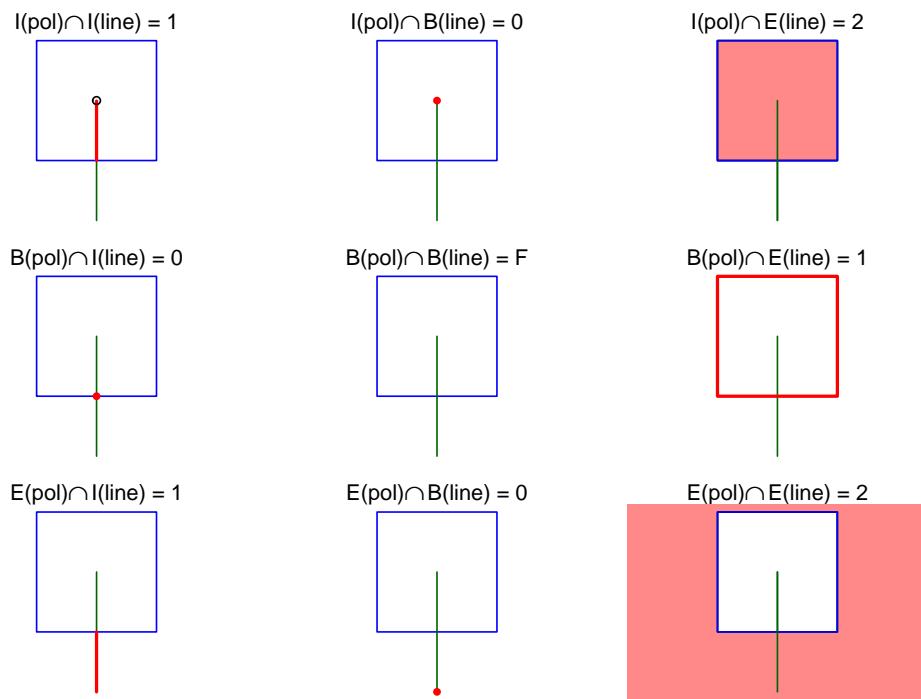


Figure 3.2: DE-9IM: intersections between the interior, boundary and exterior of a polygon (rows) and of a linestring (columns) indicated by red

Figure 3.2 shows the intersections between the I, B and E components of a polygon and a linestring indicated by red; the sub-plot title gives the dimension of these intersections (0, 1, 2 or F). The relationship between the two geometries is the concatenation of these dimensions:

```
#      [,1]
# [1,] "1020F1102"
```

Using the ability to express relationships, we can also query pairs of geometries about particular conditions expressed in a *mask string*; e.g. the string "***0*******" would evaluate TRUE when the second geometry has one or more boundary points in common with the interior of the first geometry; the symbol * standing for “any dimensionality” (0, 1, 2 or F). The mask string "**T*******" matches pairs of geometry with intersecting interiors. Here, the symbol T stands for any non-empty intersection (of dimensionality 0, 1 or 2).

Binary predicates are further described using normal-language verbs, using DE-9IM definitions. For instance, the predicate `equals` corresponds to the relationship "**T*F**FFF***". If any two geometries obey this relationship, they are (topologically) equal, but may have a different ordering of nodes.

A list of binary predicates is:

predicate	meaning	inverse of
<code>equals</code>	Two geometries A and B are topologically identical	
<code>contains</code>	None of the points of A are outside B	<code>within</code>
<code>contains_properly</code>	A contains B and B has no points in common with the boundary of A	
<code>covers</code>	No points of B lie in the exterior of A	<code>covered_by</code>
<code>covered_by</code>	Inverse of <code>covers</code>	
<code>crosses</code>	A and B have some but not all interior points in common	
<code>disjoint</code>	A and B have no points in common	<code>intersects</code>
<code>equals</code>	A and B are geometrically equal; node order number of nodes may differ;	
	identical to A contains B AND A within B	
<code>equals_exact</code>	A and B are geometrically equal, and have identical node order	
<code>intersects</code>	A and B are not disjoint	<code>disjoint</code>
<code>is_within_distance</code>	A is closer to B than a given distance	
<code>within</code>	None of the points of B are outside A	<code>contains</code>
<code>touches</code>	A and B have at least one boundary point in common, but no interior points	

predicate	meaning	inverse of
<code>overlaps</code>	A and B have some points in common; the dimension of these is identical to that of A and B	
<code>relate</code>	given a mask pattern, return whether A and B adhere to this pattern	

The Wikipedia DE-9IM page provides the `relate` patterns for each of these verbs. They are important to check out; for instance *covers* and *contains* (and their inverses) are often not completely intuitive:

- if A *contains* B, B has no points in common with the exterior *or boundary* of A
- if A *covers* B, B has no points in common with the exterior of A

3.2.3 Unary Measures

Unary measures return a measure or quantity that describes a property of the geometry:

measure	returns
<code>dimension</code>	0 for points, 1 for linear, 2 for polygons, possibly NA for empty geometries
<code>area</code>	the area of a geometry
<code>length</code>	the length of a linear geometry

3.2.4 Binary Measures

`distance` returns the distance between pairs of geometries. The qualitative measure `relate` (without mask) gives the relation pattern, a description of the geometrical relationship between two geometries explained in section 3.2.2.

3.2.5 Unary Transformers

Unary transformations work on a per-geometry basis, and for each geometry return a new geometry.

transformer	returns a geometry ...
<code>centroid</code>	of type POINT with the geometry's centroid

transformer	returns a geometry ...
buffer	that is this larger (or smaller) than the input geometry, depending on the buffer size
jitter	that was moved in space a certain amount, using a bivariate uniform distribution
wrap_dateline	cut into pieces that do no longer cover the dateline
boundary	with the boundary of the input geometry
convex_hull	that forms the convex hull of the input geometry (figure 3.3)
line_merge	after merging connecting LINESTRING elements of a MULTILINESTRING into longer LINESTRINGS .
make_valid	that is valid
node	with added nodes to linear geometries at intersections without a node; only works on individual linear geometries
point_on_surface	with a (arbitrary) point on a surface
polygonize	of type polygon, created from lines that form a closed ring
segmentize	a (linear) geometry with nodes at a given density or minimal distance
simplify	simplified by removing vertices/nodes (lines or polygons)
split	that has been split with a splitting linestring
transform	transformed or convert to a new coordinate reference system (chapter 2)
triangulate	with Delauney triangulated polygon(s) (figure 3.3)
voronoi	with the Voronoi tessellation of an input geometry (figure 3.3)
zm	with removed or added Z and/or M coordinates
collection_extract	with subgeometries from a GEOMETRYCOLLECTION of a particular type
cast	that is converted to another type
+	that is shifted over a given vector
*	that is multiplied by a scalar or matrix

3.2.6 Binary Transformers

Binary transformers are functions that return a geometry based on operating on a pair of geometries. They include:

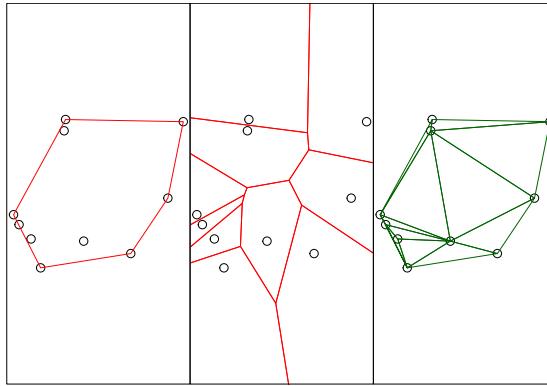


Figure 3.3: for a set of points, left: convex hull (red); middle: Voronoi polygons; right: Delaunay triangulation

function	returns	infix operator
<code>intersection</code>	the overlapping geometries for pair of geometries	&
<code>union</code>	the combination of the geometries; removes internal boundaries and duplicate points, nodes or line pieces	
<code>difference</code>	the geometries of the first after removing the overlap with the second geometry	/
<code>sym_difference</code>	the combinations of the geometries after removing where they overlap	%/%

3.2.7 N-ary Transformers

N-ary transformers operate on sets of geometries. `union` can be applied to a set of geometries to return its geometrical union. Otherwise, any set of geometries can be combined into a **MULTI**-type geometry when they have equal dimension, or else into a **GEOMETRYCOLLECTION**. Without unioning, this may lead to a geometry that is not valid, e.g. because two polygon rings have a boundary line in common.

N-ary `intersection` and `difference` take a single argument, but operate (sequentially) on all pairs, triples, quadruples, etc. Consider the plot in figure 3.4: how do we identify the area where all three boxes overlap? Using binary intersections gives us intersections for all pairs: 1-1, 1-2, 1-3, 2-1, 2-2, 2-3, 3-1, 3-2, 3-3, but does not let us identify areas where more than two geometries intersect. Figure 3.4 (right) shows the n-ary intersection: the 7 unique, non-overlapping geometries originating from intersection of one, two, or *more* geometries.

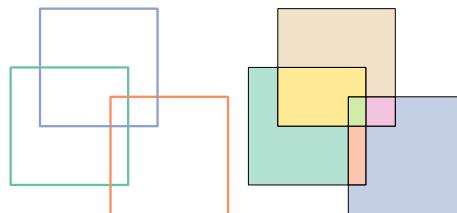


Figure 3.4: left: three overlapping boxes – how do we identify the small box where all three overlap? right: unique, non-overlapping n-ary intersections

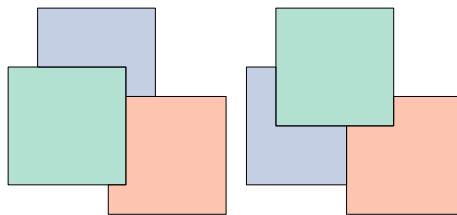


Figure 3.5: difference between subsequent boxes, left: in original order; right: in reverse order

3.3 Precision

Geometrical operations, such as finding out whether a certain point is on a line, may fail when coordinates are represented by double precision floating point numbers, such as 8-byte doubles used in R. An often chosen remedy is to limit the precision of the coordinates before the operation. For this, a *precision model* is adopted; the most common is to choose a factor p and compute rounded coordinates c' from original coordinates c by

$$c' = \text{round}(p \cdot c)/p$$

Rounding of this kind brings the coordinates to points on a regular grid with spacing $1/p$, which is beneficial for geometric computations. Of course, it also affects all computations like areas and distances, and may turn valid geometries into invalid ones. Which precision values are best for which application is often a matter of common sense combined with trial and error.

3.4 Coverages: tessellations and rasters

The Open Geospatial Consortium defines a *coverage* as a “feature that acts as a function to return values from its range for any direct position within its spatiotemporal domain” (Baumann et al., 2017). Having a *function* implies that for every “point”, i.e. every combination of spatial point and a moment in time of the spatiotemporal domain, we have *single* value for the range. This is a very common situation for spatiotemporal phenomena, a few examples can be given:

- boundary disputes aside, every point in a region (domain) belongs to a single administrative unit (range)
- at any given moment in time, every point in a region (domain) has a certain *land cover type* (range)
- every point in an area (domain) has a single elevation (range), e.g. measured with respect to a given mean sea level surface

- every spatiotemporal point in a three-dimensional body of air (domain) has single value for temperature (range)

A caveat here is that because observation or measurement always takes time and requires space, measured values are always an average over a spatiotemporal volume, and hence range variables can rarely be measured for true, zero-volume “points”; for many practical cases however the measured volume is small enough to be considered a “point”; for a variable like *land cover type* the volume needs to be chosen such that the types distinguished make sense with respect to the measured units.

In the first two of the given examples the range variable is *categorical*, in the last two the range variable is *continuous*. For categorical range variables, if large connected areas have a constant range value, an efficient way to represent these data is by storing the boundaries of the areas with constant value, such as country boundaries. Although this can be done (and is often done) by a set of simple feature geometries (polygons or multipolygons), but this brings along some challenges:

- it is hard to guarantee for such a set of simple feature polygons that they do not overlap, or that there are no gaps between them
- simple features have no way of assigning points *on* the boundary of two adjacent polygons uniquely to a single polygon, which introduces ambiguity in terms the interpretation as coverage

3.4.1 Topological models

A data model that guarantees no inadvertent gaps or overlaps of polygonal coverages is the *topological* model, examples of which are found in geographic information systems (GIS) like GRASS GIS or ArcGIS. Topological models store boundaries between polygons only once, and register which polygonal area is on either side of a boundary.

Deriving the set of (multi)polygons for each area with a constant range value from a topological model is straightforward; the other way around: reconstructing topology from a set of polygons typically involves setting thresholds on errors and handling gaps or overlaps.

3.4.2 Raster tessellations

A tessellation is a subdivision of a space (area, volume) into smaller elements by ways of polygons. A regular tessellation does this with regular polygons: triangles, squares or hexagons. Tessellations using squares are very commonly used for spatial data, and are called *raster data*. Raster data tessellate each

spatial dimension d into regular cells, formed e.g. by left-closed and right-open intervals d_i :

$$d_i = d_0 + [i \times \delta, (i + 1) \times \delta] \quad (3.1)$$

with d_0 an offset, δ the interval (cell or pixel) size, and where the cell index i is an arbitrary but consecutive set of integers. The δ value is often taken negative for the y -axis (Northing), indicating that raster row numbers increasing Southwards correspond to y -coordinates increasing Northwards.

Where in arbitrary polygon tessellations the assignment of points to polygons is ambiguous for points falling on a boundary shared by two polygons, using left-closed “[” and right-open “)” intervals in regular tessellations removes this ambiguity. This means that for rasters with negative δ values for the y -coordinate and positive for the x -coordinate, only the top-left corner point is part of each raster cell. An artifact resulting from this is shown in figure 3.6.

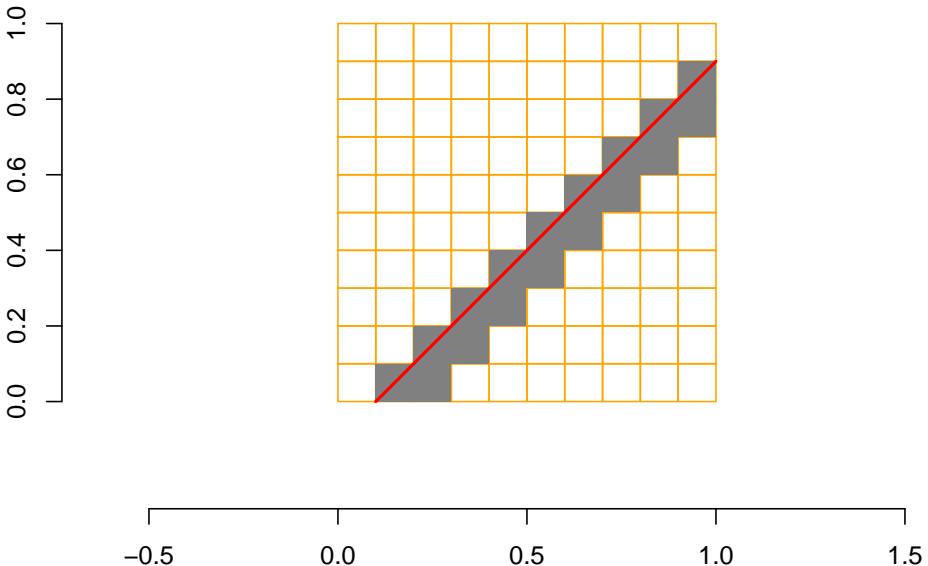


Figure 3.6: rasterization artifact: as only top-left corners are part of the raster cell, only cells below the diagonal line are rasterized

Tessellating the time dimension with left-closed, right-open intervals is very common, and reflects the implicit assumption underlying time series software such as the `xts` package in R, where time stamps indicate the start of time intervals. Different models can be combined: one could use simple feature polygons to tessellate space, and combine this with a regular tessellation of time in order to cover a space-time *vector data cube*. Raster and vector data cubes are discussed in chapter 6.

As mentioned above, besides square cells the other two shapes that can lead to regular tessellations of R^2 are triangles and hexagons. On the sphere, there

are few more, including cube, octahedron, icosahedron and dodecahedron. A spatial index that builds on the cube is s2geometry, the H3 library uses the icosahedron and densifies that with (mostly) hexagons. Mosaics that cover the entire Earth are also called *discrete global grids*.

3.5 Networks

Spatial networks are typically composed of linear (`LINESTRING`) elements, but possess further topological properties describing the network coherence:

- start and endpoints of a linestring may be connected to other linestring start or end points, forming a set of nodes and edges
- edges may be directed, to only allow for connection (flow, transport) in one way

R packages including `osmar` (Schlesinger and Eugster, 2013), `stplanr` (Lovelace et al., 2021) and `sfnetworks` (van der Meer et al., 2021) provide functionality for constructing network objects, and working with them, e.g. computing shortest or fastest routes through a network. Package `spatstat` (Baddeley et al., 2021, 2015) has infrastructure for analysing point patterns on linear networks (chapter `?(pointpatterns)`).

3.6 Exercises

For the following exercises, use R where possible.

1. Give two examples of geometries in 2-D (flat) space that cannot be represented as simple feature geometries, and create a plot of them.
2. Recompute the coordinates 10.542, 0.01, 45321.6789 using precision values 1, 1e3, 1e6, and 1e-2.
3. Describe a practical problem for which an n-ary intersection would be needed.
4. How can you create a Voronoi diagram (figure 3.3) that has one closed polygons for every single point?
5. Give the unary measure `dimension` for geometries `POINT Z (0 1 1)`, `LINESTRING Z (0 0 1,1 1 2)`, and `POLYGON Z ((0 0 0,1 0 0,1 1 0,0 0 0))`
6. Give the DE-9IM relation between `LINESTRING(0 0,1 0)` and `LINESTRING(0.5 0,0.5 1)`; explain the individual characters.
7. Can a set of simple feature polygons form a coverage? If so, under which constraints?

8. For the `nc` counties in the dataset that comes with R package `sf`, find the points touched by four counties.
9. How would figure 3.6 look like if δ for the y -coordinate was positive?

Chapter 4

Spherical Geometries

“*There are too many false conclusions drawn and stupid measurements made when geographic software, built for projected Cartesian coordinates in a local setting, is applied at the global scale*” (Chrisman, 2012)

The previous chapter discussed geometries defined on the plane, R^2 . This chapter discusses what changes when we consider geometries not on the plane, but on the sphere (S^2).

Although we learned in chapter 2 that the shape of the Earth is usually approximated by an ellipsoid, none of the libraries shown in green in figure 1.7 provide access to a comprehensive set of functions that compute on an ellipsoid. Only the s2geometry (Dunnington et al., 2021; Veach et al., 2020) library does provide it using a sphere rather than an ellipsoid. However, when compared to using a flat (projected) space we did in the previous chapter, a sphere is a much better approximation to an ellipsoid.

4.1 Straight lines

The basic premise of *simple features* of chapter 3 is that geometries are represented by sequences of points *connected by straight lines*. On R^2 (or any Cartesian space), this is trivial, but on a sphere straight lines do not exist. The shortest line connecting two points is an arc of the circle through both points and the center of the sphere, also called a *great circle segment*. A consequence is that “the” shortest distance line connecting two points on opposing sides of the sphere does not exist, as any great circle segment connecting them has equal length.

4.2 Ring direction

Any polygon on the sphere divides the sphere surface in two parts with finite area: the inside and the outside. Using the “counter clockwise rule” as was done for R^2 will not work, because the direction interpretation depends on what is defined as inside. The convention here is to define the inside as the left (or right) side of the polygon boundary when traversing its points in sequence. Reversal of the node order then switches inside and outside.

4.3 Full polygon

In addition to empty polygons, one can define the full polygon on a sphere, which comprises its entire surface. This is useful, for instance for computing the oceans as the geometric difference between the full polygon and those of the land mass.

4.4 Bounding box, rectangle, and cap

Where in R^2 one can easily define bounding boxes as the range of the x and y coordinates, for ellipsoidal coordinates these ranges are not of much use when geometries cross the antimeridian (longitude $+$ / $-$ 180) or one of the poles. The assumption in R^2 that lower x values are Westwards of higher ones does not hold when crossing the antimeridian. An alternative to delineating an area on a sphere that is more natural is the *bounding cap*, defined by its center coordinates and a radius. For Antarctica, as depicted in figure 4.1 (a) and (c), the bounding box formed by coordinate ranges is

```
#   xmin   ymin   xmax   ymax
# -180.0 -85.2 179.6 -60.5
```

which clearly does not contain the region (`ymin` being -90 and `xmax` 180). Two geometries that do contain the region are the bounding cap:

```
#   lng lat angle
# 1    0 -90  29.5
```

and the bounding *rectangle*:

```
#   lng_lo lat_lo lng_hi lat_hi
# 1    -180     -90      180    -60.5
```

For an area spanning the antimeridian, here the Fiji island country, the bounding box:

```
#   xmin    ymin    xmax    ymax
# -179.9  -21.7  180.2  -12.5
```

seems to span most of the Earth, as opposed to the bounding rectangle:

```
#   lng_lo lat_lo lng_hi lat_hi
# 1     175  -21.7   -178  -12.5
```

where a value `lng_lo` *larger* than `lng_hi` indicates that the bounding rectangle spans the antimeridian. This property could not be inferred from the coordinate ranges.

4.5 Validity on the sphere

Many global datasets are given in ellipsoidal coordinates but are prepared in a way that they “work” when interpreted on the R^2 space $[-180,180] \times [-90,90]$. This means that:

- geometries crossing the antimeridian (longitude $+/- 180$) are cut in halves, such that they no longer cross it (but nearly touch each other)
- geometries including a pole, like Antarctica, are cut at $+/- 180$ and make an excursion through $-180,-90$ and $180,-90$ (both representing the Geographic South Pole)

Figure 4.1 shows two different representation of Antarctica, plotted with ellipsoidal coordinates taken as R^2 (top) and in a Polar Stereographic projection (bottom), without (left) and with (right) an excursion through the Geographic South Pole. In the projections as plotted, polygons (b) and (c) are valid; polygon (a) is not valid as it self-intersects, polygon (d) is not valid because it traverses the same edge to the South Pole twice. On the sphere (S^2), polygon (a) is valid but (b) is not, for the same reason as (d) is not valid.

4.6 Exercises

For the following exercises, use R where possible or relevant.

1. How does the GeoJSON format define “straight” lines between ellipsoidal coordinates (section 3.1.1)? Using this definition of straight, how would `LINESTRING(0 85,180 85)` look like in a polar projection? How could this geometry be modified to have it cross the North Pole?

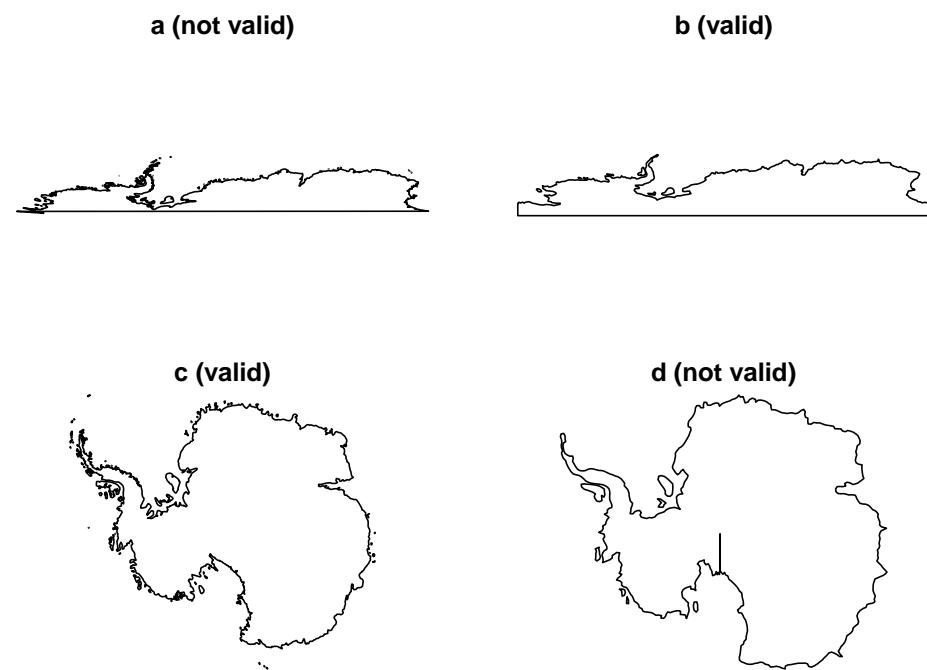


Figure 4.1: different representations of Antarctica, (a, c): with a polygon not passing through (-180 -90); (b, d): with a polygon passing through (-180 -90) and (180 -90)

2. For a typical polygon on S^2 , how can you find out ring direction?
3. Are there advantages of using bounding caps over using bounding boxes? If so, list them.
4. Why is, for small areas, the orthographic projection centered at the area a good approximation of the geometry as handled on S^2 ?
5. For `rnatu`re`earth`::`ne_countries`(country = "Fiji", returnclass="sf"), check whether the geometry is valid on R^2 , on an orthographic projection centered on the country, and on S^2 . How can the geometry be made valid on S^2 ? Plot the resulting geometry back on R^2 . Compare the centroid of the country, as computed on R^2 and on S^2 , and the distance between the two.

Chapter 5

Attributes and Support

Feature *attributes* refer to the properties of features (“things”) that do not describe the feature’s geometry. Feature attributes can be *derived* from geometry (e.g. length of a `LINESTRING`, area of a `POLYGON`) but they can also refer to non-derived properties, such as:

- the name of a street or a county
- the number of people living in a country
- the type of a road
- the soil type in a polygon from a soil map
- the opening hours of a shop
- the body weight or heart beat rate of an animal
- the NO_2 concentration measured at an air quality monitoring station

In some cases, time properties can be seen as attributes of features, e.g. the date of birth of a person or the construction year of a road. When an attribute such as for instance air quality is a function of both space and time, time is best handled on equal footing with geometry (e.g. in a data cube, see chapter 6).

Spatial data science software implementing simple features typically organizes data in tables that contain both geometries and attributes for features; this is true for `geopandas` in Python, `PostGIS` tables in PostgreSQL, and `sf` objects in R. The geometric operations described in section 3.2 operate on geometries *only*, and may occasionally yield new attributes (predicates, measures or transformations), but do not operate on attributes present.

When, while manipulating geometries, attribute *values* are retained unmodified, support problems may arise. If we look into a simple case of replacing a county polygon with the centroid of that polygon on a dataset that has attributes, we see that R package `sf` issues a warning:

```
# Warning in st_centroid.sf(.): st_centroid assumes attributes are
# constant over geometries of x
```

The reason for this is that the dataset contains variables with values that are associated with entire polygons – in this case: population counts – meaning they are not associated with a `POINT` geometry replacing the polygon.

In section 1.6 we already described that for non-point geometries (lines, polygons), feature attribute values either have *point support*, meaning that the value applies to *every point*, or they have *block support*, meaning that the value *summarizes all points* in the geometry. (More complex options, e.g. in between these two extremes, may also occur.) This chapter will describe different ways in which an attribute may relate to the geometry, its consequences on analysing such data, and ways to derive attribute data for different geometries (up- and downscaling).

5.1 Attribute-geometry relationships and support

Changing the feature geometry without changing the feature attributes does change the *feature*, since the feature is characterised by the combination of geometry and attributes. Can we, ahead of time, predict whether the resulting feature will still meaningfully relate to the attribute value when we replace all geometries for instance with their convex hull or centroid? It depends.

Take the example of a road, represented by a `LINESTRING`, which has an attribute property *road width* equal to 10 m. What can we say about the road width of an arbitrary subsection of this road? That depends on whether the attribute road length describes, for instance the road width *everywhere*, meaning that road width is constant along the road, or whether it describes an aggregate property, such as minimum or average road width. In case of the minimum, for an arbitrary subsection of the road one could still argue that the minimum road width must be at least as large as the minimum road width for the whole segment, but it may no longer be *the minimum* for that subsection. This gives us two “types” for the attribute-geometry relationship (**AGR**):

- **constant** the attribute value is valid everywhere in or over the geometry; we can think of the feature as consisting of an infinite number of points that all have this attribute value; in the geostatistical literature this is known as a variable with *point support*
- **aggregate** the attribute is an aggregate, a summary value over the geometry; we can think of the feature as a *single* observation with a value that is associated with the *entire* geometry; this is also known as a variable having *block support*

For polygon data, typical examples of **constant** AGR (point support) variables are:

- land use for a land use polygon
- rock units or geologic strata in a geological map
- soil type in a soil map
- elevation class in a elevation map that shows elevation as classes
- climate zone in a climate zone map

A typical property of such variables is that they have geometries that are not man-made and also not associated with a sensor device (such as remote sensing image pixel boundaries). Instead, the geometry follows from the variable observed.

Examples for the **aggregate** AGR (block support) variables are:

- population, either as number of persons or as population density
- other socio-economic variables, summarised by area
- average reflectance over a remote sensing pixel
- total emission of pollutants by region
- block mean NO₂ concentrations, as e.g. obtained by block kriging over square blocks or a dispersion model that predicts areal means

A typical property of such variables is that associated geometries come for instance from legislation, observation devices or analysis choices, but not intrinsically from the observed variable.

A third type of AGR arises when an attribute *identifies* a feature geometry; we call an attribute an **identity** variable when the associated geometry uniquely identifies the variable's value (there are no other geometries with the same value). An example is county name: the name identifies the county, and is still the county for any sub-area (point support), but for arbitrary sub-areas, the attribute loses the **identity** property to become a **constant** attribute. An example is:

- an arbitrary point (or region) inside a county is still part of the county and must have the same value for county name, but it does not longer identify the (entire) geometry corresponding to that county

The challenge here is that spatial information (ignoring time for simplicity) belongs to different phenomena types (e.g. Scheider et al., 2016), including:

- **fields:** where over *continuous* space, every location corresponds to a single value, e.g. elevation, air quality, or land use
- **objects:** found at a *discrete* set of locations, e.g. houses or persons

- **aggregates:** e.g. sums, totals, averages of fields, counts or densities of objects, associated with lines or regions

but that different spatial geometry types (points, lines, polygons, raster cells) have no simple mapping to these phenomena types:

- points may refer to sample locations of observations on fields (air quality) or to locations of objects
- lines may be used for objects (roads, rivers), contours of a field, or administrative borders
- raster pixels and polygons may reflect fields of a categorical variable such as land use (*coverage*), but also aggregates such as population density

Properly specifying attribute-geometry relationships, and warning against their absence or cases when change in geometry (change of support) implies a change of information can help avoiding a large class of common spatial data analysis mistakes (Stasch et al., 2014) associated with the *support* of spatial data.

5.2 Aggregating and summarising

Aggregating records in a table (or `data.frame`) involves two steps:

- grouping records based on a grouping predicate, and
- applying an aggregation function to the attribute values of a group to summarize them into a single number.

In SQL, this looks for instance like

```
SELECT GroupID, SUM(population) FROM table GROUP BY GroupID;
```

indicating the aggregation *function* (`SUM`) and the *grouping predicate* (`GroupID`).

R package `dplyr` for instance uses two steps to accomplish this: function `group_by` specifies the group membership of records, `summarize` computes data summaries (such as `sum` or `mean`) for each of the groups. R (base) function `aggregate` does both in a single function that takes the data table, the grouping predicate(s) and the aggregation function.

An example for the North Carolina counties is shown in figure 5.1. Here, we grouped counties by their position (according to the quadrant in which the county centroid is with respect to ellipsoidal coordinate `POINT(-79, 35.5)`) and counted the number of disease cases per group. The result shows that the geometries of the resulting groups have been unioned (section 3.2.6): this

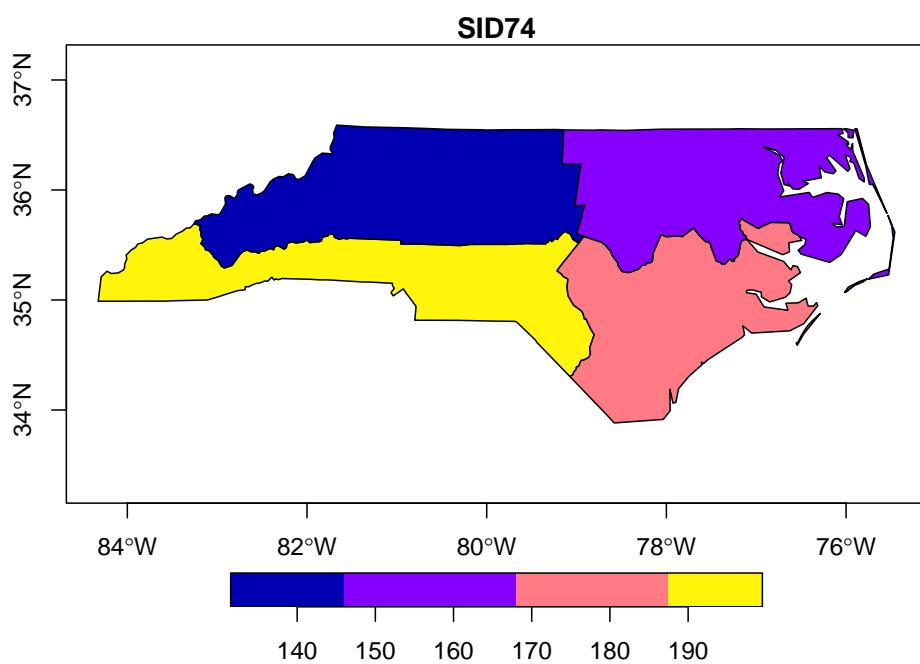


Figure 5.1: SID74 counts by county quadrant, with county polygons unioned by county quadrant

is necessary because the MULTIPOLYGON formed by just putting all the county geometries together would have many duplicate boundaries, and hence not be *valid* (section 3.1.2).

Plotting collated county polygons is technically not a problem, but for this case would raise the wrong suggestion that the group sums relate to the counties, and not the group of counties.

One particular property of aggregation in this way is that each record is assigned to a single group; this has the advantage that the sum of the group-wise sums equals the sum of the ungrouped data: for variables that reflect *amount*, nothing gets lost and nothing is added. The newly formed geometry is the result of unioning the geometries of the records.

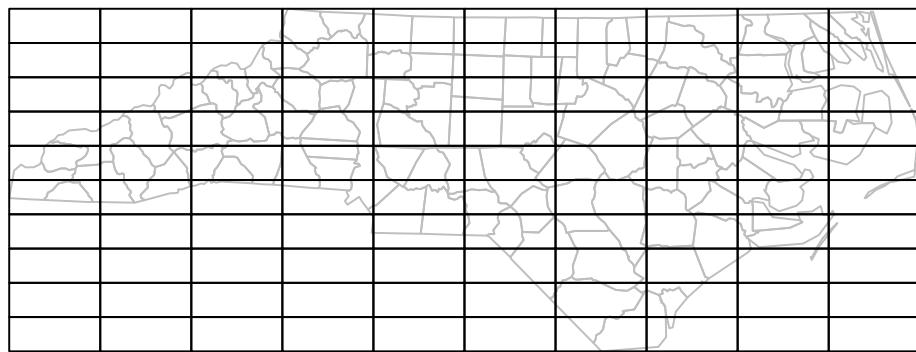


Figure 5.2: Example target blocks layed out over North Carolina counties

When we need an aggregate for a new area that is *not* a union of the geometries for a group of records, and we use a spatial predicate then single records may be matched to multiple groups. When taking the rectangles of figure 5.2 as the target areas, and summing for each rectangle the disease cases of the counties that *intersect* with the rectangles of figure 5.2, the sum of these will be much larger:

```
#    sid74_sum_counties sid74_sum_rectangles
#                      667                      2621
```

Choosing another predicate, e.g. *contains* or *covers* would on the contrary result in much smaller values, because many counties are not contained by *any* the target geometries. However, there are a few cases where this approach might be good or satisfactory:

- when we want to aggregate POINT geometries by a set of polygons, and all points are contained by a single polygon. If points fall on a shared boundary than they are assigned to both polygons (this is the case for

DE-9IM-based GEOS library; the s2geometry library has the option to define polygons as “semi-open”, which implies that points are assigned to single polygons when the polygons form a coverage)

- when aggregating many very small polygons or raster pixels over larger areas, e.g. averaging altitude from 30 m resolution raster over North Carolina counties, the error made by multiple matches may be insignificant

A more comprehensive approach to aggregating spatial data associated to areas to larger, arbitrary shaped areas is by using area-weighted interpolation.

5.3 Area-weighted interpolation

When we want to combine geometries and attributes of two datasets such that we get attribute values of a source dataset summarised for the geometries of a target, where source and target geometries are unrelated, area-weighted interpolation may be a simple approach. In effect, it considers the area of overlap of the source and target geometries, and uses that to weight the source attribute values into the target value (Goodchild and Lam, 1980; Do et al., 2015a,b, 2021). Here, we follow the notation of Do et al. (2015b).

Area-weighted interpolation computes, for each of q spatial target areas T_j , a weighted average from the values Y_i corresponding to the p spatial source areas S_i ,

$$\hat{Y}_j(T_j) = \sum_{i=1}^p w_{ij} Y_i(S_i) \quad (5.1)$$

where the w_{ij} depend on the amount of overlap of T_j and S_i , $A_{ij} = T_j \cap S_i$.

Different options exist for choosing weights, including methods using external variables (e.g. dasymetric mapping, Mennis (2003)). Two simple approaches for computing weights that do not use external variables arise, depending on whether the variable Z is *intensive* or *extensive*.

5.3.1 Spatially extensive and intensive variables

An example of a extensive variable is *population count*. It is associated with an area, and if that area is cut into smaller areas, the population count is split accordingly: not necessarily proportional to area, because population is rarely uniform, but split in such a way that the sum of the population count for the smaller areas equals that of the total. An example of a related variable that is *intensive* is population density. If an area is split into smaller areas, population density is not split similarly: the sum of the population densities for the smaller

areas is a meaningless measure, as opposed to the average of the population densities which will be similar to the density of the total area.

Extensive variables correspond to amounts, associated with a physical size (length, area, volume); for *spatially* extensive variables, if the area a value corresponds to is cut in parts, the values associated with the sub-area are split accordingly. In other words: the value is proportional to the support. Intensive variables are variables that do not have values proportional to support: if the area is split, values may vary but *on average* remain the same. The corresponding example of an intensive variable is *population density*: when we split an area into sub-areas, the sub-areas either have identical population densities (in case population is uniformly distributed) or, more realistically, have *varying* population densities that by necessity are both higher and lower than the density of the total area.

When we assume that the extensive variable Y is uniformly distributed over space, the value \hat{Y}_{ij} , derived from Y_i for a sub-area of S_i , $A_{ij} = T_j \cap S_i$ of S_i is

$$\hat{Y}_{ij}(A_{ij}) = \frac{|A_{ij}|}{|S_i|} Y_i(S_i)$$

where $|\cdot|$ denotes the spatial area. For estimating $\hat{Y}_j(T_j)$ we sum all the elements over area T_j :

$$\hat{Y}_j(T_j) = \sum_{i=1}^p \frac{|A_{ij}|}{|S_i|} Y_i(S_i) \quad (5.2)$$

For an intensive variable, under the assumption that the variable has a constant value over each area S_i , the estimate for a sub-area equals that of the total,

$$\hat{Y}_{ij} = Y_i(S_i)$$

and we can estimate the value of Y for a new spatial unit T_j by an area-weighted average of the source values:

$$\hat{Y}_j(T_j) = \sum_{i=1}^p \frac{|A_{ij}|}{|T_j|} Y_i(S_i) \quad (5.3)$$

5.3.2 Dasymetric mapping

Dasymetric mapping distributes variables, such as population, known at a coarse spatial aggregation level over finer spatial units by using other variables that are associated with population distribution, such as land use, building density, or road density. The simplest approach to dasymetric mapping is obtained for

extensive variables, where the ratio $|A_{ij}|/|S_i|$ in (5.2) is replaced by the ratio of another extensive variable $X_{ij}(S_{ij})/X_i(S_i)$, which has to be known for both the intersecting regions S_{ij} and the source regions S_i . Do et al. (2015b) discuss several alternatives for intensive Y and/or X , and cases where X is known for other areas.

5.3.3 Support in file formats

GDAL's vector API supports reading and writing so-called field domains, which can have a “split policy” and a “merge policy” indicating what should be done with attribute variables when geometries are split or merged. The values of these can be “duplicate” for split and “geometry weighted” for merge, in case of spatially intensive variables, or they can be “geometry ratio” for split and “sum” for merge, in case of spatially extensive variables. At the time of writing this, the file formats supporting this are GeoPackage and FileGDB.

5.4 Up- and Downscaling

Up- and downscaling refers in general to obtaining high-resolution information from low-resolution data (downscaling) or obtaining low-resolution information from high-resolution data (upscaling). Both are activities involve attributes' relation to geometries and both change support. They are synonymous with aggregation (upscaling) and disaggregation (downscaling).

The simplest form of downscaling is sampling (or extracting) polygon, line or grid cell values at point locations. This works well for variables with point-support (“constant” AGR), but is at best approximate when the values are aggregates.

Challenging applications for downscaling include high-resolution prediction of variables obtained by low-resolution weather prediction models or climate change models, and the high-resolution prediction of satellite image derived variables based on the fusion of sensors with different spatial and temporal resolutions.

The application of areal interpolation using (5.1) with its realisations for extensive (5.2) and intensive (5.3) variables allows moving information from any source area S_i to any target area T_j as long as the two areas have some overlap. This means that one can go arbitrarily to much larger units (aggregation) or to much smaller units (disaggregation). Of course this makes only sense to the extent that the assumptions hold: over the source regions extensive variables need to be uniformly distributed and intensive variables need to have constant value.

The ultimate disaggregation involves retrieving (extracting) point values from line or area data. For this, we cannot work with equations (5.2) or (5.3) because

$|A_{ij}| = 0$ for points, but under the assumption of having a constant value over the geometry, for intensive variables the value $Y_i(S_i)$ can be assigned to points as long as all points can be uniquely assigned to a single source area S_i . For polygon data, this implies that Y needs to be a coverage variable (section 3.4).

In cases where values associated with areas are **aggregate** values over the area, the assumptions made by area-weighted interpolation or dasymetric mapping – uniformity or constant values over the source areas – are highly unrealistic. In such cases, these simple approaches still be reasonable approximations, for instance when:

- the source and target area are nearly identical
- the variability inside source units is very small, and the variable is nearly uniform or constant

In other cases, results obtained using these methods are merely consequences of unjustified assumptions. Statistical aggregation methods that can estimate quantities for larger regions from points or smaller regions include:

- design-based methods, which require that a probability sample is available from the target region, with known inclusion probabilities (Brus (2021a), section 10.1), and
- model-based methods, which assume a random field model with spatially correlated values (block kriging, section 12.5)

Alternative disaggregation methods include:

- deterministic, smoothing-based approaches such as kernel- or spline-based smoothing methods (Tobler, 1979; Martin, 1989)
- statistical, model-based approaches: area-to-area and area-to-point kriging (Kyriakidis, 2004; Raim et al., 2021).

5.5 Exercises

Where relevant, try to make the following exercises with R.

1. When we add a variable to the `nc` dataset by `nc$State = "North Carolina"` (i.e., all counties get assigned the same state name). Which value would you attach to this variable for the attribute-geometry relationship (`agr`)?
2. Create a new `sf` object from the geometry obtained by `st_union(nc)`, and assign "North Carolina" to the variable `State`. Which `agr` can you now assign to this attribute variable?

3. Use `st_area` to add a variable with name `area` to `nc`. Compare the `area` and `AREA` variables in the `nc` dataset. What are the units of `AREA`? Are the two linearly related? If there are discrepancies, what could be the cause?
4. Is the `area` variable intensive or extensive? Is its `agr` equal to `constant`, `identity` or `aggregate`?
5. Consider figure 5.3; using the equations in section 5.3.1, compute the area-weighted interpolations for (a) the dashed cell and (b) for the square enclosing all four solid cells, first for the case where the four cells represent (i) an extensive variable, and (ii) an intensive variable. The red numbers are the data values of the source areas.

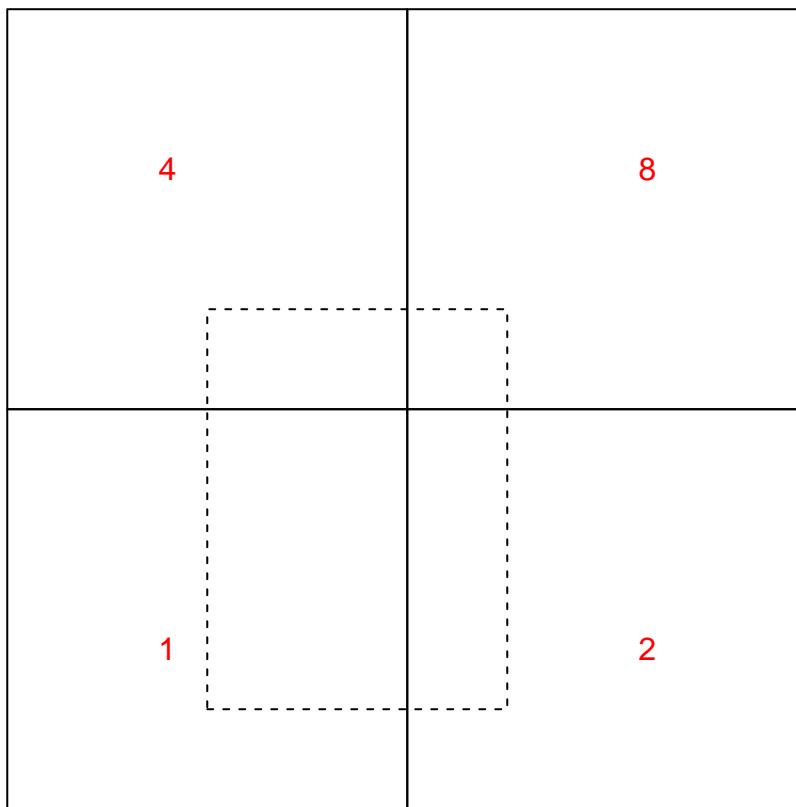


Figure 5.3: example data for area-weighted interpolation

Chapter 6

Data Cubes

Data cubes arise naturally when we observe properties of a set of geometries repeatedly over time. Time information may sometimes be considered as an attribute of a feature, e.g. when we register the year of construction of a building, or the date of birth of a person (chapter 5). In other cases it may refer to the time of observing an attribute, or the time for which a prediction of an attribute has been made. In these cases, time is on equal footing with space, and time and space together describe the physical dimensions over which we observe, model, make predictions or make forecasts.

One way of considering our world is that of a four-dimensionsal space, with three space and one time dimension. In that view, events become “things” or “objects” that have as duration their size on the time dimension (Galton, 2004). Although such a view does not align well with how we experience and describe the world, from a data analytic perspective, four numbers, along with their reference systems, suffice to describe space and time coordinates of an observation associated with a point location and time instance.

We define data cubes as array data with one or more array dimensions associated with space and/or time (Lu et al., 2018). This implies that raster data, features with attributes, and time series data are all special cases of data cubes. Since we do not restrict to three-dimensional structures, we actually mean *hypercubes* rather than cubes, and as the cube extent of the different dimensions does not have to be identical, or have comparable units, the better term would be *hyperrectangle*. For simplicity, we talk about data cubes instead.

A canonical form of a data cube is shown in figure 6.1: it shows in a perspective plot a set of raster layers for the same region that were collected (observed, or modelled) at different time steps. The three cube dimensions, longitude, latitude and time are thought of as being orthogonal. Arbitrary two-dimensional cube slices are obtained by fixing one of the dimensions at a particular value, one-dimensional slices are obtained by fixing two of the dimensions at a particular

value, and a scalar is obtained by fixing three dimensions at a particular value.

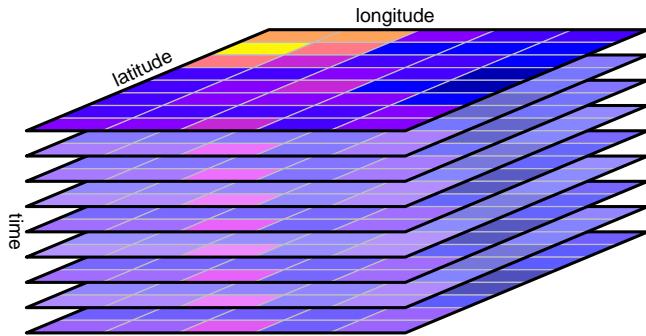


Figure 6.1: Raster data cube with dimensions latitude, longitude and time

6.1 A four-dimensional data cube

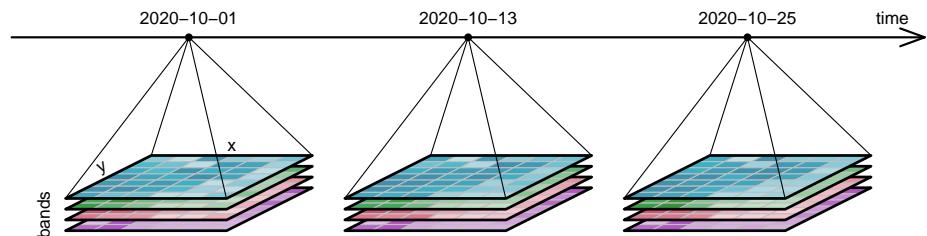


Figure 6.2: Four-dimensional raster data cube with dimensions x, y, bands and time

Figure 6.2 depicts a four-dimensional raster data cube, where three-dimensional raster data cubes with a spectral dimension (“bands”) are organised along a fourth dimension, a time axis. Color image data always has three bands (Blue, Green, Red), and this example has a fourth band (near infrared, NIR), which is commonly found in spectral remote sensing data.

Figure 6.3 shows exactly the same data, but layed out flat as a facet plot (or scatterplot matrix), where two dimensions (x and y) are aligned with (or nested within) the dimensions *bands* and *time*, respectively.

6.2 Dimensions, attributes, and support

Phenomena in space and time can be thought of as functions with domain space and time, and with range one or more observed attributes. For clearly

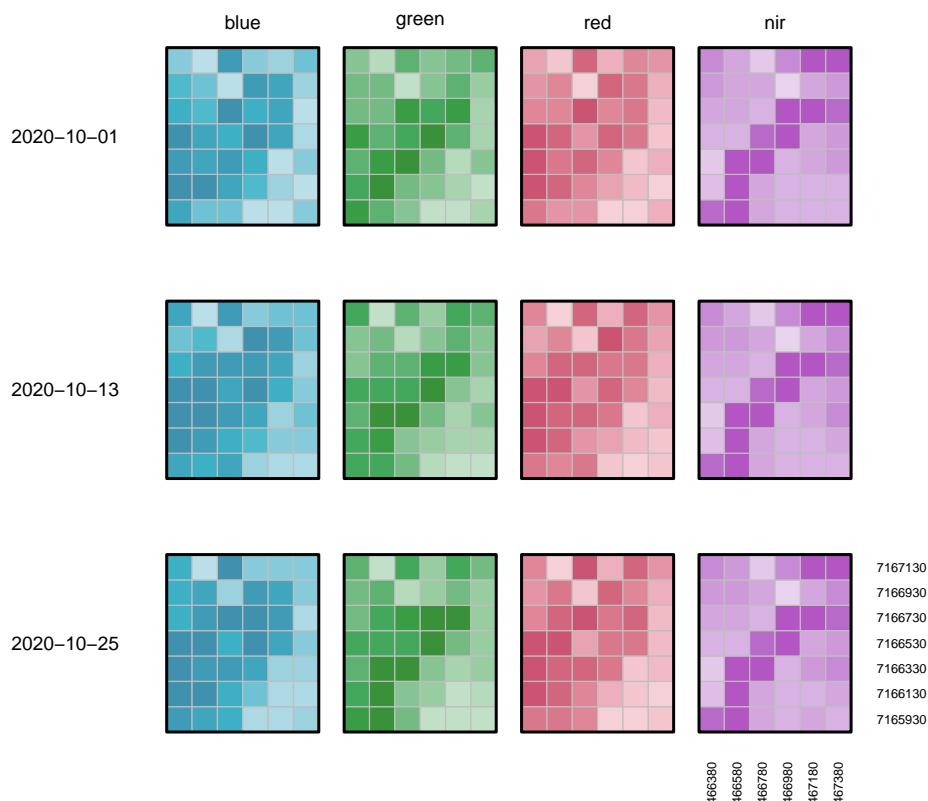


Figure 6.3: Four-dimensional raster data cube layed out flat over two dimensions

identifiable discrete events or objects, the range is typically discrete, and precise delineation involves describing the precise coordinates where a thing starts or stops, which is best suited by vector geometries. For continuous phenomena, variables that take on a value everywhere such as air temperature or land use type, there are infinitely many values to represent and a common approach is to discretize space and time *regularly* over the spatiotemporal domain (extent) of interest. This leads to a number of familiar data structures:

- time series, depicted as time lines for functions of time
- image or raster data for two-dimensional spatial data
- time sequences of images for dynamic spatial data

The third form of this, where a variable Z depends on x , y and t , as in

$$Z = f(x, y, t)$$

is the archetype of a spatiotemporal array or *data cube*: the shape of the volume where points regularly discretizing the domain forms a cube. We call the variables that form the range (here: x, y, t) the cube *dimensions*. Data cubes may have multiple attributes, as in

$$\{Z_1, Z_2, \dots, Z_p\} = f(x, y, t)$$

and if Z is functional, e.g. reflectance values measured over the electromagnetic spectrum, the spectral wavelengths λ may form an additional dimension, as in $Z = f(x, y, t, \lambda)$; section 6.5 discusses the alternative of representing color bands as attributes.

Multiple time dimensions arise for instance when making forecasts for different times in the future t' at different times t , or when time is split into multiple dimensions (e.g. year, day-of-year, hour-of-day). The most general definition of a data cube is a functional mapping from n dimensions to p attributes:

$$\{Z_1, Z_2, \dots, Z_p\} = f(D_1, D_2, \dots, D_n)$$

Here, we will consider any dataset with one or more space dimensions and zero or more time dimensions as data cubes. That includes:

- simple features (section 3.1)
- time series for sets of features
- raster data
- multi-spectral raster data (images)
- time series of multi-spectral raster data (video)

6.2.1 Regular dimensions, GDAL’s geotransform

Data cubes are usually stored in multi-dimensional arrays, and the usual relationship between 1-based array index i and an associated regularly discretized dimension variable x is

$$x = o_x + (i - 1)d_x$$

with o_x the origin, and d_x the grid spacing for this dimension.

For more general cases like those in figure 1.6b-c, the relation between x and y and array indexes i and j is

$$x = o_x + (i - 1)d_x + (j - 1)a_1$$

$$y = o_y + (i - 1)a_2 + (j - 1)d_y$$

With two affine parameters a_1 and a_2 ; this is the so-called *geotransform* as used in GDAL. When $a_1 = a_2 = 0$, this reduces to the regular raster of figure 1.6a with square cells if $d_x = d_y$. For integer indexes, the coordinates are that of the starting *edge* of a grid cell, and the cell area (pixel) spans a range corresponding to index values ranging from i (inclusive) to $i+1$ (exclusive). For most common imagery formats, d_y is negative, indicating that image row index increases with decreasing y values (southward). To get the x - and y -coordinate of the grid cell *center* of the top left grid cell (in case of a negative d_y), we use $i = 1.5$ and $j = 1.5$.

For rectilinear rasters, a table that maps array index to dimension values is needed. NetCDF files for instance always stores all values of spatial dimension variables that correspond to the center of spatial grid cells, and may in addition store grid cell boundaries (which is needed to define rectilinear dimensions unambiguously).

For curvilinear rasters an array that maps every combination of i, j into x, y pairs is needed, or a parametric function that does this (e.g. an projection or its inverse). NetCDF files often provide both, when available.

6.2.2 Support along cube dimensions

Section 5.1 defined *spatial* support of an attribute variable as the size (length, area, volume) of a geometry a particular observation or prediction is associated with. The same notion applies to temporal support. Although time is rarely reported by explicit time periods having a start- and end-time, in many cases either the time stamp implies a period (e.g. ISO-8601 indications like “2021” for

a full year, “2021-01” for full month) or the time period is taken as the period from the time stamp of the current record up to but not including the time stamp of the next record.

An example is MODIS satellite imagery, where vegetation indexes (NDVI and EVI) are available as 16-day composites, meaning that over 16-day periods all available imagery is aggregated into a single image; such composites have temporal “block support”. Sentinel-2 or Landsat-8 data on the other hand are “snapshot” images and have temporal “point support”. When temporally aggregating data with temporal point support e.g. to monthly values one would select all images falling in the target time interval. When aggregating temporal block support imagery such as the MODIS 16-day composite, one might weigh images, e.g. according to the amount of overlap of the 16-day composite period and the target period, similar to area-weighted interpolation but over the time dimension.

6.3 Operations on data cubes

6.3.1 Slicing a cube: filter

Data cubes can be sliced into sub-cubes by fixing a dimension at a particular value. Figure 6.4 shows the sub-cubes obtained by doing with each of the dimensions. In this figure, the spatial filtering does not happen by fixing a single spatial dimension at a particular value, but by selecting a particular subregion, which is a more common operation. Fixing x or y would give a sub-cube along a transect of constant x or y , which can be used to show a Hovmöller diagram, where an attribute is plotted (colored) in the space of one space and one time dimension.

6.3.2 Applying functions to dimensions

A common analysis involves applying a function over one or more cube dimensions. Simple cases arise where a function such as `abs`, `sin` or `sqrt` is applied to all values in the cube, or when a function takes all values in the cube and returns a single scalar, e.g. when one computes the mean or maximum value over the entire cube. Other options include applying the function to selected dimensions, e.g. applying a temporal low-pass filter to every individual (pixel/band) time series as shown in figure 6.6, or applying a *spatial* low-pass filter to every spatial slice (i.e. every band/time combination), shown in figure 6.5.

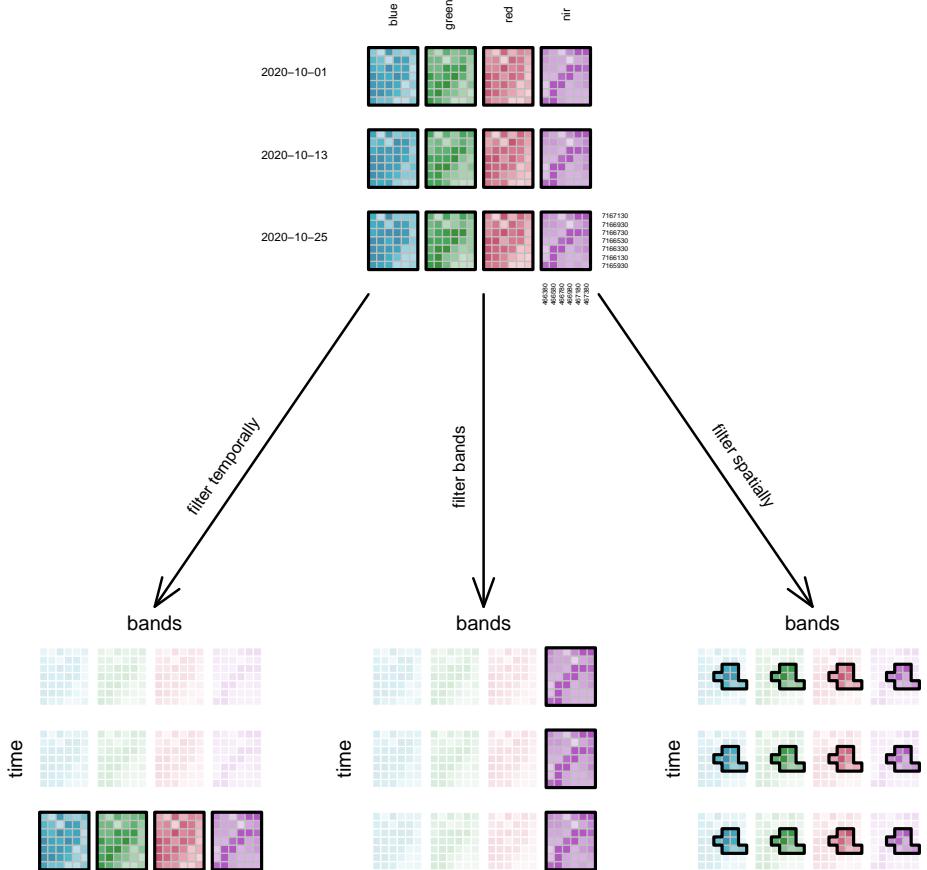


Figure 6.4: Data cube filtering by time, band or spatially

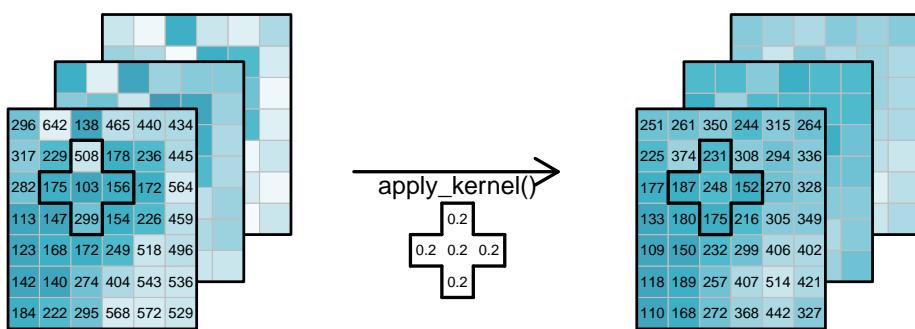


Figure 6.5: Low pass filtering of spatial slices

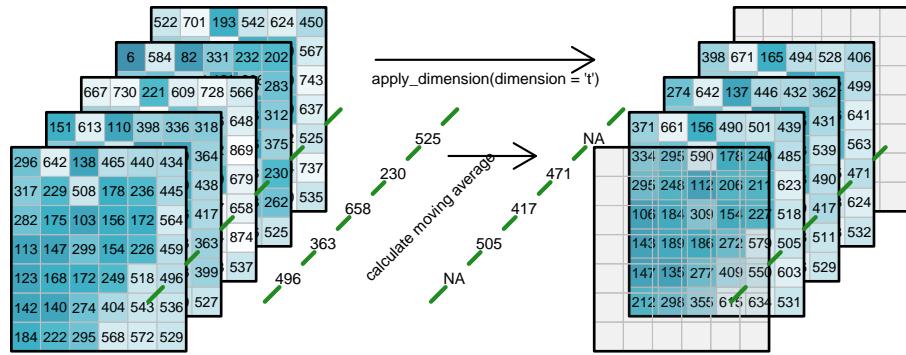


Figure 6.6: Low pass filtering of time series

6.3.3 Reducing dimensions

When applying function `mean` to an entire data cube, all dimensions vanish: the resulting “data cube” has dimensionality zero. We can also apply functions to a limited set of dimensions such that selected dimensions vanish, or are *reduced*. We already saw that filtering is a special case of this, but more in general we could for instance compute the maximum of every time series, the mean over every spatial slice, or a band index such as NDVI that summarizes different spectral values into a single new “band” with the index value. Figure 6.7 illustrates these options.

6.4 Aggregating raster to vector cubes

Figure 6.8 illustrates how a four-dimensional raster data cube can be aggregated to a three-dimensional *vector data cube*. Pixels in the raster are grouped by spatial intersection with a set of vector geometries, and each group is then reduced to a single value by an aggregation function such as `mean` or `max`. In the example, the *two* spatial dimensions *x* and *y* reduce to a single dimension, the one-dimensional sequence of feature geometries, with geometries that are defined in the space of *x* and *y*. Grouping geometries can also be `POINT` geometries, in which case the aggregation function is obsolete as single values at the `POINT` locations are *extracted*, e.g. by querying a `pixels` value or by interpolating from the nearest pixels.

Further examples of vector data cubes include air quality data, where we could have PM_{10} measurements over two dimensions:

- a sequence of monitoring stations, and
- a sequence of time intervals

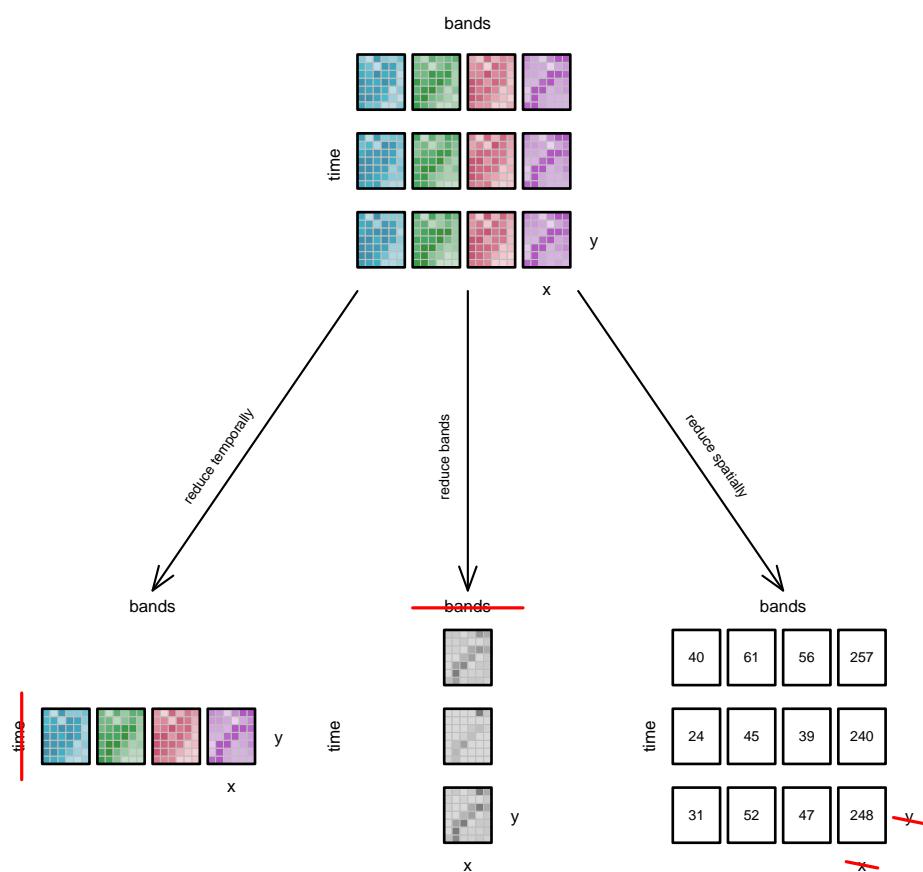


Figure 6.7: Reducing data cube dimensions

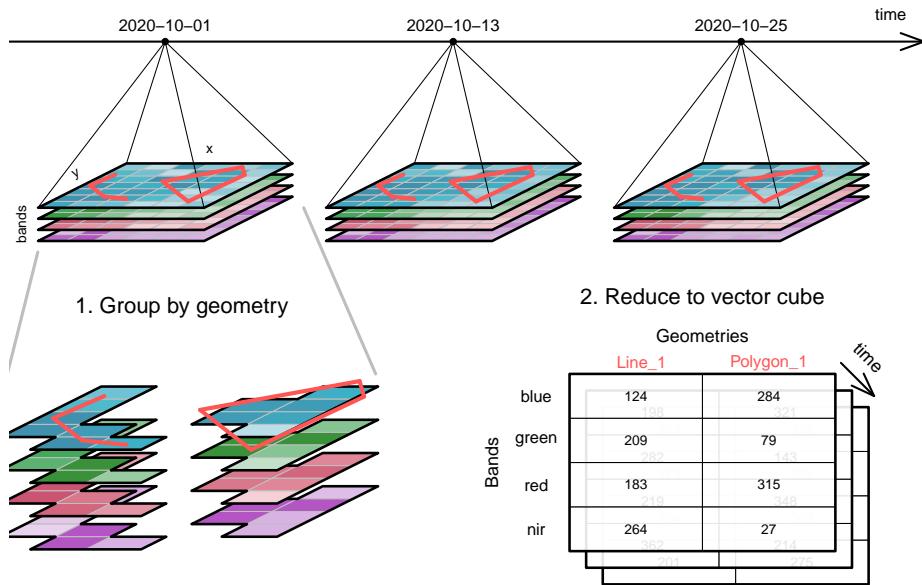


Figure 6.8: Aggregating a raster data cube to a vector data cube

or where we consider time series of demographic or epidemiological data, consisting of (population, disease) counts, with number of persons

- by region, for a sequence of n regions
- by age class, for m age classes, and
- by year, for p years

which forms an array with $nm p$ elements.

For spatial data science, support of vector and raster data cubes is extremely useful, because many variables are both spatially and temporally varying, and because we often want to either change dimensions or aggregate them out, but in a fully flexible manner and order. Examples of changing dimensions are:

- interpolating air quality measurements to values on a regular grid (raster; chapter 12)
- estimating density maps from points or lines, e.g. with the number of flights passing by per week within a range of 1 km (chapter 11)
- aggregating climate model predictions to summary indicators for administrative regions
- combining Earth observation data from different sensors, e.g. MODIS (250 m pixels, every 16 days) with Sentinel-2 (10 m, every 5 days)

Examples of aggregating one or more full dimensions are assessments of:

- which air quality monitoring stations indicate unhealthy conditions (time)
- which region has the highest increase in disease incidence (space, time)
- global warming (e.g. in degrees per year)

6.5 Switching dimension with attributes

When we accept that a dimension can also reflect an unordered, categorical variable, then one can easily swap a set of attributes for a single dimension, by replacing

$$\{Z_1, Z_2, \dots, Z_p\} = f(D_1, D_2, \dots, D_n)$$

with

$$Z = f(D_1, D_2, \dots, D_n, D_{n+1})$$

where D_{n+1} has cardinality p and has as labels (the names of) Z_1, Z_2, \dots, Z_p . Figure 6.9 shows a vector data cube for air quality stations where one cube dimension reflects air quality parameters. When the Z_i have incompatible measurement units, as in figure 6.9, one would have to take care when reducing the “parameter” dimension D_{n+1} : numeric functions like `mean` or `max` would be meaningless. Counting the number of variables that exceed their respective threshold values may however be meaningful.

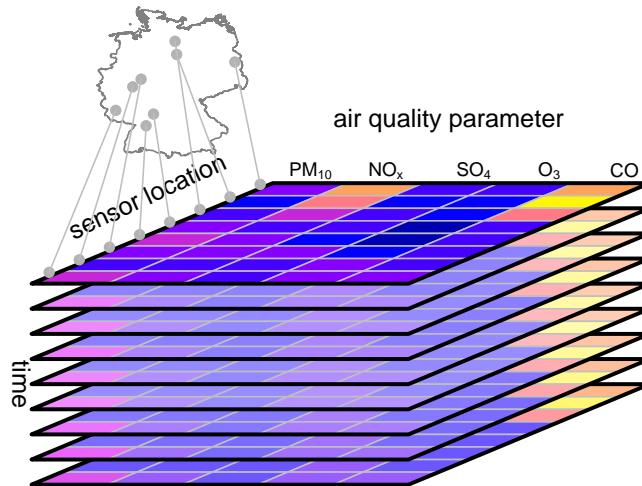


Figure 6.9: Vector data cube with air quality time series

Being able to swap dimensions to attributes flexibly and vice-versa leads to extremely flexible analysis possibilities, as e.g. shown by the array database SciDB (Brown, 2010).

6.6 Other dynamic spatial data

We have seen several dynamic raster and vector data examples that match the data cube structure well. Other data examples do less so: in particular spatiotemporal point patterns (chapter 11) and trajectories (movement data; for a recent review, see Joo et al. (2020)) are often more straightforward to not handle as a data cube. Spatiotemporal point patterns are the sets of spatiotemporal coordinates of events or objects: accidents, disease cases, traffic jams, lightning strikes, and so on. Trajectory data are time sequences of spatial locations of moving objects (persons, cars, satellites, animals). For such data, the primary information is in the coordinates, and shifting these to a limited set of regularly discretized grid cells covering the space may help some analysis, e.g. to quickly explore patterns in areas of higher densities, but the loss of the exact coordinates also hinders a number of analysis approaches involving distance, direction or speed calculations. Nevertheless, for such data often the first computational steps involves generation of data cube representations by aggregating to a time-fixed spatial and/or space-fixed temporal discretization.

Using sparse array representations of data cubes to represent point pattern or trajectory data, as e.g. offered by SciDB (Brown, 2010) or TileDB (Papadopoulos et al., 2016), may strongly limit the loss of coordinate accuracy by choosing dimensions that represent an extremely dense grid, and storing only those grid cell that contain data points. For trajectory data, such representations would need to add a grouping dimension to identify individuals, or individual sequences of consecutive movement observations.

6.7 Exercises

Use words to solve the following exercises, if needed or relevant use R code to illustrate the argument(s).

1. Why is it difficult to represent trajectories, sequences of (x, y, t) obtained by tracking moving objects, by data cubes as described in this chapter?
2. In a socio-economic vector data cube with variables population, life expectancy, and gross domestic product ordered by dimensions country and year, which variables have block support for the spatial dimension, and which have block support for the temporal dimension?

3. The Sentinel-2 satellites collect images in 12 spectral bands; list advantages and disadvantages to represent them as (i) different data cubes, (ii) a data cube with 12 attributes, one for each band, and (iii) a single attribute data cube with a spectral dimension.
4. Explain why a curvilinear raster as shown in figure 1.6 can be considered a special case of a data cube.
5. Explain how the following problems can be solved with data cube operations `filter`, `apply`, `reduce` and/or `aggregate`, and in which order. Also mention for each which function is applied, and what the dimensionality of the resulting data cube is (if any):
 - from hourly PM_{10} measurements for a set of air quality monitoring stations, compute per station the amount of days per year that the average daily PM_{10} value exceeds $50 \mu g/m^3$
 - for a sequence of aerial images of an oil spill, find the time at which the oil spill had its largest extent, and the corresponding extent
 - from a 10-year period with global daily sea surface temperature (SST) raster maps, find the area with the 10% largest and 10% smallest temporal trends in SST values.

Part II

R for Spatial Data Science

Chapter 7

Introduction to sf and stars

This chapter introduces R packages **sf** and **stars**. **sf** provides a table format for simple features, where feature geometries are carried in a list-column. R package **stars** was written to support raster and vector datacubes (Chapter 6), and has raster data stacks and feature time series as special cases. **sf** first appeared on CRAN in 2016, **stars** in 2018. Development of both packages received support from the R Consortium as well as strong community engagement. The packages were designed to work together.

All functions operating on **sf** or **stars** objects start with **st_**, making it easy to recognize them or to search for them when using command line completion.

7.1 Package sf

Intended to succeed and replace R packages **sp**, **rgeos** and the vector parts of **rgdal**, R Package **sf** (Pebesma, 2018) was developed to move spatial data analysis in R closer to standards-based approaches seen in the industry and open source projects, to build upon more modern versions of the open source geospatial software stack (figure 1.7), and to allow for integration of R spatial software with the tidyverse if desired.

To do so, R package **sf** provides simple features access (Herring et al., 2011), natively, to R. It provides an interface to several **tidyverse** packages, in particular to **ggplot2**, **dplyr** and **tidyr**. It can read and write data through GDAL, execute geometrical operations using GEOS (for projected coordinates) or s2geometry (for ellipsoidal coordinates), and carry out coordinate transformations or conversions using PROJ. External C++ libraries are interfaced using **Rcpp** (Eddelbuettel, 2013).

Package **sf** represents sets of simple features in **sf** objects, a sub-class of a **data.frame** or **tibble**. **sf** objects contain at least one *geometry list-column* of

class **sfc**, which for each element contains the geometry as an R object of class **sfg**. A geometry list-column acts as a variable in a **data.frame** or tibble, but has a more complex structure than e.g. numeric or character variables. Following the convention of PostGIS, all operations (functions, method) that operate on **sf** objects or related start with **st_**.

An **sf** object has the following meta-data:

- the name of the (active) geometry column, held in attribute **sf_column**
- for each non-geometry variable, the attribute-geometry relationships (section 5.1), held in attribute **agr**

An **sfc** geometry list-column has the following meta-data:

- the coordinate reference system held in attribute **crs**
- the bounding box held in attribute **bbox**
- the precision held in attribute **precision**
- the number of empty geometries held in attribute **n_empty**

These attributes may best be accessed or set by using functions like **st_bbox**, **st_crs**, **st_set_crs**, **st_agr**, **st_set_agr**, **st_precision**, and **st_set_precision**.

7.1.1 Creation

One could create an **sf** object from scratch e.g. by

```
library(sf)
p1 = st_point(c(7.35, 52.42))
p2 = st_point(c(7.22, 52.18))
p3 = st_point(c(7.44, 52.19))
sfc = st_sfc(list(p1, p2, p3), crs = 'OGC:CRS84')
st_sf(elev = c(33.2, 52.1, 81.2), marker = c("Id01", "Id02", "Id03"),
      geom = sfc)
# Simple feature collection with 3 features and 2 fields
# Geometry type: POINT
# Dimension: XY
# Bounding box: xmin: 7.22 ymin: 52.2 xmax: 7.44 ymax: 52.4
# Geodetic CRS: WGS 84
#   elev marker           geom
# 1 33.2   Id01 POINT (7.35 52.4)
# 2 52.1   Id02 POINT (7.22 52.2)
# 3 81.2   Id03 POINT (7.44 52.2)
```

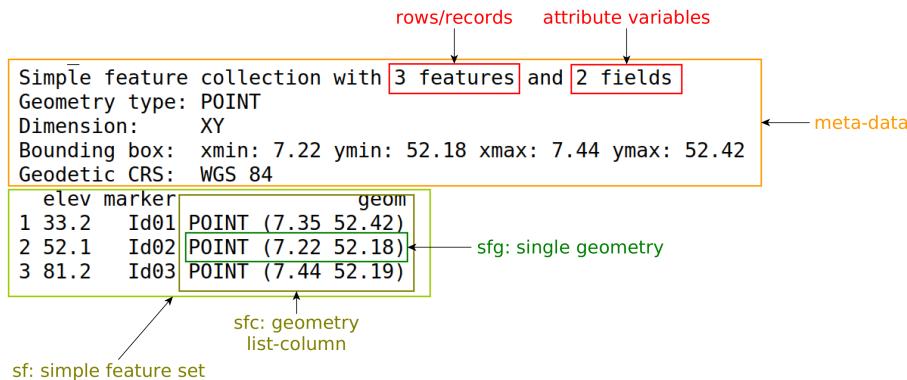
Figure 7.1: components of an `sf` object

Figure 7.1 gives an explanation of the components printed. Rather than creating objects from scratch, spatial data in R are typically read from an external source, which can be

- an external file,
- a request to a web service, or
- a dataset held in some form in another R package.

The next section introduces reading from files; section 8.1 discusses handling of datasets too large to fit into working memory.

7.1.2 Reading and writing

Reading datasets from an external “data source” (file, web service, or even string) is done using `st_read`:

```
library(sf)
(file = system.file("gpkg/nc.gpkg", package="sf"))
# [1] "/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/sf/gpkg/nc.gpkg"
nc = st_read(file)
# Reading layer `nc.gpkg' from data source
#   '/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/sf/gpkg/nc.gpkg'
#   using driver `GPKG'
# Simple feature collection with 100 features and 14 fields
# Geometry type: MULTIPOLYGON
# Dimension:      XY
# Bounding box:  xmin: -84.3 ymin: 33.9 xmax: -75.5 ymax: 36.6
# Geodetic CRS:  NAD27
```

Here, the file name and path `file` is read from the `sf` package, which has a different path on every machine, and hence is guaranteed to be present on every `sf` installation.

Command `st_read` has two arguments: the *data set name* (`dsn`) and the *layer*. In the example above, the *geopackage* file contains only a single layer that is being read. If it had contained multiple layers, then the first layer would have been read and a warning would have been emitted. The available layers of a data set can be queried by

```
st_layers(file)
# Driver: GPKG
# Available layers:
#   layer_name geometry_type features fields
# 1 nc.gpkg Multi Polygon      100      14
```

Simple feature objects can be written with `st_write`, as in

```
(file = tempfile(fileext = ".gpkg"))
# [1] "/tmp/Rtmp2fxnVb/filefaf227111bc79.gpkg"
st_write(nc, file, layer = "layer_nc")
# Writing layer `layer_nc' to data source
#   `/tmp/Rtmp2fxnVb/filefaf227111bc79.gpkg' using driver `GPKG'
# Writing 100 features with 14 fields and geometry type Multi Polygon.
```

where the file format (GPKG) is derived from the file name extension.

7.1.3 Subsetting

A very common operation is to subset objects; base R can use `[` for this. The rules that apply to `data.frame` objects also apply to `sf` objects, e.g. that records 2-5 and columns 3-7 are selected by

```
nc[2:5, 3:7]
```

but with a few additional features, in particular

- the `drop` argument is by default `FALSE` meaning that the geometry column is *always* selected, and an `sf` object is returned; when it is set to `TRUE` and the geometry column *not* selected, it is dropped and a `data.frame` is returned;
- selection with a spatial (`sf`, `sfc` or `sfg`) object as first argument leads to selection of the features that spatially *intersect* with that object (see next section); other predicates than *intersects* can be chosen by setting parameter `op` to a function such as `st_covers` or or any other binary predicate function listed in section 3.2.2

7.1.4 Binary predicates

Binary predicates like `st_intersects`, `st_covers` etc (section 3.2.2) take two sets of features or feature geometries and return for all pairs whether the predicate is TRUE or FALSE. For large sets this would potentially result in a huge matrix, typically filled mostly with FALSE values and for that reason a sparse representation is returned by default:

```
nc5 = nc[1:5, ]
nc7 = nc[1:7, ]
(i = st_intersects(nc5, nc7))
# Sparse geometry binary predicate list of length 5, where
# the predicate was `intersects'
# 1: 1, 2
# 2: 1, 2, 3
# 3: 2, 3
# 4: 4, 7
# 5: 5, 6
```



Figure 7.2: First seven North Carolina counties

Figure 7.2 shows how the intersections of the first five with the first seven counties can be understood. We can transform the sparse logical matrix into a dense matrix by

```
as.matrix(i)
#      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]
# [1,] TRUE TRUE FALSE FALSE FALSE FALSE FALSE
# [2,] TRUE TRUE TRUE FALSE FALSE FALSE FALSE
# [3,] FALSE TRUE TRUE FALSE FALSE FALSE FALSE
# [4,] FALSE FALSE FALSE TRUE FALSE FALSE TRUE
# [5,] FALSE FALSE FALSE FALSE TRUE TRUE FALSE
```

The number of countries that each of `nc5` intersects with is

```
lengths(i)
# [1] 2 3 2 2 2
```

and the other way around, the number of counties in `nc5` that intersect with each of the counties in `nc7` is

```
lengths(t(i))
# [1] 2 3 2 1 1 1 1
```

The object `i` is of class `sgbp` (sparse geometrical binary predicate), and is a list of integer vectors, with each element representing a row in the logical predicate matrix holding the column indices of the TRUE values for that row. It further holds some metadata like the predicate used, and the total number of columns. Methods available for `sgbp` objects include

```
methods(class = "sgbp")
# [1] as.data.frame as.matrix      dim           Ops
# [5] print          t
# see '?methods' for accessing help and source code
```

where the only `Ops` method available is `!`, the negation operation.

7.1.5 tidyverse

The tidyverse is a collection of data science packages that work together, described e.g. in (Wickham and Grolemund, 2017; Wickham et al., 2019). Package `sf` has tidyverse-style read and write functions, `read_sf` and `write_sf`, which return a tibble rather than a `data.frame`, do not print any output, and overwrite existing data by default.

Further tidyverse generics with methods for `sf` objects include `filter`, `select`, `group_by`, `ungroup`, `mutate`, `transmute`, `rowwise`, `rename`, `slice`, `summarise`, `distinct`, `gather`, `pivot_longer`, `spread`, `nest`, `unnest`, `unite`, `separate`, `separate_rows`, `sample_n`, and `sample_frac`. Most of these methods simply manage the metadata of `sf` objects, and make sure the geometry remains present. In case a user wants the geometry to be removed, one can use `st_set_geometry(nc, NULL)` or simply coerce to a `tibble` or `data.frame` before selecting:

```
library(tidyverse)
nc %>% as_tibble() %>% select(BIR74) %>% head(3)
# # A tibble: 3 x 1
#   BIR74
#   <dbl>
# 1 1091
# 2 487
# 3 3188
```

The `summarise` method for `sf` objects has two special arguments:

- `do_union` (default `TRUE`) determines whether grouped geometries are unioned on return, so that they form a valid geometry;
- `is_coverage` (default `FALSE`) in case the geometries grouped form a coverage (do not have overlaps), setting this to `TRUE` speeds up the unioning

The `distinct` method selects distinct records, where `st_equals` is used to evaluate distinctness of geometries.

`filter` can be used with the usual predicates; when one wants to use it with a spatial predicate, e.g. to select all counties less than 50 km away from Orange county, one could use

```
orange <- nc %>% filter(NAME == "Orange")
wd = st_is_within_distance(nc, orange, units::set_units(50, km))
o50 <- nc %>% filter(lengths(wd) > 0)
nrow(o50)
# [1] 17
```

Figure 7.3 shows the results of this analysis, and in addition a buffer around the county borders; note that this buffer serves for illustration, it was *not* used to select the counties.

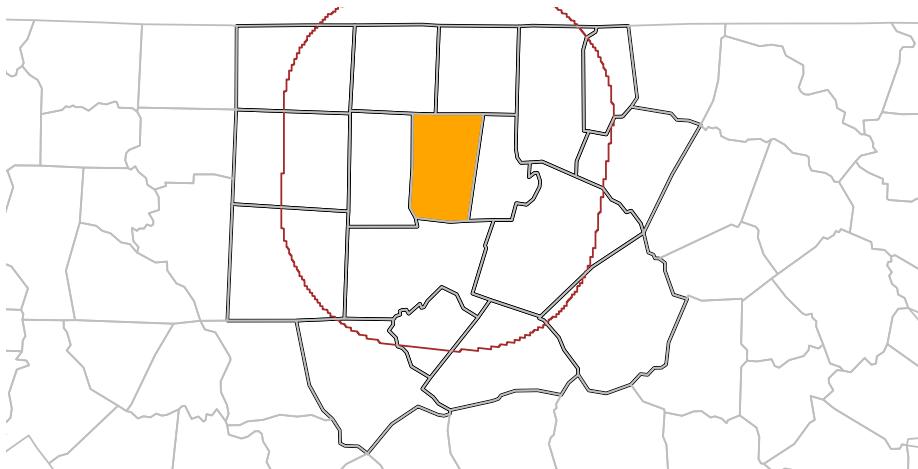


Figure 7.3: Orange county (orange), counties within a 50 km radius (black), a 50 km buffer around Orange county (brown), and remaining counties (grey)

7.2 Spatial joins

In regular (left, right or inner) joins, *joined* records from a pair of tables are reported when one or more selected attributes match (are identical) in both

tables. A spatial join is similar, but the criterion to join records is not equality of attributes but a spatial predicate. This leaves a wide variety of options in order to define *spatially* matching records, using binary predicates listed in section 3.2.2. The concepts of “left”, “right”, “inner” or “full” joins remain identical to the non-spatial join as the options for handling records that have no spatial match.

When using spatial joins, each record may have several matched records, yielding a large result table. A way to reduce this complexity may be to select from the matching records the one with the largest overlap with the target geometry. An example of this is shown (visually) in figure 7.4; this is done using `st_join` with argument `largest = TRUE`.

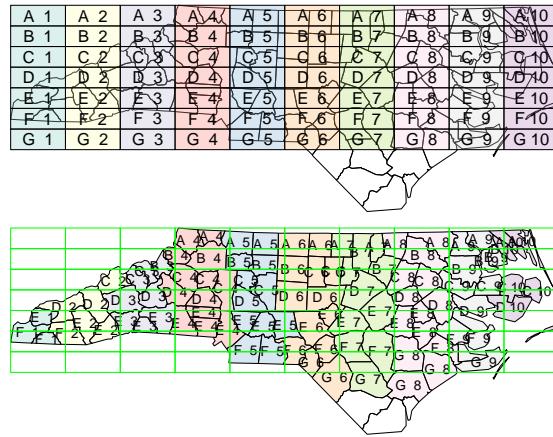


Figure 7.4: example of `st_join` with `largest = TRUE`: the label of the polygon in the top figure with the largest intersection with polygons in the bottom figure is assigned to the polygons of the bottom figure.

Another way to reduce the result set is to use `aggregate` after a join, all matching records, and union their geometries; see Section 5.4.

7.2.1 sampling, gridding, interpolating

Several convenience functions are available in package `sf`, some of which will be discussed here. Function `st_sample` generates a sample of points randomly sampled from target geometries, where target geometries can be point, line or polygon geometries. Sampling strategies can be (completely) random, regular or (with polygons) triangular. Chapter 11 explains how spatial sampling (or point pattern simulation) methods available in package `spatstat` are interfaced through `st_sample`.

Function `st_make_grid` creates a square, rectangular or hexagonal grid over a

region, or points with the grid centers or corners. It was used to create the rectangular grid in figure 7.4.

Function `st_interpolate_aw` “interpolates” area values to new areas, as explained in section 5.3, both for intensive and extensive variables.

7.3 Package stars

Athough package `sp` has always had limited support for raster data, over the last decade R package `raster` (Hijmans, 2021a) has clearly been dominant as the prime package for powerful, flexible and scalable raster analysis. The raster data model of package `raster` (and its successor, `terra` (Hijmans, 2021b)) is that of a 2D regular raster, or a set of raster layers (a “raster stack”). This aligns with the classical static “GIS world view”, where the world is modelled as a set of layers, each representing a different theme. A lot of data available today however is dynamic, and comes as time series of rasters or raster stacks. A raster stack does not meaningfully reflect this, requiring the user to keep a register of which layer represents what.

Also, the `raster` package, and its successer `terra` do an excellent job in scaling computations up to datasizes no larger than the local storage (the computer’s hard drives), and doing this fast. Recent datasets however, including satellite imagery, climate model or weather forecasting data, often no longer fit in local storage (chapter 8). Package `spacetime` (Pebesma, 2012) addresses the analysis of time series of vector geometries or raster grid cells, but does not extend to higher-dimensional arrays.

Here, we introduce package `stars` for analysing raster and vector data cubes. The package

- allows for representing dynamic (time varying) raster stacks,
- aims at being scalable, also beyond local disk size,
- provides a strong integration of raster functions in the GDAL library
- in addition to regular grids handles rotated, sheared, rectilinear and curvilinear rasters (figure 1.6),
- provides a tight integration with package `sf`,
- also handles array data with non-raster spatial dimensions, the *vector data cubes*, and
- follows the tidyverse design principles.

Vector data cubes include for instance time series for simple features, or spatial graph data such as potentially dynamic origin-destination matrices. The concept of spatial vector and raster data cubes was explained in chapter 6.

7.3.1 Reading and writing raster data

Raster data typically are read from a file. We use a dataset containing a section of Landsat 7 scene, with the 6 30m-resolution bands (bands 1-5 and 7) for a region covering the city of Olinda, Brazil. We can read the example GeoTIFF file holding a regular, non-rotated grid from the package `stars`:

```
tif = system.file("tif/L7_ETMs.tif", package = "stars")
library(stars)
(r = read_stars(tif))
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif    1      54     69 68.9     86   255
# dimension(s):
#       from to offset delta          refsys point values x/y
# x       1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y       1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band    1   6     NA     NA             NA     NA  NULL
```

where we see the offset, cellsize, coordinate reference system, and dimensions. The dimension table reports the following for each dimension:

- `from`: the starting index
- `to`: the ending index
- `offset`: the dimension value at the start (edge) of the first pixel
- `delta`: the cell size; negative `delta` values indicate that pixel index increases with decreasing dimension values
- `refsys`: the reference system
- `point`: boolean, indicating whether cell values have point support or cell support
- `values`: a list with values or labels associated with each of the dimension's values
- `x/y`: an indicator whether a dimension is associated with a spatial raster x- or y-axis

For regular, rotated or sheared grids or other regularly discretized dimensions (e.g. time), `offset` and `delta` are not `NA` and `values` is `NULL`; for other cases, `offset` and `delta` are `NA` and `values` contains one of:

- the sequence of values, or intervals, in case of a rectlinear spatial raster or irregular time dimension
- in case of a vector data cube, geometries associated with the spatial dimension

- in case of a curvilinear raster, the matrix with coordinate values for each raster cell
- in case of a discrete dimension, the band names or labels associated with the dimension values

The object `r` is of class `stars` and is a simple list of length one, holding a three-dimensional array:

```
length(r)
# [1] 1
class(r[[1]])
# [1] "array"
dim(r[[1]])
#   x   y band
# 349 352    6
```

and in addition holds an attribute with a dimensions table with all the metadata required to know what the array dimensions refer to, obtained by

```
st_dimensions(r)
```

We can get the spatial extent of the array by

```
st_bbox(r)
#   xmin   ymin   xmax   ymax
# 288776 9110729 298723 9120761
```

Raster data can be written to local disk using `write_stars`:

```
tf = tempfile(fileext = ".tif")
write_stars(r, tf)
```

where again the data format (in this case, GeoTIFF) is derived from the file extension. As for simple features, reading and writing uses the GDAL library; the list of available drivers for raster data is obtained by

```
st_drivers("raster")
```

7.3.2 Subsetting stars data cubes

Data cubes can be subsetted using `[`, or using tidyverse verbs. The first options, `[` uses for the arguments: attributes first, followed by dimension. This means that `r[1:2, 101:200, , 5:10]` selects from `r` attributes 1-2, index 101-200 for

dimension 1, and index 5-10 for dimension 3; omitting dimension 2 means that no subsetting takes place. For attributes, attributes names or logical vectors can be used. For dimensions, logical vectors are not supported; Selecting discontinuous ranges supported when it is a regular sequence. By default, `drop` is FALSE, when set to TRUE dimensions with a single value are dropped:

```
r[,1:100, seq(1, 250, 5), 4] %>% dim()
#   x   y band
# 100  50    1
r[,1:100, seq(1, 250, 5), 4, drop = TRUE] %>% dim()
#   x   y
# 100  50
```

For selecting particular ranges of dimension *values*, one can use `filter` (after loading `dplyr`):

```
library(dplyr, warn.conflicts = FALSE)
filter(r, x > 289000, x < 290000)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif      5      51     63 64.3      75 242
# dimension(s):
#   from to offset delta          refsys point values x/y
# x     1 35 289004 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y     1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band  1   6        1       1             NA    NA  NULL
```

which changes the offset of the *x* dimension. Particular cube slices can also be obtained with `slice`, e.g.

```
slice(r, band, 3)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif     21      49     63 64.4      77 255
# dimension(s):
#   from to offset delta          refsys point values x/y
# x     1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y     1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
```

which drops the singular dimension. `mutate` can be used on `stars` objects to add new arrays as functions of existing ones, `transmute` drops existing ones.

7.3.3 Cropping

Further subsetting can be done using spatial objects of class `sf`, `sfc` or `bbox`, e.g. when using the sample raster,

```
b = st_bbox(r) %>%
  st_as_sfc() %>%
  st_centroid() %>%
  st_buffer(units::set_units(500, m))

r[b]
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# L7_ETMs.tif    22      54     66 67.7    78.2   174 2184
# dimension(s):
#       from to offset delta      refsys point values x/y
# x      157 193 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      159 194 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band    1   6     NA     NA            NA     NA  NULL
```

selects the circular center region with a diameter of 500 metre, for the first band shown in figure 7.5,

```
# downsample set to c(0,0,1)
```

where we see that pixels outside the spatial object are assigned `NA` values. This object still has dimension indexes relative to the `offset` and `delta` values of `r`; we can reset these to a new `offset` with

```
r[b] %>% st_normalize() %>% st_dimensions()
#       from to offset delta      refsys point values x/y
# x      1 37 293222 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 36 9116258 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band    1   6     NA     NA            NA     NA  NULL
```

By default, the resulting raster is cropped to the extent of the selection object; an object with the same dimensions as the input object is obtained with

```
r[b, crop = FALSE]
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# L7_ETMs.tif    22      54     66 67.7    78.2   174 731280
# dimension(s):
```

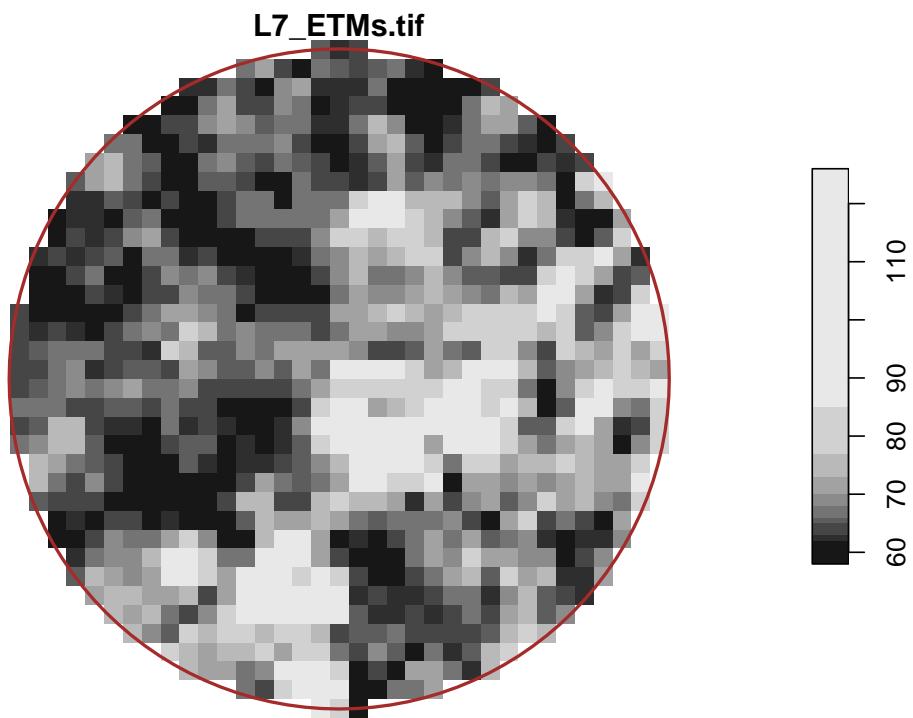


Figure 7.5: circular center region of the Landsat 7 scene (band 1)

```
#      from to offset delta      refsys point values x/y
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band  1   6     NA     NA           NA     NA  NULL
```

Cropping a `stars` object can alternatively be done directly with `st_crop`, as in

```
st_crop(r, b)
```

7.3.4 redimensioning stars objects

Package `stars` uses package `abind` (Plate and Heiberger, 2016) for a number of array manipulations. One of them is `aperm` which transposes an array by permuting it. A method for `stars` objects is provided, and

```
aperm(r, c(3, 1, 2))
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif    1      54     69 68.9      86 255
# dimension(s):
#      from to offset delta      refsys point values x/y
# band  1   6     NA     NA           NA     NA  NULL
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
```

permutes the order of dimensions of the resulting object.

Attributes and dimensions can be swapped, using `merge` and `split`:

```
(rs = split(r))
# stars object with 2 dimensions and 6 attributes
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# X1     47      67     78 79.1      89 255
# X2     32      55     66 67.6      79 255
# X3     21      49     63 64.4      77 255
# X4      9      52     63 59.2      75 255
# X5      1      63     89 83.2     112 255
# X6      1      32     60 60.0      88 255
# dimension(s):
#      from to offset delta      refsys point values x/y
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
```

```
merge(rs)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#   Min. 1st Qu. Median Mean 3rd Qu. Max.
# X      1      54      69 68.9      86 255
# dimension(s):
#           from to offset delta      refsys point
# x          1 349 288776 28.5 UTM Zone 25, S... FALSE
# y          1 352 9120761 -28.5 UTM Zone 25, S... FALSE
# attributes 1 6     NA    NA             NA    NA
#           values x/y
# x          NULL [x]
# y          NULL [y]
# attributes X1,...,X6
```

split distributes the `band` dimension over 6 attributes of a 2-dimensional array, `merge` reverses this operation. `st_redimension` can be used for more generic operations, such as splitting a single array dimension over two new dimensions:

```
st_redimension(r, c(349, 352, 3, 2))
# stars object with 4 dimensions and 1 attribute
# attribute(s):
#   Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif 1      54      69 68.9      86 255
# dimension(s):
#   from to offset delta      refsys point values
# X1      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL
# X2      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL
# X3      1  3     NA    NA             NA    NA  NULL
# X4      1  2     NA    NA             NA    NA  NULL
```

7.3.5 extracting point samples, aggregating

A very common use case for raster data cube analysis is the extraction of values at certain locations, or computing aggregations over certain geometries. `st_extract` extracts point values. We will do this for a few randomly sampled points over the bounding box of `r`:

```
pts = st_bbox(r) %>% st_as_sf() %>% st_sample(20)
(e = st_extract(r, pts))
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#   Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif 12     41.8     63     61    80.5 145
```

```
# dimension(s):
#           from to offset delta      refsys point
# geometry    1 20     NA     NA UTM Zone 25, S... TRUE
# band        1   6     NA     NA             NA     NA
#                                         values
# geometry POINT (293002 9115516),...,POINT (290941 9114128)
# band                                NULL
```

which results in a vector data cube with 4 points and 6 bands.

7.3.6 Predictive models

The typical model prediction workflow in R works as follows:

- use a `data.frame` with response and predictor variables (covariates)
- create a model object based on this `data.frame`
- call `predict` with this model object and the `data.frame` with target predictor variable values

Package `stars` provides a `predict` method for `stars` objects that essentially wraps the last step, by creating the `data.frame`, calling the `predict` method for that, and reconstructing a `stars` object with the predicted values.

We will illustrate this with a trivial two-class example mapping land from sea in the example Landsat data set, using the sample points extracted above, shown in figure 7.6.

```
# downsample set to c(0,0,1)
```

From this figure, we read “by eye” that the points 8, 14, 15, 18 and 19 are on water, the others on land. Using a linear discriminant (“maximum likelihood”) classifier, we find model predictions as shown in figure 7.7.

```
rs = split(r)
trn = st_extract(rs, pts)
trn$cls = rep("land", 20)
trn$cls[c(8, 14, 15, 18, 19)] = "water"
model = MASS::lda(cls ~ ., st_set_geometry(trn, NULL))
pr = predict(rs, model)
```

Here, we used the `MASS::` prefix to avoid loading `MASS`, as that would mask `select` from `dplyr`. Another way would be to load `MASS` and unload it later on with `detach()`.

We also see that the layer plotted in figure 7.7 is a `factor` variable, with class labels.

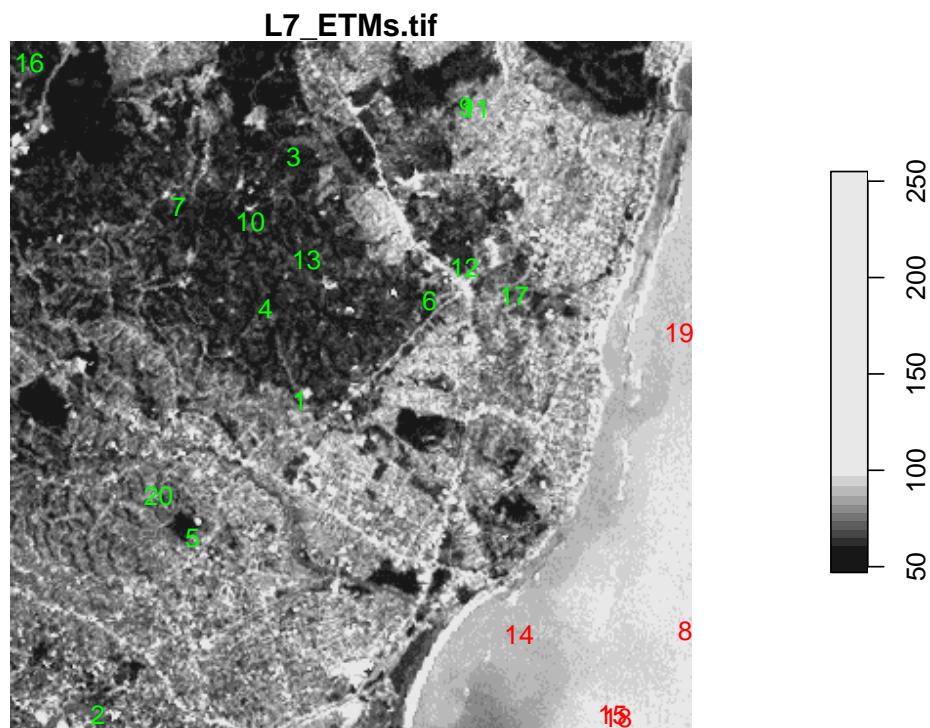


Figure 7.6: randomly chosen sample locations for training data; red: water, green: land

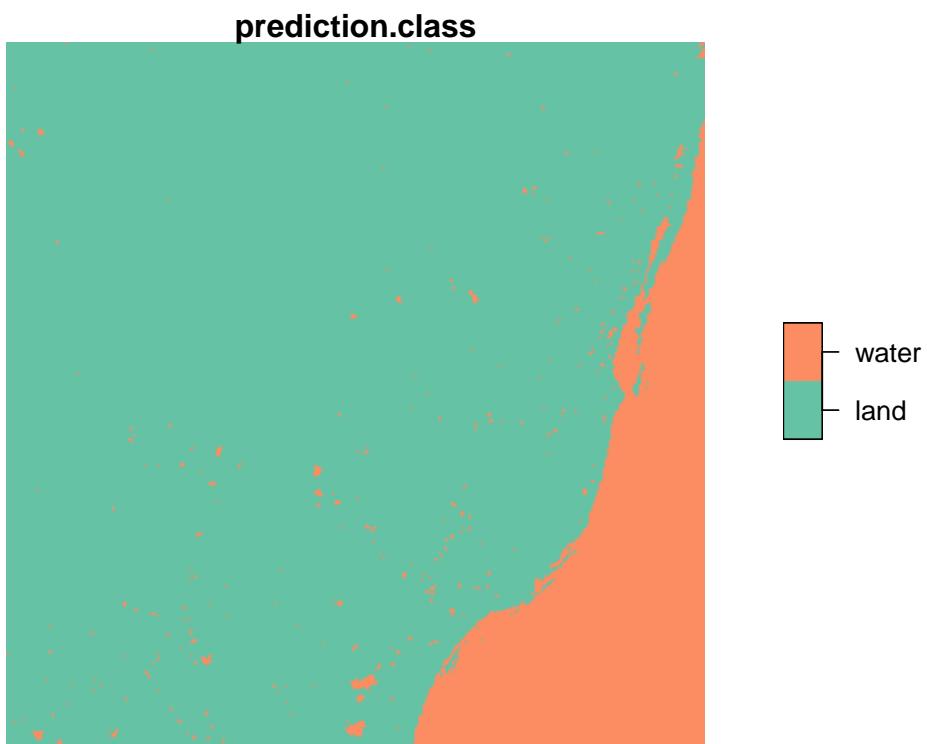


Figure 7.7: Linear discriminant classifier for land/water, based on training data of figure 7.6

7.3.7 GDAL utils

7.3.8 Plotting raster data

We can use the base `plot` method for `stars` objects, where the plot created with `plot(r)` is shown in figure 7.8.

```
# downsample set to c(2,2,1)
```

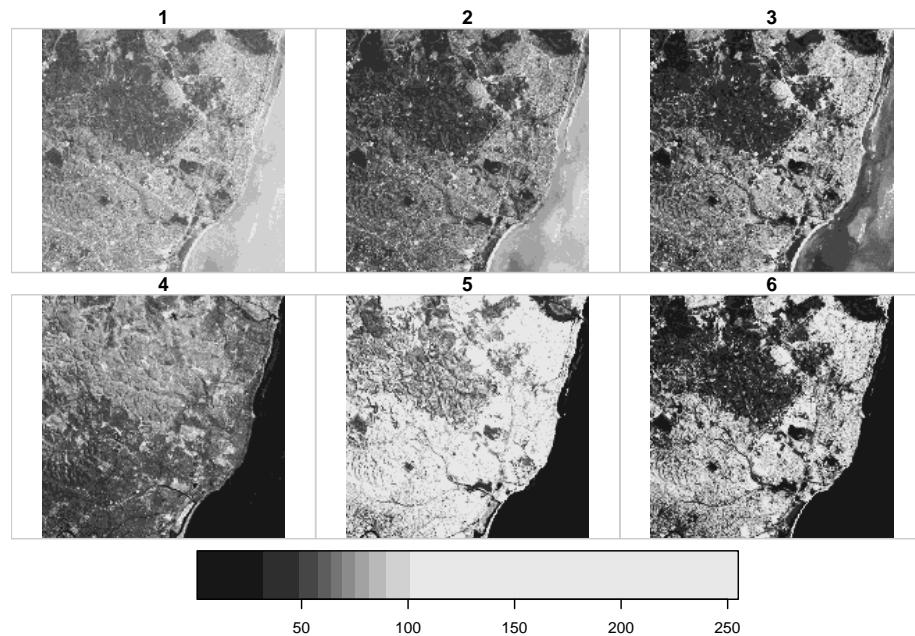


Figure 7.8: 6 30m Landsat bands downsampled to 90m for Olinda, Br.

is shown in figure 7.8. The default color scale uses grey tones, and stretches these such that color breaks correspond to data quantiles over all bands (“histogram equalization”). A more familiar view is the RGB or false color composite shown in figure 7.9.

```
# downsample set to c(2,2,1)
# downsample set to c(2,2,1)
```

Further details and options are given in Chapter 9.

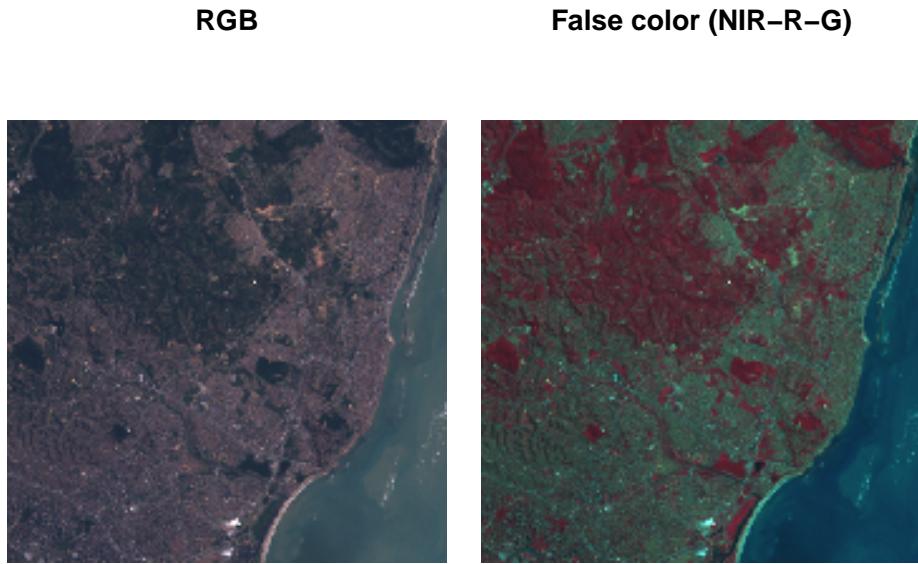


Figure 7.9: Two color composites

7.3.9 Analysing raster data

Element-wise mathematical functions (section 6.3.2) on `stars` objects are just passed on to the arrays. This means that we can call functions and create expressions:

```
log(r)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif      0    3.99   4.23 4.12    4.45 5.54
# dimension(s):
#       from to offset delta          refsys point values x/y
# x       1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y       1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band    1   6     NA     NA             NA     NA  NULL
r + 2 * log(r)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif      1      62   77.5 77.1    94.9 266
# dimension(s):
#       from to offset delta          refsys point values x/y
# x       1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
```

```
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band   1   6     NA     NA             NA     NA  NULL
```

or even mask out certain values:

```
r2 = r
r2[r < 50] = NA
r2
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.  NA's
# L7_ETMs.tif    50      64     75   79      90  255 149170
# dimension(s):
#       from to offset delta      refsys point values x/y
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band   1   6     NA     NA             NA     NA  NULL
```

or un-mask areas:

```
r2[is.na(r2)] = 0
r2
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif    0      54     69   63      86  255
# dimension(s):
#       from to offset delta      refsys point values x/y
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
# band   1   6     NA     NA             NA     NA  NULL
```

Dimension-wise, we can apply functions to selected array dimensions (section 6.3.3) of stars objects similar to how `apply` does this to arrays. For instance, we can compute for each pixel the mean of the 6 band values by

```
st_apply(r, c("x", "y"), mean)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# mean  25.5    53.3   68.3 68.9      82  255
# dimension(s):
#       from to offset delta      refsys point values x/y
# x      1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y      1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
```

A more meaningful function would e.g. compute the NDVI (normalized differenced vegetation index):

```
ndvi = function(b1, b2, b3, b4, b5, b6) (b4 - b3)/(b4 + b3)
st_apply(r, c("x", "y"), ndvi)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#       Min. 1st Qu. Median Mean 3rd Qu. Max.
# ndvi -0.753 -0.203 -0.0687 -0.0643 0.187 0.587
# dimension(s):
#   from to offset delta           refsys point values x/y
# x     1 349 288776 28.5 UTM Zone 25, S... FALSE  NULL [x]
# y     1 352 9120761 -28.5 UTM Zone 25, S... FALSE  NULL [y]
```

Alternatively, one could have defined

```
ndvi2 = function(x) (x[4]-x[3])/(x[4]+x[3])
```

which is more convenient if the number of bands is large, but which is also much slower than `ndvi` as it needs to be called for every pixel whereas `ndvi` can be called once for all pixels, or for large chunks of pixels. The mean for each band over the whole image is computed by

```
as.data.frame(st_apply(r, c("band"), mean))
#   band mean
# 1    1 79.1
# 2    2 67.6
# 3    3 64.4
# 4    4 59.2
# 5    5 83.2
# 6    6 60.0
```

the result of which is small enough to be printed here as a `data.frame`. In these two examples, entire dimensions disappear. Sometimes, this does not happen (section 6.3.2); we can for instance compute the three quartiles for each band

```
st_apply(r, c("band"), quantile, c(.25, .5, .75))
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#       Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif 32 60.8 66.5 69.8 78.8 112
# dimension(s):
#   from to offset delta           refsys point      values
# quantile    1 3 NA NA NA 25%, 50%, 75%
# band        1 6 NA NA NA  NULL
```

and see that this *creates* a new dimension, `quantile`, with three values. Alternatively, the three quantiles over the 6 bands for each pixel are obtained by

```
st_apply(r, c("x", "y"), quantile, c(.25, .5, .75))
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif    4      55   69.2 67.2    81.2 255
# dimension(s):
#           from to offset delta          refsys point
# quantile  1   3     NA   NA          NA   NA
# x         1 349 288776 28.5 UTM Zone 25, S... FALSE
# y         1 352 9120761 -28.5 UTM Zone 25, S... FALSE
#           values x/y
# quantile 25%, 50%, 75%
# x           NULL [x]
# y           NULL [y]
```

7.3.10 Curvilinear rasters

There are several reasons why non-regular rasters occur (figure 1.6). For one, when the data is Earth-bound, a regular raster does not fit the Earth's surface, which is curved. Other reasons are:

- when we convert or transform a regular raster data into another coordinate reference system, it will become curvilinear unless we resample (warp; section 7.7); warping always goes at the cost of some loss of data and is not reversible.
- observation may lead to irregular rasters; e.g. for satellite swaths, we may have a regular raster in the direction of the satellite (not aligned with *x* or *y*), and rectilinear in the direction perpendicular to that (e.g. if the sensor discretizes the viewing *angle* in equal sections)

7.4 Vector data cube examples

7.4.1 Example: aggregating air quality time series

Air quality data from package `spacetime` were obtained from the airBase European air quality data base. Daily average PM₁₀ values were downloaded for rural background stations in Germany, 1998-2009.

We can create a `stars` object from the `air` matrix, the `dates` Date vector and the `stations` `SpatialPoints` objects by

```

library(spacetime)
data(air) # this loads several datasets in .GlobalEnv
dim(air)
# space time
# 70 4383
d = st_dimensions(station = st_as_sfc(stations), time = dates)
(aq = st_as_stars(list(PM10 = air), dimensions = d))
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# PM10      0    9.92   14.8 17.7     22   274 157659
# dimension(s):
#      from to offset delta      refsys point
# station  1  70    NA    NA +proj=longlat ... TRUE
# time      1 4383 1998-01-01 1 days           Date FALSE
#                                         values
# station POINT (9.59 53.7),...,POINT (9.45 49.2)
# time             NULL

```

We can see from figure 7.10 that the time series are quite long, but also have large missing value gaps. Figure 7.11 shows the spatial distribution measurement stations and mean PM_{10} values.

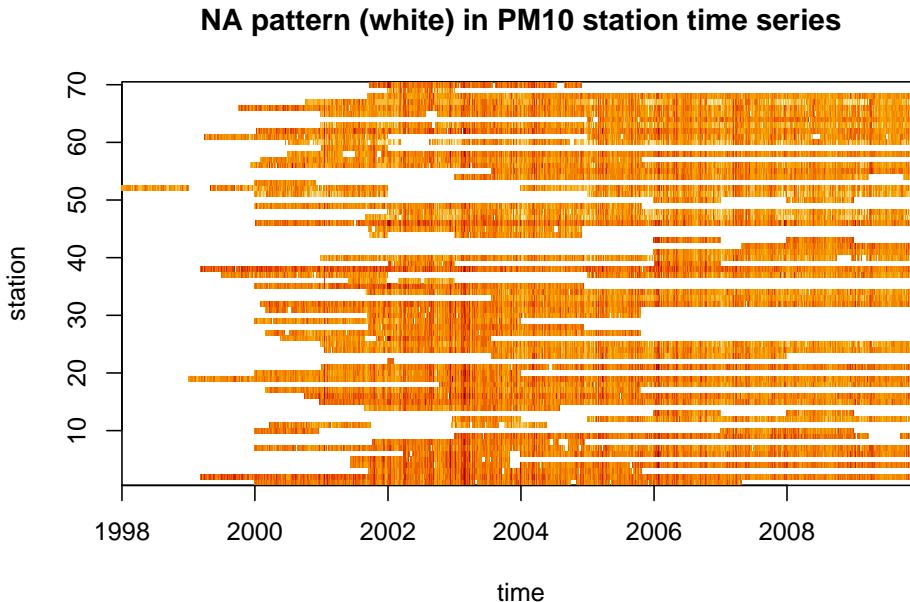


Figure 7.10: space-time diagram of PM_{10} measurements by time and station

```
# Warning in sp::proj4string(obj): CRS object has comment, which is
# lost in output
```

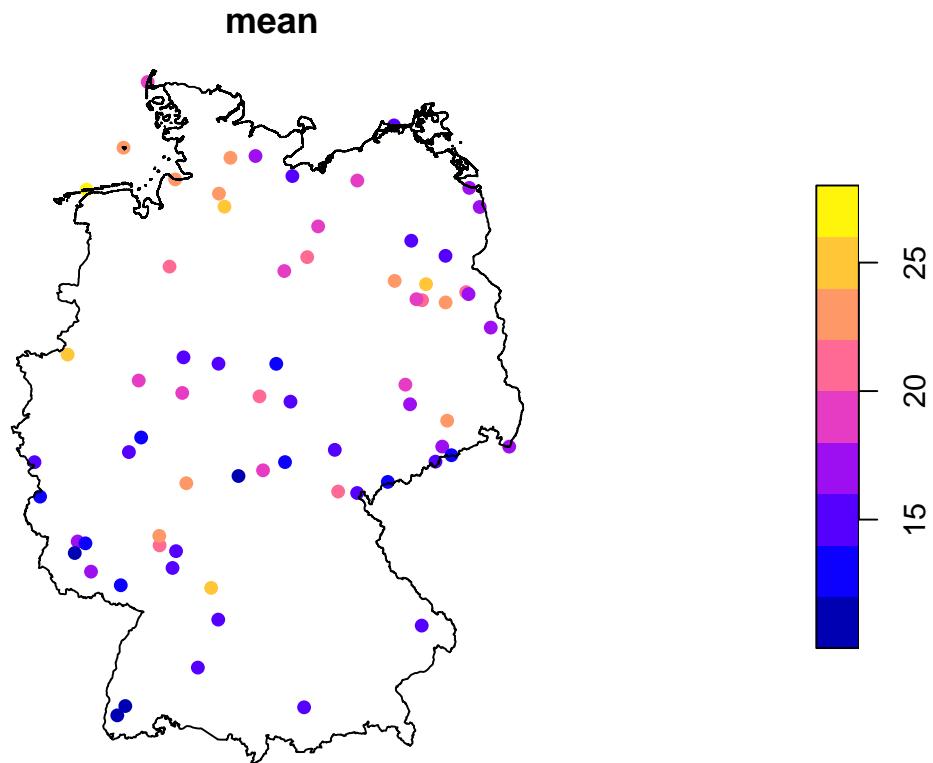


Figure 7.11: locations of PM₁₀ measurement stations, showing mean values

We can aggregate these station time series to area means, mostly as a simple exercise. For this, we use the `aggregate` method for `stars` objects

```
(a = aggregate(aq, st_as_sf(DE_NUTS1), mean, na.rm = TRUE))
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# PM10  1.08    10.9   15.3 17.9    21.8  172 25679
# dimension(s):
#       from     to     offset      delta          refsys point
# geometry 1     16      NA      NA +proj=longlat ... FALSE
# time      1 4383 1998-01-01 1 days           Date FALSE
#
# geometry MULTIPOLYGON (((9.65 49.8, ..., ..., MULTIPOLYGON (((10.8 51.6, ...
# time                                NULL
```

and we can now for instance show the maps for six arbitrarily chosen days (figure 7.12), using

```
library(tidyverse)
a %>% filter(time >= "2008-01-01", time < "2008-01-07") %>%
  plot(key.pos = 4)
```

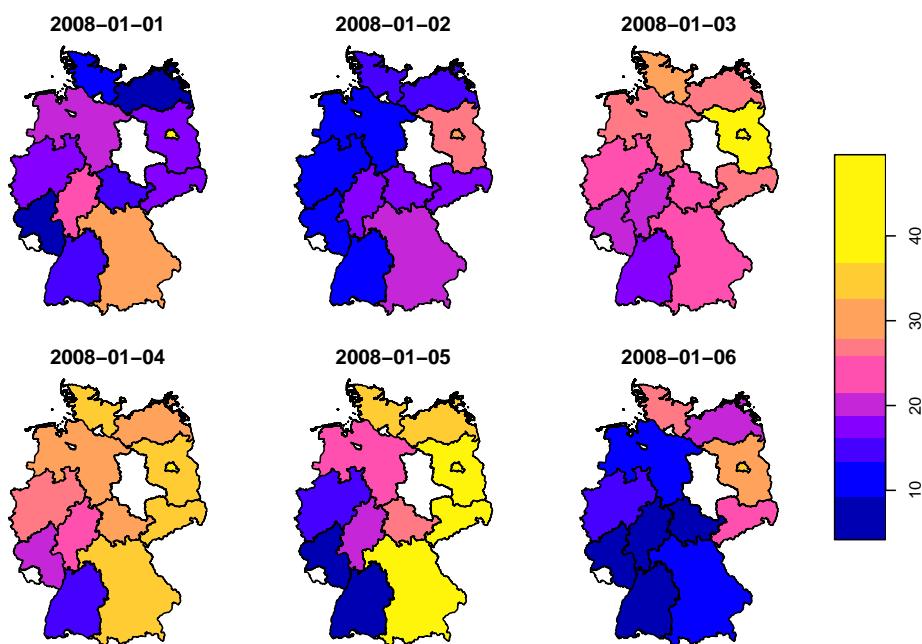


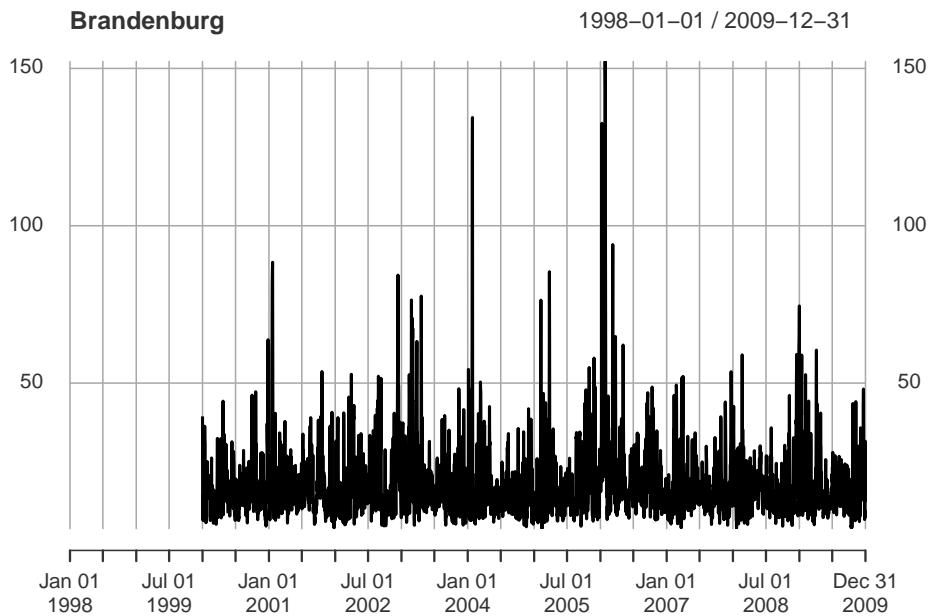
Figure 7.12: areal mean PM₁₀ values, for six days

or create a time series plot of mean values for a single state (figure 7.13) by

```
suppressPackageStartupMessages(library(xts))
plot(as.xts(a)[,4], main = DE_NUTS1$NAME_1[4])
```

7.4.2 Example: Bristol origin-destination datacube

The data used for this example come from (Lovelace et al., 2019), and concern origin-destination (OD) counts: the number of persons going from region A to region B, by transportation mode. We have feature geometries for the 102 origin and destination regions, shown in figure 7.14.

Figure 7.13: areal mean PM₁₀ values, for six days

```
library(spDataLarge)
plot(st_geometry(bristol_zones), axes = TRUE, graticule = TRUE)
plot(st_geometry(bristol_zones)[33], col = 'red', add = TRUE)
```

and the OD counts come in a table with OD pairs as records, and transportation mode as variables:

```
head(bristol_od)
# # A tibble: 6 x 7
#   o          d      all bicycle foot car_driver train
#   <chr>     <chr>  <dbl>    <dbl> <dbl>    <dbl>   <dbl>
# 1 E02002985 E02002985  209      5   127      59     0
# 2 E02002985 E02002987  121      7   35       62     0
# 3 E02002985 E02003036   32      2     1       10     1
# 4 E02002985 E02003043  141      1     2       56    17
# 5 E02002985 E02003049   56      2     4       36     0
# 6 E02002985 E02003054   42      4     0       21     0
```

We see that many combinations of origin and destination are implicit zeroes, otherwise these two numbers would have been similar:

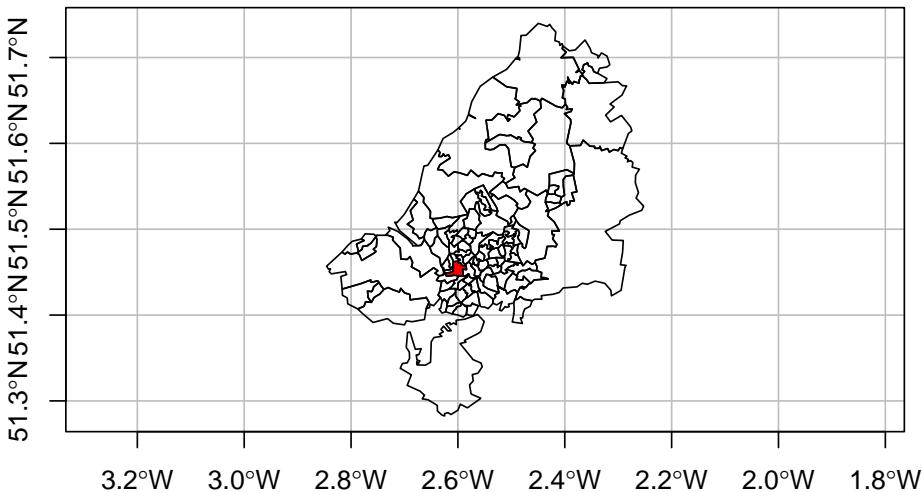


Figure 7.14: Origin destination data zones for Bristol, UK, with zone 33 (E02003043) colored red

```
nrow(bristol_zones)^2 # all combinations
# [1] 10404
nrow(bristol_od) # non-zero combinations
# [1] 2910
```

We will form a three-dimensional vector datacube with origin, destination and transportation mode as dimensions. For this, we first “tidy” the `bristol_od` table to have origin (o), destination (d), transportation mode (mode), and count (n) as variables, using `pivot_longer`:

```
# create O-D-mode array:
bristol_tidy <- bristol_od %>%
  select(-all) %>%
  pivot_longer(3:6, names_to = "mode", values_to = "n")
head(bristol_tidy)
# # A tibble: 6 x 4
#   o          d      mode     n
#   <chr>    <chr>  <chr>    <dbl>
# 1 E02002985 E02002985 bicycle     5
# 2 E02002985 E02002985 foot       127
# 3 E02002985 E02002985 car_driver  59
# 4 E02002985 E02002985 train      0
# 5 E02002985 E02002987 bicycle     7
# 6 E02002985 E02002987 foot       35
```

Next, we form the three-dimensional array `a`, filled with zeroes:

```
od = bristol_tidy %>% pull("o") %>% unique()
nod = length(od)
mode = bristol_tidy %>% pull("mode") %>% unique()
nmode = length(mode)
a = array(0L, c(nod, nod, nmode),
          dimnames = list(o = od, d = od, mode = mode))
dim(a)
# [1] 102 102 4
```

We see that the dimensions are named with the zone names (`o`, `d`) and the transportation mode name (`mode`). Every row of `bristol_tidy` denotes an array entry, and we can use this to fill the non-zero entries of the `bristol_tidy` table with their appropriate value (`n`):

```
a[as.matrix(bristol_tidy[c("o", "d", "mode")])] = bristol_tidy$n
```

To be sure that there is not an order mismatch between the zones in `bristol_zones` and the zone names in `bristol_tidy`, we can get the right set of zones by:

```
order = match(od, bristol_zones$geo_code) # it happens this equals 1:102
zones = st_geometry(bristol_zones)[order]
```

(It happens that the order is already correct, but it is good practice to not assume this).

Next, with zones and modes we can create a stars dimensions object:

```
library(stars)
(d = st_dimensions(o = zones, d = zones, mode = mode))
#      from to offset delta refsys point
# o      1 102     NA     NA WGS 84 FALSE
# d      1 102     NA     NA WGS 84 FALSE
# mode   1  4     NA     NA     NA FALSE
#
# o      MULTIPOLYGON (((-2.51 51.4,...,....,MULTIPOLYGON (((-2.55 51.5,....
# d      MULTIPOLYGON (((-2.51 51.4,...,....,MULTIPOLYGON (((-2.55 51.5,....
# mode                                         values
# mode                                         bicycle,...,train
```

and finally build or stars object from `a` and `d`:

```

(odm = st_as_stars(list(N = a), dimensions = d))
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#       Min. 1st Qu. Median Mean 3rd Qu. Max.
# N        0        0        0  4.8        0 1296
# dimension(s):
#       from    to offset delta refsys point
# o        1 102     NA     NA WGS 84 FALSE
# d        1 102     NA     NA WGS 84 FALSE
# mode     1    4     NA     NA      NA FALSE
#
#                                         values
# o    MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
# d    MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
# mode                                bicycle, ..., train

```

We can take a single slice through from this three-dimensional array, e.g. for zone 33 (figure 7.14), by `odm[, , 33]`, and plot it with

```

plot(odm[, , 33] + 1, logz = TRUE)
# Warning in st_as_sf.stars(x): working on the first sfc dimension
# only
# Warning in st_bbox.dimensions(st_dimensions(obj), ...): returning
# the bounding box of the first geometry dimension

# Warning in st_bbox.dimensions(st_dimensions(obj), ...): returning
# the bounding box of the first geometry dimension

```

the result of which is shown in figure 7.15. Subsetting this way, we take all attributes (there is only one: N) since the first argument is empty, we take all origin regions (second argument empty), we take destination zone 33 (third argument), and all transportation modes (fourth argument empty, or missing).

We plotted this particular zone because it has the largest number of travelers as its destination. We can find this out by summing all origins and travel modes by destination:

```
d = st_apply(odm, 2, sum)
which.max(d[[1]])
# [1] 33
```

Other aggregations we can carry out include: total transportation by OD (102 x 102):

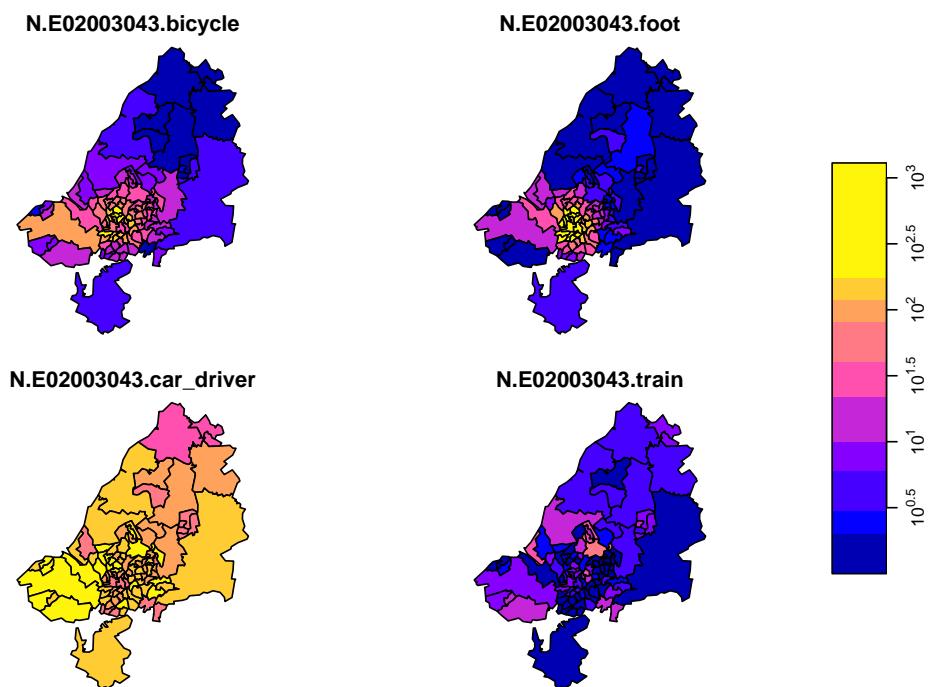


Figure 7.15: OD matrix sliced for destination zone 33, by transportation mode

```
st_apply(odm, 1:2, sum)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#      Min. 1st Qu. Median Mean 3rd Qu. Max.
# sum      0       0       0 19.2      19 1434
# dimension(s):
#   from to offset delta refsys point
# o     1 102      NA      NA WGS 84 FALSE
# d     1 102      NA      NA WGS 84 FALSE
#
#                                                 values
# o MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
# d MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
```

Origin totals, by mode:

```
st_apply(odm, c(1,3), sum)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#      Min. 1st Qu. Median Mean 3rd Qu. Max.
# sum      1    57.5    214   490    771 2903
# dimension(s):
#   from to offset delta refsys point
# o     1 102      NA      NA WGS 84 FALSE
# mode   1  4      NA      NA      NA FALSE
#
#                                                 values
# o      MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
# mode                                bicycle, ..., train
```

Destination totals, by mode:

```
st_apply(odm, c(2,3), sum)
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#      Min. 1st Qu. Median Mean 3rd Qu. Max.
# sum      0      13    104   490    408 12948
# dimension(s):
#   from to offset delta refsys point
# d     1 102      NA      NA WGS 84 FALSE
# mode   1  4      NA      NA      NA FALSE
#
#                                                 values
# d      MULTIPOLYGON (((-2.51 51.4, ..., ..., MULTIPOLYGON (((-2.55 51.5, ...
# mode                                bicycle, ..., train
```

Origin totals, summed over modes:

```
o = st_apply(odm, 1, sum)
```

Destination totals, summed over modes (we had this):

```
d = st_apply(odm, 2, sum)
```

We plot o and d together after joining them by

```
x = (c(o, d, along = list(od = c("origin", "destination"))))
plot(x, logz = TRUE)
```

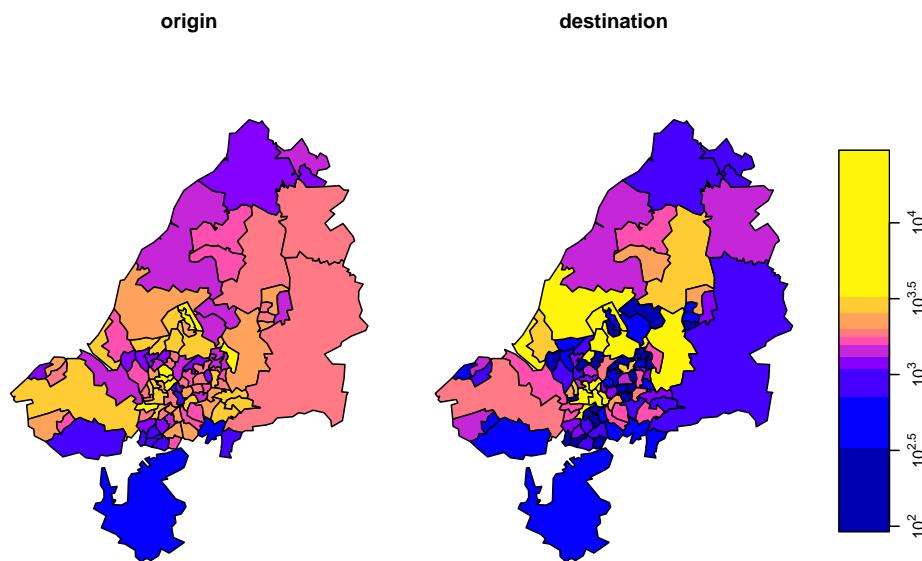


Figure 7.16: total commutes, summed by origin (left) or destination (right).

the result of which is shown in figure 7.16.

There is something to say for the argument that such maps give the wrong message, as both amount (color) and polygon size give an impression of amount. To take out the amount in the count, we can compute densities (count / km²), by

```
library(units)
# udunits database from /usr/share/xml/udunits/udunits2.xml
a = set_units(st_area(st_as_sf(o)), km^2)
o$sum_km = o$sum / a
d$sum_km = d$sum / a
od = c(o["sum_km"], d["sum_km"], along = list(od = c("origin", "destination")))
plot(od, logz = TRUE)
```

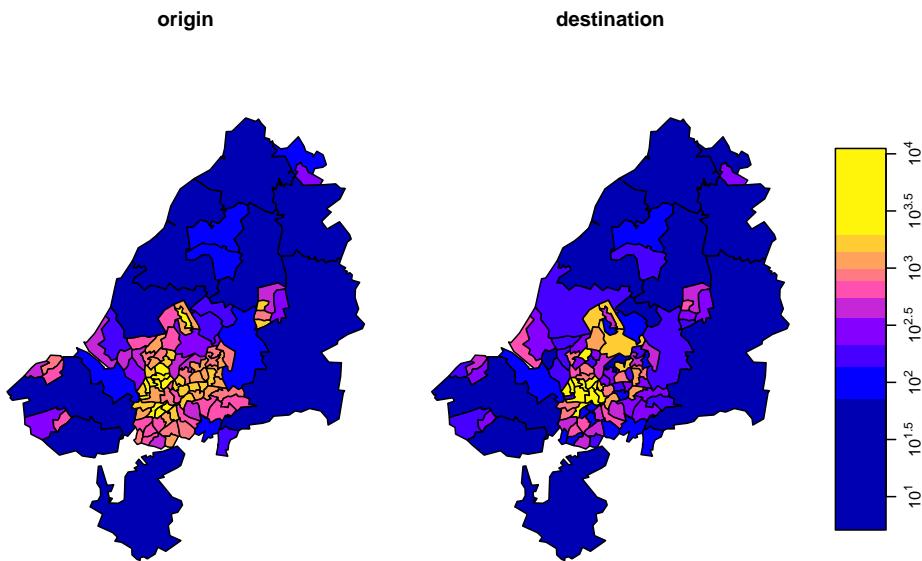


Figure 7.17: total commutes per square km, by area of origin (left) or destination (right)

shown in figure 7.17. Another way to normalize these totals would be to divide them by population size.

7.4.3 Tidy array data

The *tidy data* paper (Wickham, 2014b) may suggest that such array data should be processed not as an array, but in a long table where each row holds (region, class, year, value), and it is always good to be able to do this. For primary handling and storage however, this is often not an option, because

- a lot of array data are collected or generated as array data, e.g. by imagery or other sensory devices, or e.g. by climate models
- it is easier to derive the long table form from the array than vice versa
- the long table form requires much more memory, since the space occupied by dimension values is $O(nmp)$, rather than $O(n + m + p)$
- when missing-valued cells are dropped, the long table form loses the implicit indexing of the array form

To put this argument to the extreme, consider for instance that all image, video and sound data are stored in array form; few people would make a real case for storing them in a long table form instead. Nevertheless, R packages like `tsibble` take this approach, and have to deal with ambiguous ordering of multiple records

with identical time steps for different spatial features and index them, which is solved for both *automatically* by using the array form – at the cost of using dense arrays, in package **stars**.

Package **stars** tries to follow the tidy manifesto to handle array sets, and has particularly developed support for the case where one or more of the dimensions refer to space, and/or time.

7.5 raster-to-vector, vector-to-raster

Section 1.3 already showed some examples of raster-to-vector and vector-to-raster conversions, this section will add some code details and examples.

7.5.1 vector-to-raster

`st_as_stars` is meant as a method to transform objects into **stars** objects. However, not all stars objects are **raster** objects, and the method for **sf** objects creates a vector data cube with the geometry as its spatial (vector) dimension, and attributes as attributes. When given a feature *geometry* (**sfc**) object, `st_as_stars` will rasterize it, as shown in section 7.7, and in figure 7.18.

```
(file = system.file("gpkg/nc.gpkg", package="sf"))
# [1] "/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/sf/gpkg/nc.gpkg"
read_sf(file) %>%
  st_geometry() %>%
  st_as_stars() %>%
  plot()
```

Here, `st_as_stars` can be parameterized to control cell size, number of cells, and/or extent. The cell values returned are 0 for cells with center point outside the geometry and 1 for cell with center point inside or on the border of the geometry. Rasterizing existing features is done using `st_rasterize`, as also shown in figure 1.5:

```
library(dplyr)
read_sf(file) %>%
  mutate(name = as.factor(NAME)) %>%
  select(SID74, SID79, name) %>%
  st_rasterize()
# stars object with 2 dimensions and 3 attributes
# attribute(s):
#   SID74           SID79           name
#   Min.    : 0       Min.    : 0       Sampson : 655
```

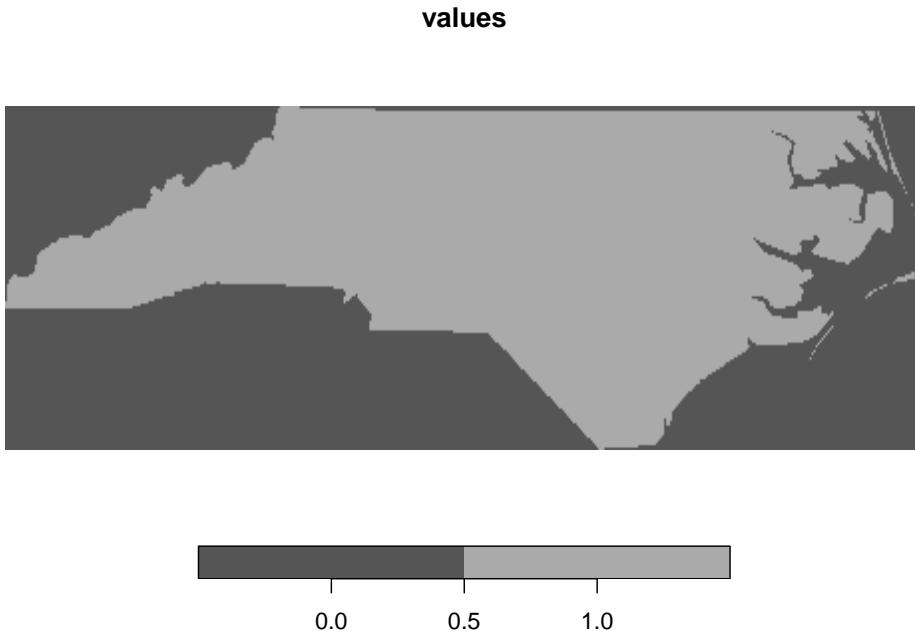


Figure 7.18: rasterizing vector geometry using `st_as_stars`

```
# 1st Qu.: 3      1st Qu.: 3      Columbus: 648
# Median : 5      Median : 6      Robeson : 648
# Mean   : 8      Mean   :10     Bladen   : 604
# 3rd Qu.:10     3rd Qu.:13     Wake    : 590
# Max.   :44     Max.   :57     (Other) :30952
# NA's    :30904   NA's    :30904   NA's    :30904
# dimension(s):
#   from   to   offset   delta   refsys point values x/y
# x     1 461 -84.3239  0.0192484 NAD27 FALSE  NULL [x]
# y     1 141  36.5896 -0.0192484 NAD27 FALSE  NULL [y]
```

Similarly, line and point geometries can be rasterized, as shown in figure 7.19.

```
read_sf(file) %>%
  st_cast("MULTILINESTRING") %>%
  select(CNTY_ID) %>%
  st_rasterize() %>%
  plot()
```

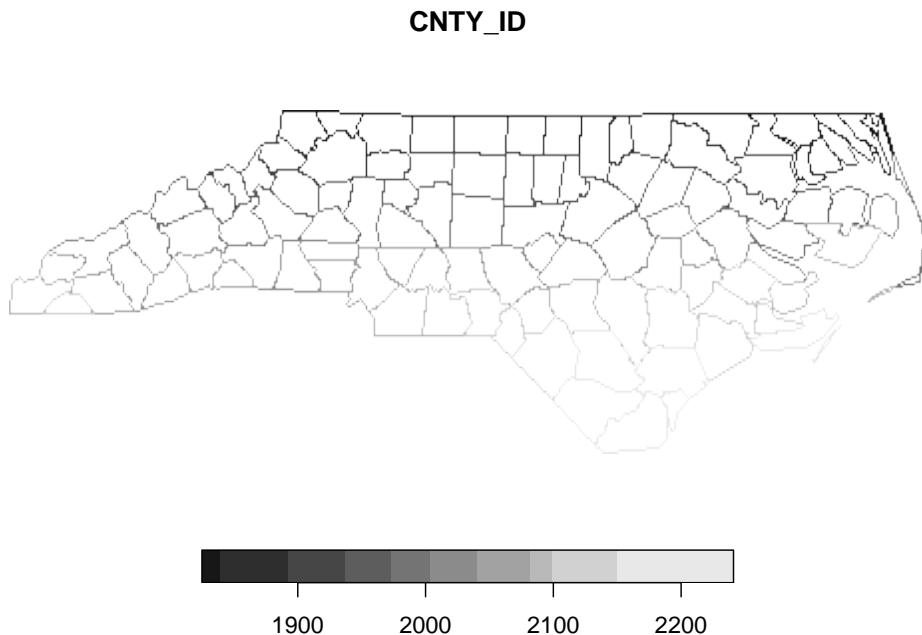


Figure 7.19: rasterization of North Carolina boundaries

7.6 Coordinate transformations and conversions

7.6.1 `st_crs`

Spatial objects of class `sf` or `stars` contain a coordinate reference system that can be get or replaced with `st_crs`, or be set or replaced in a pipe with `st_set_crs`. Coordinate reference systems can be set with an EPSG code, like `st_crs(4326)` which will be converted to `st_crs('EPSG:4326')`, or with a PROJ.4 string like "`+proj=utm +zone=25 +south`", a name like "WGS84", or a name preceded by an authority like "OGC:CRS84"; alternatives include a coordinate reference system definition in WKT, WKT-2 (section 2.5) or PROJJSON.

The object returned contains two fields:

- `wkt` with the WKT-2 representation
- `input` with the user input, if any, or a human readable description of the coordinate reference system, if available

Note that PROJ.4 strings can be used to *define* some coordinate reference systems, but they cannot be used to *represent* coordinate reference systems. Conversion of a WKT-2 in a `crs` object to a proj4string using the `$proj4string` method, as in

```
x = st_crs("OGC:CRS84")
x$proj4string
# [1] "+proj=longlat +datum=WGS84 +no_defs"
```

may succeed but is not in general lossless or invertible. Using PROJ.4 strings, for instance to *define* a parameterized, projected coordinate reference system is fine as long as it is associated with the WGS84 datum.

7.6.2 `st_transform`, `sf_project`

Coordinate transformations or conversions (section 2.4) for `sf` or `stars` objects are carried out with `st_transform`, which takes as its first argument a spatial object of class `sf` or `stars` that has a coordinate reference system set, and as a second argument with an `crs` object (or something that can be converted to it with `st_crs`). When PROJ finds more than one possibility to transform or convert from the source `crs` to the target `crs`, it chooses the one with the highest declared accuracy. More fine-grained control over the options is explained in section 7.6.5.

A lower-level function to transform or convert coordinates *not* in `sf` or `stars` objects is `sf_project`: it takes a matrix with coordinates and a source and target `crs`, and returns transformed or converted coordinates.

7.6.3 `sf_proj_info`

Function `sf_proj_info` can be used to query available projections, ellipsoids, units and prime meridians available in the PROJ software. It takes a single parameter, `type`, which can have the following values:

- `type = "proj"` lists the short and long names of available projections; short names can be used in a “`+proj=name`” string.
- `type = "ellps"` lists available ellipses, with name, long name, and ellipsoidal parameters.
- `type = "units"` lists the available length units, with conversion constant to meters
- `type = "prime_meridians"` lists the prime meridians with their position with respect to the Greenwich meridian.

7.6.4 `proj.db`, `datum grids`, `cdn.proj.org`, local cache

Datum grids (section 2.4) can be installed locally, or be read from the PROJ datum grid CDN at <https://cdn.proj.org/>. If installed locally, they are read from the PROJ search path, which is shown by

```
sf_proj_search_paths()
# [1] "/home/edzer/.local/share/proj"
# [2] "/usr/share/proj"
```

The main PROJ database is `proj.db`, an sqlite3 database typically found at

```
paste0(tail(sf_proj_search_paths(), 1), .Platform$file.sep, "proj.db")
# [1] "/usr/share/proj/proj.db"
```

which can be read. The version of the snapshot of the EPSG database included in each PROJ release is stated in the "`metadata`" table of `proj.db`; the version of the PROJ runtime used by `sf` is shown by

```
sf_extSoftVersion() ["PROJ"]
#      PROJ
# "7.2.1"
```

If for a particular coordinate transformation datum grids are not locally found, PROJ will search for online datum grids in the PROJ CDN when

```
sf_proj_network()
# [1] FALSE
```

returns TRUE. By default it is set to FALSE, but

```
sf_proj_network(TRUE)
# [1] "https://cdn.proj.org"
```

sets it to TRUE and returns the URL of the network resource used; this resource can also be set to another resource, that may be faster or less limited.

After querying a datum grid on the CDN, PROJ writes the *portion* of the grid queried (not, by default, the entire grid) to a local cache, which is another sqlite3 database found locally in a user directory, e.g. at

```
list.files(sf_proj_search_paths()[1], full.names = TRUE)
# [1] "/home/edzer/.local/share/proj/cache.db"
```

that will be searched first in subsequent datum grid queries.

7.6.5 transformation pipelines

Internally, PROJ uses a so-called *coordinate operation pipeline*, to represent the sequence of operations to get from a source CRS to a target CRS. Given multiple options to go from source to target, `st_transform` chooses the one with highest accuracy. We can query the options available by

```
(p = sf_proj_PIPELINES("EPSG:4326", "EPSG:22525"))
# Candidate coordinate operations found: 5
# Strict containment: FALSE
# Axis order auth compl: FALSE
# Source: EPSG:4326
# Target: EPSG:22525
# Best instantiable operation has accuracy: 2 m
# Description: axis order change (2D) + Inverse of Corrego Alegre
#               1970-72 to WGS 84 (2) + UTM zone 25S
# Definition: +proj=pipeline +step +proj=unitconvert +xy_in=deg
#              +xy_out=rad +step +inv
#              +proj=hgridshift
#              +grids=br_ibge_CA7072_003.tif +step
#              +proj=utm +zone=25 +south +ellps=intl
```

and see that pipeline with the highest accuracy is summarised; we can see that it specifies use of a datum grid. Had we not switched on the network search, we would have obtained a different result:

```
sf_proj_NETWORK(FALSE)
# character(0)
sf_proj_PIPELINES("EPSG:4326", "EPSG:22525")
# Candidate coordinate operations found: 5
# Strict containment: FALSE
# Axis order auth compl: FALSE
# Source: EPSG:4326
# Target: EPSG:22525
# Best instantiable operation has accuracy: 5 m
# Description: axis order change (2D) + Inverse of Corrego Alegre
#               1970-72 to WGS 84 (4) + UTM zone 25S
# Definition: +proj=pipeline +step +proj=unitconvert +xy_in=deg
#              +xy_out=rad +step +proj=push +v_3
#              +step +proj=cart +ellps=WGS84 +step
#              +proj=helmert +x=206.05 +y=-168.28
#              +z=3.82 +step +inv +proj=cart
#              +ellps=intl +step +proj=pop +v_3 +step
#              +proj=utm +zone=25 +south +ellps=intl
# Operation 4 is lacking 1 grid with accuracy 2 m
```

```
# Missing grid: br_ibge_CA7072_003.tif
# URL: https://cdn.proj.org/br_ibge_CA7072_003.tif
```

and a report that a datum grid is missing. The object returned by `sf_proj_pipelines` is a sub-classed `data.frame`, with columns

```
names(p)
# [1] "id"           "description"   "definition"    "has_inverse"
# [5] "accuracy"     "axis_order"    "grid_count"   "instantiable"
# [9] "containment"
```

and we can list for instance the accuracies by

```
p$accuracy
# [1] 5 5 8 2 NA
```

Here, NA refers to “ballpark accuracy”, which may be anything in the 30-120 m range:

```
p[is.na(p$accuracy),]
# Candidate coordinate operations found: 1
# Strict containment: FALSE
# Axis order auth compl: FALSE
# Source: EPSG:4326
# Target: EPSG:22525
# Best instantiable operation has only ballpark accuracy
# Description: axis order change (2D) + Ballpark geographic offset
#               from WGS 84 to Corrego Alegre 1970-72
#               + UTM zone 25S
# Definition: +proj=pipeline +step +proj=unitconvert +xy_in=deg
#              +xy_out=rad +step +proj=utm +zone=25
#              +south +ellps=intl
```

The default, most accurate pipeline chosen by `st_transform` can be overridden by specifying `pipeline` argument, as selected from the set of options in `p$definition`.

7.6.6 Axis order

As mentioned in section 2.5, the definition of EPSG:4326,

```
# GEOGCRS["WGS 84",
```

```

#      DATUM["World Geodetic System 1984",
#             ELLIPSOID["WGS 84",6378137,298.257223563,
#                     LENGTHUNIT["metre",1]]],
#             PRIMEM["Greenwich",0,
#                     ANGLEUNIT["degree",0.0174532925199433]],
#             CS[ellipsoidal,2],
#                 AXIS["geodetic latitude (Lat)",north,
#                     ORDER[1],
#                     ANGLEUNIT["degree",0.0174532925199433]],
#                 AXIS["geodetic longitude (Lon)",east,
#                     ORDER[2],
#                     ANGLEUNIT["degree",0.0174532925199433]],
#             USAGE[
#                 SCOPE["Horizontal component of 3D system."],
#                 AREA["World."],
#                 BBOX[-90,-180,90,180]],
#             ID["EPSG",4326]

```

indicates that the first axis is associated with latitude and the second with longitude; this is also the case for a number of other ellipsoidal coordinate reference systems. Although this is how the authority (EPSG) prescribes this, it is not how most datasets are currently stored! As most other software, package `sf` by default ignores this, and interprets ellipsoidal coordinates as (longitude, latitude) by default. If however data needs to be read e.g. from a WFS service that wants to be compliant to the authority, one can set

```
st_axis_order(TRUE)
```

to globally instruct `sf`, when calling GDAL and PROJ routines, that authority compliance (latitude, longitude order) is assumed. It is anticipated that problems may happen in case of authority compliance, e.g. with plotting data. At the time of writing this, the `plot` method for `sf` objects respects the axis order flag and will swap coordinates using the transformation pipeline "`+proj=pipeline +step +proj=axisswap +order=2,1`" before plotting them, but e.g. `geom_sf()` in `ggplot2` has not been modified to do this. As mentioned earlier, the ambiguity of EPSG:4326 is resolved by replacing it with OGC:CRS84.

7.7 Transforming and warping rasters

When using `st_transform` on a raster data set, as e.g. in

```
tif = system.file("tif/L7_ETMs.tif", package = "stars")
read_stars(tif) %>%
```

```

st_transform(4326)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max.
# L7_ETMs.tif      1      54     69 68.9      86   255
# dimension(s):
#       from to offset delta refsys point
# x      1 349     NA     NA WGS 84 FALSE
# y      1 352     NA     NA WGS 84 FALSE
# band  1   6     NA     NA     NA     NA
#                           values x/y
# x      [349x352] -34.9165,...,-34.8261 [x]
# y      [349x352] -8.0408,...,-7.94995 [y]
# band                         NULL
# curvilinear grid

```

we see that a *curvilinear* is created, which means that for every grid cell the coordinates are computed in the new CRS, which no longer form a *regular* grid. Plotting such data is extremely slow, as small polygons are computed for every grid cell and then plotted. The advantage of this is that no information is lost: grid cell values remain identical after the projection.

When we start with a raster on a regular grid and want to obtain a *regular* grid in a new coordinate reference system, we need to *warp* the grid: we need to recreate a grid at new locations, and use some rule to assign values to new grid cells. Rules can involve using the nearest value, or using some form of interpolation. This operation is not lossless and not invertible.

The best options for warping is to specify the target grid as a **stars** object. When only a target CRS is specified, default options for the target grid are picked that may be completely inappropriate for the problem at hand. An example workflow that uses only a target CRS is

```

read_stars(tif) %>%
  st_warp(crs = st_crs(4326)) %>%
  st_dimensions()
#       from to offset      delta refsys point values x/y
# x      1 350 -34.9166 0.000259243 WGS 84     NA    NULL [x]
# y      1 352 -7.94982 -0.000259243 WGS 84     NA    NULL [y]
# band  1   6     NA          NA     NA     NA    NULL

```

which creates a pretty close raster, but then the transformation is also relatively modest. For a workflow that creates a target raster first, here with exactly the same number of rows and columns as the original raster one could use:

```
r = read_stars(tif)
grd = st_bbox(r) %>%
      st_as_sfc() %>%
      st_transform(4326) %>%
      st_bbox() %>%
      st_as_stars(nx = dim(r)["x"], ny = dim(r)["y"])
st_warp(r, grd)
# stars object with 3 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# L7_ETMs.tif    1      54     69 68.9      86   255 6180
# dimension(s):
#       from to offset      delta refsys point values x/y
# x       1 349 -34.9166 0.000259666 WGS 84     NA    NULL [x]
# y       1 352 -7.94982 -0.000258821 WGS 84     NA    NULL [y]
# band    1   6     NA             NA     NA    NA    NULL
```

7.8 Exercises

Use R to solve the following exercises.

1. Find the names of the `nc` counties that intersect `LINESTRING(-84 35, -78 35)`; use `[` for this, and as an alternative use `st_join()` for this.
2. Repeat this after setting `sf_use_s2(FALSE)`, and *compute* the difference (hint: use `setdiff()`), and color the counties of the difference using color '#88000088'.
3. Plot the two different lines in a single plot; note that R will plot a straight line always straight in the projection currently used; `st_segmentize` can be used to add points on straight line, or on a great circle for ellipsoidal coordinates.
4. NDVI, normalized differenced vegetation index, is computed as $(\text{NIR}-\text{R})/(\text{NIR}+\text{R})$, with NIR the near infrared and R the red band. Read the `L7_ETMs.tif` file into object `x`, and distribute the band dimensions over attributes by `split(x, "band")`. Then, add attribute NDVI to this object by using an expression that uses the NIR (band 4) and R (band 3) attributes directly.
5. Compute NDVI for the `L7_ETMs.tif` image by reducing the band dimension, using `st_apply` and a function `ndvi = function(x) {(x[4]-x[3])/(x[4]+x[3])}`. Plot the result, and write the result to a GeoTIFF.
6. Use `st_transform` to transform the `stars` object read from `L7_ETMs.tif` to EPSG:4326. Print the object. Is this a regular grid? Plot the first band using arguments `axes=TRUE` and `border=NA`, and explain why this takes such a long time.

7. Use `st_warp` to warp the `L7_ETMs.tif` object to `EPSG:4326`, and plot the resulting object with `axes=TRUE`. Why is the plot created much faster than after `st_transform`?
8. Using a vector representation of the raster `L7_ETMs`, plot the intersection with a circular area around `POINT(293716 9113692)` with radius 75 m, and compute the area-weighted mean pixel values for this circle. Compare the area-weighted values with those obtained by `aggregate` using the vector data, and by `aggregate` using the raster data, using `exact=FALSE` (default) and `exact=TRUE`. Explain the differences.

Chapter 8

Large data sets

This chapter describes how large spatial and spatiotemporal datasets can be handled with R, with a focus on packages `sf` and `stars`. For practical use, we classify large data sets as either

- too large to fit in working memory, or
- also too large to fit on the local hard drive, or
- also too large to download it to locally managed compute infrastructure (such as network attached storage)

these three categories correspond very roughly to Gigabyte-, Terabyte- and Petabyte-sized data sets.

8.1 Vector data: `sf`

8.1.1 Reading from disk

Function `st_read` reads vector data from disk, using GDAL, and then keeps the data read in working memory. In case the file is too large to be read in working memory, several options exist to read parts of the file. The first is to set argument `wkt_filter` with a WKT text string containing a geometry; only geometries from the target file that intersect with this geometry will be returned. An example is

```
library(sf)
(file = system.file("gpkg/nc.gpkg", package="sf"))
# [1] "/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/sf/gpkg/nc.gpkg"
bb = "POLYGON ((-81.7 36.2, -80.4 36.2, -80.4 36.5, -81.7 36.5, -81.7 36.2))"
```

```
nc.1 = st_read(file, wkt_filter = bb)
# Reading layer `nc.gpkg' from data source
#   `/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/sf/gpkg/nc.gpkg'
#   using driver `GPKG'
# Simple feature collection with 8 features and 14 fields
# Geometry type: MULTIPOLYGON
# Dimension:      XY
# Bounding box:  xmin: -81.9 ymin: 36 xmax: -80 ymax: 36.6
# Geodetic CRS:  NAD27
```

The second option is to use the `query` argument to `st_read`, which can be any query in “OGR SQL” dialect, which can be used to select features from a layer, and limit fields. An example is:

```
q = paste("select BIR74,SID74,geom from 'nc.gpkg' where BIR74 > 1500")
nc.2 = st_read(file, query = q)
# Reading query `select BIR74,SID74,geom from 'nc.gpkg' where BIR74 > 1500' from data ...
#   using driver `GPKG'
# Simple feature collection with 61 features and 2 fields
# Geometry type: MULTIPOLYGON
# Dimension:      XY
# Bounding box:  xmin: -83.3 ymin: 33.9 xmax: -76.1 ymax: 36.6
# Geodetic CRS:  NAD27
```

note that `nc.gpkg` is the *layer name*, which can be obtained from `file` using `st_layers`. Sequences of records can be read using `LIMIT` and `OFFSET`, to read records 51-60 use

```
q = paste("select BIR74,SID74,geom from 'nc.gpkg' LIMIT 10 OFFSET 50")
nc.2 = st_read(file, query = q)
# Reading query `select BIR74,SID74,geom from 'nc.gpkg' LIMIT 10 OFFSET 50' from data ...
#   using driver `GPKG'
# Simple feature collection with 10 features and 2 fields
# Geometry type: MULTIPOLYGON
# Dimension:      XY
# Bounding box:  xmin: -84 ymin: 35.2 xmax: -75.5 ymax: 36.2
# Geodetic CRS:  NAD27
```

Further query options include selection on geometry type, polygon area. When the dataset queried is a spatial database, then the query is passed on to the database and not interpreted by GDAL; this means that more powerful features will be available. Further information is found in the GDAL documentation under “OGR SQL dialect”.

Very large files or directories that are zipped can be read without the need to unzip them, using the `/vsizip` (for zip), `/vsigzip` (for gzip) or `/vsitar` (for

tar files) prefix to files; this is followed by the path to the zip file, and then followed by the file inside this zip file. Reading files this way may come at some computatational cost.

8.1.2 Reading from databases, dbplyr

Although GDAL has support for several spatial databases, and as mentioned above it passes on SQL in the `query` argument to the database, it is sometimes beneficial to directly read from and write to a spatial database using the R database drivers for this. An example of this is:

```
pg <- DBI::dbConnect(
  RPostgres::Postgres(),
  host = "localhost",
  dbname = "postgis")
st_read(pg, query = "select BIR74,wkb_geometry from nc limit 3")
# Simple feature collection with 3 features and 1 field
# Geometry type: MULTIPOLYGON
# Dimension: XY
# Bounding box: xmin: -81.7 ymin: 36.2 xmax: -80.4 ymax: 36.6
# Geodetic CRS: NAD27
#   bir74           wkb_geometry
# 1 1091 MULTIPOLYGON (((-81.5 36.2,...
```

A spatial query might look like

```
q = "SELECT BIR74,wkb_geometry FROM nc WHERE \
  ST_Intersects(wkb_geometry, 'SRID=4267;POINT (-81.49826 36.4314)');"
st_read(pg, query = q)
# Simple feature collection with 1 feature and 1 field
# Geometry type: MULTIPOLYGON
# Dimension: XY
# Bounding box: xmin: -81.7 ymin: 36.2 xmax: -81.2 ymax: 36.6
# Geodetic CRS: NAD27
#   bir74           wkb_geometry
# 1 1091 MULTIPOLYGON (((-81.5 36.2,...
```

Here, the intersection is done in the database, and uses the spatial index typically present.

The same mechanism works when using `dplyr` with a database backend:

```
library(dplyr, warn.conflicts = FALSE)
nc_db = tbl(pg, "nc")
```

Spatial queries can be formulated and are passed on to the database:

```
nc_db %>%
  filter(ST_Intersects(wkb_geometry, 'SRID=4267;POINT (-81.49826 36.4314)')) %>%
  collect()
# # A tibble: 1 x 16
#   ogc_fid area perimeter cnty_id name    fips  fipsno
#       <int>  <dbl>     <dbl> <dbl> <chr> <chr> <dbl>
# 1      1 0.114     1.44  1825  1825 Ashe  37009 37009
# # ... with 8 more variables: cress_id <int>, bir74 <dbl>,
# #   sid74 <dbl>, nwbir74 <dbl>, bir79 <dbl>, sid79 <dbl>,
# #   nwbir79 <dbl>, wkb_geometry <pq_gmtry>
nc_db %>% filter(ST_Area(wkb_geometry) > 0.1) %>% head(3)
# # Source: lazy query [?? x 16]
# # Database: postgres [edzer@localhost:5432/postgis]
#   ogc_fid area perimeter cnty_id name    fips  fipsno
#       <int>  <dbl>     <dbl> <dbl> <chr> <chr> <dbl>
# 1      1 0.114     1.44  1825  1825 Ashe  37009 37009
# 2      3 0.143     1.63  1828  1828 Surry 37171 37171
# 3      5 0.153     2.21  1832  1832 Northampton 37131 37131
# # ... with 8 more variables: cress_id <int>, bir74 <dbl>,
# #   sid74 <dbl>, nwbir74 <dbl>, bir79 <dbl>, sid79 <dbl>,
# #   nwbir79 <dbl>, wkb_geometry <pq_gmtry>
```

It should be noted that PostGIS' `ST_Area` computes the same area as the `AREA` field in `nc`, which is the meaningless value obtained by assuming the coordinates are projected, although they are ellipsoidal.

8.1.3 Reading from online resources or web services

GDAL drivers support reading from online recourses, by prepending `/vsicurl/` before the URL starting with e.g. `https://`. A number of similar drivers specialized for particular clouds include `/vsis3` for Amazon S3, `/vsigs` for Google Cloud Storage, `/vsiaz` for Azure, `/vsioss` for Alibaba Cloud, or `/vsiswift` for OpenStack Swift Object Storage. These prepositions can be combined e.g. with `/vsizip/` to read a zipped online resource. Depending on the file format used, reading information this way may always involve reading the entire file, or reading it multiple times, and may not always be the most efficient way of handling resources. A format like “cloud-optimized geotiff” (COG) has been specially designed to be efficient and resource-friendly in many cases, e.g. for only reading the metadata, or for only reading overviews (low-resolutions versions of the full

imagery) or spatial segments. COGs can also be created using the GeoTIFF driver of GDAL, and setting the right dataset creation options in a `write_stars` call.

8.1.4 API's, OpenStreetMap

Although online resource do not have to be stored files but could be created server-side on the fly, typical web services for geospatial data create data on the fly, and give access to this through an API. As an example, data from OpenStreetMap can be bulk downloaded and read locally, e.g. using the GDAL vector driver, but more typical a user wants to obtain a small subset of the data or use the data for a small query. Several R packages exist that query openstreetmap data:

- Package `OpenStreetMap` downloads data as raster tiles, typically used as backdrop or reference for plotting other features
- Package `osmdata` downloads vector data as points, lines or polygons in `sf` or `sp` format
- Package `osmar` returns vector data, but in addition the network topology (as an `igraph` object) that contains how road elements form a network, and has functions that compute the shortest route.

When provided with a correctly formulated API call in the URL the highly configurable GDAL OSM driver (in `st_read`) can read an “.osm” file (xml) and returns a dataset with five layers: `points` that have significant tags, `lines` with non-area “way” features, `multilinestrings` with “relation” features, `multipolygons` with “relation” features and `other_relations`. A simple and very small bounding box query to OpenStreetMap could look like

```
download.file(
  "https://openstreetmap.org/api/0.6/map?bbox=7.595,51.969,7.598,51.970",
  "data/ms.osm", method = "auto")
```

and from this file we can read the layer `lines`, and plot its first attribute by

```
o = read_sf("data/ms.osm", "lines")
p = read_sf("data/ms.osm", "multipolygons")
bb = st_bbox(c(xmin=7.595, ymin = 51.969, xmax = 7.598, ymax = 51.970),
             crs = 4326)
plot(st_as_sfc(bb), axes = TRUE, lwd = 2, lty = 2, cex.axis = .5)
plot(o[,1], lwd = 2, add = TRUE)
plot(st_geometry(p), border = NA, col = '#88888888', add = TRUE)
```

the result of which is shown in figure 8.1. The overpass API provides a more generic and powerful query functionality to OpenStreetMap data.

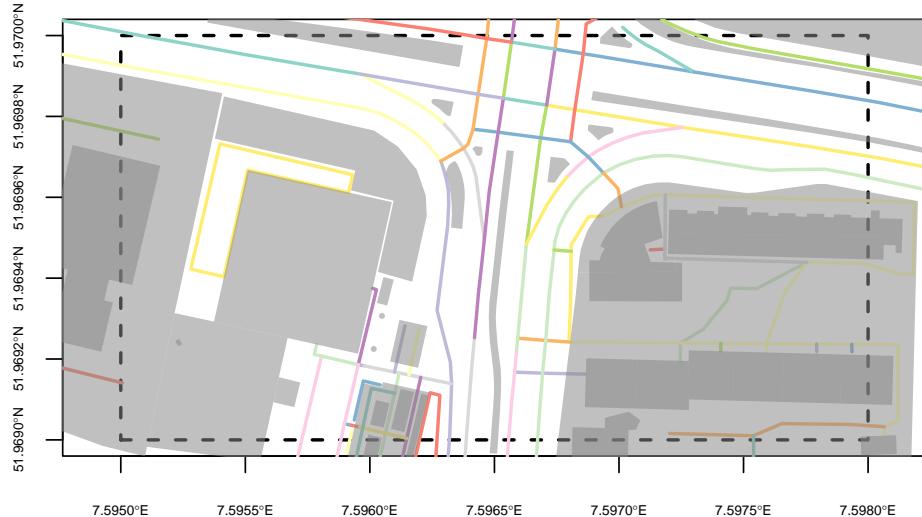


Figure 8.1: OpenStreetMap vector data

8.2 Raster data: stars

A common challenge with raster datasets is not only that they come in large files (single Sentinel-2 tiles are around 1 Gb), but that many of these files, potentially thousands, are needed to address the area and time period of interest. At time of writing this, the Copernicus program that runs all Sentinel satellites publishes 160 Tb of images per day. This means that a classic pattern in using R, consisting of

- downloading data to local disc,
- loading the data in memory,
- analysing it

is not going to work.

Cloud-based Earth Observation processing platforms like Google Earth Engine (Gorelick et al., 2017) or Sentinel Hub recognize this and let users work with datasets up to the petabyte range rather easily and with a great deal of interactivity. They share the following properties:

- computations are postponed as long as possible (lazy evaluation),
- only the data you ask for are being computed and returned, and nothing more,
- storing intermediate results is avoided in favour of on-the-fly computations,

- maps with useful results are generated and shown quickly to allow for interactive model development.

This is similar to the `dbplyr` interface to databases and cloud-based analytics environments, but differs in the aspect of *what* we want to see quickly: rather than the first n records of a `dbplyr` table, we want a quick *overview* of the results, in the form of a map covering the whole area, or part of it, but at screen resolution rather than native (observation) resolution.

If for instance we want to “see” results for the United States on screen with 1000 x 1000 pixels, we only need to compute results for this many pixels, which corresponds roughly to data on a grid with 3000 m x 3000 m grid cells. For Sentinel-2 data with 10 m resolution, this means we can subsample with a factor 300, giving 3 km x 3 km resolution. Processing, storage and network requirements then drop a factor $300^2 \approx 10^5$, compared to working on the native 10 m x 10 m resolution. On the platforms mentioned, zooming in the map triggers further computations on a finer resolution and smaller extent.

A simple optimisation that follows these lines is how `stars`’ plot method works: in the case of plotting large rasters, it subsamples the array before it plots, drastically saving time. The degree of subsampling is derived from the plotting region size and the plotting resolution (pixel density). For vector devices, such as pdf, R sets plot resolution to 75 dpi, corresponding to 0.3 mm per pixel. Enlarging plots may reveal this, but replotting to an enlarged devices will create a plot at target density.

8.2.1 stars proxy objects

To handle datasets that are too large to fit in memory, `stars` provides `stars_proxy` objects. To demonstrate its use, we will use the `starsdata` package, an R data package with larger datasets (around 1 Gb total). It can be installed by

```
options(timeout = 600) # or large in case of slow network
install.packages("starsdata", repos = "http://pebesma.staff.ifgi.de",
  type = "source")
```

We can “load” a Sentinel-2 image from it by

```
f = "sentinel/S2A_MSIL1C_20180220T105051_N0206_R051_T32ULE_20180221T134037.zip"
granule = system.file(file = f, package = "starsdata")
file.size(granule)
# [1] 7.69e+08
base_name = strsplit(basename(granule), ".zip")[[1]]
s2 = paste0("SENTINEL2_L1C:/vsizip/", granule, "/", base_name,
```

```

".SAFE/MTD_MSIL1C.xml:10m:EPSG_32632")
(p = read_stars(s2, proxy = TRUE))
# stars_proxy object with 1 attribute in 1 file(s):
# $`MTD_MSIL1C.xml:10m:EPSG_32632`
# [1] "[...]/MTD_MSIL1C.xml:10m:EPSG_32632"
#
# dimension(s):
#      from     to offset delta          refsys point    values
# x      1 10980 3e+05   10 WGS 84 / UTM z...    NA    NULL
# y      1 10980 6e+06  -10 WGS 84 / UTM z...    NA    NULL
# band  1     4     NA     NA                  NA    NA B4,...,B8
#      x/y
# x      [x]
# y      [y]
# band
object.size(p)
# 11160 bytes

```

and we see that this does not actually load *any* of the pixel values, but keeps the reference to the dataset and fills the dimensions table. (The convoluted `s2` name is needed to point GDAL to the right file inside the `.zip` file containing 115 files in total).

The idea of a proxy object is that we can build expressions like

```
p2 = p * 2
```

but that the computations for this are postponed. Only when we really need the data, e.g. because we want to plot it, is `p * 2` evaluated. We need data when either

- we want to `plot` data, or
- we want to write an object to disk, with `write_stars`, or
- we want to explicitly load an object in memory, with `st_as_stars`

In case the entire object does not fit in memory, `plot` and `write_stars` choose different strategies to deal with this:

- `plot` fetches only the pixels that can be seen, rather than all pixels available, and
- `write_stars` reads, processes, and writes data chunk by chunk.

Downsampling and chunking is implemented for spatially dense images, not e.g. for dense time series, or other dense dimensions.

As an example, the output of `plot(p)`, shown in figure 8.2

```
# downsample set to c(56)
```

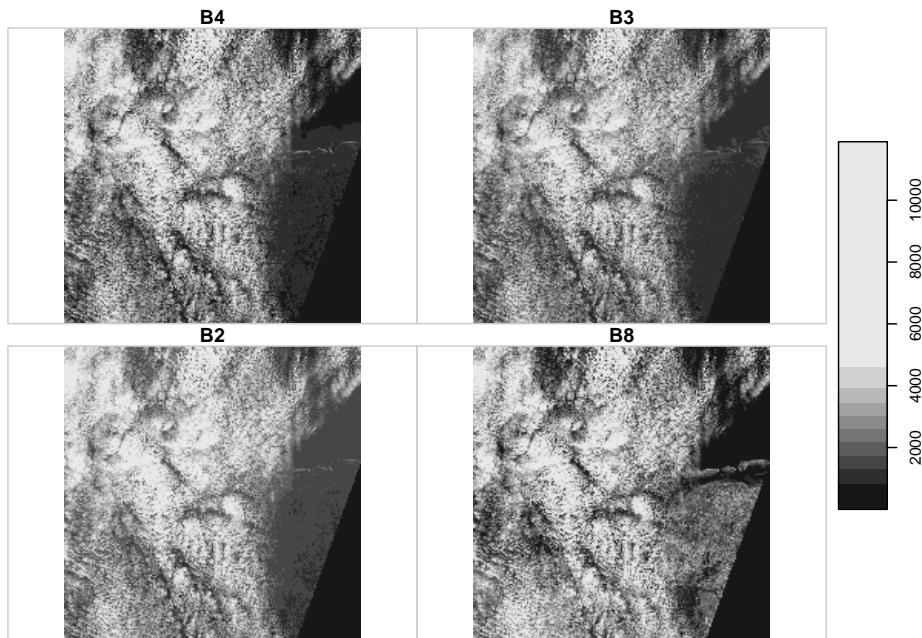


Figure 8.2: Plot of downsampled 10 m bands of a Sentinel-2 scene

only fetches the pixels that can be seen on the plot device, rather than the 10980 x 10980 pixels available in each band. The downsampling ratio taken is

```
(ds = floor(sqrt(prod(dim(p)) / prod(dev.size("px")))))
# [1] 56
```

meaning that for every 56×56 sub-image in the original image, only one pixel is read, and plotted. This value is still a bit too low as it ignores the white space and space for the key on the plotting device.

8.2.2 Operations on proxy objects

A few dedicated methods are available for `stars_proxy` objects:

```
methods(class = "stars_proxy")
# [1] [l] [[[<- [<- [adrop
# [5] aggregate [aperm [as.data.frame [c
# [9] coerce [dim [droplevels [filter
# [13] hist [initialize [is.na [mapView
```

```
# [17] Math          merge      mutate      Ops
# [21] plot         predict     print       pull
# [25] replace_na   select     show        slice
# [29] slotsFromS3  split      st_apply   st_as_sf
# [33] st_as_stars  st_crop    st_mosaic  st_redimension
# [37] st_sample    st_set_bbox transmute  write_stars
# see '?methods' for accessing help and source code
```

We have seen `plot` and `print` in action; `dim` reads out the dimension from the dimensions metadata table.

The three methods that actually fetch data are `st_as_stars`, `plot` and `write_stars`. `st_as_stars` reads the actual data into a `stars` object, its argument `downsample` controls the downsampling rate. `plot` does this too, choosing an appropriate `downsample` value from the device resolution, and plots the object. `write_stars` writes a `star_proxy` object to disc.

All other methods for `stars_proxy` objects do not actually operate on the raster data but add the operations to a *to do* list, attached to the object. Only when actual raster data are fetched, e.g. by calling `plot` or `st_as_stars`, the commands in this list are executed.

`st_crop` limits the extent (area) of the raster that will be read. `c` combines `stars_proxy` objects, but still doesn't read any data. `adrop` drops empty dimensions, `aperm` changes dimension order.

`write_stars` reads and processes its input chunk-wise; it has an argument `chunk_size` that lets users control the size of spatial chunks.

8.3 Very large data cubes

At some stage, data sets need to be analysed that are so large that downloading them is no longer feasible; even when local storage would be sufficient, network bandwidth may become limiting. Examples are satellite image archives such as those from Landsat and Copernicus (Sentinel-x), or model computations such as the ERA5 (Hersbach et al., 2020), a model reanalysis of the global atmosphere, land surface and ocean waves from 1950 onwards. In such cases it may be most helpful to either gain access to (typically: rent) virtual machines in a cloud where these data are available and nearby (i.e., the data should be stored in the same data center as where the virtual machine is), or to use a system that lets the user carry out such computations without having to worry about virtual machines. Both options will be discussed.

8.3.1 Finding and processing assets

When working on a virtual machine on a cloud, a first task is usually to find the assets (files) to work on. It looks attractive to obtain a file listing, and then parse file names such as

`S2A_MSIL1C_20180220T105051_N0206_R051_T32ULE_20180221T134037.zip`

for their metadata including the date of acquisition and the code of the spatial tile covered. Obtaining such a file listing however is usually computationally very demanding, as is the processing of the result, when the number of tiles runs in the many millions.

A solution to this is to use a catalogue. The recently developed and increasingly deployed STAC, short for *spatiotemporal asset catalogue*, provides an API that can be used to query image collections by properties like bounding box, date, band, and cloud coverage. The R package `rstac` (Brazil Data Cube Team, 2021) provides an R interface to create queries, and manage the information returned.

Processing the resulting files may involve creating a data cube at a lower spatial and/or temporal resolution, from images that may span a range of coordinate reference systems (e.g., several UTM zones). An R package that can do that is `gdalcubes` (Appel, 2020; Appel and Pebesma, 2019), which can also directly use STAC output (Marius Appel, 2021).

8.3.2 Processing data: GEE, openEO

Platforms that do not require the management and programming of virtual machines *in* the cloud but provide direct access to the imagery managed include GEE, openEO, and the climate data store.

Google Earth Engine (GEE) is a cloud platform that allows users to compute on large amounts of Earth Observation data as well as modelling products (Gorelick et al., 2017). It has powerful analysis capabilities, including most of the data cube operations explained in section 6.3. It has an IDE where scripts can be written in JavaScript, and a Python interface to the same functionality. The code of GEE is not open source, and cannot be extended by arbitrary user-defined functions in languages like Python or R. R package `rgee` (Aybar, 2021) provides an R client interface to GEE.

Cloud-based data cube processing platforms built entirely around open source software are emerging, several of which using the openEO API (Schramm et al., 2021). This API allows for user-defined functions (UDFs) written in Python or R that are being passed on through the API and executed at the pixel level, e.g. to aggregate or reduce dimensions. UDFs in R represent the data chunk to be processed as a `stars` object, in Python `xarray` objects are used.

Other platforms include the Copernicus climate data store (Raoult et al., 2017) or atmosphere data store, which allow processing of atmospheric or climate data from ECMWF, including ERA5. An R package with an interface to both data stores is `ecmwfr` (Hufkens, 2020).

8.4 Exercises

Use R to solve the following exercises.

1. For the S2 image (above), find out in which order the bands are by using `st_get_dimension_values()`, and try to find out (e.g. by internet search) which spectral bands / colors they correspond to.
2. Compute NDVI for the S2 image, using `st_apply` and an appropriate `ndvi` function. Plot the result to screen, and then write the result to a GeoTIFF. Explain the difference in runtime between plotting and writing.
3. Plot an RGB composite of the S2 image, using the `rgb` argument to `plot()`, and then by using `st_rgb()` first.
4. Select five random points from the bounding box of `S2`, and extract the band values at these points; convert the object returned to an `sf` object.
5. For the 10 km radius circle around `POINT(390000 5940000)`, use `aggregate` to compute the mean pixel values of the S2 image when downsampling the images with factor 30, and on the original resolution. Compute the relative difference between the results.
6. Use `hist` to compute the histogram on the downsampled S2 image. Also do this for each of the bands. Use `ggplot2` to compute a single plot with all four histograms.
7. Use `st_crop` to crop the S2 image to the area covered by the 10 km circle. Plot the results. Explore the effect of setting argument `crop = FALSE`.
8. With the downsampled image, compute the logical layer where all four bands have pixel values higher than 1000. Use a raster algebra expression on the four bands (use `split` first), or use `st_apply` for this.

Chapter 9

Plotting spatial data

Together with timelines, maps belong to the most powerful graphs, perhaps because we can immediately relate where we are, or have been, on the space of the plot. Two recent books on visualisation (Healy, 2018; Wilke, 2019) contain chapters on visualising geospatial data or maps. Here, we will not try to preach the do's and don'ts of maps, but rather point out a number of possibilities how to do things, point out challenges along the way and ways to mitigate them.

9.1 Every plot is a projection

The world is round, but plotting devices are flat. As mentioned in section 2.2.2, any time we visualise, in any way, the world on a flat device, we project: we convert ellipsoidal coordinates into Cartesian coordinates. This includes the cases where we think we “do nothing” (figure 9.1, left), or where show the world “as it is”, e.g. as seen from space (figure 9.1, right).

```
library(sf)
library(rnaturalearth)
w <- ne_countries(scale = "medium", returnclass = "sf")
suppressWarnings(st_crs(w) <- st_crs(4326))
layout(matrix(1:2, 1, 2), c(2,1))
par(mar = rep(0, 4))
plot(st_geometry(w))

# sphere:
library(s2)
g = as_s2_geography(TRUE) # Earth
co = s2_data_countries()
oc = s2_difference(g, s2_union_agg(co)) # oceans
```

```
b = s2_buffer_cells(as_s2_geography("POINT(-30 -10)"), 9800000) # visible half
i = s2_intersection(b, oc) # visible ocean
co = s2_intersection(b, co)
plot(st_transform(st_as_sf(i), "+proj=ortho +lat_0=-10 +lon_0=-30"), col = 'lightblue')
plot(st_transform(st_as_sf(co), "+proj=ortho +lat_0=-10 +lon_0=-30"), col = NA, add = TRUE)
```

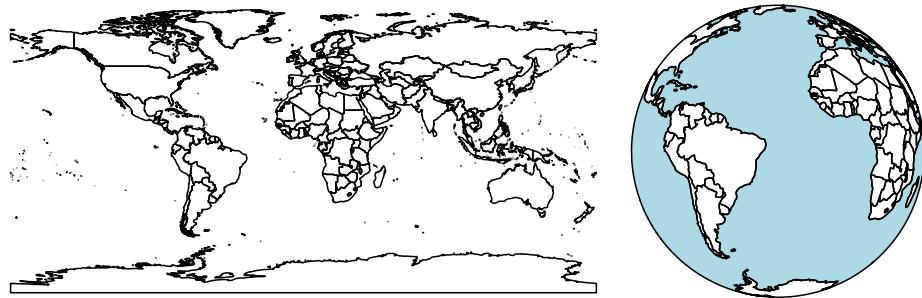


Figure 9.1: Earth country boundaries; left: mapping long/lat linearly to x and y (plate carrée); right: as seen from infinite distance (orthographic)

The left plot of figure 9.1 was obtained by

```
library(sf)
library(rnaturalearth)
w <- ne_countries(scale = "medium", returnclass = "sf")
plot(st_geometry(w))
```

and we see that this is the default projection for data with ellipsoidal coordinates, as indicated by

```
st_is_longlat(w)
# [1] TRUE
```

The projection taken in figure 9.1 (left) is the equirectangular (or equidistant cylindrical) projection, which maps longitude and latitude linearly to the x and y axis, keeping an aspect ratio of 1. Were we to do this for smaller areas not on the equator, it makes sense to choose a plot ratio such that one distance unit E-W equals one distance unit N-S on the center of the plotted area, and this is the default behaviour of the `plot()` method for unprojected `sf` or `stars` datasets, as well as the default for `ggplot2::geom_sf()` (section @rer(geomsf)).

We can also carry out this projection before plotting. Say we want to plot Germany, then after loading the (rough) country outline, we use `st_transform` to project:

```
DE = st_geometry(ne_countries(country = "germany", returnclass = "sf"))
DE.eqc = st_transform(DE, "+proj=eqc +lat_ts=51.14 +lon_0=90w")
```

The `eqc` refers to the “equidistant cylindrical” projection of PROJ; the projection parameter here is `lat_ts`, the latitude of true scale (i.e., one length unit N-S equals one length unit E-W), which was here chosen as the middle of the bounding box latitudes

```
print(mean(st_bbox(DE)[c("ymin", "ymax")]), digits = 4)
# [1] 51.14
```

When we now plot both maps (figure 9.2), they look identical up to the values along the axes: degrees for ellipsoidal (left), and metres for projected (Cartesian) coordinates.

```
par(mfrow = c(1, 2))
plot(DE, axes = TRUE)
plot(DE.eqc, axes = TRUE)
```

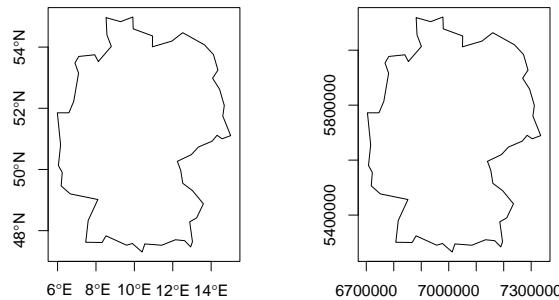


Figure 9.2: Germany in equirectangular projection: with axis units degrees (left) and metres in the equidistant cylindrical projection (right)

9.1.1 What is a good projection for my data?

There is unfortunately no silver bullet here. Projections that maintain all distances do not exist; only globes do. The most used projections try to preserve

- areas (equal area),
- directions (conformal, e.g. Mercator),
- some properties of distances (e.g. equirectangular preserves distances along meridians, azimuthal equidistant preserves distances to a central point)

or some compromise of these. Parameters of projections decide what is shown in the center of a map and what on the fringes, which areas are up and which are down, and which areas are most enlarged. All these choices are in the end political decisions.

It is often entertaining and at times educational to play around with the different projections and understand their consequences. When the primary purpose of the map however is not to entertain or educate projection varieties, it may be preferable to choose a well-known or less surprising projection, and move the discussion which projection should be preferred to a decision process on its own.

9.2 Plotting points, lines, polygons, grid cells

Since maps are just a special form of plots of statistical data, the usual rules hold. Frequently occurring challenges include:

- polygons may be very small, and vanish when plotted,
- depending on the data, polygons for different features may well overlap, and be visible only partially; using transparent fill colors may help identify them
- when points are plotted with symbols, they may easily overlap and be hidden; density maps (chapter 11) may be more helpful
- lines may be hard to read when coloured and may overlap regardless line width

9.2.1 Colors

When plotting polygons filled with colors, one has the choice to plot polygon boundaries, or to suppress these. If polygon boundaries draw too much attention, an alternative is to colour them in a grey tone, or another color that doesn't interfere with the fill colors. When suppressing boundaries entirely, polygons with (nearly) identical colors will no longer be visually distinguishable. If the property indicating the fill color is constant over the region, such as land cover type, then this is not a problem but if the property is an aggregation, the region over which it was aggregated gets lost, and by that the proper interpretation: especially for extensive variables, e.g. the amount of people living in a polygon, this strongly misleads. But even with polygon boundaries, using filled polygons for such variables may not be a good idea.

The use of continuous color scales that have no noticeable color breaks for continuously varying variables may look attractive, but is often more fancy than useful:

- it is impractical to match a color on the map with a legend value

- colors ramps often stretch non-linearly over the value range, making it hard to convey magnitude

Only for cases where the identification of values is less important than the continuity of the map, such as the coloring of a high resolution digital terrain model, it does serve its goal. Good colors scales are e.g. found in packages **RColorBrewer** (Neuwirth, 2014), **viridis** (Garnier, 2021) or **colorspace** (Ihaka et al., 2021; Zeileis et al., 2020).

9.2.2 Color breaks: `classInt`

When plotting continuous geometry attributes using a limited set of colors (or symbols), classes need to be made from the data. R package **classInt** (Bivand, 2020a) provides a number of methods to do so. The default method is “quantile”:

```
library(classInt)
# set.seed(1) needed ?
r = rnorm(100)
(cI <- classIntervals(r))
# style: quantile
#   one of 1.49e+10 possible partitions of this variable into 8 classes
#   [-2.61,-1.26]  [-1.26,-0.356]  [-0.356,-0.131]  [-0.131,0.091]
#           13          12          13          12
#   [0.091,0.433]  [0.433,0.623]  [0.623,1.11]  [1.11,2.76]
#           12          13          12          13
cI$brks
# [1] -2.612 -1.257 -0.356 -0.131  0.091  0.433  0.623  1.113  2.755
```

it takes argument `n` for the number of intervals, and a `style` that can be one of “fixed”, “sd”, “equal”, “pretty”, “quantile”, “kmeans”, “hclust”, “bclust”, “fisher” or “jenks”. Style “pretty” may not obey `n`; if `n` is missing, ‘nclass.Sturges’ is used; two other methods are available for choosing `n` automatically. If the number of observations is greater than 3000, a 10% sample is used to create the breaks for “fisher” and “jenks”.

9.2.3 Graticule and other navigation aids

A graticules is a network of lines on a map that follow constant latitude or longitude. On figure 1.1 a graticule is drawn in grey, on figure 1.2 in white. Graticules are often drawn in maps to give reference where something is. In our first map in figure 1.1 we can read that the area plotted is near 35° North and 80° West. Had we plotted the lines in the projected coordinate system, they would have been straight and their actual numbers would not have been very

informative, apart from giving an interpretation of size or distances when the unit is known, and familiar to the map reader. Graticules, by that, also shed light on which projection was used: equirectangular or Mercator projections have straight vertical and horizontal lines, conic projections have straight but diverging meridians, equal area may have curved meridians.

The real navigation aid on figure 9.1 and most other maps are geographical features like the state outline, country outlines, coast lines, rivers, roads, railways and so on. If these are added sparsely and sufficiently, a graticule can as well be omitted. In such cases, maps look good without axes, tics, and labels, leaving up a lot of plotting space to be filled with actual map data.

9.3 Base plot

The `plot` method for `sf` and `stars` objects try to make quick, useful, exploratory plots; for higher quality plots and more configurability, alternatives with more control and/or better defaults are offered for instance by packages `ggplot2` (Wickham et al., 2021), `tmap` (Tennekes, 2021, 2018) or `maps` (Giraud, 2021).

By default, the `plot` method tries to plot “all” it is given. This means that:

- given a geometry only (`sfc`), the geometry is plotted, without colors,
- given a geometry and an attribute, the geometry is colored according to the values of the attribute, using a qualitative color scale for `factor` or `logical` attributes and a continuous scale otherwise,
- given multiple attributes, multiple maps are plotted, each with a color scale but a legend is omitted by default, as color assignment is done on a per sub-map basis,
- for `stars` objects with multiple attributes, only the first attribute is plotted; for three-dimensional raster cubes, all slices over the third dimension are plotted.

9.3.1 Adding to plots with legends

The `plot` methods for `stars` and `sf` objects may show a color key on one of the sides (e.g., figure 1.1). To do this with `base::plot`, the plot region is split in two and two plots are created: one with the map, and one with the legend. By default, the `plot` function resets the graphics device (using `layout(matrix(1))`) so that subsequent plots are not hindered by the device being split in two. If one wants to *add* to an existing plot having a color legend, this is however what is needed, and resetting the plotting device needs to be prevented by setting argument `reset = FALSE`, and use `add = TRUE` in a subsequent call to `plot`, an example is

```
library(sf)
nc = read_sf(system.file("gpkg/nc.gpkg", package="sf"))
plot(nc["BIR74"], reset = FALSE, key.pos = 4)
plot(st_buffer(nc[1,1], units::set_units(10, km)), col = 'NA',
     border = 'red', lwd = 2, add = TRUE)
```

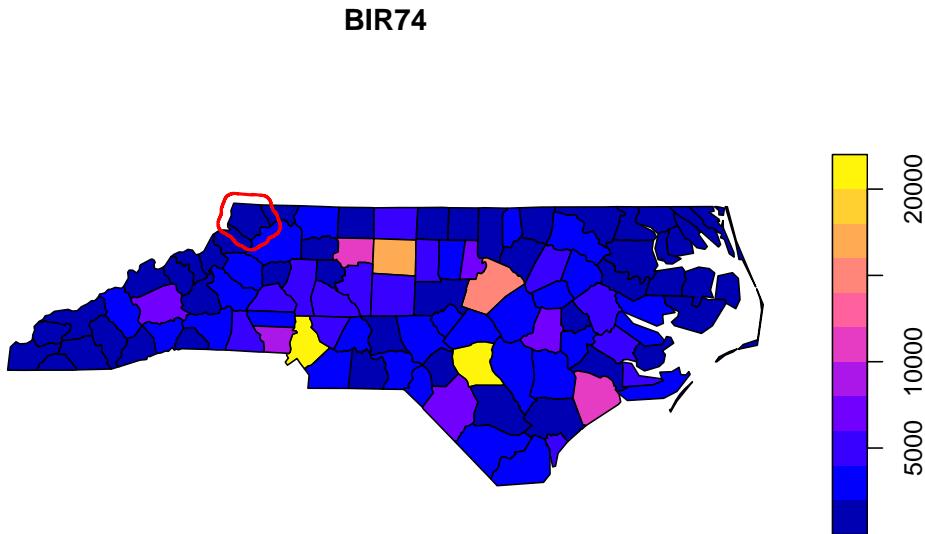


Figure 9.3: Annotating base plots that have a legend

which is shown in figure 9.3. Annotating `stars` plots can be done in the same way when a *single* stars layer is shown. Annotating `stars` plots with multiple cube slices can be done by adding a “hook” function that will be called on every slice shown, as in

```
library(stars)
r = read_stars(system.file("tif/L7_ETMs.tif", package = "stars"))
circ = st_bbox(r) %>% st_as_sfc() %>% st_sample(5) %>% st_buffer(300)
hook = function() plot(circ, col = NA, border = 'yellow', add = TRUE)
plot(r, hook = hook, key.pos = 4)
# downsample set to c(2,2,1)
```

and as shown in figure 9.4.

Base plot methods have access to the resolution of the screen device and hence the base plot method for `stars` and `stars_proxy` object will downsample dense rasters and only plot pixels at a density that makes sense for the device available.

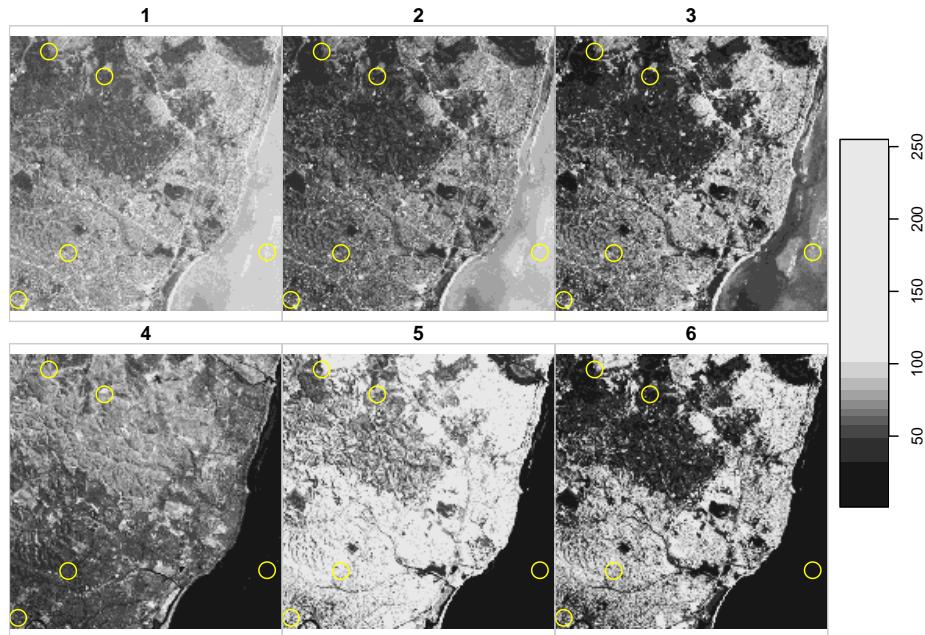


Figure 9.4: annotated multi-slice stars plot

9.3.2 Projections in base plots

The base `plot` method plots data with ellipsoidal coordinates using the equirectangular projection, using a latitude parameter equal to the middle latitude of the data bounding box (figure 9.2). To control this parameter, either a projection to another equirectangular can be applied before plotting, or the parameter `asp` can be set to override, e.g. `asp=1` would lead to plate carrée (figure 9.1 left). Subsequent plots need to be in the same coordinate reference system in order to make sense with overplotting, this is not being checked.

9.3.3 Colors and color breaks

In base plots, `nbreaks` can be used to set the number of color breaks, and `breaks` either to the numeric vector with actual breaks, or to a value for the `style` argument in `classInt::classIntervals`.

9.4 Maps with `ggplot2`

Package `ggplot2` (Wickham et al., 2021; Wickham, 2016) can create more complex and nicer looking graphs; it has a geometry `geom_sf` that was developed in

conjunction with the development of `sf`, and helps creating beautiful maps; an introduction to this is found in (Moreno and Basille, 2018), a first example is shown in figure 1.2. The code used for this plot is:

```
suppressPackageStartupMessages(library(tidyverse))
nc_32119 = st_transform(nc, 32119)
year_labels = c("SID74" = "1974 - 1978", "SID79" = "1979 - 1984")
nc_32119 %>% select(SID74, SID79) %>%
  pivot_longer(starts_with("SID")) -> nc_longer

ggplot() + geom_sf(data = nc_longer, aes(fill = value)) +
  facet_wrap(~ name, ncol = 1, labeller = labeller(name = year_labels)) +
  scale_y_continuous(breaks = 34:36) +
  scale_fill_gradientn(colors = sf.colors(20)) +
  theme(panel.grid.major = element_line(color = "white"))
```

where we see that two attributes had to be stacked (`pivot_longer`) before plotting them as facets: this is the idea of “tidy” data, and the `pivot_longer` method for `sf` objects automatically stacks the geometry column too.

Because `ggplot2` creates graphics *objects* before plotting them, it can control the coordinate reference system of all elements involved, and will transform or convert all subsequent objects to the coordinate reference system of the first. It will also draw a graticule for the (default) thin white lines on a grey background, and uses a datum (by default: WGS84) for this. `geom_sf()` can be combined with other geoms, for instance to allow for annotating plots.

For package `stars`, a `geom_stars` has, at the moment of writing this, rather limited scope: it uses `geom_sf` for map layout and vector data cubes, and adds `geom_raster` for regular rasters and `geom_rect` for rectilinear rasters. It down-samples if the user specifies a downsampling rate, but has no access to the screen dimensions to automatically choose a downsampling rate. This may be just enough, for instance figure 9.5 is created by the following commands:

```
library(ggplot2)
library(stars)
r = read_stars(system.file("tif/L7_ETMs.tif", package = "stars"))
ggplot() + geom_stars(data = r) +
  facet_wrap(~band) + coord_equal() +
  theme_void() +
  scale_x_discrete(expand=c(0,0)) +
  scale_y_discrete(expand=c(0,0)) +
  scale_fill_viridis_c()
```

More elaborate `ggplot2`-based plots with `stars` objects may be obtained using package `ggspatial` (Dunnington, 2021). Non-compatible but nevertheless

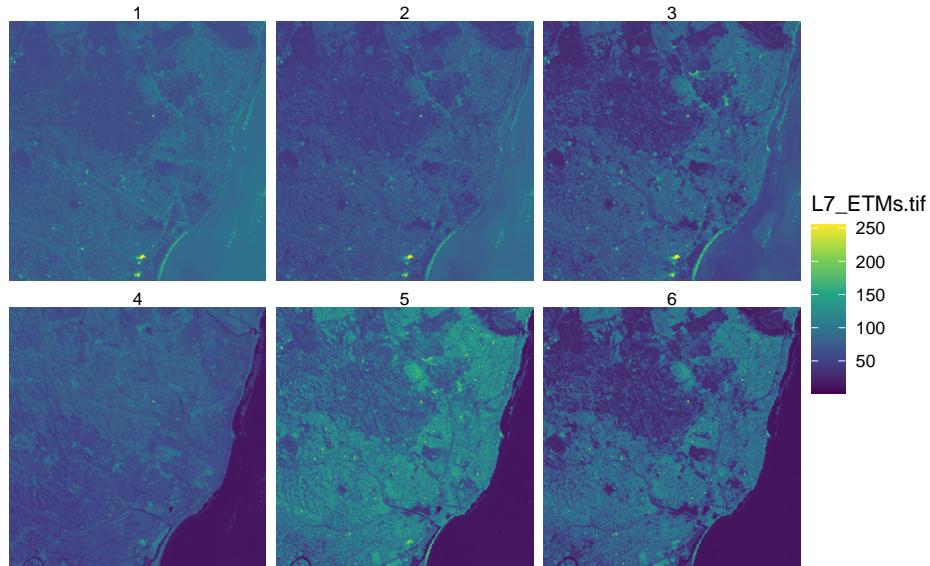


Figure 9.5: Simple raster plot with ggplot2

ggplot2-style plots can be created with `tmap`, a package dedicated to creating high quality maps

When combining several feature sets with varying coordinate reference systems, using `geom_sf`, all sets are transformed to the reference system of the first set. To get further control over the “base” coordinate reference system, `coord_sf` can be used. This allows for instance working in a projected system, while combining graphics elements that are *not* `sf` objects but regular `data.frames` with ellipsoidal coordinates associated to WGS84. A twitter thread by Claus Wilke illustrating this is found [here](#).

9.5 Maps with `tmap`

Package `tmap` (Tennekes, 2021, 2018) takes a fresh look on plotting spatial data in R; it resembles `ggplot2` in the sense that it composes graphics objects before printing, by building on the `grid` package, and by concatenating map elements with a `+` between them, but otherwise it is entirely independent from, and incompatible with, `ggplot2`. It has a number of options that allow for highly professional looking maps, and many defaults have been carefully chosen. To recreate figure 1.2, for instance, we use

```
library(tmap)
system.file("gpkg/nc.gpkg", package = "sf") %>%
```

```
read_sf() %>%
  st_transform('EPSG:32119') -> nc.32119
tm_shape(nc.32119) + tm_polygons(c("SID74", "SID79"))
```

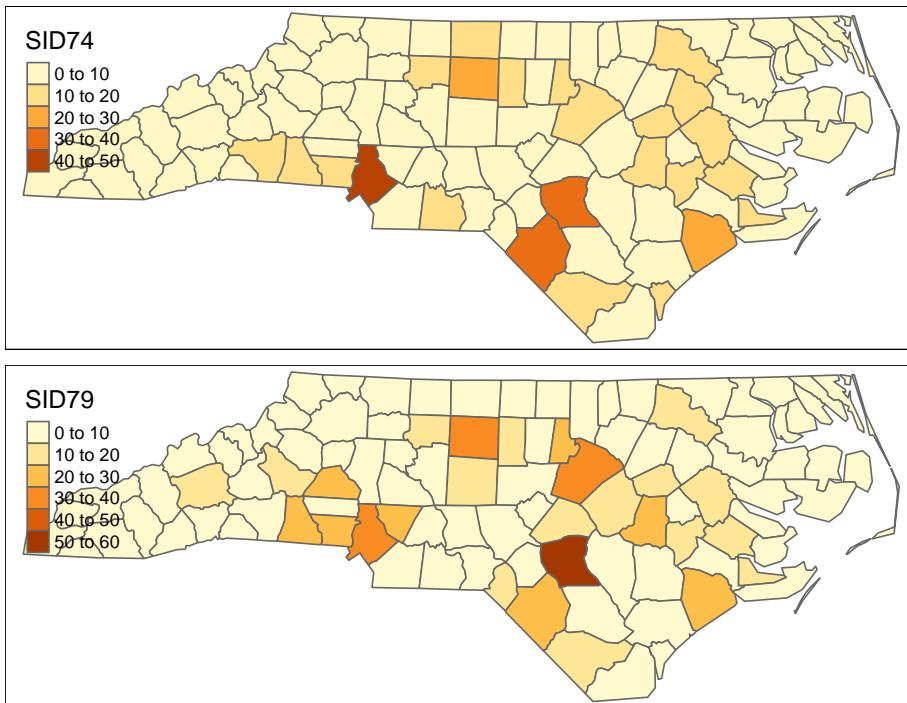


Figure 9.6: tmap: using ... with two attributes

to create figure 9.6 and

```
nc.32119 %>% select(SID74, SID79) %>%
  pivot_longer(starts_with("SID"), values_to = "SID") -> nc_longer
tm_shape(nc_longer) + tm_polygons("SID") + tm_facets(by = "name")
```

to create figure 9.7.

Package **tmap** also has support for **stars** objects, an example created with

```
tm_shape(r) + tm_raster()
```

is shown in figure 9.8. More examples of the use of **tmap** are given in Chapter 14.

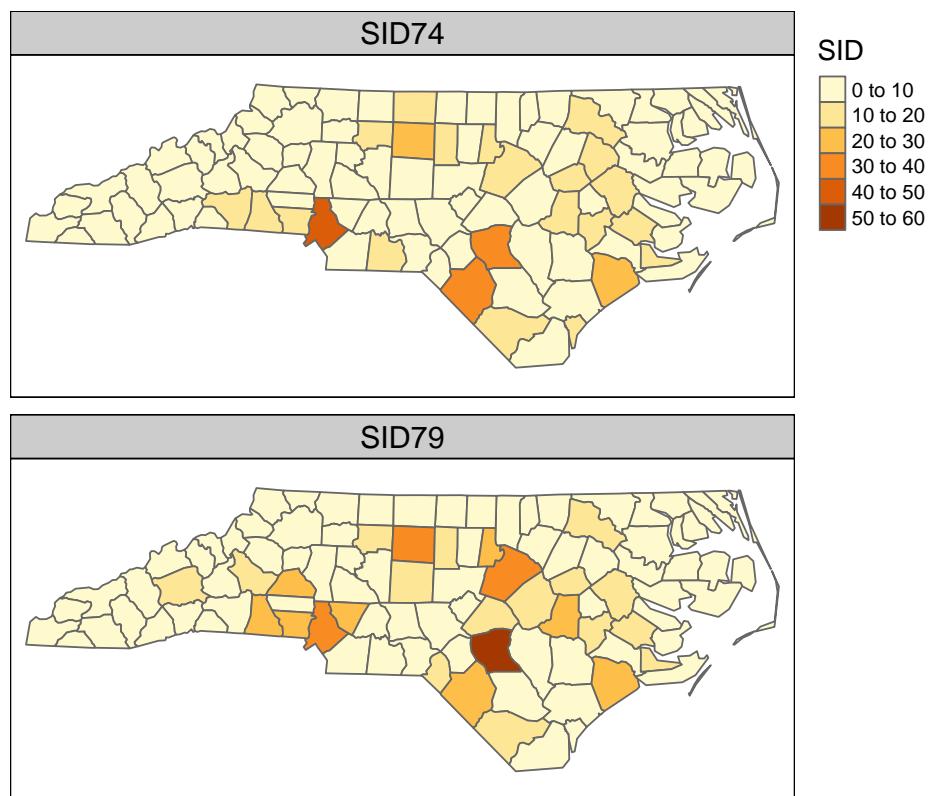


Figure 9.7: tmap: using ...

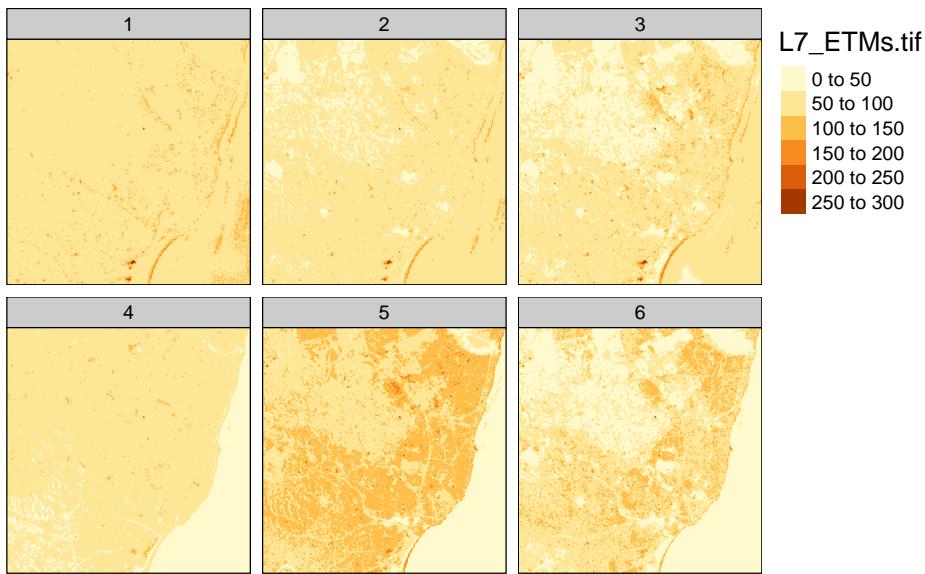


Figure 9.8: Simple raster plot with tmap

9.6 Interactive maps: `leaflet`, `mapview`, `tmap`

Interactive maps as shown in figure 1.3 can be created with R packages `leaflet`, `mapview` or `tmap`. `mapview` adds a number of capabilities to `leaflet` including a map legend, configurable pop-up windows when clicking features, support for raster data, and scalable maps with very large feature sets using the file geobuf file format, as well as facet maps that react synchronously to zoom and pan actions. Package `tmap` has the nice option that after giving

```
tmap_mode("view")
```

all usual `tmap` commands are applied to an interactive html/leaflet widget, whereas after

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

again all output is sent to R own graphics device.

9.7 Exercises

1. For the countries Indonesia and Canada, create individual plots using equirectangular, orthographic, and Lambert equal area projections, while choosing projection parameters sensible for the area.

2. Recreate the plot in figure 9.3 with `ggplot2` and with `tmap`.
3. Recreate the plot in figure 9.8 using the `viridis` color ramp.
4. View the interactive plot in figure 9.8 using the “view” (interactive) mode of `tmap`, and explore which interactions are possible; also explore adding `+ tm_facets(as.layers=TRUE)` and try switching layers on and off.

Part III

Models for Spatial Data

Chapter 10

Statistical modelling of spatial data

Spatial data almost always (and everywhere) has the property that it is spatially structured: observations done closeby in space tend to be more similar than observations done at larger distance from each other. This phenomenon, in the geography domain attributed to Waldo Tobler (as in “Waldo Tobler’s first law of geography”) was already noted by (Fisher et al., 1937) and was a motivation for developing randomized block design in agricultural experiments: allocating treatments randomly to blocks avoids that spatial structure gets mixed up (or: confounds) with a signal caused by the treatment.

The often heard argument that spatially structured data *means* that the data is spatially correlated, which would *exclude* estimation methods that assume independent observations is false. Correlation is a property of two random variables, and there are different ways in which spatial data can be approached with random variables: either the observation locations are random (leading to design-based inference) or the observed values are random (leading to model-based inference). The next section points out the difference between these two.

10.1 Design-based and model-based inference

Statistical inference means the action of estimating parameters about a population from sample data. Suppose we denote the variable of interest with $z(s)$, where z is the attribute value measured at location s , and we are interested in estimating the mean value of $z(s)$ over a domain D ,

$$z(s) = \frac{1}{|D|} \int_{u \in D} z(u) du,$$

with $|D|$ the area of D , from sample data $z(s_1), \dots, z(s_n)$.

Then, there are two possibilities to proceed: model-based, or design-based. A model-based approach considers $z(s)$ to be a realisation of a superpopulation $Z(s)$ (using capital letters to indicate random variables), and could for instance postulate a model for its spatial variability in the form of

$$Z(s) = m + e(s), \quad E(e(s)) = 0, \quad \text{Cov}(e(s)) = \Sigma(\theta)$$

which would require choosing the covariance function $\Sigma()$ and estimating its parameters θ from $z(s)$, and then computing a block kriging prediction $\hat{Z}(D)$ (section 12.5). This approach makes no assumptions about the sample $z(s)$, but of course it should allow for choosing the covariance function and estimating its parameters; inference is conditional to the validity of the postulated model.

Rather than assuming a superpopulation model, the design-based approach (De Grujter and Ter Braak, 1990; Brus, 2021a; Breidt et al., 2017) assumes randomness in the locations, which is justified (only) when using random sampling. It *requires* that the sample data were obtained by probability sampling, meaning that some form of spatial random sampling was used where all elements of $z(s)$ had a known and positive probability of being included in the sample obtained. The random process is that of sampling: $z(s_1)$ is a realisation of the random process $z(S_1)$, the first observation taken *over repeated random sampling*. Design-based estimators only need these inclusion probabilities to estimate mean values with standard errors. This means that for instance given a simple random sample, the unweighted sample mean is used to estimate the population mean, and no model parameters need to be fit.

The misconception here, as explained in [brus2021], is that this is only the case when working under model-based approaches: $Z(s_1)$ and $Z(s_2)$ may well be correlated (“model-dependent”), but although in a particular random sampling (realisation) $z(s_1)$ and $z(s_2)$ *may* be close in space, the corresponding random variables $z(S_1)$ and $z(S_2)$ considered over repeated random sampling are not close together, and are design-independent. Both situations can co-exist without contradiction, and are a consequence of choosing to work under one inference framework or the other.

The choice whether to work under a design-based or model-based framework depends on the purpose of the study and the data collection process. The model-based framework lends itself best for cases * where predictions are required for individual locations, or for areas too small to be sampled * when the available data were not collected using a known random sampling scheme (i.e., the inclusion probabilities are unknown, or are zero over particular areas or/and times) Design-based approaches are most suitable when * observations were collected using a spatial random sampling process * aggregated properties of the entire sample region (or sub-region) are needed. * estimates are required that are not sensitive to potential model misspecification, e.g. when needed for regulatory or legal purposes.

In case a sampling procedure is to be planned (De Gruijter et al., 2006), some form of spatial random sampling is definitely worth considering since it opens up the possibility of following both inference frameworks.

10.2 Predictive models with coordinates

In data science projects, coordinates may be seen as features in a larger set of predictors (or features, or covariates) and treated accordingly. There are some catches with doing so.

As usual when working with predictors, it is good to choose predictive methods that are not sensitive to shifts in origin or shifts in unit (scale). Assuming a two-dimensional problem, predictive models should also not be sensitive to arbitrary rotations of the x- and y- or latitude and longitude axes. For projected (2D, Cartesian) coordinates this can be assured e.g. by using polynomials of order n as $(x+y)^n$, rather than $(x)^n + (y)^n$; for a second order polynomial this involves including the term xy , so that an ellipsoidal-shape trend surface does not have to be aligned with the x - or y -axis. For a GAM model with spline components, one would use a spline in two dimensions rather than two independent splines in x and y . An exception to this “rule” is when e.g. a pure latitude effect is desired, for instance to account for solar energy influx.

When the area covered by the data is large, the difference between using ellipsoidal coordinates and projected coordinates will automatically become larger, and hence choosing one of both will have an effect on predictive modelling. For very large extents, e.g. global models, polynomials or splines in latitude and longitude will not make sense as they ignore the circular nature of longitude and the coordinate singularities at the poles. Here, spherical harmonics, base functions that are continuous on the sphere with increasing spatial frequencies can replace polynomials or be used as spline base functions.

In many cases, the spatial coordinates over which samples were collected also define the space over which predictions are made, setting them apart from other features. Many simple predictive approaches, including most machine learning methods, assume sample data to be independent. When samples are collected by spatially random sampling over the spatial target area, this assumption may be justified when working under a design-based context (Brus, 2021b). This context however treats the coordinate space as the variable over which we randomize, which affords predicting values for a new *randomly chosen* location but rules out making predictions for fixed locations; this implies that averages over areas over which samples were collected can be obtained, but not spatial interpolations. In case predictions for fixed locations are required, or in case data were not collected by spatial random sampling, a model-based approach (as taken in chapter 12) is needed and typically some form of spatial and/or temporal autocorrelation of residuals must be assumed.

A common case is where sample data are collected opportunistically (“whatever could be found”), and are then used in a predictive framework that does not weigh them. This has a consequence that the resulting model may be biased towards over-represented areas (in predictor space and/or in spatial coordinates space), and that simple (random) cross validation statistics may be over-optimistic when taken as performance measures for spatial prediction (Meyer and Pebesma, 2020). Adaptive cross validation measures, e.g. spatial cross validation may help getting more relevant measures for predictive performance.

10.3 Further reading

There is a large number of papers and books available on analysing and statistical modelling of spatial and spatiotemporal data, and a very large number of R packages help doing so. Several CRAN task views try to maintain an overview of the R packages, e.g. on

- spatial data (Bivand, 2021a)
- spatiotemporal data (Pebesma, 2021a)
- tracking data (Joo et al., 2021), see also (Joo et al., 2020)

The introductions to subsequent chapters contain more pointers to relevant literature references. Introductions to using the integrated nested Laplace approximation (INLA) for analysing spatial data are given in Blangiardo et al. (2013), Blangiardo and Cameletti (2015), and Gómez-Rubio (2020a). Krainski et al. (2018) combine the INLA approach with stochastic partial differential equations. Spatiotemporal Bayesian modelling of change of support problems is presented in Raim et al. (2021) and Raim et al. (2020).

Chapter 11

Point Pattern Analysis

Point pattern analysis is concerned with describing patterns of points over space, and making inference about the process that could have generated an observed pattern. The main focus here lies on the information carried in the locations of the points, and typically these locations are not controlled by sampling but a result of a process we're interested in, such as animal sightings, accidents, disease cases, or tree locations. This is opposed to geostatistical processes (chapter 12) where we have values of some phenomenon everywhere but observations limited to a set of locations *that we can control*, at least in principle. Hence, in geostatistical problems the prime interest is not in the observation locations but in estimating the value of the observed phenomenon at unobserved locations. Point pattern analysis typically assumes that for an observed area, all points are available, meaning that locations without a point are not unobserved as in a geostatistical process, but are observed and contain no point. In terms of random processes, in point processes locations are random variables, where in geostatistical processes the measured variable is a random field with locations fixed.

This chapter is confined to describing the very basics of point pattern analysis, using package **spatstat** (Baddeley et al., 2021), and related packages by the same authors. The **spatstat** book of Baddeley et al. (2015) gives a comprehensive introduction to point pattern theory and the use of the **spatstat** package family, which we will not try to copy. Inclusion of particular topics in this chapter should not be seen as an expression that these are more relevant than others. In particular, this chapter tries to illustrate interfaces existing between **spatstat** and the more spatial data science oriented packages **sf** and **stars**. A further books that introduces point patterns analysis is Stoyan et al. (2017). An R package for analysing spatiotemporal point processes is discussed in Gabriel et al. (2013).

Important concepts of point patterns analysis are the distinction between a point *pattern* and a point *process*: the latter is the stochastic process that,

when sampled, generates a point pattern. A data set is always a point pattern, and inference involves figuring out what kind of process could have generated a pattern like the one we observed. Properties of a spatial point process are:

- first order properties or intensity function, which measures the number of points per area unit; this function is spatially varying for a *inhomogeneous* point process. *
- second order properties, e.g. pairwise interactions: given a constant or varying intensity function, are points distributed independently *from one another*, or do they tend to attract each other (clustering) or repulse each other (appear regularly distributed, compared to independence)

11.1 Observation window

Point patterns have an observation window. Consider the points randomly generated randomly by

```
library(sf)
n = 30
xy = data.frame(x = runif(n), y = runif(n)) %>% st_as_sf(coords = c("x", "y"))
```

then these points are (by construction) uniformly distributed, or completely spatially random, over the domain $[0, 1] \times [0, 1]$. For a larger domain, they are not uniform, for the two square windows `w1` and `w2` created by

```
w1 = st_bbox(c(xmin = 0, ymin = 0, xmax = 1, ymax = 1)) %>%
      st_as_sfc()
w2 = st_sfc(st_point(c(1, 0.5))) %>% st_buffer(1.2)
```

this is shown in figure 11.1.

Point patterns in `spatstat` are objects of class `ppp` that contain points and an observation window (an object of class `owin`). We can create a `ppp` from points by

```
suppressPackageStartupMessages(library(spatstat))
as.ppp(xy)
# Planar point pattern: 30 points
# window: rectangle = [0.009, 0.999] x [0.103, 0.996] units
```

where we see that the bounding box of the points is used as observation window when no window is specified. If we add a polygonal geometry as the first feature of the dataset, then this is used as observation window:

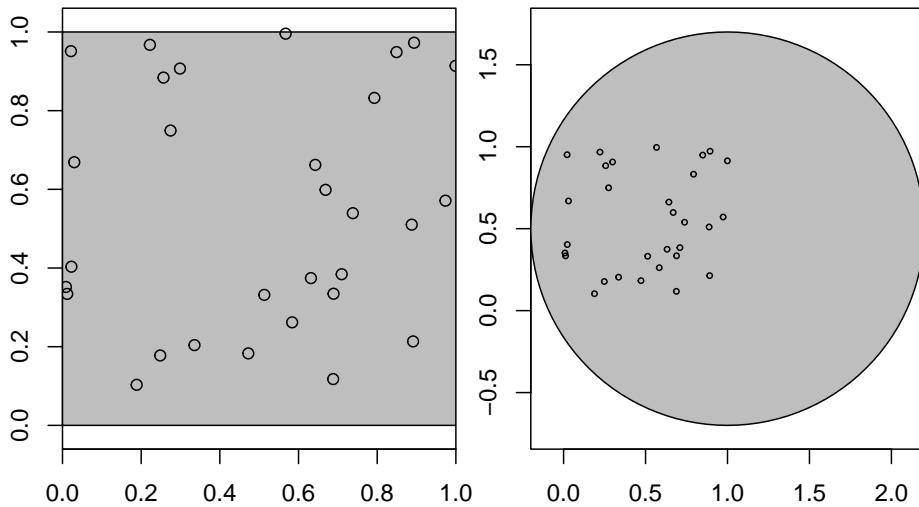


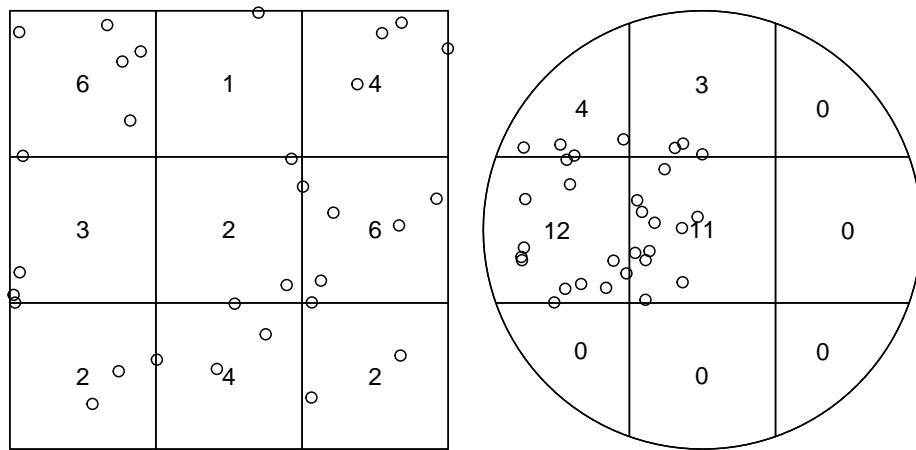
Figure 11.1: Depending on the observation window (grey), the same point pattern can appear completely spatially random (left), or clustered (right)

```
(pp1 = c(w1, st_geometry(xy)) %>% as.ppp())
# Planar point pattern: 30 points
# window: polygonal boundary
# enclosing rectangle: [0, 1] x [0, 1] units
c1 = st_buffer(st_centroid(w2), 1.2)
(pp2 = c(c1, st_geometry(xy)) %>% as.ppp())
# Planar point pattern: 30 points
# window: polygonal boundary
# enclosing rectangle: [-0.2, 2.2] x [-0.7, 1.7] units
```

To test for homogeneity, one could carry out a quadrat count, using an appropriate quadrat layout (a 3×3 layout is shown in figure 11.2)

and carry out a χ^2 test on these counts:

```
quadrat.test(pp1, nx=3, ny=3)
# Warning: Some expected counts are small; chi^2 approximation may
# be inaccurate
#
# Chi-squared test of CSR using quadrat counts
#
# data: pp1
# X2 = 8, df = 8, p-value = 0.9
# alternative hypothesis: two.sided
#
```

Figure 11.2: 3×3 quadrat counts for the two point patterns

```
# Quadrats: 9 tiles (irregular windows)
quadrat.test(pp2, nx=3, ny=3)
# Warning: Some expected counts are small; chi^2 approximation may
# be inaccurate
#
# Chi-squared test of CSR using quadrat counts
#
# data: pp2
# X2 = 43, df = 8, p-value = 2e-06
# alternative hypothesis: two.sided
#
# Quadrats: 9 tiles (irregular windows)
```

where we should take the p-values with a large grain of salt because we have too small expected counts.

Kernel densities can be computed using `density`, where kernel shape and bandwidth can be controlled. Here, cross validation is used by function `bw.diggle` to specify the bandwidth parameter `sigma`; plots are shown in figure 11.3.

```
den1 <- density(pp1, sigma = bw.diggle)
den2 <- density(pp2, sigma = bw.diggle)
```

The density maps created this way are obviously raster images, and we can convert them into stars object, e.g. by

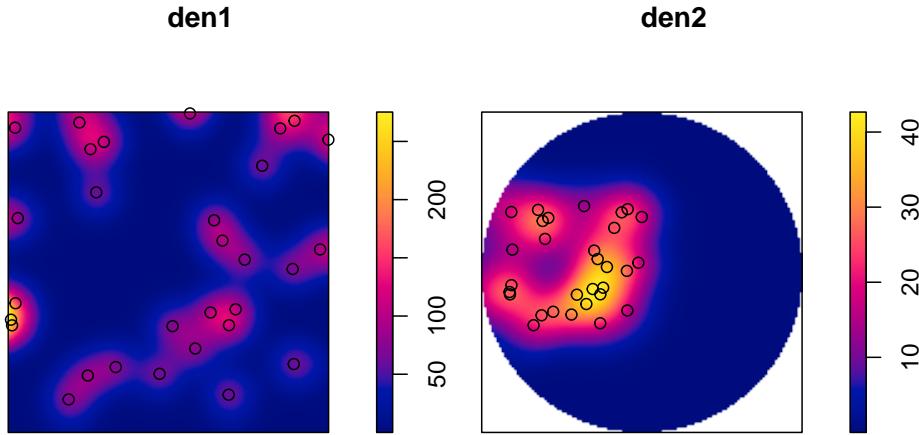


Figure 11.3: Kernel densities for both point patterns

```

library(stars)
s1 = st_as_stars(den1)
(s2 = st_as_stars(den2))
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# v 1.03e-14 0.000153 0.304 6.77 13.1 42.7 3492
# dimension(s):
#   from to offset delta refsys point values x/y
# x    1 128 -0.2 0.01875     NA     NA    NULL [x]
# y    1 128 -0.7 0.01875     NA     NA    NULL [y]

```

and we can verify that the area under the density surface is similar to the sample size (30), by

```

sum(s1[[1]], na.rm = TRUE)*st_dimensions(s1)$x$delta^2
# [1] 29
sum(s2[[1]], na.rm = TRUE)*st_dimensions(s2)$x$delta^2
# [1] 30.7

```

More exciting applications involve e.g. modelling the density surface as a function of external variables. Suppose we want to model the density of `pp2` as a Poisson point process (meaning that points do not interact with each other), where the intensity is a function of distance to the center of the “cluster”, and these distance are available in a `stars` object:

```
pt = st_sfc(st_point(c(0.5, 0.5)))
s2$dist = st_distance(st_as_sf(s2, as_points=TRUE, na.rm = FALSE), pt)
```

we can then model the densities using `ppm`, where the *name* of the point pattern object is used as the left-hand-side of the `formula`:

```
(m = ppm(pp2 ~ dist, data = list(dist = as.im(s2["dist"]))))
# Nonstationary Poisson process
#
# Log intensity: ~dist
#
# Fitted trend coefficients:
# (Intercept)      dist
#        4.54       -4.25
#
#           Estimate  S.E. CI95.lo CI95.hi Ztest  Zval
# (Intercept)    4.54 0.341     3.87     5.21 *** 13.32
# dist          -4.25 0.701    -5.62    -2.88 *** -6.06
```

The returned object is of class `ppm`, and can be plotted: figure 11.4 shows the predicted surface, the prediction standard error can also be plotted.

Fitted trend

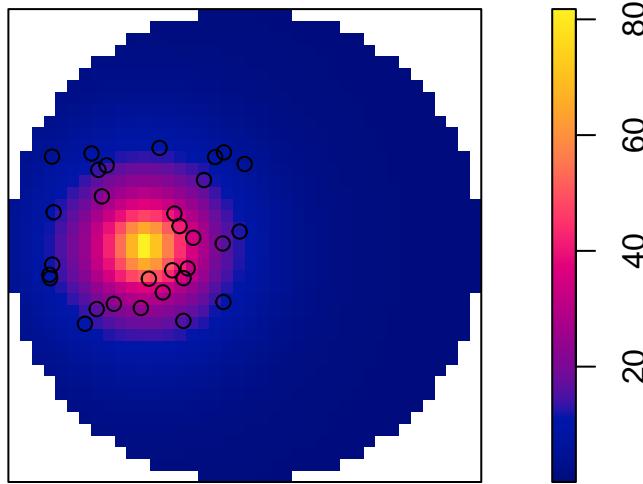


Figure 11.4: Predicted densities of a `ppm` model

The model also has a `predict` method, which returns an `im` object that can be converted into a `stars` object by

```

predict(m, covariates = list(dist = as.im(s2["dist"]))) %>%
  st_as_stars()
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# v  0.0694    0.527   2.12 6.62      7.3 89.9 3492
# dimension(s):
#   from to offset delta refsys point values x/y
# x    1 128    -0.2 0.01875     NA     NA    NULL [x]
# y    1 128    -0.7 0.01875     NA     NA    NULL [y]

```

11.2 Coordinate reference systems

All routines in **spatstat** are layed out for two-dimensional data with Cartesian coordinates. If we try to convert an object with ellipsoidal coordinates, we get an error:

```

system.file("gpkg/nc.gpkg", package = "sf") %>%
  read_sf() %>%
  st_geometry() %>%
  st_centroid() %>%
  as.ppp()
# Error: Only projected coordinates may be converted to spatstat class objects

```

Also, when converting to a **spatstat** data structure (e.g. to a ppp, create a density image, convert back to stars) we loose the coordinate reference system we started with. It can be set back e.g. by using **st_set_crs**.

11.3 Marked point patterns, points on linear networks

A few more data types can be converted to and from **spatstat**. Marked point patterns are point patterns that have a “mark”, which is either a categorical label or a numeric label for each point. A dataset available in **spatstat** with marks is the **longleaf** pines dataset, containing diameter at breast height as numeric mark:

```

longleaf
# Marked planar point pattern: 584 points
# marks are numeric, of storage type 'double'
# window: rectangle = [0, 200] x [0, 200] metres

```

```
ll = st_as_sf(longleaf)
head(ll)
# Simple feature collection with 6 features and 2 fields
# Geometry type: GEOMETRY
# Dimension: XY
# Bounding box: xmin: 0 ymin: 0 xmax: 200 ymax: 200
# CRS: NA
#   mark  label           geom
# 1  NA    point      POINT (200 8.8)
# 2  32.9 point      POINT (199 10)
# 3  53.5 point      POINT (194 22.4)
# 4  68.0 point      POINT (168 35.6)
# 5  17.7 point      POINT (184 45.4)
```

Values can be converted back to ppp with

```
as.ppp(ll)
# Warning in as.ppp.sf(ll): only first attribute column is used for
# marks
# Marked planar point pattern: 584 points
# marks are numeric, of storage type 'double'
# window: polygonal boundary
# enclosing rectangle: [0, 200] x [0, 200] units
```

Line segments, in **spatstat** objects of class **psp** can be converted back and forth to simple feature with LINESTRING geometries following a POLYGON feature with the observation window, as in

```
print(st_as_sf(copper$SouthLines), n = 5)
# Simple feature collection with 91 features and 1 field
# Geometry type: GEOMETRY
# Dimension: XY
# Bounding box: xmin: -0.335 ymin: 0.19 xmax: 35 ymax: 158
# CRS: NA
# First 5 features:
#   label           geom
# 1  window POLYGON ((-0.335 0.19, 35 0...
# 2 segment LINESTRING (3.36 0.19, 10.4...
# 3 segment LINESTRING (12.5 0.263, 11....
# 4 segment LINESTRING (11.2 0.197, -0....
# 5 segment LINESTRING (6.35 12.8, 16.5...
```

Finally, point patterns on linear networks, in **spatstat** represented by **lpp** objects, can be converted to **sf** by

```
print(st_as_sf(chicago), n = 5)
# Simple feature collection with 620 features and 4 fields
# Geometry type: GEOMETRY
# Dimension: XY
# Bounding box: xmin: 0.389 ymin: 153 xmax: 1280 ymax: 1280
# CRS: NA
# First 5 features:
#   label seg tp marks          geom
# 1 window NA NA <NA> POLYGON ((0.389 153, 1282 1...
# 2 segment NA NA <NA> LINESTRING (0.389 1254, 110...
# 3 segment NA NA <NA> LINESTRING (110 1252, 111 1...
# 4 segment NA NA <NA> LINESTRING (110 1252, 198 1...
# 5 segment NA NA <NA> LINESTRING (198 1277, 198 1...
```

where we only see the first five features; the points are also in this object, as variable `label` indicates

```
table(st_as_sf(chicago)$label)
#
#   point segment window
#     116      503       1
```

Potential information about network *structure*, as of which LINESTRING is connected to others, is not present in the `sf` object. Package `sfnetworks` (van der Meer et al., 2021) would be a candidate package to hold such information, or e.g. to pass on network data imported from OpenStreetMaps to `spatstat`.

11.4 Spatial sampling and simulating a point process

Package `sf` contains an `st_sample` method that samples points from MULTIPOLYPOINT, linear or polygonal geometries, using different spatial sampling strategies. It natively supports strategies “random”, “hexagonal” and “regular”, where “regular” refers to sampling on a square regular grid, and “hexagonal” essentially gives a triangular grid. For type “random”, it can return exactly the number of requested points, for other types this is approximate.

`st_sample` also interfaces point process simulation functions of `spatstat`, when other values for sampling type are chosen. For instance the `spatstat` function `rThomas` is invoked when setting `type = Thomas` (figure 11.5):

```

kappa = 30 / st_area(w2) # intensity
th = st_sample(w2, kappa = kappa, mu = 3, scale = 0.05, type = "Thomas")
nrow(th)
# [1] 82

```

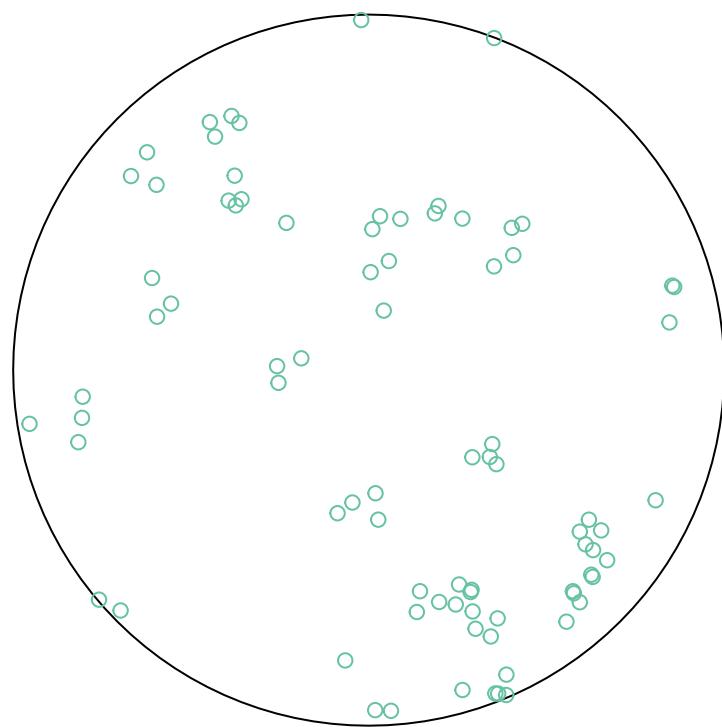


Figure 11.5: Thomas process with $\mu = 3$ and $\text{scale} = 0.05$

The help function obtained by `?rThomas` details the meaning of the parameters `kappa`, `mu` and `scale`. Simulating point processes means that the intensity is given, not the sample size. The sample size within the observation window obtained this way is a random variable.

11.5 Simulating points on the globe

Another spatial random sampling type supported by `sf` natively (in `st_sample`) is simulation of random points on the sphere. An example of this is shown in figure 11.6, where points were constrained to those in oceans.

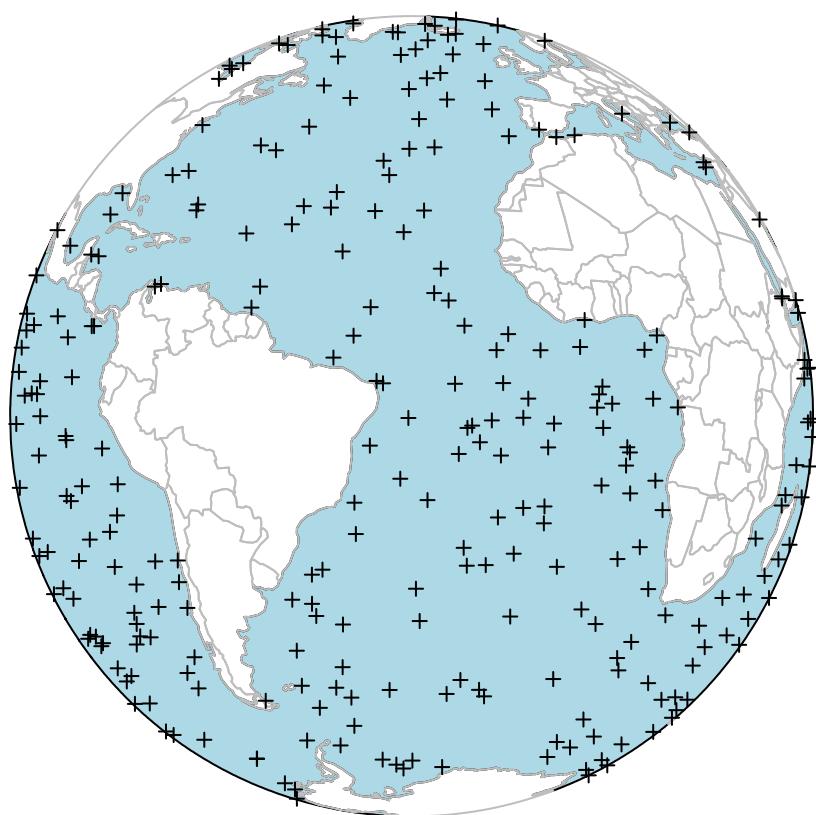


Figure 11.6: Points randomly sampled over the oceans

11.6 Exercises

1. After loading `spatstat`, recreate the plot obtained by `plot(longleaf)` by using `ggplot2` and `geom_sf()`, and by `sf:::plot()`.
2. Convert the sample locations of the NO₂ data used in chapter 12 to a `ppp` object, with a proper window.
3. Compute and plot the density of the NO₂ dataset, import the density as a `stars` object and compute the volume under the surface.

Chapter 12

Spatial Interpolation

Spatial interpolation is the activity of estimating values spatially continuous variables for spatial locations where they have not been observed, based on observations. The statistical methodology for spatial interpolation, called geostatistics, is concerned with the modelling, prediction and simulation of spatially continuous phenomena. The typical problem is a missing value problem: we observe a property of a phenomenon $Z(s)$ at a limited number of sample locations $s_i, i = 1, \dots, n$, and are interested in the property value at all locations s_0 covering an area of interest, so we have to predict it for unobserved locations. This is also called *kriging*, or Gaussian Process prediction. In case $Z(s)$ contains a white noise component ϵ , as in $Z(s) = S(s) + \epsilon(s)$ (possibly reflecting measurement error) an alternative but similar goal is to predict $S(s)$, which may be called spatial filtering or smoothing.

In this chapter we will show simple approaches for handling geostatistical data, will demonstrate simple interpolation methods, explore modelling spatial correlation, spatial prediction and simulation. We will use package `gstat` (Pebesma and Graeler, 2021; Pebesma, 2004), which offers a fairly wide palette of models and options for non-Bayesian geostatistical analysis. Bayesian methods with R implementations are found in e.g. Diggle et al. (1998), Diggle and Ribeiro Jr. (2007), Blangiardo and Cameletti (2015), and Wikle et al. (2019). An overview and comparisons of methods for large datasets is given in Heaton et al. (2018).

12.1 A first dataset

We can read NO₂ data, which is prepared in chapter 13, from package `gstat` using

```

library(tidyverse)
no2 = read_csv(system.file("external/no2.csv", package = "gstat"))
#
# -- Column specification -----
# cols(
#   .default = col_character(),
#   station_start_date = col_date(format = ""),
#   station_end_date = col_logical(),
#   station_longitude_deg = col_double(),
#   station_latitude_deg = col_double(),
#   station_altitude = col_double(),
#   lau_level1_code = col_double(),
#   lau_level2_code = col_double(),
#   NO2 = col_double()
# )
# i Use `spec()` for the full column specifications.

```

and convert it into an `sf` object using

```

library(sf)
crs = st_crs("EPSG:32632")
no2.sf = st_as_sf(no2, coords = c("station_longitude_deg", "station_latitude_deg"), crs = st_transform(crs))

```

Next, we can load country boundaries and plot these data using `ggplot`, shown in figure 12.1.

```

data(air, package = "spacetime") # this loads German boundaries into DE_NUTS1
de <- st_transform(st_as_sf(DE_NUTS1), crs)

```

If we want to interpolate, we first need to decide where. This is typically done on a regular grid covering the area of interest. Starting with the country outline `de` we can create a regular grid with 10 km grid cells (pixels) by

```

# build a grid over Germany:
library(stars)
st_bbox(de) %>%
  st_as_stars(dx = 10000) %>%
  st_crop(de) -> grd
grd
# stars object with 2 dimensions and 1 attribute
# attribute(s):
#           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# values      0       0     0    0       0     0 2076

```

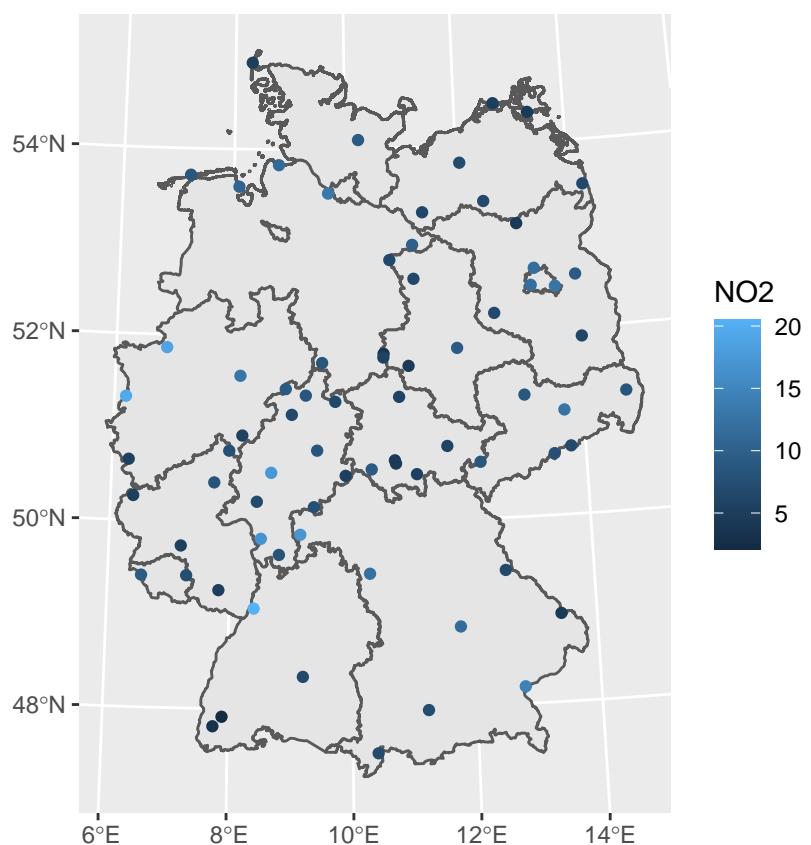


Figure 12.1: Mean NO_2 concentrations in air for rural background stations in Germany over 2017

```
# dimension(s):
#   from to offset delta           refsys point values x/y
# x    1 65 280741 10000 WGS 84 / UTM z... NA NULL [x]
# y    1 87 6101239 -10000 WGS 84 / UTM z... NA NULL [y]
```

Here, we chose grid cells to be not too fine, so that we still see them in plots.

Perhaps the simplest interpolation method is inverse distance weighted interpolation, which is a weighted average, using weights inverse proportional to distances from the interpolation location:

$$\hat{z}(s_0) = \frac{\sum_{i=1}^n w_i z(s_i)}{\sum_{i=1}^n w_i}$$

with $w_i = |s_0 - s_i|^p$, and the inverse distance power typically taken as 2, or optimized using cross validation. We can compute inverse distance interpolated values using `gstat::idw`,

```
library(gstat)
#
# Attaching package: 'gstat'
# The following object is masked from 'package:spatstat.core':
#
#       idw
i = idw(NO2~1, no2.sf, grd)
# [inverse distance weighted interpolation]
```

and plot them in figure 12.2.

```
ggplot() + geom_stars(data = i, aes(fill = var1.pred, x = x, y = y)) +
  geom_sf(data = st_cast(de, "MULTILINESTRING"), aes(col = 'grey')) +
  geom_sf(data = no2.sf)
```

12.2 Sample variogram

In order to make spatial predictions using geostatistical methods, we first need to identify a model for the mean and for the spatial correlation. In the simplest model, $Z(s) = m + e(s)$, the mean is an unknown constant m , and in this case the spatial correlation can be modelled using the variogram, $\gamma(h) = 0.5E(Z(s) - Z(s + h))^2$. For processes with a finite variance $C(0)$, the variogram is related to the covariogram or covariance function through $\gamma(h) = C(0) - C(h)$.

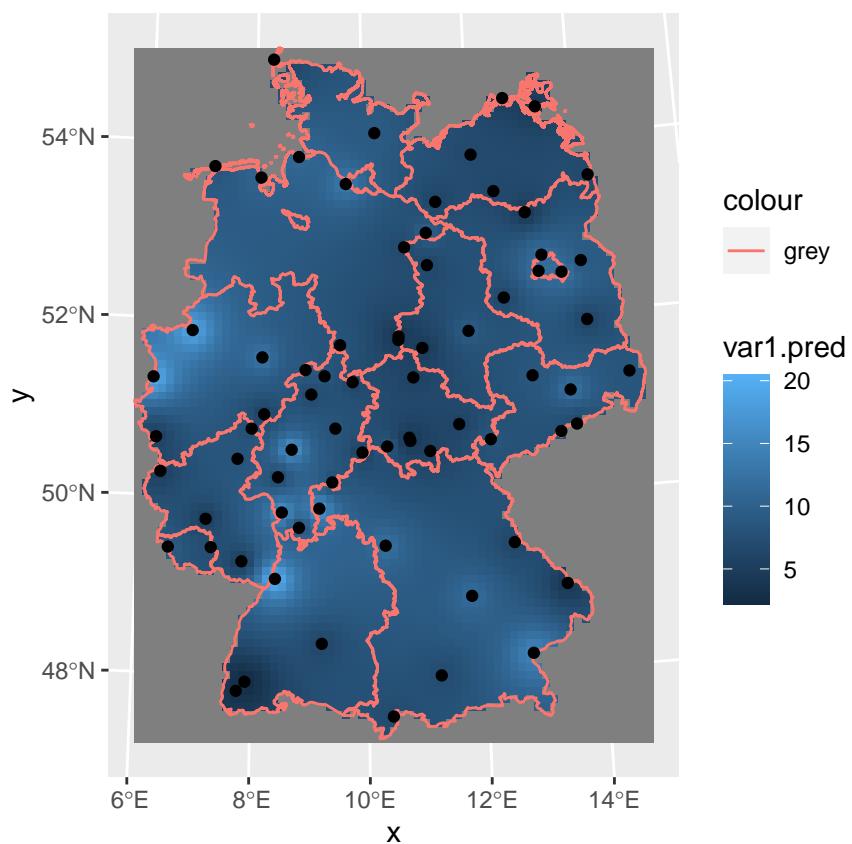


Figure 12.2: Inverse distance weighted interpolated values for NO_2 over Germany

The sample variogram is obtained by computing estimates of $\gamma(h)$ for distance intervals, $h_i = [h_{i,0}, h_{i,1}]$:

$$\hat{\gamma}(h_i) = \frac{1}{2N(h_i)} \sum_{j=1}^{N(h_i)} (z(s_i) - z(s_i + h'))^2, \quad h_{i,0} \leq h' < h_{i,1}$$

with $N(h_i)$ the number of sample pairs available for distance interval h_i . Function `gstat::variogram` computes sample variograms,

```
v = variogram(NO2~1, no2.sf)
```

and the result of plotting this is shown in figure 12.3.

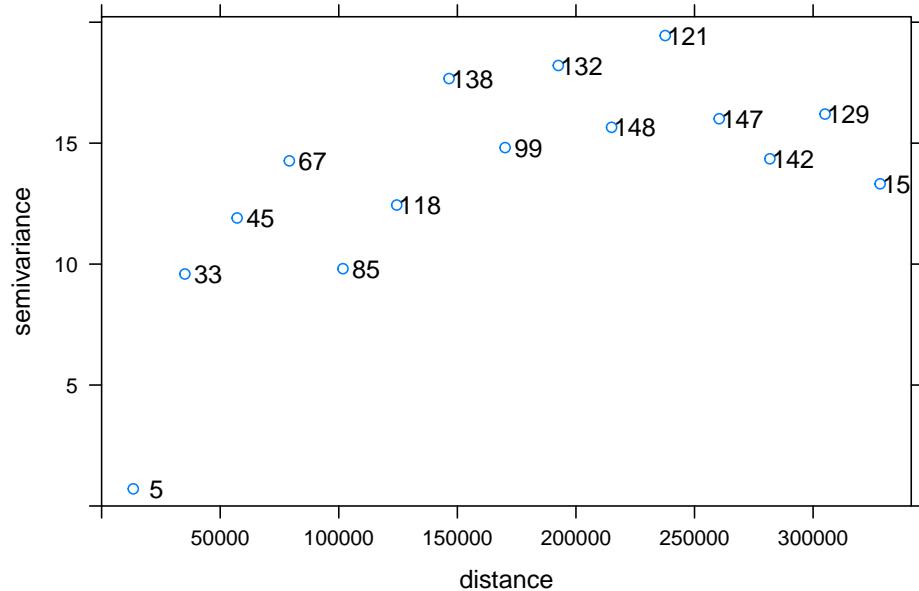


Figure 12.3: Sample variogram plot

Function `variogram` chooses default for maximum distance (`cutoff`: one third of the length of the bounding box diagonal) and (constant) interval widths (`width`: cutoff divided by 15). These defaults can be changed, e.g. by

```
library(gstat)
v0 = variogram(NO2~1, no2.sf, cutoff = 100000, width = 10000)
```

shown in figure 12.4.

```
plot(v0, plot.numbers = TRUE)
```

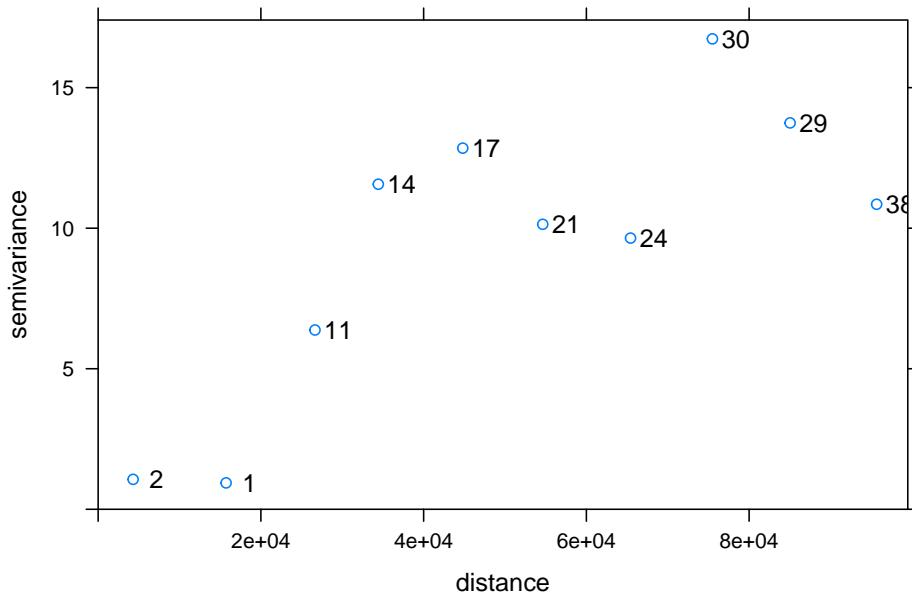


Figure 12.4: Sample variogram plot with adjusted cutoff and lag width

Note that the formula `N02~1` is used to select the variable of interest from the data file (`N02`), and to specify the mean model: `~1` refers to an intercept-only (unknown, constant mean) model.

12.3 Fitting variogram models

In order to progress toward spatial predictions, we need a variogram *model* $\gamma(h)$ for (potentially) all distances h , rather than the set of estimates derived above: in case we would for instance connect these estimates with straight lines, or assume they reflect constant values over their respective distance intervals, this would lead to statistical models with non-positive covariance matrices, which is a complicated way of expressing that you are in a lot of trouble.

To avoid these troubles we fit parametric models $\gamma(h)$ to the estimates $\hat{\gamma}(h_i)$, where we take h_i as the mean value of all the h' values involved in estimating $\hat{\gamma}(h_i)$. For this, when we fit a model like the exponential variogram, fitted by

```
v.m = fit.variogram(v, vgm(1, "Exp", 50000, 1))
```

and shown in figure 12.5.

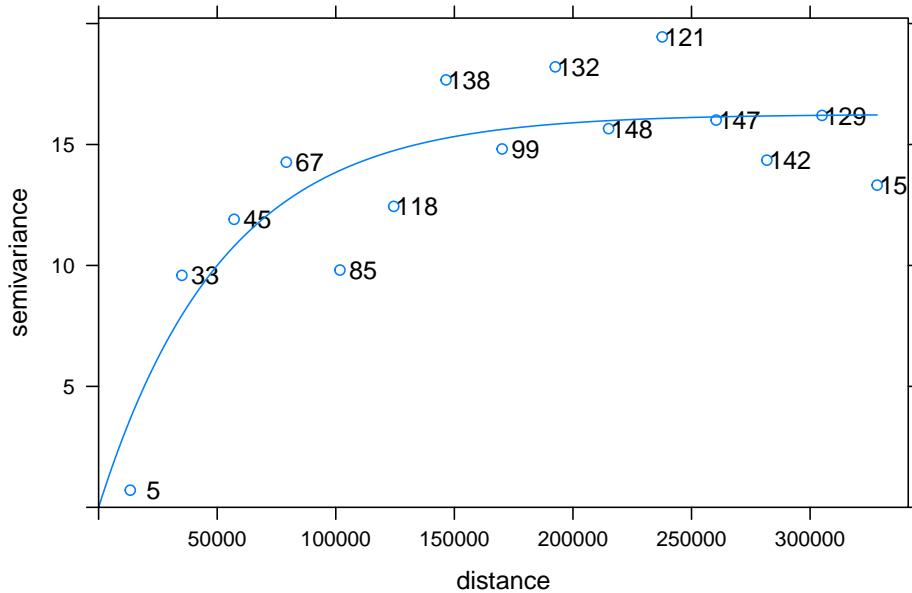


Figure 12.5: Sample variogram with fitted model

The fitting is done by weighted least squares, minimizing $\sum_{i=1}^n w_i(\gamma(h_i) - \hat{\gamma}(h_i))^2$, with w_i by default equal to $N(h_i)/h^2$, other fitting schemes are available through argument `fit.method`.

12.4 Kriging interpolation

Typically, when we interpolate a variable, we do that on points on a regular grid covering the target area. We first create a `stars` object with a raster covering the target area, and NA's outside it:

Kriging involves the prediction of $Z(s_0)$ at arbitrary locations s_0 . We can krigie NO₂ by using `gstat::krige`, with the model for the trend, the data, the prediction grid, and the variogram model (figure 12.6):

```
k = krige(NO2~1, no2.sf, grd, v.m)
# [using ordinary kriging]
```

12.5 Areal means: block kriging

Computing areal means can be done in several ways. The simples is to take the average of point samples falling inside the target polygons:

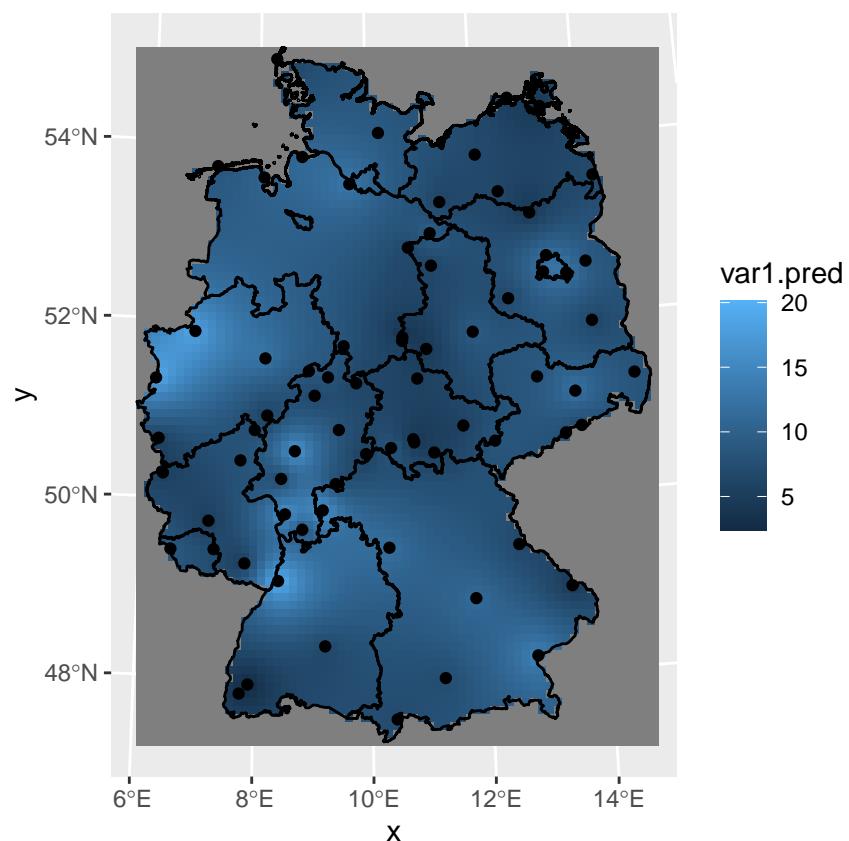


Figure 12.6: Kriged NO_2 concentrations over Germany

```
a = aggregate(no2.sf[["NO2"]], by = de, FUN = mean)
```

A more complicated way is to use *block kriging* (Journel and Huijbregts, 1978), which uses *all* the data to estimate the mean of the variable over the target area. With `krige`, this can be done by giving the target areas (polygons) as the `newdata` argument:

```
b = krige(NO2~1, no2.sf, de, v.m)
# [using ordinary kriging]
```

we can now merge the two maps into a single object to create a single plot (figure 12.7):

```
b$sample = a$NO2
b$kriging = b$var1.pred
```

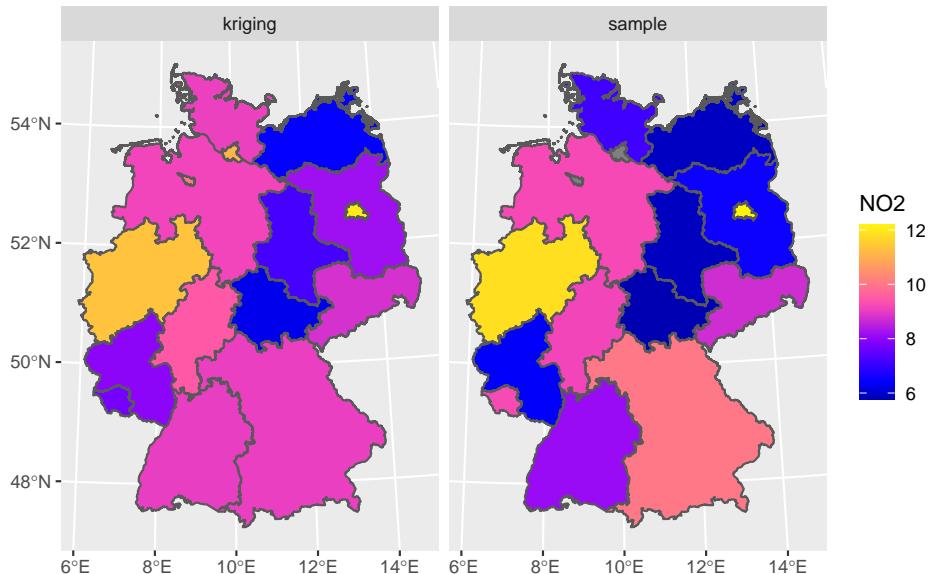


Figure 12.7: Aggregated NO₂ values from simple averaging (left) and block kriging (right)

We see that the signal is similar, but that the simple means are more variable than the block kriging values; this may be due to the smoothing effect of kriging: data points outside the target area are weighted, too.

To compare the standard errors of means, for the sample mean we can get a rough guess of the standard error by $\sqrt{(\sigma^2/n)}$:

```
SE = function(x) sqrt(var(x)/length(x))
a = aggregate(no2.sf[["NO2"]], de, SE)
```

which would have been the actual estimate in design-based inference if the sample was obtained by spatially random sampling. The block kriging variance is the model-based estimate, and is a by-product of kriging. We combine and rename the two:

```
b$sample = a$NO2
b$kriging = sqrt(b$var1.var)

{r aggrSE,fig.cap = 'Standard errors for mean NO2 values obtained
by simple averaging (left) and block kriging (right)', b %>%
select(sample, kriging) %>%
pivot_longer(1:2, names_to
= "var", values_to = "NO2") -> b2 ggplot() + geom_sf(data = b2,
mapping = aes(fill = NO2)) + facet_wrap(~var) + scale_fill_gradientn(colors
= sf.colors(20)) where we see that the simple approach gives clearly more
variability and mostly larger values for prediction errors of areal means,
compared to block kriging.
```

12.6 Conditional simulation

In case one or more conditional realisation of the field $Z(s)$ are needed rather than their conditional mean, we can obtain this by *conditional simulation*. A reason for wanting this may be the need to estimate areal mean values of $g(Z(s))$ with $g(\cdot)$ a non-linear function; a simple example is the areal fraction where $Z(s)$ exceeds a threshold.

The standard approach used by `gstat` is to use the sequential simulation algorithm for this. This is a simple algorithm that randomly steps through the prediction locations and at each location:

- carries out a kriging prediction
- draws a random variable from the normal distribution with mean and variance equal to the kriging variance
- adds this value to the conditioning dataset
- finds a new random simulation location

until all locations have been visited.

This is carried out by `gstat::krige` when `nsim` is set to a positive value. In addition, it is good to constrain `nmax`, the (maximum) number of nearest neighbours to include in kriging estimation, because the dataset grows each step, leading

otherwise quickly to very long computing times and large memory requirements (figure 12.8):

```
s = krig(N02~1, no2.sf, grd, v.m, nmax = 30, nsim = 10)
# drawing 10 GLS realisations of beta...
# [using conditional Gaussian simulation]
```

```
# Loading required package: viridisLite
#
# Attaching package: 'viridis'
# The following object is masked from 'package:scales':
#
#     viridis_pal
# The following object is masked from 'package:maps':
#
#     unemp
```

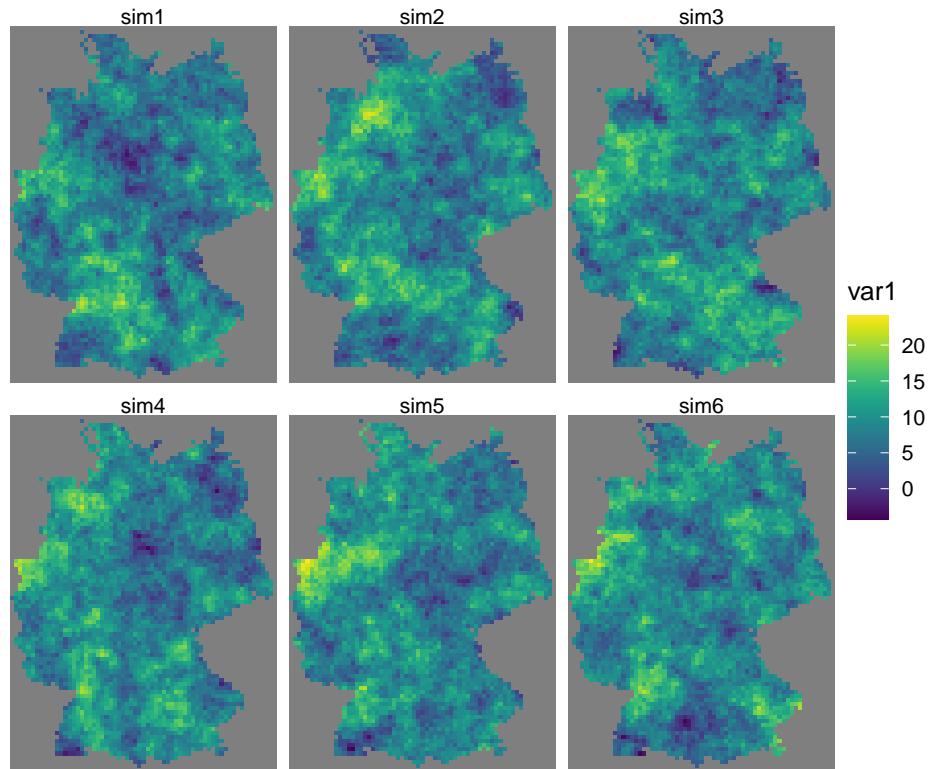


Figure 12.8: Ten conditional simulations for NO₂ values

Alternative methods for conditional simulation have recently been added to

`gstat`, and include `krigeSimCE` implementing the circular embedding method (Davies and Bryant, 2013), and `krigeSTSimTB` implementing the turning bands method (Schlather, 2011). These are of particular interest for larger datasets or conditional simulations of spatiotemporal data.

12.7 Trend models

Kriging and conditional simulation, as used so far in this chapter, assume that all spatial variability is a random process, characterized by a spatial covariance model. In case we have other variables that are meaningfully correlated with the target variable, we can use them in a linear regression model for the trend,

$$Z(s) = \sum_{j=0}^p \beta_j X_p(s) + e(s)$$

with $X_0(s) = 1$ and β_0 an intercept, but with the other β_j regression coefficients. This typically reduces both the spatial correlation in the residual $e(s)$, as well as its variance, and leads to more accurate predictions and more similar conditional simulations.

12.7.1 A population grid

As a potential predictor for NO₂ in the air, we use population density. NO₂ is mostly caused by traffic, and traffic is stronger in more densely populated areas. Population density is obtained from the 2011 census, and is downloaded as a csv file with the number of inhabitants per 100 m grid cell. We can aggregate these data to the target grid cells by summing the inhabitants:

```
v = vroom::vroom("aq/pop/Zensus_Bevoelkerung_100m-Gitter.csv")
# Rows: 35785840 Columns: 4
# -- Column specification --
# Delimiter: ";"
# chr (1): Gitter_ID_100m
# dbl (3): x_mp_100m, y_mp_100m, Einwohner
#
# i Use `spec()` to retrieve the full column specification for this data.
# i Specify the column types or set `show_col_types = FALSE` to quiet this message.
v %>% filter(Einwohner > 0) %>%
  select(-Gitter_ID_100m) %>%
  st_as_sf(coords = c("x_mp_100m", "y_mp_100m"), crs = 3035) %>%
  st_transform(st_crs(grd)) -> b
a = aggregate(b, st_as_sf(grd, na.rm = FALSE), sum)
```

Now we have the population counts per grid cell in `a`. To get to population density, we need to find the area of each cell; for cells crossing the country border, this will be less than 10 x 10 km:

```
grd$ID = 1:prod(dim(grd)) # so we can find out later which grid cell we have
ii = st_intersects(grd["ID"], st_cast(st_union(de), "MULTILINESTRING"))
# Warning in st_intersects.stars(grd["ID"], st_cast(st_union(de),
# "MULTILINESTRING")): as_points is NA: assuming here that raster
# cells are small polygons, not points
grd_sf = st_as_sf(grd["ID"], na.rm = FALSE)[lengths(ii) > 0,]
iii = st_intersection(grd_sf, st_union(de))
# Warning: attribute variables are assumed to be spatially constant
# throughout all geometries
grd$area = st_area(grd)[[1]] + units::set_units(grd$values, m^2) # NA's
grd$area[iii$ID] = st_area(iii)
```

Instead of doing the two-stage procedure above: first finding cells that have a border crossing it, then computing its area, we could also directly use `st_intersection` on all cells, but that takes considerably longer. From the counts and areas we can compute densities, and verify totals (figure 12.9):

```
grd$pop_dens = a$Einwohner / grd$area
sum(grd$pop_dens * grd$area, na.rm = TRUE) # verify
# 80323301 [1]
sum(b$Einwohner)
# [1] 80324282
```

We need to divide the number of inhabitants by the number of 100 m grid points contributing to it, in order to convert population counts into population density.

To obtain population density values at monitoring network stations, we can use

```
(a = aggregate(grd["pop_dens"], no2.sf, mean))
# stars object with 1 dimensions and 1 attribute
# attribute(s):
#           Min.   1st Qu.   Median   Mean   3rd Qu.   Max.
# pop_dens 3.37e-06 4.98e-05 8.93e-05 0.000195 0.000237 0.00224
#           NA's
# pop_dens    1
# dimension(s):
#           from to offset delta          refsys point
# geometry    1 74     NA     NA WGS 84 / UTM z...  TRUE
#                           values
# geometry POINT (545414 5930802),...,POINT (835252 5630738)
no2.sf$pop_dens = st_as_sf(a)[[1]]
```

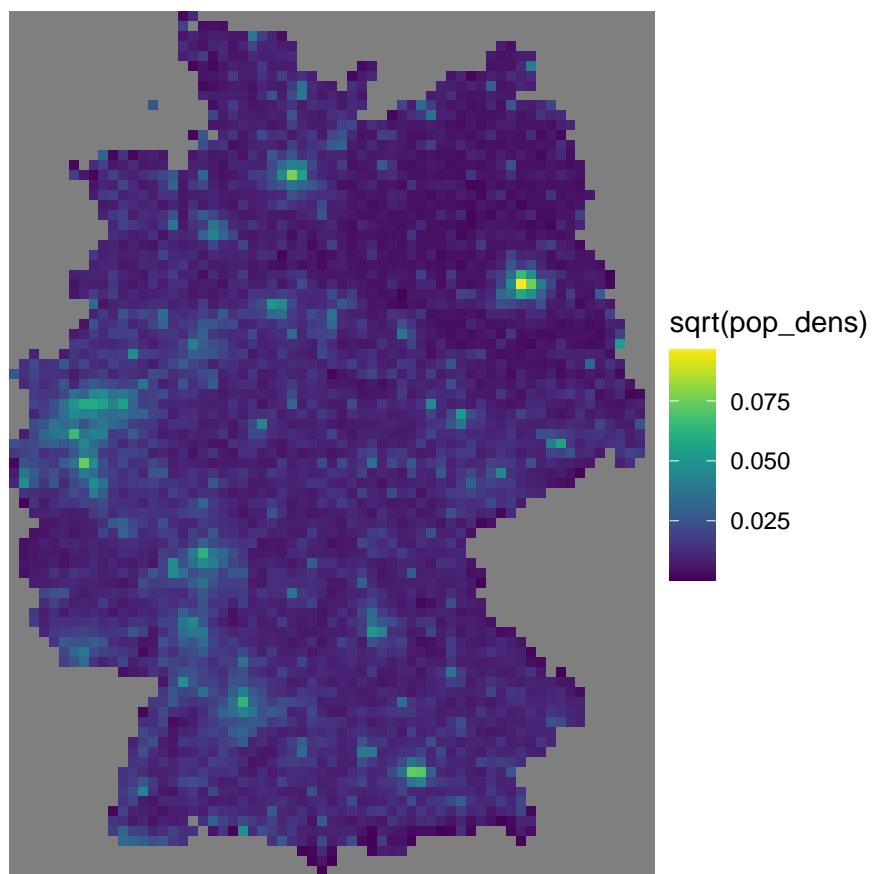


Figure 12.9: Population density for 100 m x 100 m grid cells

```

summary(lm(NO2~sqrt(pop_dens), no2.sf))
#
# Call:
# lm(formula = NO2 ~ sqrt(pop_dens), data = no2.sf)
#
# Residuals:
#   Min     1Q Median     3Q    Max
# -7.96  -2.15  -0.50  1.60  8.10
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 4.561      0.697   6.54 8.0e-09 ***
# sqrt(pop_dens) 325.006    49.927   6.51 9.2e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 3.15 on 71 degrees of freedom
# (1 observation deleted due to missingness)
# Multiple R-squared:  0.374,  Adjusted R-squared:  0.365
# F-statistic: 42.4 on 1 and 71 DF,  p-value: 9.19e-09

```

and the corresponding scatterplot is shown in 12.10.

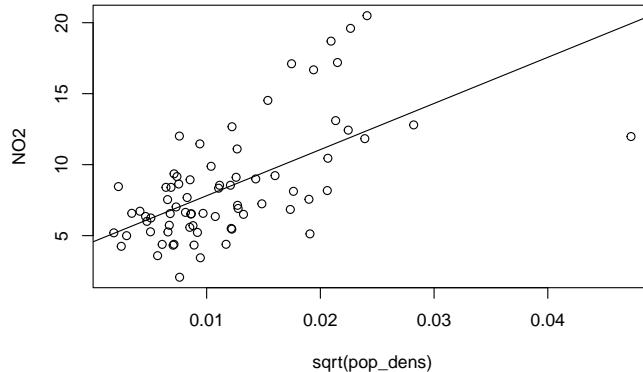


Figure 12.10: Scatter plot of 2017 annual mean NO2 concentration against population density, for rural background air quality stations

Prediction under this new model involves first modelling a residual variogram (figure 12.11):

```

no2.sf = no2.sf[!is.na(no2.sf$pop_dens),]
vr = variogram(NO2~sqrt(pop_dens), no2.sf)
vr.m = fit.variogram(vr, vgm(1, "Exp", 50000, 1))

```

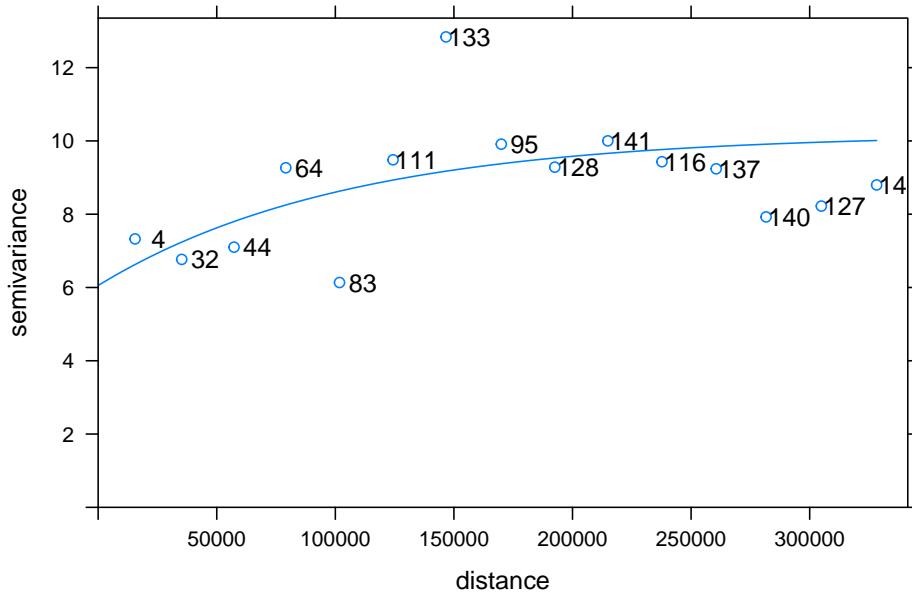


Figure 12.11: Residual variogram after subtracting population density trend

and subsequently, kriging prediction is done by (figure 12.12)

```

kr = krige(N02~sqrt(pop_dens), no2.sf, grd["pop_dens"], vr.m)
# [using universal kriging]
k$kr1 = k$var1.pred
k$kr2 = kr$var1.pred
st_redimension(k[c("kr1", "kr2")],
  along = list(what = c("kriging", "residual kriging"))) %>%
  setNames("N02") -> km

# Coordinate system already present. Adding new coordinate system, which will replace the existing one

```

where, critically, the `pop_dens` values are now available for prediction locations in object `grd`. We see some clear differences: the map with population density in the trend follows the extremes of the population density rather than those of the measurement stations, and has a range that extends that of the former. It should be taken with a large grain of salt however, since the stations used were filtered for the category “rural background”, indicating that they represent conditions of lower populations density. The scatter plot of Figure 12.10 reveals that the the population density at the locations of stations is much more limited than that in the population density map, and hence the right-hand side map is based on strongly extrapolating the relationship shown in 12.10.

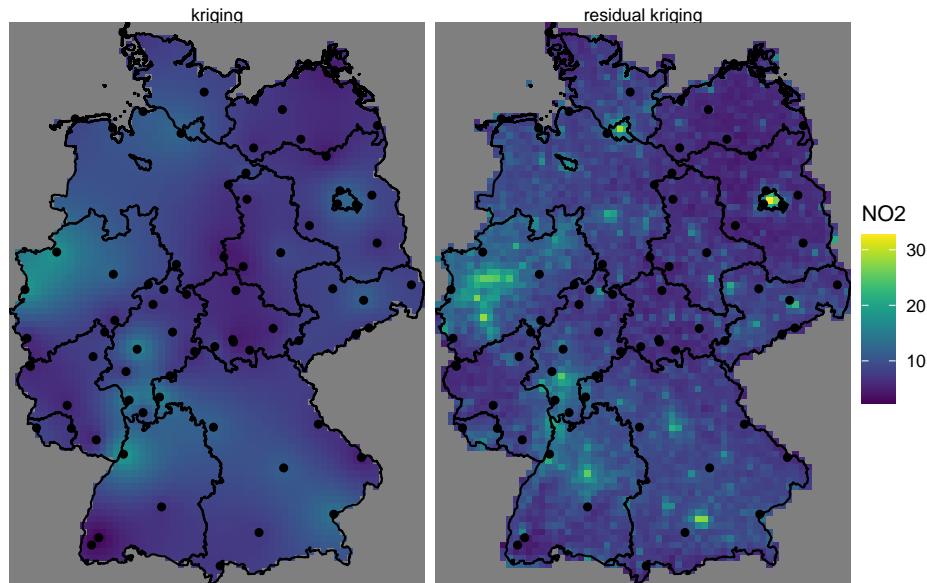


Figure 12.12: Kriging NO₂ values using population density as a trend variable

12.8 Exercises

1. Create a plot like the one in figure @ref(fig:residualkriging} that has the inverse distance interpolated map of figure 12.2 added on left side.
2. Create a scatter plot of the map values of the idw and kriging map, and a scatter plot of map values of idw and residual kriging.
3. Carry out a *block kriging* by setting the `block` argument in `krige()`, and do this for block sizes of 10 km (the grid cell size), 50 km and 200 km. Compare the resulting maps of estimates, and resulting map of kriging standard errors.
4. Based on the residual kriging results obtained above, compute maps of the lower and upper boundary of a 95% confidence interval, when assuming that the kriging error is normally distributed, and shown them in a plot with a single (joint) legend.

Chapter 13

Multivariate and Spatiotemporal Geostatistics

Building on the simple interpolation methods presented in chapter 12, this chapter works out a case study for spatiotemporal interpolation, using NO₂ air quality data, and populations density as covariate.

13.1 Preparing the air quality dataset

The dataset we work with is an air quality dataset obtained from the European Environmental Agency (EEA). European member states report air quality measurements to this Agency. So-called *validated* data are quality controlled by member states, and are reported on a yearly basis. They form the basis for policy compliancy evaluations.

The EEA's air quality e-reporting website gives access to the data reported by European member states. We decided to download hourly (time series) data, which is the data primarily measured. A web form helps convert simple selection criteria into an http GET request. The following URL

https://fme.discomap.eea.europa.eu/fmedatastreaming/AirQualityDownload/AQData_Extract.fmw?Country

was created to select all validated (`Source=E1a`) NO₂ (`Pollutant=8`) data for 2017 (`Year_from, Year_to`) from Germany (`CountryCode=DE`). It returns a text file with a set of URLs to CSV files, each containing the hourly values for the

whole period for a single measurement station. These files were downloaded and converted to the right encoding using the `dos2unix` command line utility.

In the following, we will read all the files into a list,

```
files = list.files("aq", pattern = "*.csv", full.names = TRUE)
r = lapply(files[-1], function(f) read.csv(f))
```

then convert the time variable into a `POSIXct` variable, and put them in time order by

```
Sys.setenv(TZ = "UTC") # make sure times are not interpreted in local time zone
r = lapply(r, function(f) {
  f$t = as.POSIXct(f$DatetimeBegin)
  f[order(f$t), ]
})
```

We remove smaller datasets, which for this dataset have no hourly data:

```
r = r[sapply(r, nrow) > 1000]
names(r) = sapply(r, function(f) unique(f$AirQualityStationEoICode))
length(r) == length(unique(names(r)))
# [1] TRUE
```

and then combine all files using `xts::cbind`, so that they are matched based on matching times:

```
library(xts)
r = lapply(r, function(f) xts(f$Concentration, f$t))
aq = do.call(cbind, r)
```

A usual further selection for this dataset is to select stations for which 75% of the hourly values measured are valid, i.e. drop those with more than 25% hourly values missing. Knowing that `mean(is.na(x))` gives the *fraction* of missing values in a vector `x`, we can apply this function to the columns (stations):

```
sel = apply(aq, 2, function(x) mean(is.na(x)) < 0.25)
aqsel = aq[, sel]
```

Next, the station metadata was read and filtered for rural background stations in Germany ("DE") by

```
library(tidyverse)
read.csv("aq/AirBase_v8_stations.csv", sep = "\t", stringsAsFactors = FALSE) %>%
  as_tibble() %>%
  filter(country_iso_code == "DE", station_type_of_area == "rural",
    type_of_station == "Background") -> a2
```

These stations contain coordinates, and an `sf` object with (static) station metadata is created by

```
library(sf)
a2.sf = st_as_sf(a2, coords = c("station_longitude_deg", "station_latitude_deg"), crs = 'EPSG:4326')
```

We now subset the air quality measurements to include only stations that are of type rural background:

```
sel = colnames(aqsel) %in% a2$station_european_code
aqsel = aqsel[, sel]
```

We can compute station means, and join these to stations locations by

```
tb = tibble(NO2 = apply(aqsel, 2, mean, na.rm = TRUE), station_european_code = colnames(aqsel))
crs = 32632
right_join(a2.sf, tb) %>% st_transform(crs) -> no2.sf
# Joining, by = "station_european_code"
```

Station mean NO₂ concentrations, along with country borders, are shown in figure 12.1.

13.2 Multivariable geostatistics

Multivariable geostatistics involves the *joint* modelling, prediction and simulation of multiple variables,

$$Z_1(s) = X_1\beta_1 + e_1(s)$$

...

$$Z_n(s) = X_n\beta_n + e_n(s).$$

In addition to having observations, trend models, and variograms for each variable, the *cross* variogram for each pair of residual variables, describing the covariance of $e_i(s), e_j(s+h)$, is required. If this cross covariance is non-zero, knowledge of $e_j(s+h)$ may help predict (or simulate) $e_i(s)$. This is especially true if $Z_j(s)$ is more densely sample than $Z_i(s)$. Prediction and simulation under this model are called cokriging and cosimulation. Examples using gstat are found when running the demo scripts

```
library(gstat)
demo(cokriging)
demo(cosimulation)
```

and are further illustrated and discussed in (Bivand et al., 2013).

In case the different variables considered are observed at the same set of locations, for instance different air quality parameters, then the statistical *gain* of using cokriging as opposed to direct (univariable) kriging is often modest, when not negligible. A gain may however be that the prediction is truly multivariable: in addition to the prediction vector $Z(s_0) = (\hat{Z}_1(s_0), \dots, \hat{Z}_n(s_0))$ we get the full covariance matrix of the prediction error (Ver Hoef and Cressie, 1993). This means for instance that if we are interested in some linear combination of $\hat{Z}(s_0)$, such as $\hat{Z}_2(s_0) - \hat{Z}_1(s_0)$, that we can get the standard error of that combination because we have the correlations between the prediction errors.

Although sets of direct and cross variograms can be computed and fitted automatically, multivariable geostatistical modelling becomes quickly hard to manage when the number of variables gets large, because the number of direct and cross variograms required is $n(n + 1)/2$.

In case different variables refer to the same variable take at different time steps, one could use a multivariable (cokriging) prediction approach, but this would not allow for e.g. interpolation between two time steps. For this, and for handling the case of having data observed at many time instances, one can also model its variation as a function of continuous space *and* time, as of $Z(s, t)$, which we will do in the next section.

13.3 Spatiotemporal geostatistics

Spatiotemporal geostatistical processes are modelled as variables having a value everywhere in space and time, $Z(s, t)$, with s and t the continuously indexed space and time index. Given observations $Z(s_i, t_j)$ and a variogram (covariance) model $\gamma(s, t)$ we can predict $Z(s_0, t_0)$ at arbitrary space/time locations (s_0, t_0) using standard Gaussian process theory.

Several books have been written recently about modern approaches to handling and modelling spatiotemporal geostatistical data, including (Wikle et al., 2019) and (Blangiardo and Cameletti, 2015). Here, we will use (Gräler et al., 2016) and give some simple examples using the dataset also used for the previous chapter.

13.3.1 A spatiotemporal variogram model

Starting with the spatiotemporal matrix of NO₂ data in `aq` constructed at the beginning of this chapter, we will first select the measurements taken at rural background stations:

```
aqx = aq[ , colnames(aq) %in% a2$station_european_code]
```

Then we will select the spatial locations for these stations by

```
sfc = st_geometry(a2.sf)[match(colnames(aqx), a2.sf$station_european_code)]
```

and finally build a `stars` object with time and station as dimensions:

```
library(stars)
st_as_stars(NO2 = as.matrix(aqx)) %>%
  st_set_dimensions(names = c("time", "station")) %>%
  st_set_dimensions("time", index(aqx)) %>%
  st_set_dimensions("station", sfc) -> no2.st
```

From this, we can compute the spatiotemporal variogram using

```
v.st = variogramST(NO2~1, no2.st[,1:(24*31)], tlags = 0:48,
cores =getOption("mc.cores", 2))
```

which is shown in figure 13.1.

To this sample variogram, we can fit a variogram model. One relatively flexible model we try here is the product-sum model (Gräler et al., 2016), fitted by

```
# product-sum
prodSumModel <- vgmST("productSum",
  space=vgm(150, "Exp", 200, 0),
  time= vgm(20, "Sph", 40, 0),
  k=2)
StAni = estiStAni(v.st, c(0,200000))
(fitProdSumModel <- fit.StVariogram(v.st, prodSumModel, fit.method = 7,
  StAni = StAni, method = "L-BFGS-B",
  control = list(parscale = c(1,10,1,1,0.1,1,10)),
  lower = rep(0.0001, 7)))
# space component:
#   model psill range
# 1   Nug  26.3     0
# 2   Exp 140.5   432
```

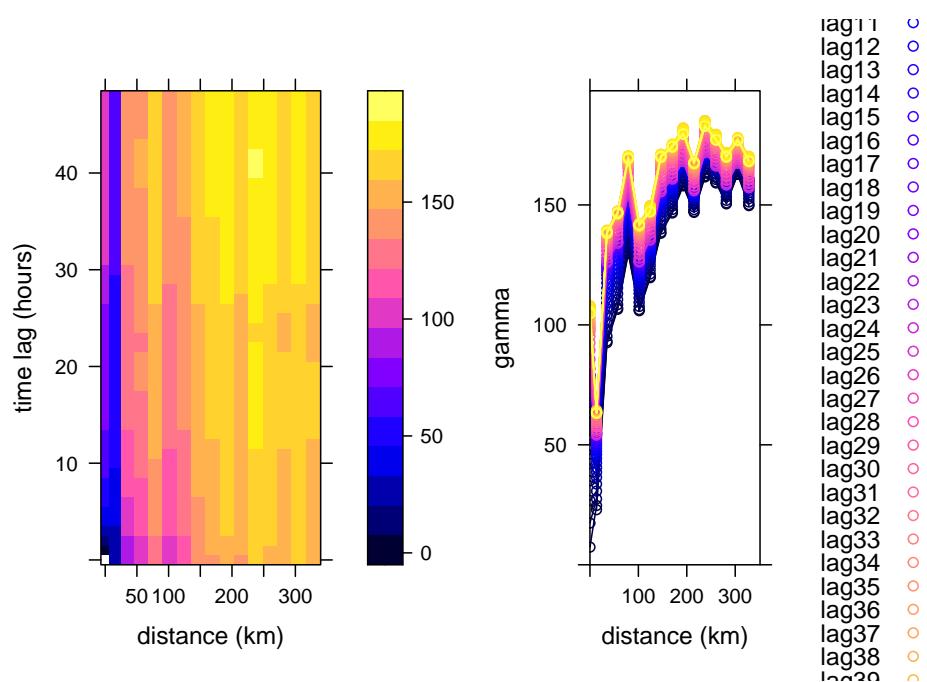


Figure 13.1: Spatiotemporal sample variogram for hourly NO_2 concentrations at rural background stations in Germany over 2027

```
# time component:
# model psill range
# 1 Nug 1.21 0.0
# 2 Sph 15.99 40.1
# k: 0.0322469094848839
```

and shown in figure 13.2.

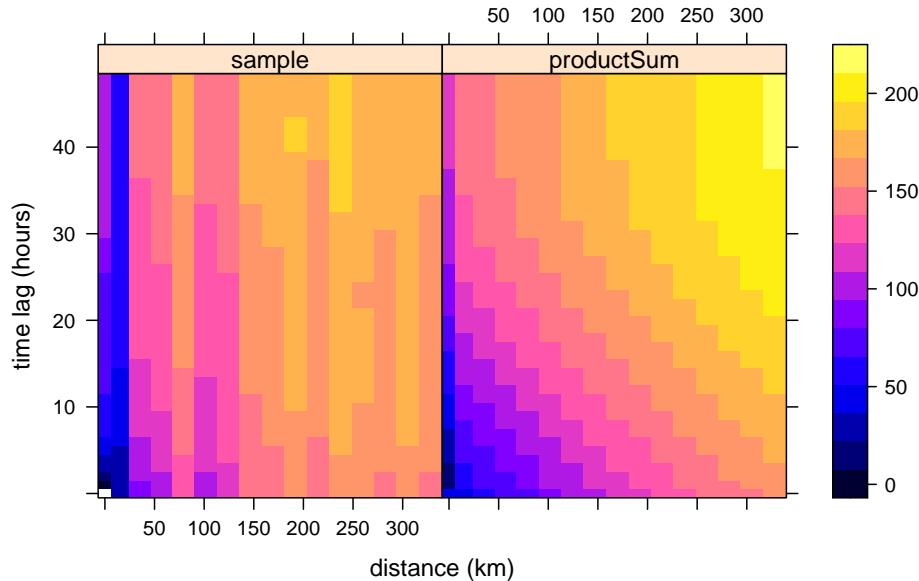


Figure 13.2: Product-sum model, fitted to the spatiotemporal sample variogram

which can also be plotted as wireframes, shown in figure 13.3.

Hints about the fitting strategy and alternative models for spatiotemporal variograms are given in (Gräler et al., 2016).

With this fitted model, and given the observations, we can carry out kriging or simulation at arbitrary points in space and time. For instance, we could estimate (or simulate) values in the time series that are now missing: this occurs regularly, and in section 12.4 we used means over time series based on simply ignoring up to 25% of the observations: substituting these with estimated or simulated values based on neighbouring (in space and time) observations before computing yearly mean values seems a more reasonable approach.

More in general, we can estimate at arbitrary locations and time points, and we will illustrate this with predicting time series at particular locations, and predicting spatial slices (Gräler et al., 2016). We can create a `stars` object for two randomly picked spatial points and all time instances is created by

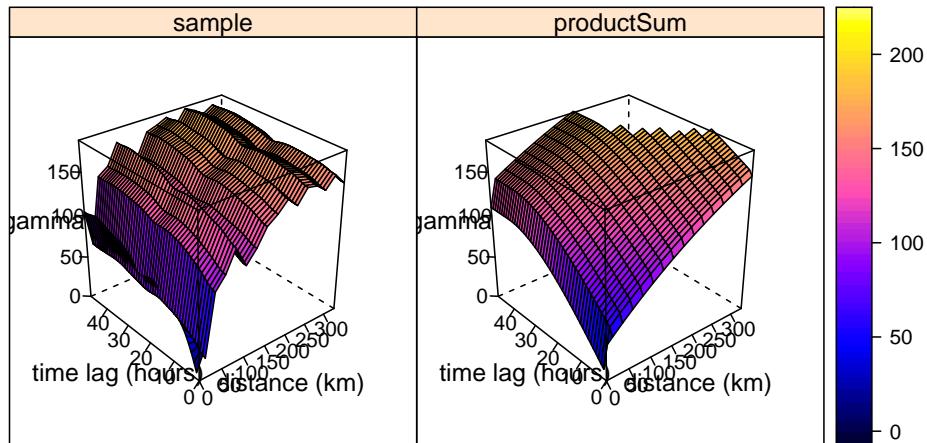


Figure 13.3: Wireframe plot of the fitted spatiotemporal variogram model

```

pt = st_sample(de, 2)
t = st_get_dimension_values(no2.st, 1)
st_as_stars(list(pts = matrix(1, length(t), length(pt)))) %>%
  st_set_dimensions(names = c("time", "station")) %>%
  st_set_dimensions("time", t) %>%
  st_set_dimensions("station", pt) -> new_pt

```

and we obtain the spatiotemporal predictions at these two points using `krigeST` by

```

no2.st <- st_transform(no2.st, crs)
new_ts <- krigeST(NO2~1, data = no2.st[["NO2"]], newdata = new_pt,
  nmax = 50, stAni = StAni, modelList = fitProdSumModel,
  progress = FALSE)

```

where the results are shown in figure 13.4.

Alternatively, we can create spatiotemporal predictions for a set of time-stamped raster maps, evenly spaced over the year 2017, created by

```

t4 = t[(1:4 - 0.5) * (3*24*30)]
d = dim(grd)
st_as_stars(pts = array(1, c(d[1], d[2], time=length(t4)))) %>%
  st_set_dimensions("time", t4) %>%
  st_set_dimensions("x", st_get_dimension_values(grd, "x")) %>%
  st_set_dimensions("y", st_get_dimension_values(grd, "y")) %>%
  st_set_crs(crs) -> grd.st

```

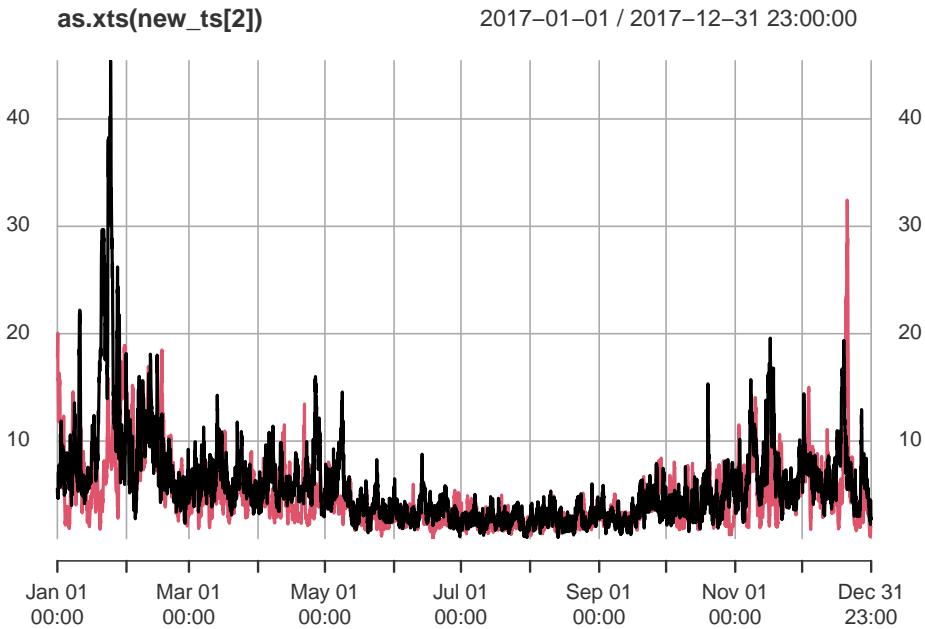


Figure 13.4: Time series plot of spatiotemporal predictions for two points

and the subsequent predictions are obtained by

```
new_int <- krigeST(NO2~1, data = no2.st["NO2"], newdata = grd.st,
                     nmax = 200, stAni = StAni, modelList = fitProdSumModel,
                     progress = FALSE)
names(new_int)[2] = "NO2"

# Coordinate system already present. Adding new coordinate system, which will replace the existing
```

and shown in figure 13.5.

A larger value for `nmax` was needed here to decrease the visible disturbance (sharp edges) caused by discrete neighbourhood selections, which are now done in space *and* time.

13.4 Exercises

1. Which fraction of the stations is removed in section 13.1 when the criterion applied that a station must be 75% complete?
2. From the hourly time series in `no2.st`, compute daily mean concentrations using `aggregate`, and compute the spatiotemporal variogram of this. How does it compare to the variogram of hourly values?

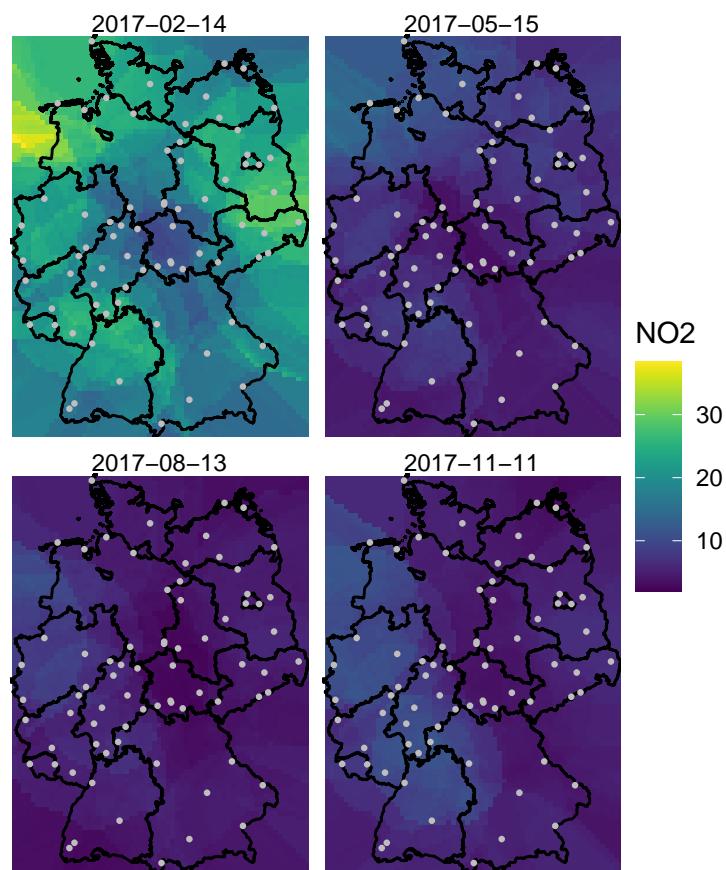


Figure 13.5: Spatiotemporal predictions for four selected time slices

3. Carry out a spatiotemporal interpolation for daily mean values for the days corresponding to those shown in 13.5, and compare the results.
4. Following the example in the demo scripts pointed at in section 13.2, carry out a cokriging on the daily mean station data for the four days shown in 13.5. What are the differences of this approach to spatiotemporal kriging?

Chapter 14

Proximity and Areal Data

Areal units of observation are very often used when simultaneous observations are aggregated within non-overlapping boundaries. The boundaries may be those of administrative entities, and may be related to underlying spatial processes, such as commuting flows, but are usually arbitrary. If they do not match the underlying and unobserved spatial processes in one or more variables of interest, proximate areal units will contain parts of the underlying processes, engendering spatial autocorrelation. By proximity, we mean *closeness* in ways that make sense for the data generation processes thought to be involved. In cross-sectional geostatistical analysis with point support, measured distance makes sense for typical data generation processes. In similar analysis of areal data, sharing a border may make more sense, because that is what we do know, but we cannot measure the distance between the areas in as adequate a way.

With support of data we mean the physical size (length, area, volume) associated with an individual observational unit (measurement). It is possible to represent the support of areal data by a point, despite the fact that the data have polygonal support. The centroid of the polygon may be taken as a representative point, or the centroid of the largest polygon in a multi-polygon object. When data with intrinsic point support are treated as areal data, the change of support goes the other way, from the known point to a non-overlapping tessellation such as a Voronoi diagram or Dirichlet tessellation or Thiessen polygons often through a Delaunay triangulation using projected coordinates. Here, different metrics may also be chosen, or distances measured on a network rather than on the plane. There is also a literature using weighted Voronoi diagrams in local spatial analysis (see for example Boots and Okabe, 2007; Okabe et al., 2008; She et al., 2015).

When the intrinsic support of the data is as points, but the underlying process is between proximate observations rather than driven chiefly by distance however measured between observations, the data may be aggregate counts or totals

(polling stations, retail turnover) or represent a directly observed characteristic of the observation (opening hours of the polling station). Obviously, the risk of mis-representing the footprint of the underlying spatial processes remains in all of these cases, not least because the observations are taken as encompassing the entirety of the underlying process in the case of tessellation of the whole area of interest. This is distinct from the geostatistical setting in which observations are rather samples taken using some scheme within the area of interest. It is also partly distinct from the practice of taking areal sample plots within the area of interest but covering only a small proportion of the area, typically used in ecological and environmental research.

In order to explore and analyse areal data of these kinds in Chapters 15, 16 and 17, methods are needed to represent the proximity of observations. This chapter then considers a subset of the such methods, where the spatial processes are considered as working through proximity understood in the first instance as contiguity, as a graph linking observations taken as neighbours. This graph is typically undirected and unweighted, but may be directed and/or weighted in certain settings, which then leads to further issues with regard to symmetry. In principle, proximity would be expected to operate symmetrically in space, that is that the influence of i on j and of j on i based on their relative positions should be equivalent. Edge effects are not considered in standard treatments.

14.1 Representing proximity in `spdep`

Handling spatial autocorrelation using relationships to neighbours on a graph takes the graph as given, chosen by the analyst. This differs from the geostatistical approach in which the analyst chooses the binning of the empirical variogram and function used, and then the way the variogram is fitted. Both involve a priori choices, but represent the underlying correlation in different ways (Wall, 2004). In Bavaud (1998) and work citing his contribution, attempts have been made to place graph-based neighbours in a broader context.

One issue arising in the creation of objects representing neighbourhood relationships is that of no-neighbour areal units (Bivand and Portnov, 2004). Islands or units separated by rivers may not be recognised as neighbours when the units have areal support and when using topological relationships such as shared boundaries. In some settings, for example `mrf` (Markov Random Field) terms in `mgcv::gam()` and similar model fitting functions that require undirected connected graphs, a requirement which is violated when there are disconnected subgraphs.

No-neighbour observations can also occur when a distance threshold is used between points, where the threshold is smaller than the maximum nearest neighbour distance. Shared boundary contiguities are not affected by using geographical, unprojected coordinates, but all point-based approaches use distance in one way or another, and need to calculate distances in an appropriate way.

The **spdep** package provides an **nb** class for neighbours, a list of length equal to the number of observations, with integer vector components. No-neighbours are encoded as an integer vector with a single element 0L, and observations with neighbours as sorted integer vectors containing values in 1L:n pointing to the neighbouring observations. This is a typical row-oriented sparse representation of neighbours. **spdep** provides many ways of constructing **nb** objects, and the representation and construction functions are widely used in other packages.

spdep builds on the **nb** representation (undirected or directed graphs) with the **listw** object, a list with three components, an **nb** object, a matching list of numerical weights, and a single element character vector containing the single letter name of the way in which the weights were calculated. The most frequently used approach in the social sciences is calculating weights by row standardization, so that all the non-zero weights for one observation will be the inverse of the cardinality of its set of neighbours ($1/\text{card}(\text{nb}[[\text{i}]])$).

We will be using election data from the 2015 Polish Presidential election in this chapter, with 2495 municipalities and Warsaw boroughs (see Figure 14.1 for a **tmap** map (section 9.5) of the municipality types), and complete count data from polling stations aggregated to these areal units. The data are an **sf** **sf** object:

```
library(sf)

data(pol_pres15, package="spDataLarge")
pol_pres15 %>%
  subset(select=c(TERYT, name, types)) %>%
  head()

# Simple feature collection with 6 features and 3 fields
# Geometry type: MULTIPOLYGON
# Dimension: XY
# Bounding box: xmin: 235000 ymin: 367000 xmax: 281000 ymax: 413000
# Projected CRS: ETRS89 / Poland CS92
#       TERYT           name      types
# 1 020101    BOLESŁAWIEC    Urban
# 2 020102    BOLESŁAWIEC   Rural
# 3 020103     GROMADKA   Rural
# 4 020104  NOWOGRODZIEC Urban/rural
# 5 020105    OSIECZNICA   Rural
# 6 020106 WARTA BOLESŁAWIECKA   Rural
#                           geometry
# 1 MULTIPOLYGON (((261089 3855...
# 2 MULTIPOLYGON (((254150 3837...
# 3 MULTIPOLYGON (((275346 3846...
# 4 MULTIPOLYGON (((251770 3770...
# 5 MULTIPOLYGON (((263424 4060...
```

```
# 6 MULTIPOLYGON (((267031 3870...
```

```
library(tmap)
tm_shape(pol_pres15) + tm_fill("types")
```

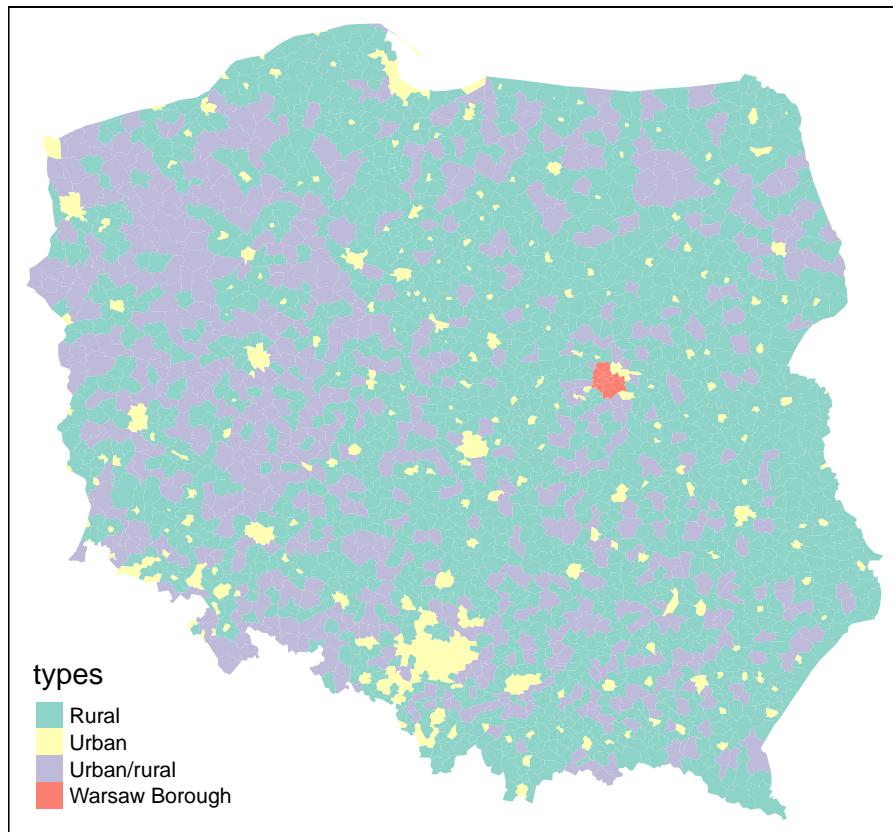


Figure 14.1: Polish municipality types 2015

For safety's sake, we impose topological validity:

```
if (!all(st_is_valid(pol_pres15))) pol_pres15 <- st_make_valid(pol_pres15)
```

Between early 2002 and April 2019, **spdep** contained functions for constructing and handling neighbour and spatial weights objects, tests for spatial autocorrelation, and model fitting functions. The latter have been split out into **spatialreg**, and will be discussed in the next chapter. **spdep** (Bivand, 2021b) now accommodates objects represented using **sf** classes and **sp** classes directly.

```
library(spdep)
# Loading required package: sp
# Loading required package: spData
```

14.2 Contiguous neighbours

The `poly2nb()` function in **spdep** takes the boundary points making up the polygon boundaries in the object passed as the `pl=` argument, typically an "`sf`" or "`sfc`" object with "POLYGON" or "MULTIPOLYGON" geometries. For each observation, the function checks whether at least one (`queen=TRUE`, default), or at least two (`rook, queen=FALSE`) points are within `snap=` distance units of each other. The distances are planar in the raw coordinate units, ignoring geographical projections. Once the required number of sufficiently close points is found, the search is stopped.

```
args(poly2nb)
# function (pl, row.names = NULL, snap = sqrt(.Machine$double.eps),
#         queen = TRUE, useC = TRUE, foundInBox = NULL, small_n = 500)
# NULL
```

From **spdep** 1.1-7, the GEOS interface of the **sf** package is used within `poly2nb()` if `foundInBox=NULL` to find the candidate neighbours and populate `foundInBox` internally. In this case, this use of spatial indexing (STRtree queries) in GEOS through **sf** is the default:

```
system.time(pol_pres15 %>% poly2nb(queen=TRUE) -> nb_q)
#    user  system elapsed
#   1.93    0.00   1.93
```

Earlier, `foundInBox=` accepted the output of the **rgeos** `gUnarySTRtreeQuery()` function to list candidate neighbours, that is polygons whose bounding boxes intersect the bounding boxes of other polygons. The print method shows the summary structure of the neighbour object:

```
nb_q
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 14242
# Percentage nonzero weights: 0.229
# Average number of links: 5.71
```

From **sf** version 1.0-0, the **s2** package (Dunnington et al., 2021) is used by default for spherical geometries, as `st_intersects()` used in `poly2nb()` passes calculation to `s2::s2_intersects_matrix()` (see 4). From **spdep** version 1.1-9, if

`sf_use_s2()` is TRUE, spherical intersection is used to find candidate neighbours; as with GEOS, the underlying `s2` library uses fast spatial indexing.

```
sf_use_s2(TRUE)
```

```
(pol_pres15 %>% st_transform("OGC:CRS84") -> pol_pres15_ll) %>%
  poly2nb(queen=TRUE) -> nb_q_s2
```

Spherical and planar intersection of the input polygons yield the same contiguity neighbours in this case; in both cases valid input geometries are desirable:

```
all.equal(nb_q, nb_q_s2, check.attributes=FALSE)
# [1] TRUE
```

Note that `nb` objects record both symmetric neighbour relationships, because these objects admit asymmetric relationships as well, but these duplications are not needed for object construction.

Most of the `spdep` functions for constructing neighbour objects take a `row.names=` argument, the value of which is stored as a `region.id` attribute. If not given, the values are taken from `row.names()` of the first argument. These can be used to check that the neighbours object is in the same order as data. If `nb` objects are subsetted, the indices change to continue to be within `1:length(subsetted_nb)`, but the `region.id` attribute values point back to the object from which it was constructed. This is used in out-of-sample prediction from spatial regression models discussed briefly in section @ref(spatcon_pred).

We can also check that this undirected graph is connected using the `n.comp.nb()` function; while some model estimation techniques do not support graphs that are not connected, it is helpful to be aware of possible problems (Freni-Starrantino et al., 2018):

```
(nb_q %>% n.comp.nb())$nc
# [1] 1
```

This approach is equivalent to treating the neighbour object as a graph and using graph analysis on that graph (Csardi and Nepusz, 2006; file., 2020), by first coercing to a binary sparse matrix (Bates and Maechler, 2021):

```
library(Matrix, warn.conflicts=FALSE)
nb_q %>%
  nb2listw(style="B") %>%
  as("CsparseMatrix") -> smat
library(igraph, warn.conflicts=FALSE)
```

```
(smat %>%
  graph.adjacency() -> g1) %>%
  count_components()
# [1] 1
```

Neighbour objects may be exported and imported in GAL format for exchange with other software, using `write.nb.gal()` and `read.gal()`:

```
tf <- tempfile(fileext=".gal")
write.nb.gal(nb_q, tf)
```

14.3 Graph-based neighbours

If areal units are an appropriate representation, but only points on the plane have been observed, contiguity relationships may be approximated using graph-based neighbours. In this case, the imputed boundaries tessellate the plane such that points closer to one observation than any other fall within its polygon. The simplest form is by using triangulation, here using the `deldir()` function in the `deldir` package. Because the function returns from i and to j identifiers, it is easy to construct a long representation of a `listw` object, as used in the S-Plus SpatialStats module and the `sn2listw()` function internally to construct an `nb` object (ragged wide representation). Alternatives such as GEOS often fail to return sufficient information to permit the neighbours to be identified.

The output of these functions is then converted to the `nb` representation using `graph2nb()`, with the possible use of the `sym=` argument to coerce to symmetry. We take the centroids of the largest component polygon for each observation as the point representation; population-weighted centroids might have been a better choice if they were available:

```
pol_pres15 %>%
  st_geometry() %>%
  st_centroid(of_largest_polygon=TRUE) -> coords
(coords %>% tri2nb() -> nb_tri)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 14930
# Percentage nonzero weights: 0.24
# Average number of links: 5.98
```

The average number of neighbours is similar to the Queen boundary contiguity case, but if we look at the distribution of edge lengths using `nbdists()`, we can see that although the upper quartile is about 15 km, the maximum is almost

300 km, an edge along much of one side of the convex hull. The short minimum distance is also of interest, as many centroids of urban municipalities are very close to the centroids of their surrounding rural counterparts.

```
nb_tri %>%
  nbdists(coords) %>%
  unlist() %>%
  summary()
#      Min. 1st Qu. Median     Mean 3rd Qu.     Max.
#    247    9847  12151   13485  14994  296974
```

Triangulated neighbours also yield a connected graph:

```
(nb_tri %>% n.comp.nb())$nc
# [1] 1
```

Graph-based approaches include `soi.graph()` - discussed here, `relativeneigh()` and `gabrielneigh()`.

The Sphere of Influence `soi.graph()` function takes triangulated neighbours and prunes off neighbour relationships represented by edges that are unusually long for each point, especially around the convex hull (Avis and Horton, 1985).

```
(nb_tri %>%
  soi.graph(coords) %>%
  graph2nb() -> nb_soi)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 12792
# Percentage nonzero weights: 0.205
# Average number of links: 5.13
```

Unpicking the triangulated neighbours does however remove the connected character of the underlying graph:

```
(nb_soi %>% n.comp.nb() -> n_comp)$nc
# [1] 16
```

The algorithm has stripped out longer edges leading to urban and rural municipality pairs where their centroids are very close to each other because the rural ones completely surround the urban, giving 15 pairs of neighbours unconnected to the main graph:

```
table(n_comp$comp.id)
#
#   1   2   3   4   5   6   7   8   9   10  11  12  13
# 2465  2   2   2   2   2   2   2   2   2   2   2   2
#   14  15  16
#   2   2   2
```

The largest length edges along the convex hull have been removed, but “holes” have appeared where the unconnected pairs of neighbours have appeared. The differences between `nb_tri` and `nb_soi` are shown in orange in Figure 14.2.

```
opar <- par(mar=c(0,0,0,0)+0.5)
pol_pres15 %>%
  st_geometry() %>%
  plot(border="grey", lwd=0.5)
nb_soi %>%
  plot(coords=coords, add=TRUE, points=FALSE, lwd=0.5)
nb_tri %>%
  diffnb(nb_soi) %>%
  plot(coords=coords, col="orange", add=TRUE, points=FALSE, lwd=0.5)

par(opar)
```

14.4 Distance-based neighbours

Distance-based neighbours can be constructed using `dneareigh()`, with a distance band with lower `d1=` and upper `d2=` bounds controlled by the `bounds=` argument. If spherical coordinates are used and either specified in the coordinates object `x` or with `x` as a two column matrix and `longlat=TRUE`, great circle distances in km will be calculated assuming the WGS84 reference ellipsoid, or if `use_s2=TRUE`, a 200-cell `s2` buffer is constructed around each point, points falling within the buffer chosen, and then chosen points beyond `d2=` dropped. From tests, it appears that spherical spatial indexing is not used, and so is slower than the legacy brute-force approach (see chapter 4).

From `spdep` 1.1-7, two arguments have been added, to use functionality in the `dbscan` package (Hahsler and Piekenbrock, 2021) for finding neighbours using planar spatial indexing in two or three dimensions by default, and not to test the symmetry of the output neighbour object.

The `kneareigh()` function for k -nearest neighbours returns a `knn` object, converted to an `nb` object using `knn2nb()`. It can also use great circle distances, not least because nearest neighbours may differ when upprojected coordinates are

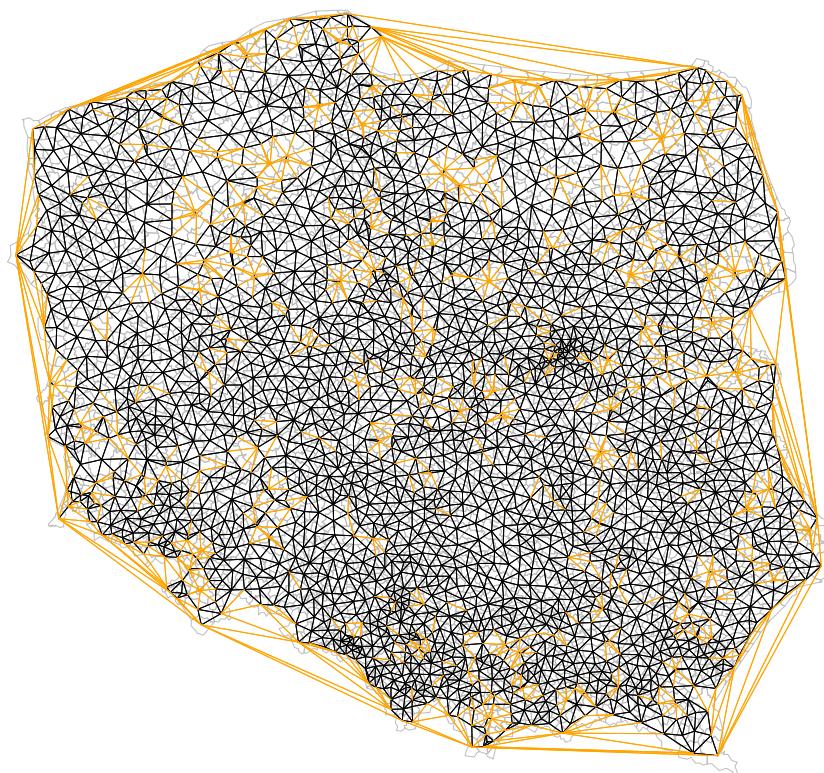


Figure 14.2: Triangulated (orange + black) and sphere of influence neighbours (black)

treated as planar. `k=` should be a small number. For projected coordinates, the `dbSCAN` package is used to compute nearest neighbours more efficiently. Note that `nb` objects constructed in this way are most unlikely to be symmetric, hence `knn2nb()` has a `sym=` argument to permit the imposition of symmetry, which will mean that all units have at least `k=` neighbours, not that all units will have exactly `k=` neighbours. From `spdep` version 1.1-9 and when `sf_use_s2()` is TRUE, `knearneigh()` will use fast spherical spatial indexing when the input object is of class "sf" or "sfc".

The `nbdists()` function returns the length of neighbour relationship edges in the units of the coordinates if the coordinates are projected, in km otherwise. In order to set the upper limit for distance bands, one may first find the maximum first nearest neighbour distance, using `unlist()` to remove the list structure of the returned object. From `spdep` version 1.1-9 and when `sf_use_s2()` is TRUE, `nbdists()` will use fast spherical distance calculations when the input object is of class "sf" or "sfc".

```
coords %>%
  knearneigh(k=1) %>%
  knn2nb() %>%
  nbdists(coords) %>%
  unlist() %>%
  summary()
#   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#   247    6663    8538    8275   10124   17979
```

Here the largest first nearest neighbour distance is just under 18 km, so using this as the upper threshold gives certainty that all units will have at least one neighbour:

```
system.time(coords %>% dnearneigh(0, 18000) -> nb_d18)
#   user  system elapsed
# 0.272   0.008   0.281
```

For this moderate number of observations, use of spatial indexing does not yield advantages in run times:

```
system.time(coords %>% dnearneigh(0, 18000, use_kd_tree=FALSE) -> nb_d18a)
#   user  system elapsed
# 0.291   0.000   0.291
```

and the output objects are the same:

```
all.equal(nb_d18, nb_d18a, check.attributes=FALSE)
# [1] TRUE
```

```
nb_d18
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 20358
# Percentage nonzero weights: 0.327
# Average number of links: 8.16
```

However, even though there are no no-neighbour observations (their presence is reported by the print method for `nb` objects), the graph is not connected, as a pair of observations are each others' only neighbours.

```
(nb_d18 %>% n.comp.nb() -> n_comp)$nc
# [1] 2
```

```
table(n_comp$comp.id)
#
#      1     2
# 2493    2
```

Adding 300 m to the threshold gives us a neighbour object with no no-neighbour units, and all units can be reached from all others across the graph.

```
(coords %>% dnearneigh(0, 18300) -> nb_d183)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 21086
# Percentage nonzero weights: 0.339
# Average number of links: 8.45
```

```
(nb_d183 %>% n.comp.nb())$nc
# [1] 1
```

One characteristic of distance-based neighbours is that more densely settled areas, with units which are smaller in terms of area (Warsaw boroughs are much smaller on average, but have almost 30 neighbours). Having many neighbours smooths the neighbour relationship across more neighbours.

For use later, we also construct a neighbour object with no-neighbour units, using a threshold of 16 km:

```
(coords %>% dnearest(0, 16000) -> nb_d16)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 15850
# Percentage nonzero weights: 0.255
# Average number of links: 6.35
# 7 regions with no links:
# 569 1371 1522 2374 2385 2473 2474
```

It is possible to control the numbers of neighbours directly using k -nearest neighbours, either accepting asymmetric neighbours:

```
((coords %>% knearneigh(k=6) -> knn_k6) %>% knn2nb() -> nb_k6)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 14970
# Percentage nonzero weights: 0.24
# Average number of links: 6
# Non-symmetric neighbours list
```

or imposing symmetry:

```
(knn_k6 %>% knn2nb(sym=TRUE) -> nb_k6s)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 16810
# Percentage nonzero weights: 0.27
# Average number of links: 6.74
```

Here the size of `k=` is sufficient to ensure connectedness, although the graph is not planar as edges cross at locations other than nodes, which is not the case for contiguous or graph-based neighbours.

```
(nb_k6s %>% n.comp.nb())$nc
# [1] 1
```

In the case of points on the sphere (see chapter 4), the output of `st_centroid()` will differ, so rather than inverse projecting the points, we extract points as geographical coordinates from the inverse projected polygon geometries:

```
sf_use_s2(TRUE)
```

```
pol_pres15_ll %>%
  st_geometry() %>%
  st_centroid(of_largest_polygon=TRUE) -> coords_ll
```

From **spdep** version 1.1-9, fast spatial indexing in **s2** is used to find the nearest neighbours:

```
(coords_ll %>% knearneigh(k=6) %>% knn2nb() -> nb_k6_ll)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 14970
# Percentage nonzero weights: 0.24
# Average number of links: 6
# Non-symmetric neighbours list
```

These neighbours differ from the planar $k=6$ nearest neighbours as would be expected:

```
isTRUE(all.equal(nb_k6, nb_k6_ll, check.attributes=FALSE))
# [1] TRUE
```

The **nbdists()** function also uses **s2** to find distances on the sphere when the "sf" or "sfc" input object is in geographical coordinates (distances returned in kilometres):

```
nb_q %>% nbdists(coords_ll) %>% unlist() %>% summary()
#      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
#      0.2    9.8   12.2    12.7   15.1   33.1
```

These differ a little for the same weights object when planar coordinates are used (distances returned in the metric of the points):

```
nb_q %>% nbdists(coords) %>% unlist() %>% summary()
#      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
#     247    9822   12173   12651   15117   33102
```

14.5 Weights specification

Once neighbour objects are available, further choices need to be made in specifying the weights objects. The **nb2listw()** function is used to create a **listw** weights object with an **nb** object, a matching list of weights vectors, and a style specification. Because handling no-neighbour observations now begins to matter, the

`zero.policy=` argument is introduced. By default, this is FALSE, indicating that no-neighbour observations will cause an error, as the spatially lagged value for an observation with no neighbours is not available. By convention, zero is substituted for the lagged value, as the cross product of a vector of zero-valued weights and a data vector, hence the name of `zero.policy`.

```
args(nb2listw)
# function (neighbours, glist = NULL, style = "W", zero.policy = NULL)
# NULL
```

We will be using the helper function `spweights.constants()` below to show some consequences of varying style choices. It returns constants for a `listw` object, n is the number of observations, $n1$ to $n3$ are $n - 1, \dots, nn$ is n^2 and S_0, S_1 and S_2 are constants, S_0 being the sum of the weights. There is a full discussion of the constants in Bivand and Wong (2018).

```
args(spweights.constants)
# function (listw, zero.policy = NULL, adjust.n = TRUE)
# NULL
```

The "B" binary style gives a weight of unity to each neighbour relationship, and typically upweights units with no boundaries on the edge of the study area.

```
(nb_q %>%
  nb2listw(style="B") -> lw_q_B) %>%
  spweights.constants() %>%
  data.frame() %>%
  subset(select=c(n, S0, S1, S2))
#      n      S0      S1      S2
# 1 2495 14242 28484 357280
```

The "W" row-standardized style upweights units around the edge of the study area that necessarily have fewer neighbours. This style first gives a weight of unity to each neighbour relationship, then divides these weights by the per unit sums of weights. Naturally this leads to division by zero where there are no neighbours, a not-a-number result, unless the chosen policy is to permit no-neighbour observations. We can see that S_0 is now equal to n .

```
(nb_q %>%
  nb2listw(style="W") -> lw_q_W) %>%
  spweights.constants() %>%
  data.frame() %>%
  subset(select=c(n, S0, S1, S2))
#      n      S0      S1      S2
# 1 2495 2495 958 10406
```

Inverse distance weights are used in a number of scientific fields. Some use dense inverse distance matrices, but many of the inverse distances are close to zero, so have little practical contribution, especially as the spatial process matrix is itself dense. Inverse distance weights may be constructed by taking the lengths of edges, changing units to avoid most weights being too large or small (here from m to km), taking the inverse, and passing through the `glist=` argument to `nb2listw()`:

```
nb_d183 %>%
  nbdistss(coords) %>%
  lapply(function(x) 1/(x/1000)) -> gwts
(nb_d183 %>% nb2listw(glist=gwts, style="B") -> lw_d183_idw_B) %>%
  spweights.constants() %>%
  data.frame() %>%
  subset(select=c(n, S0, S1, S2))
#      n    S0    S1    S2
# 1 2495 1841 534 7265
```

No-neighbour handling is by default to prevent the construction of a weights object, making the analyst take a position on how to proceed.

```
try(nb_d16 %>% nb2listw(style="B") -> lw_d16_B)
# Error in nb2listw(., style = "B") : Empty neighbour sets found
```

Use can be made of the `zero.policy=` argument to many functions used with `nb` and `listw` objects.

```
nb_d16 %>%
  nb2listw(style="B", zero.policy=TRUE) %>%
  spweights.constants(zero.policy=TRUE) %>%
  data.frame() %>%
  subset(select=c(n, S0, S1, S2))
#      n    S0    S1    S2
# 1 2488 15850 31700 506480
```

Note that by default the `adjust.n=` argument to `spweights.constants()` is set by default to TRUE, subtracting the count of no-neighbour observations from the observation count, so `n` is smaller with possible consequences for inference. The complete count can be retrieved by changing the argument.

14.6 Higher order neighbours

We recall the characteristics of the neighbour object based on Queen contiguities:

```
nb_q
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 14242
# Percentage nonzero weights: 0.229
# Average number of links: 5.71
```

If we wish to create an object showing i to k neighbours, where i is a neighbour of j , and j in turn is a neighbour of k , so taking two steps on the neighbour graph, we can use `nblag()`, which automatically removes i to i self-neighbours:

```
(nb_q %>% nblag(2) -> nb_q2)[[2]]
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 32930
# Percentage nonzero weights: 0.529
# Average number of links: 13.2
```

The `nblag_cumul()` function cumulates the list of neighbours for the whole list of lags:

```
nblag_cumul(nb_q2)
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 47172
# Percentage nonzero weights: 0.758
# Average number of links: 18.9
```

while the set operation `union.nb()` takes two objects, giving here the same outcome:

```
union.nb(nb_q2[[2]], nb_q2[[1]])
# Neighbour list object:
# Number of regions: 2495
# Number of nonzero links: 47172
# Percentage nonzero weights: 0.758
# Average number of links: 18.9
```

Returning to the graph representation of the same neighbour object, we can ask how many steps might be needed to traverse the graph:

```
diameter(g1)
# [1] 52
```

We step out from each observation across the graph to establish the number of steps needed to reach each other observation by the shortest path, once again finding the same count, and that the municipality is called Lutowiska, close to the Ukrainian border in the far south east of the country.

```
g1 %>% shortest.paths() -> sps
(sps %>% apply(2, max) -> spmax) %>% max()
# [1] 52

mr <- which.max(spmax)
pol_pres15$name0[mr]
# [1] "Lutowiska"
```

Figure 14.3 shows that contiguity neighbours represent the same kinds of relationships with other observations as distance. Some approaches prefer distance neighbours on the basis that, for example, inverse distance neighbours show clearly how all observations are related to each other. However, the development of tests for spatial autocorrelation and spatial regression models has involved the inverse of a spatial process model, which in turn can be represented as the sum of a power series of the product of a coefficient and a spatial weights matrix, so intrinsically acknowledging the relationships of all observations with all other. Sparse contiguity neighbour objects accommodate rich dependency structures without the need to make the structures explicit.

```
gridExtra::grid.arrange(tmap_grob(tm1), g1, nrow=1)
# Note that the aspect ratio for which the grob has been generated is 1.44
```

14.7 Exercises

1. Which kinds of geometry support are appropriate for which functions creating neighbour objects?
2. Which functions creating neighbour objects are only appropriate for planar representations?
3. What difference might the choice of `rook` rather than `queen` contiguities make on a chessboard?
4. What are the relationships between neighbour set cardinalities (neighbour counts) and row-standardized weights, and how do they open analyses up to edge effects? Use the chessboard you constructed in exercise 3 for both `rook` and `queen` neighbours.

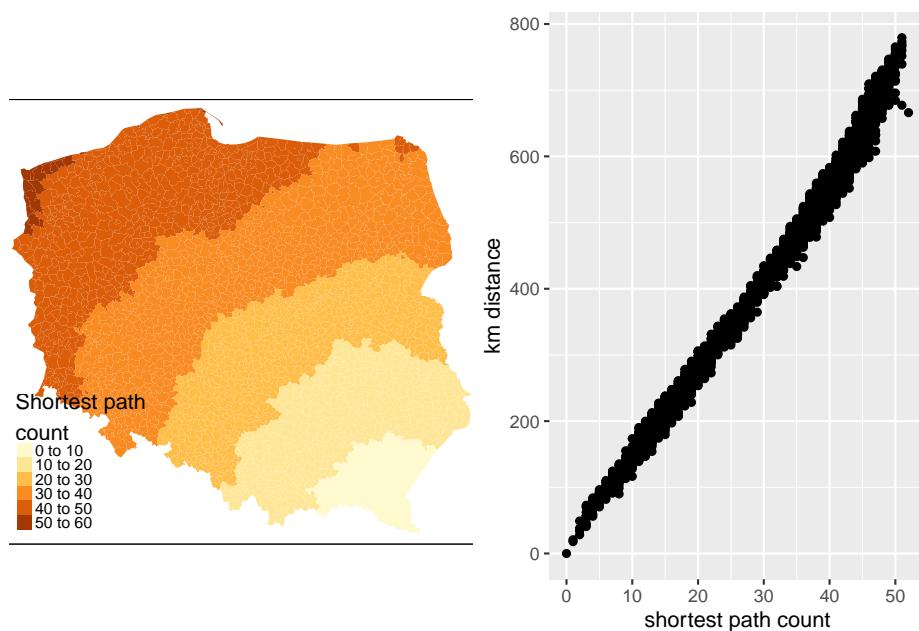


Figure 14.3: Relationship of shortest paths to distance for Lutowiska; left panel: shortest path counts from Lutowiska; right panel: plot of shortest paths from Lutowiska to other observations and distances (km) from Lutowiska to other observations

Chapter 15

Measures of spatial autocorrelation

When analysing areal data, it has long been recognised that, if present, spatial autocorrelation changes how we may infer, relative to the default position of independent observations. In the presence of spatial autocorrelation, we can predict the values of observation i from the values observed at $j \in N_i$, the set of its proximate neighbours. Early results (Moran, 1948; Geary, 1954), entered into research practice gradually, for example the social sciences (Duncan et al., 1961). These results were then collated and extended to yield a set of basic tools of analysis (Cliff and Ord, 1973, 1981).

Cliff and Ord (1973) generalised and extended the expression of the spatial weights matrix representation as part of the framework for establishing the distribution theory for join count, Moran's I and Geary's C statistics. This development of what have become known as global measures, returning a single value of autocorrelation for the total study area, has been supplemented by local measures returning values for each areal unit (Getis and Ord, 1992; Anselin, 1995).

15.1 Measures and process mis-specification

It is not and has never been the case that Tobler's first law of geography: "Everything is related to everything else, but near things are more related than distant things" always holds absolutely. This is and has always been an oversimplification, disguising the underlying entitation, support and other mis-specification problems. Are the units of observation appropriate for the scale of the underlying spatial process? Could the spatial patterning of the variable of interest for the chosen entitation be accounted for by another variable?

Tobler (1970) was published in the same special issue of *Economic Geography* as Olsson (1970), but Olsson does grasp the important point that spatial autocorrelation is not inherent in spatial phenomena, but often is engendered by inappropriate entitation, by omitted variables and/or inappropriate functional form. The key quote from Olsson is on p. 228:

The existence of such autocorrelations makes it tempting to agree with Tobler (1970, 236 [the original refers to the pagination of a conference paper]) that ‘everything’ is related to everything else, but near things are more related than distant things.’ On the other hand, the fact that the autocorrelations seem to hide systematic specification errors suggests that the elevation of this statement to the status of ‘the first law of geography’ is at best premature. At worst, the statement may represent the spatial variant of the post hoc fallacy, which would mean that coincidence has been mistaken for a causal relation.

The status of the “first law” is very similar to the belief that John Snow induced the cause of cholera as water-borne from a map. It may be a good way of selling GIS, but it is inaccurate; Snow had a strong working hypothesis prior to visiting Soho, and the map was prepared after the Broad street pump was disabled as documentation that the hypothesis held (Brody et al., 2000).

Measures of spatial autocorrelation unfortunately pick up other mis-specifications in the way that we model data (Schabenberger and Gotway, 2005; McMillen, 2003). For reference, Moran’s I is given as (Cliff and Ord, 1981, page 17):

$$I = \frac{n \sum_{(2)} w_{ij} z_i z_j}{S_0 \sum_{i=1}^n z_i^2}$$

where $x_i, i = 1, \dots, n$ are n observations on the numeric variable of interest,

$z_i = x_i - \bar{x}$, $\bar{x} = \sum_{i=1}^n x_i / n$, $\sum_{(2)} = i \neq j$, w_{ij} are the spatial weights, and $S_0 = \sum_{(2)} w_{ij}$. First we test a random variable using the Moran test, here under the normality assumption (argument `randomisation=FALSE`, default `TRUE`). Inference is made on the statistic $Z(I) = \frac{I - E(I)}{\sqrt{\text{Var}(I)}}$, the z-value compared with the Normal distribution for $E(I)$ and $\text{Var}(I)$ for the chosen assumptions; this x does not show spatial autocorrelation with these spatial weights:

```
glance_htest <- function(ht) c(ht$estimate,
  "Std deviate"=unname(ht$statistic),
  "p.value"=unname(ht$p.value))
set.seed(1)
(pol_pres15 %>%
```

```

nrow() %>%
  rnorm() -> x) %>%
moran.test(lw_q_B, randomisation=FALSE, alternative="two.sided") %>%
glance_htest()
# Moran I statistic      Expectation      Variance
#      -0.004772          -0.000401        0.000140
# Std deviate           p.value
#      -0.369320          0.711889

```

The test however detects quite strong positive spatial autocorrelation when we insert a gentle trend into the data, but omit to include it in the mean model, thus creating a missing variable problem but finding spatial autocorrelation instead:

```

beta <- 0.0015
coords %>%
  st_coordinates() %>%
  subset(select=1, drop=TRUE) %>%
  (function(x) x/1000)() -> t
(x + beta * t -> x_t) %>%
  moran.test(lw_q_B, randomisation=FALSE, alternative="two.sided") %>%
glance_htest()
# Moran I statistic      Expectation      Variance
#      0.043403          -0.000401        0.000140
# Std deviate           p.value
#      3.701491          0.000214

```

If we test the residuals of a linear model including the trend, the apparent spatial autocorrelation disappears:

```

lm(x_t ~ t) %>%
  lm.morantest(lw_q_B, alternative="two.sided") %>%
glance_htest()
# Observed Moran I      Expectation      Variance
#      -0.004777          -0.000789        0.000140
# Std deviate           p.value
#      -0.337306          0.735886

```

A comparison of implementations of measures of spatial autocorrelation shows that a wide range of measures is available in R in a number of packages, chiefly in the **spdep** package (Bivand, 2021b), and that differences from other implementations can be attributed to design decisions (Bivand and Wong, 2018). The **spdep** package also includes the only implementations of exact and saddlepoint approximations to global and local Moran's I for regression residuals (Tiefelsdorf, 2002; Bivand et al., 2009).

15.2 Global measures

Global measures consider the average level of spatial autocorrelation across all observations; they can of course be biased (as most spatial statistics) by edge effects where important spatial process components fall outside the study area.

15.2.1 Join-count tests for categorical data

We will begin by examining join count statistics, where `joincount.test()` takes a "factor" vector of values `fx=` and a `listw` object, and returns a list of `htest` (hypothesis test) objects defined in the `stats` package, one `htest` object for each level of the `fx=` argument. The observed counts are of neighbours with the same factor levels, known as same-colour joins.

```
args(joincount.test)
# function (fx, listw, zero.policy = NULL, alternative = "greater",
#         sampling = "nonfree", spChk = NULL, adjust.n = TRUE)
# NULL
```

The function takes an `alternative=` argument for hypothesis testing, a `sampling=` argument showing the basis for the construction of the variance of the measure, where the default "`nonfree`" choice corresponds to analytical permutation; the `spChk=` argument is retained for backward compatibility. For reference, the counts of factor levels for the type of municipality or Warsaw borough are:

```
(pol_pres15 %>%
  st_drop_geometry() %>%
  subset(select=types, drop=TRUE) -> Types) %>%
  table()
#
#          Rural           Urban      Urban/rural Warsaw Borough
#          1563            303           611             18
```

Since there are four levels, we re-arrange the list of `htest` objects to give a matrix of estimated results. The observed same-colour join counts are tabulated with their expectations based on the counts of levels of the input factor, so that few joins would be expected between for example Warsaw boroughs, because there are very few of them. The variance calculation uses the underlying constants of the chosen `listw` object and the counts of levels of the input factor. The z-value is obtained in the usual way by dividing the difference between the observed and expected join counts by the square root of the variance.

The join count test was subsequently adapted for multi-colour join counts (Upton and Fingleton, 1985). The implementation as `joincount.mult()` in `spdep` returns a table based on nonfree sampling, and does not report p-values.

Types %>% joincount.multi(listw=lw_q_B)	Joincount	Expected	Variance	z-value
#	3087.000	2793.920	1126.534	8.73
# Rural:Rural	110.000	104.719	93.299	0.55
# Urban:Urban	656.000	426.526	331.759	12.60
# Urban/rural:Urban/rural	41.000	0.350	0.347	68.96
# Warsaw Borough:Warsaw Borough	668.000	1083.941	708.209	-15.63
# Urban:Rural	2359.000	2185.769	1267.131	4.87
# Urban/rural:Rural	171.000	423.729	352.190	-13.47
# Warsaw Borough:Rural	12.000	64.393	46.460	-7.69
# Warsaw Borough:Urban	9.000	12.483	11.758	-1.02
# Warsaw Borough:Urban/rural	8.000	25.172	22.354	-3.63
# Jtot	3227.000	3795.486	1496.398	-14.70

So far, we have used binary weights, so the sum of join counts multiplied by the weight on that join remains integer. If we change to row standardised weights, where the weights almost always fractions of 1, the counts, expectations and variances change, but there are few major changes in the z-values.

Using an inverse distance based `listw` object does, however, change the z-values markedly, because closer centroids are upweighted relatively strongly:

Types %>% joincount.multi(listw=lw_d183_idw_B)	Joincount	Expected	Variance	z-value
#	3.46e+02	3.61e+02	4.93e+01	-2.10
# Rural:Rural	2.90e+01	1.35e+01	2.23e+00	10.39
# Urban:Urban	4.65e+01	5.51e+01	9.61e+00	-2.79
# Urban/rural:Urban/rural	1.68e+01	4.53e-02	6.61e-03	206.38
# Warsaw Borough:Warsaw Borough	2.02e+02	1.40e+02	2.36e+01	12.73
# Urban:Rural	2.25e+02	2.83e+02	3.59e+01	-9.59
# Urban/rural:Rural	3.65e+01	5.48e+01	8.86e+00	-6.14
# Warsaw Borough:Rural	5.65e+00	8.33e+00	1.73e+00	-2.04
# Warsaw Borough:Urban	9.18e+00	1.61e+00	2.54e-01	15.01
# Warsaw Borough:Urban/rural	3.27e+00	3.25e+00	5.52e-01	0.02
# Jtot	4.82e+02	4.91e+02	4.16e+01	-1.38

15.2.2 Moran's *I*

The implementation of Moran's *I* in `spdep` in the `moran.test()` function has similar arguments to those of `joincount.test()`, but `sampling=` is replaced by `randomisation=` to indicate the underlying analytical approach used for calculating the variance of the measure. It is also possible to use ranks rather than numerical values (Cliff and Ord, 1981, p. 46). The `drop.EI2=` argument may be used to reproduce results where the final component of the variance term is omitted as found in some legacy software implementations.

```
args(moran.test)
# function (x, listw, randomisation = TRUE, zero.policy = NULL,
#         alternative = "greater", rank = FALSE, na.action = na.fail,
#         spChk = NULL, adjust.n = TRUE, drop.EI2 = FALSE)
# NULL
```

The default for the `randomisation=` argument is `TRUE`, but here we will simply show that the test under normality is the same as a test of least squares residuals with only the intercept used in the mean model; the analysed variable is first round turnout proportion of registered voters in municipalities and Warsaw burroughs in the 2015 Polish presidential election. The spelling of randomisation is that of Cliff and Ord (1973).

```
(pol_pres15 %>%
  st_drop_geometry() %>%
  subset(select=I_turnout, drop=TRUE) -> z) %>%
  moran.test(listw=lw_q_B, randomisation=FALSE) %>%
  glance_htest()
# Moran I statistic      Expectation      Variance
#          0.691434       -0.000401      0.000140
# Std deviate            p.value
#          58.461349       0.000000
```

The `lm.morantest()` function also takes a `resfun=` argument to set the function used to extract the residuals used for testing, and clearly lets us model other salient features of the response variable (Cliff and Ord, 1981, p. 203). To compare with the standard test, we are only using the intercept here and, as can be seen, the results are the same.

```
lm(I_turnout ~ 1, pol_pres15) %>%
  lm.morantest(listw=lw_q_B) %>%
  glance_htest()
# Observed Moran I      Expectation      Variance
#          0.691434       -0.000401      0.000140
# Std deviate            p.value
#          58.461349       0.000000
```

The only difference between tests under normality and randomisation is that an extra term is added if the kurtosis of the variable of interest indicates a flatter or more peaked distribution, where the measure used is the classical measure of kurtosis. Under the default randomisation assumption of analytical randomisation, the results are largely unchanged.

```
(z %>%
  moran.test(listw=lw_q_B) -> mtr) %>%
  glance_htest()
# Moran I statistic      Expectation      Variance
#          0.691434        -0.000401       0.000140
# Std deviate           p.value
#          58.459835        0.000000
```

From the very beginning in the early 1970s, interest was shown in Monte Carlo tests, also known as Hope-type tests and as permutation bootstrap. By default, `moran.mc()` returns a "htest" object, but may simply use `boot::boot()` internally and return a "boot" object when `return_boot=TRUE`. In addition the number of simulations needs to be given as `nsim=`; that is the number of times the values of the observations are shuffled at random.

```
set.seed(1)
z %>%
  moran.mc(listw=lw_q_B, nsim=999, return_boot = TRUE) -> mmc
```

The bootstrap permutation retains the outcomes of each of the random permutations, reporting the observed value of the statistic, here Moran's I , the difference between this value and the mean of the simulations under randomisation (equivalent to $E(I)$), and the standard deviation of the simulations under randomisation.

If we compare the Monte Carlo and analytical variances of I under randomisation, we typically see few differences, arguably rendering Monte Carlo testing unnecessary.

```
c("Permutation bootstrap"=var(mmc$t),
  "Analytical randomisation"=uname(mtr$estimate[3]))
#   Permutation bootstrap Analytical randomisation
#                 0.000144            0.000140
```

Geary's global C is implemented in `geary.test()` largely following the same argument structure as `moran.test()`. The Getis-Ord G test includes extra arguments to accommodate differences between implementations, as Bivand and Wong (2018) found multiple divergences from the original definitions, often to omit no-neighbour observations generated when using distance band neighbours. It is given by (Getis and Ord, 1992, page 194). For G^* , the $\sum_{(2)}$ constraint is relaxed by including i as a neighbour of itself (thereby also removing the no-neighbour problem, because all observations have at least one neighbour).

Finally, the empirical Bayes Moran's I takes account of the denominator in assessing spatial autocorrelation in rates data (Assunção and Reis, 1999). Until

now, we have considered the proportion of valid votes cast in relation to the numbers entitled to vote by spatial entity, but using `EBImoran.mc()` we can try to accommodate uncertainty in extreme rates in entities with small numbers entitled to vote. There is, however, little impact on the outcome in this case.

Global measures of spatial autocorrelation using spatial weights objects based on graphs of neighbours are, as we have seen, rather blunt tools, which for interpretation depend critically on a reasoned mean model of the variable in question. If the mean model is just the intercept, the global measures will respond to all kinds of mis-specification, not only spatial autocorrelation. The choice of entities for aggregation of data will typically be a key source of mis-specification.

15.3 Local measures

Building on insights from the weaknesses of global measures, local indicators of spatial association began to appear in the first half of the 1990s (Anselin, 1995; Getis and Ord, 1992, 1996).

In addition, the Moran plot was introduced, plotting the values of the variable of interest against their spatially lagged values, typically using row-standardised weights to make the axes more directly comparable (Anselin, 1996). The `moran.plot()` function also returns an influence measures object used to label observations exerting more than proportional influence on the slope of the line representing global Moran's I . In Figure 15.1, we can see that there are many spatial entities exerting such influence. These pairs of observed and lagged observed values make up in aggregate the global measure, but can also be explored in detail. The quadrants of the Moran plot also show low-low pairs in the lower left quadrant, high-high in the upper right quadrant, and fewer low-high and high-low pairs in the upper left and lower right quadrants.

```
z %>%
  moran.plot(listw=lw_q_W, labels=pol_pres15$TERYT, cex=1, pch=".",
             xlab="I round turnout", ylab="lagged turnout") -> infl_W
```

If we extract the hat value influence measure from the returned object, Figure 15.2 suggests that some edge entities exert more than proportional influence (perhaps because of row standardisation), as do entities in or near larger urban areas.

```
pol_pres15$hat_value <- infl_W$hat
tm_shape(pol_pres15) + tm_fill("hat_value")
```

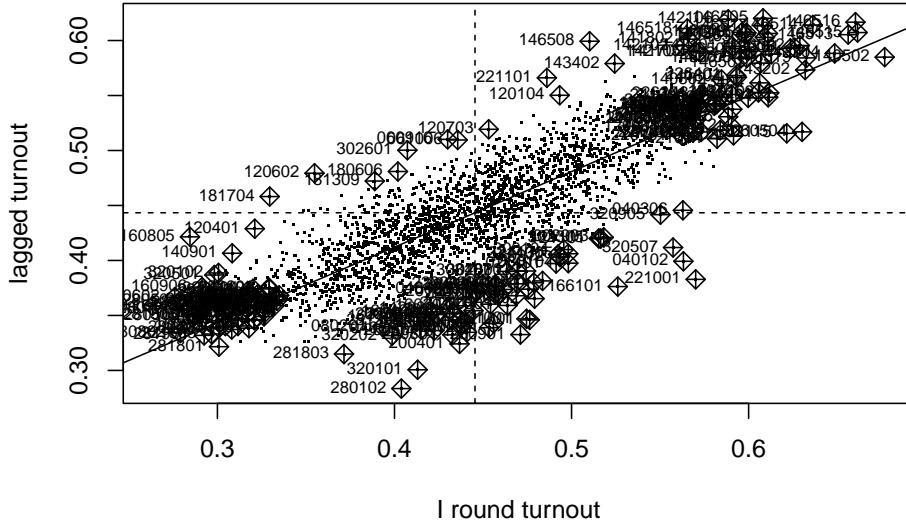


Figure 15.1: Moran plot of I round turnout, row standardised weights

15.3.1 Local Moran's I_i

Bivand and Wong (2018) discuss issues impacting the use of local indicators, such as local Moran's I_i and local Getis-Ord G_i . Some issues affect the calculation of the local indicators, others inference from their values. Because n statistics may be being calculated from the same number of observations, there are multiple comparison problems that need to be addressed. Although the apparent detection of hotspots from values of local indicators has been quite widely adopted, it remains fraught with difficulty because adjustment of the inferential basis to accommodate multiple comparisons is not often chosen, and as in the global case, mis-specification also remains a source of confusion. Further, interpreting local spatial autocorrelation in the presence of global spatial autocorrelation is challenging (Ord and Getis, 2001; Tiefelsdorf, 2002; Bivand et al., 2009).

```
args(localmoran)
# function (x, listw, zero.policy = NULL, na.action = na.fail,
#         conditional = FALSE, alternative = "greater", p.adjust.method = "none",
#         mlvar = TRUE, spChk = NULL, adjust.x = FALSE)
# NULL
```

The `mlvar=` and `adjust.x=` arguments to `localmoran()` are discussed in Bivand and Wong (2018), and permit matching with other implementations. The `p.adjust.method=` argument uses an untested speculation implemented in `p.adjustSP()` that adjustment should only take into account the cardinality of

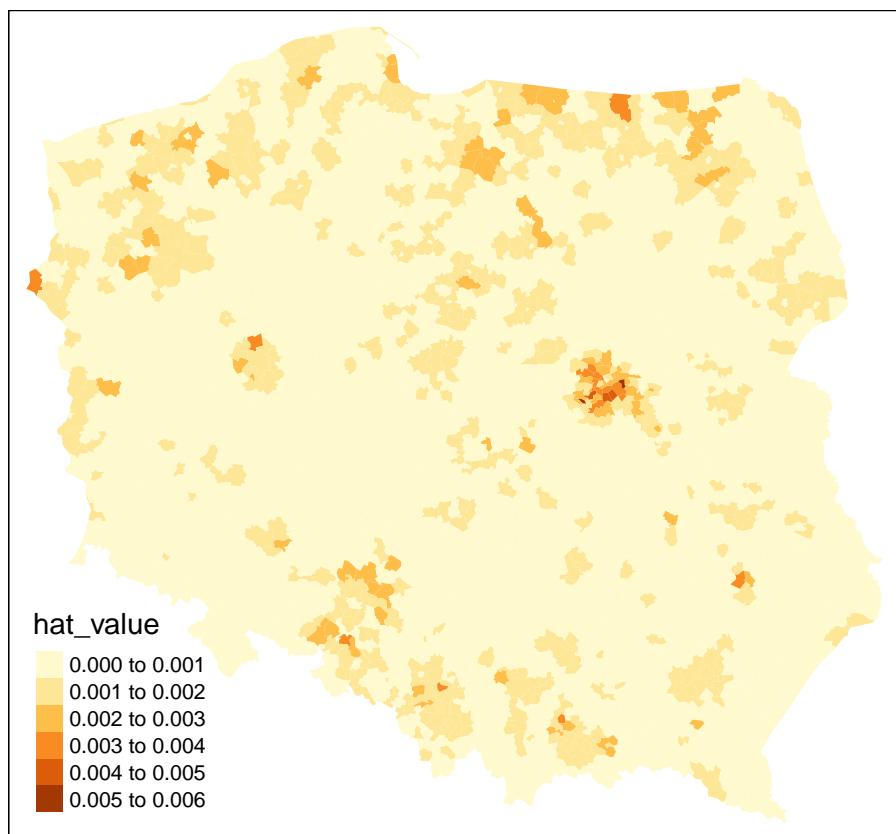


Figure 15.2: Moran plot hat values, row standardised neighbours

the neighbour set of each observation when adjusting for multiple comparisons; using `stats::p.adjust()` is preferable.

Taking "two.sided" p-values, we obtain:

```
z %>%
  localmoran(listw=lw_q_W, alternative="two.sided") -> locm
```

The I_i local indicators when summed and divided by the sum of the spatial weights equal global Moran's I , showing the possible presence of positive and negative local spatial autocorrelation:

```
all.equal(sum(locm[,1])/Szero(lw_q_W), unname(moran.test(z, lw_q_W)$estimate[1]))
# [1] TRUE
```

Using `stats::p.adjust()` to adjust for multiple comparisons, we see that almost 29% of the 2495 local measures have p-values < 0.05 if no adjustment is applied, but only 12% using Bonferroni adjustment to control the familywise error rate, with two other choices shown: `fdr` is the Benjamini and Hochberg (1995) false discovery rate and `BY` (Benjamini and Yekutieli, 2001), another false discovery rate adjustment:

```
pva <- function(pv) cbind("none"=pv, "bonferroni"=p.adjust(pv, "bonferroni"),
  "fdr"=p.adjust(pv, "fdr"), "BY"=p.adjust(pv, "BY"))
locm %>%
  subset(select="Pr(z != 0)", drop=TRUE) %>%
  pva() -> pvsp
f <- function(x) sum(x < 0.05)
apply(pvsp, 2, f)
#      none bonferroni      fdr        BY
#      715      297      576      424
```

In the global measure case, bootstrap permutations may be used as an alternative to analytical methods for possible inference, where both the theoretical development of the analytical variance of the measure, and the permutation scheme, shuffle all of the observed values. In the local case, conditional permutation may be used, fixing the value at observation i and randomly sampling from the remaining $n - 1$ values to find randomised values at neighbours, and is provided as `localmoran_perm()`, which may use multiple nodes to sample in parallel if provided, and permits the setting of a seed for the random number generator across the compute nodes:

```
library(parallel)
set.coresOption(ifelse(detectCores() == 1, 1, detectCores()-1L))
# NULL
```

```
system.time(z %>%
  localmoran_perm(listw=lw_q_W, nsim=499, alternative="two.sided",
  iseed=1) -> locm_p)
#   user  system elapsed
# 1.16    3.50    1.75
```

The outcome is that almost 32% of observations have two sided p-values < 0.05 without multiple comparison adjustment, and under 3% with Bonferroni adjustment.

```
locm_p %>%
  subset(select="Pr(z != 0)", drop=TRUE) %>%
  pva() -> pvsp
apply(pvsp, 2, f)
#       none bonferroni      fdr        BY
#       797         76       463       161
```

We can see what is happening by tabulating counts of the standard deviate of local Moran's I , where the two-sided $\alpha = 0.05$ bounds would be 0.025 and 0.975, but Bonferroni adjustment is close to 0.00001 and 0.99999. Without adjustment, almost 800 observations are significant, with Bonferroni adjustment, only almost 70 in the conditional permutation case:

```
brks <- qnorm(c(0, 0.00001, 0.0001, 0.001, 0.01, 0.025, 0.5, 0.975, 0.99, 0.999,
  0.99999, 1))
(locm_p %>%
  subset(select=Z.Ii, drop=TRUE) %>%
  cut(brks) %>%
  table())-> tab
# .
#  (-Inf,-4.26] (-4.26,-3.72] (-3.72,-3.09] (-3.09,-2.33]
#          0          0          1          4
# (-2.33,-1.96]     (-1.96,0]     (0,1.96]     (1.96,2.33]
#          5         459        1239        195
#  (2.33,3.09]     (3.09,3.72]     (3.72,4.26]     (4.26, Inf]
#         316         145          55          76
sum(tab[c(1:5, 8:12)])
# [1] 797
sum(tab[c(1, 12)])
# [1] 76
```

In an important clarification, Sauer et al. (2021) show that the comparison of standard deviates for local Moran's I_i based on analytical formulae and conditional permutation in Bivand and Wong (2018) was based on a misunderstanding. Sokal et al. (1998) provide alternative analytical formulae for standard deviates of local Moran's I_i based either on total or conditional permutation, but

the analytical formulae used in Bivand and Wong (2018), based on earlier practice, only use total permutation, and consequently do not match the simulation conditional permutations. Thanks to a timely pull request, `localmoran()` now has a `conditional=` argument using alternative formulae from the appendix of Sokal et al. (1998):

```
z %>%
  localmoran(listw=lw_q_W, conditional=TRUE, alternative="two.sided") -> locm_c
```

yielding standard deviates that correspond closely to those from simulation:

```
locm_c %>%
  subset(select="Pr(z != 0)", drop=TRUE) %>%
  pva() -> pvsp
apply(pvsp, 2, f)
#      none bonferroni      fdr      BY
#    789        69        468      156

pol_pres15$locm_Z <- locm[, "Z.Ii"]
pol_pres15$locm_c_Z <- locm_c[, "Z.Ii"]
pol_pres15$locm_p_Z <- locm_p[, "Z.Ii"]
tm_shape(pol_pres15) + tm_fill(c("locm_Z", "locm_c_Z", "locm_p_Z"), breaks=brks,
  midpoint=0, title="Standard deviates of\nLocal Moran's I") +
  tm_facets(free.scales=FALSE, ncol=2) + tm_layout(panel.labels=c("Analytical total",
  "Analytical conditional", "Conditional permutation"))
```

Figure 15.3 shows that conditional permutation scales back the proportion of standard deviate values taking extreme values, especially positive values. The analytical total standard deviates of local Moran's I should probably not be used since alternatives are available, not least thanks to the clarification by Sauer et al. (2021).

In presenting local Moran's I , use is often made of “hotspot” maps. Because I_i takes high values both for strong positive autocorrelation of low and high values of the input variable, it is hard to show where “clusters” of similar neighbours with low or high values of the input variable occur. The quadrants of the Moran plot are used, by creating a categorical quadrant variable interacting the input variable and its spatial lag split at their means. The quadrant categories are then set to NA if, for the chosen standard deviate and adjustment, I_i would be considered insignificant. Here, for the Bonferroni adjusted conditional analytical standard deviates, 14 observations belong to “Low-Low clusters”, and 55 to “High-High clusters”:

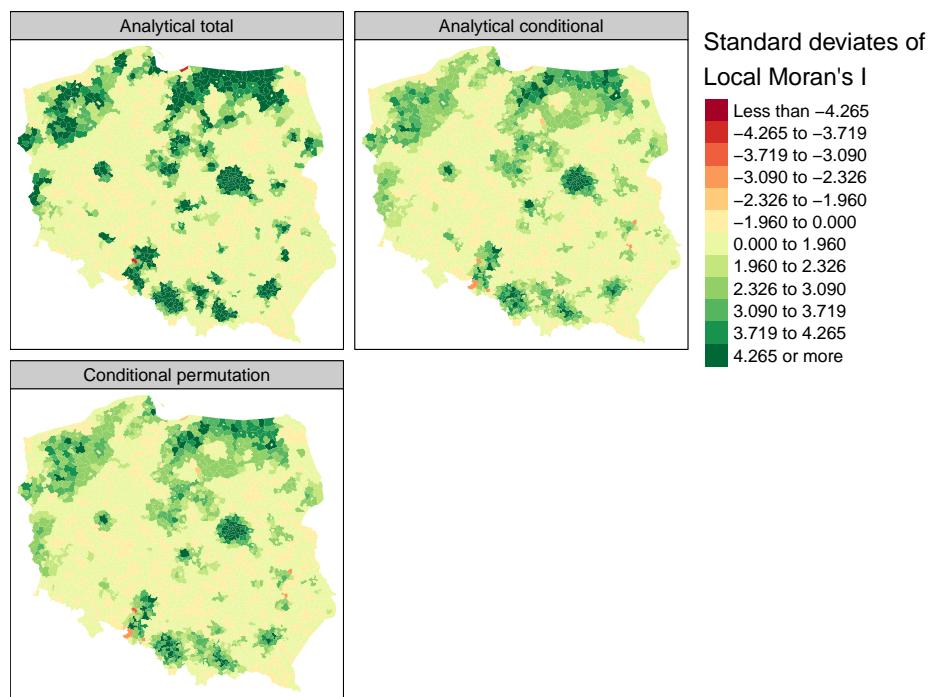


Figure 15.3: Analytical total and conditional permutation, and bootstrap conditional permutation standard deviates of local Moran's I for first round turnout, row-standardised neighbours

```

quadr <- interaction(
  cut(infl_W$x, c(-Inf, mean(infl_W$x), Inf), labels=c("Low X", "High X")),
  cut(infl_W$wx, c(-Inf, mean(infl_W$wx), Inf), labels=c("Low WX", "High WX")),
  sep=" : ")
a <- table(quadr)
pol_pres15$hs_an_q <- quadr
is.na(pol_pres15$hs_an_q) <- !(pol_pres15$locm_Z < brks[6] |
  pol_pres15$locm_Z > brks[8])
b <- table(pol_pres15$hs_an_q)
pol_pres15$hs_ac_q <- quadr
is.na(pol_pres15$hs_ac_q) <- !(pol_pres15$locm_c_Z < brks[2] |
  pol_pres15$locm_c_Z > brks[12])
c <- table(pol_pres15$hs_ac_q)
pol_pres15$hs_cp_q <- quadr
is.na(pol_pres15$hs_cp_q) <- !(pol_pres15$locm_p_Z < brks[2] |
  pol_pres15$locm_p_Z > brks[12])
d <- table(pol_pres15$hs_cp_q)
t(rbind("Moran plot quadrants"=a, "Unadjusted analytical total"=b,
  "Bonferroni analytical cond."=c, "Bonferroni cond. perm."=d))
#
# Moran plot quadrants Unadjusted analytical total
# Low X : Low WX          1040           370
# High X : Low WX         264            3
# Low X : High WX         213            0
# High X : High WX        978          342
#
# Bonferroni analytical cond.
# Low X : Low WX           14
# High X : Low WX          0
# Low X : High WX          0
# High X : High WX         55
#
# Bonferroni cond. perm.
# Low X : Low WX           18
# High X : Low WX          0
# Low X : High WX          0
# High X : High WX         58
tm_shape(pol_pres15) + tm_fill(c("hs_an_q", "hs_ac_q", "hs_cp_q"), colorNA="grey95",
  textNA="Not significant", title="Turnout hotspot status\nLocal Moran's I") +
  tm_facets(free.scales=FALSE, ncol=2) + tm_layout(panel.labels=
    c("Unadjusted analytical total", "Bonferroni analytical cond.",
      "Cond. perm. with Bonferroni"))

```

Figure 15.4 shows the impact of using analytical or conditional permutation standard deviates, and no or Bonferroni adjustment, reducing the counts of observations in “Low-Low clusters” from 370 to 14, and “High-High clusters” from 342 to 54; the “High-High clusters” are metropolitan areas.

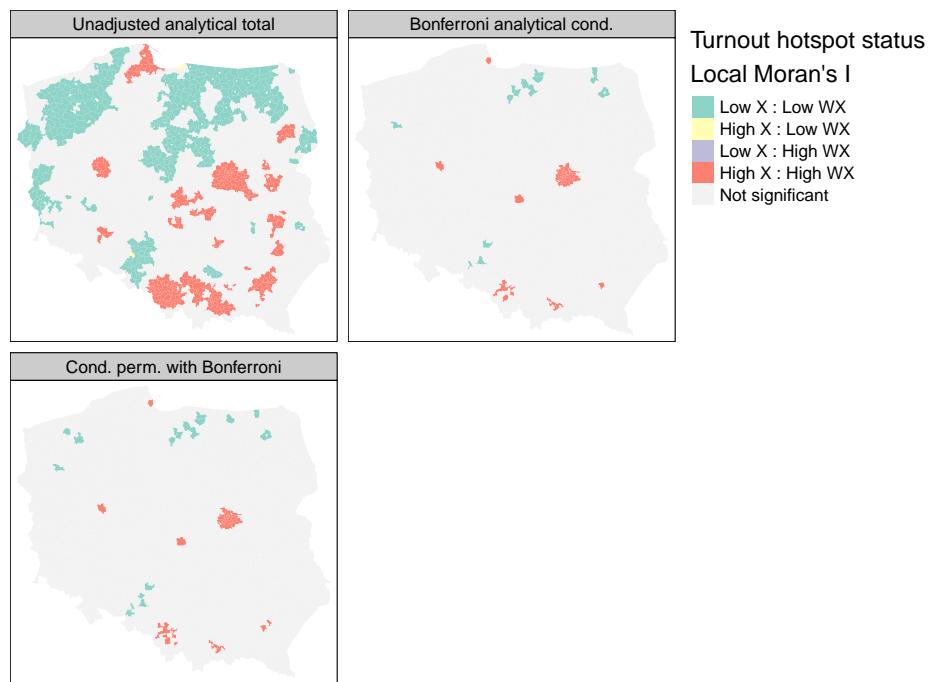


Figure 15.4: Local Moran's I hotspot maps $\alpha = 0.05$: left panel upper: unadjusted analytical standard deviates; right upper panel: Bonferroni adjusted analytical conditional standard deviates; left lower panel: Bonferroni adjusted bootstrap conditional permutation standard deviates, first round turnout, row-standardised neighbours

Tiefelsdorf (2002) argues that standard approaches to the calculation of the standard deviates of local Moran's I_i should be supplemented by numerical estimates, and shows that Saddlepoint approximations are a computationally efficient way of achieving this goal. The `localmoran.sad()` function takes a fitted linear model as its first argument, so we first fit a null (intercept only) model, but use case weights because the numbers entitled to vote vary greatly between observations:

```
lm(z ~ 1, weights=pol_pres15$I_entitled_to_vote) -> lm_null
```

Saddlepoint approximation is much more computationally intensive than conditional permutation:

```
system.time(lm_null %>% localmoran.sad(nb=nb_q, style="W", alternative="two.sided") %>%
  summary() -> locm_sad_null)
#   user  system elapsed
# 19.769   0.104 19.874
```

However, standard approaches do not permit richer mean models with covariates or case weights. Next we add the categorical variable distinguishing between rural, urban and other types of observational unit:

```
lm(z ~ Types, weights=pol_pres15$I_entitled_to_vote) -> lm_types

system.time(lm_types %>% localmoran.sad(nb=nb_q, style="W", alternative="two.sided") %>%
  summary() -> locm_sad_types)
#   user  system elapsed
# 21.052   0.004 21.088
```

To conclude, we add the spatially lagged categories, although the spatial lag of a categorical variable, represented by dummies in the model matrix, is not well defined:

```
spatialreg::lmSLX(z ~ Types, listw=lw_q_W,
  weights=pol_pres15$I_entitled_to_vote) -> lm_Dtypes

system.time(lm_Dtypes %>% localmoran.sad(nb=nb_q, style="W", alternative="two.sided") %>%
  summary() -> locm_sad_Dtypes)
#   user  system elapsed
# 21.869   0.008 21.906
```

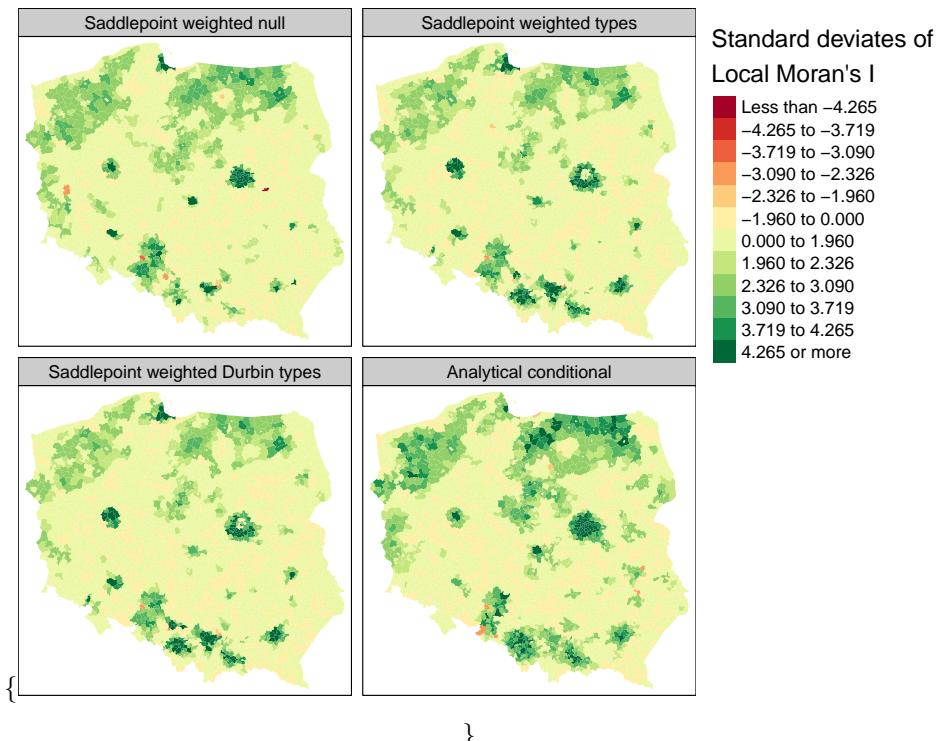
(ref:localmoranZfc_sad) Saddlepoint weighted null model, weighted types model, weighted Durbin types model, and analytical conditional standard deviates of local Moran's I for first round turnout, row-standardised neighbours

```

pol_pres15$locm_sad_null_Z <- locm_sad_null[, "Saddlepoint"]
pol_pres15$locm_sad_types_Z <- locm_sad_types[, "Saddlepoint"]
pol_pres15$locm_sad_Dtypes_Z <- locm_sad_Dtypes[, "Saddlepoint"]
tm_shape(pol_pres15) + tm_fill(c("locm_sad_null_Z", "locm_sad_types_Z",
  "locm_sad_Dtypes_Z", "locm_c_Z"), breaks=brks, midpoint=0,
  title="Standard deviates of\nnLocal Moran's I") +
  tm_facets(free.scales=FALSE, ncol=2) + tm_layout(panel.labels=c(
    "Saddlepoint weighted null",
    "Saddlepoint weighted types",
    "Saddlepoint weighted Durbin types",
    "Analytical conditional"))

```

\begin{figure}



\caption{(ref:localmoranZfc_sad)}(#fig:localmoranZ_sad) \end{figure}

Figure @ref(fig:localmoranZ_sad) includes the analytical conditional standard deviates for comparison (lower right panel), but in general it can be seen that the Saddlepoint approximation standard deviates lie closer to zero, possibly because some of the mis-specification in the mean model has been removed by using richer versions, and possibly because the approximation approach is inherently local, relating regression residual values at i to those of its neighbours. It is also possible to use Saddlepoint approximation where the

global spatial process has been incorporated, removing the conflation of global and local spatial autocorrelation in standard approaches. The same can also be accomplished using exact methods (Bivand et al., 2009).

15.3.2 Local Getis-Ord G_i

The local Getis-Ord G measure is reported as a standard deviate, and may also take the G_i^* form where self-neighbours are inserted into the neighbour object using `include.self()`. The observed and expected values of local G with their analytical variances may also be returned if `return_internals=TRUE`.

```
system.time(z %>%
  localG(lw_q_W) -> locG)
#   user  system elapsed
# 0.024 0.000 0.024
system.time(z %>%
  localG_perm(lw_q_W, nsim=499, iseed=1) -> locG_p)
#   user  system elapsed
# 0.025 1.024 1.420
```

Once again we face the problem of multiple comparisons, with the count of areal unit p-values < 0.05 being reduced by an order of magnitude when employing Bonferroni correction:

```
locG %>%
  c() %>%
  abs() %>%
  pnorm(lower.tail = FALSE) %>%
  (function(x) x*2)() %>%
  pva() -> pvsp
apply(pvsp, 2, f)
#      none bonferroni      fdr      BY
#      789         69       468      156

library(ggplot2)
p1 <- ggplot(data.frame(Zi=locm_c[,4], Zi_perm=locm_p[,4])) + geom_point(aes(x=Zi,
  y=Zi_perm), alpha=0.2) + xlab("Analytical conditional") +
  ylab("Bootstrap conditional") + coord_fixed() + ggtitle("Local Moran's I")
p2 <- ggplot(data.frame(Zi=c(locG), Zi_perm=c(locG_p))) + geom_point(aes(x=Zi,
  y=Zi_perm), alpha=0.2) + xlab("Analytical conditional") +
  ylab("Bootstrap conditional") + coord_fixed() + ggtitle("Local G")
gridExtra::grid.arrange(p1, p2, nrow=1)
```

Figure 15.5 shows that, when using analytical conditional standard deviates, the values from analytical and bootstrap estimates coincide for both I_i and G_i .

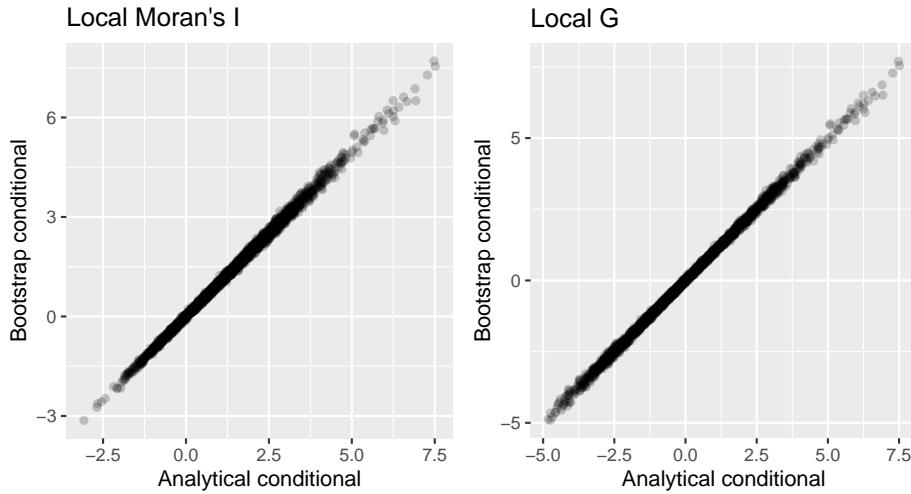


Figure 15.5: Plots of analytical conditional against bootstrap standard deviates; left: local Moran's I; right: local G; first round turnout, row-standardised neighbours

In both cases, one may argue that the bootstrap approach is superfluous in exploratory spatial data analysis.

```
pol_pres15$locG_Z <- c(locG)
pol_pres15$hs_G <- cut(c(locG), c(-Inf, brks[2], brks[12], Inf),
  labels=c("Low", "Not significant", "High"))
table(pol_pres15$hs_G)
#
#          Low Not significant           High
#             14            2426            55

m1 <- tm_shape(pol_pres15) + tm_fill(c("locG_Z"), midpoint=0, title="Standard\ndeviate")
m2 <- tm_shape(pol_pres15) + tm_fill(c("hs_G"),
  title="Local G Bonferroni\ncorrected hotspot status")
tmap_arrange(m1, m2, nrow=1)
```

As can be seen from Figure 15.5, we do not need to contrast the two estimation methods, and showing the mapped standard deviate is as informative as the “hotspot” status for the chosen adjustment (Figure 15.6). In the case of G_i , the values taken by the measure reflect the values of the input variable, so a “High cluster” is found for observations with high values of the input variable, here high turnout in metropolitan areas.

Very recently, Geoda has been wrapped for R as **rgeoda** (Li and Anselin, 2021), and will provide very similar functionalities for the exploration of

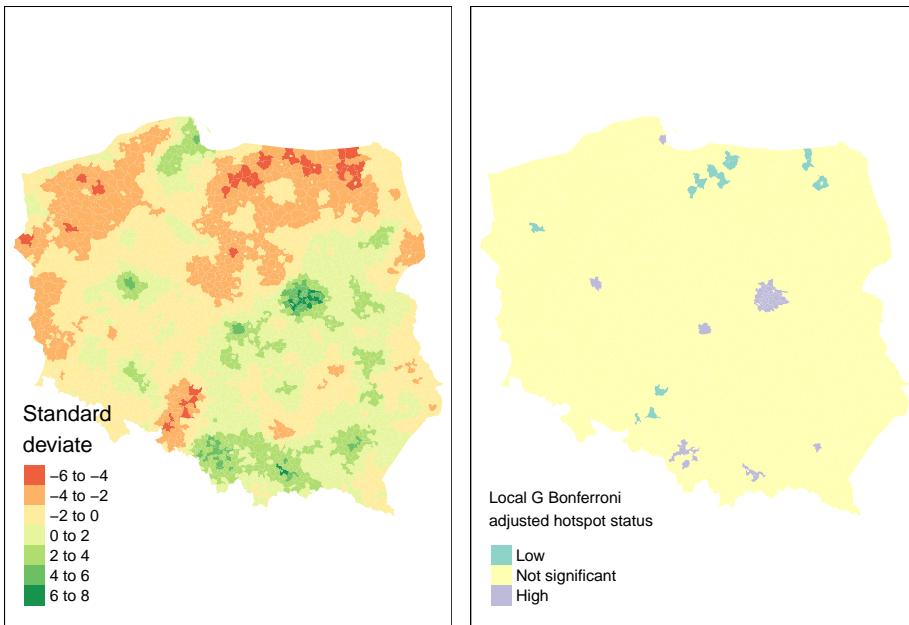


Figure 15.6: Left: analytical standard deviates of local G; right: Bonferroni hotspots; first round turnout, row-standardised neighbours

spatial autocorrelation in areal data as **spdep**. The active objects are kept as pointers to a compiled code workspace; using compiled code for all operations (as in Geoda itself) makes **rgeoda** perform fast, but leaves less flexible when modifications or enhancements are desired.

The contiguity neighbours it constructs are the same as those found by **poly2nb()**, as almost are the I_i measures. The difference is as established by Bivand and Wong (2018), that **localmoran()** calculates the input variable variance dividing by n , but Geoda uses $(n - 1)$, as can be reproduced by setting **mlvar=FALSE**:

```
library(rgeoda)
# Loading required package: digest
#
# Attaching package: 'rgeoda'
# The following object is masked from 'package:spdep':
#
#     skater
system.time(Geoda_w <- queen_weights(pol_pres15))
# here
#   user  system elapsed
# 0.136  0.008  0.144
```

```

summary(Geoda_w)
#               name          value
# 1 number of observations:      2495
# 2      is symmetric:        TRUE
# 3      sparsity: 0.00228786229774178
# 4      # min neighbors:       1
# 5      # max neighbors:      13
# 6      # mean neighbors:    5.70821643286573
# 7      # median neighbors:   6
# 8      has isolates:        FALSE
system.time(lisa <- local_moran(Geoda_w, pol_pres15["I_turnout"],
  cpu_threads=ifelse(parallel::detectCores() == 1, 1, parallel::detectCores()-1L),
  permutations=499, seed=1))
# user system elapsed
# 0.437  0.004  0.115
all.equal(card(nb_q), lisa_num_nbrs(lisa), check.attributes=FALSE)
# [1] TRUE
all.equal(lisa_values(lisa), localmoran(pol_pres15$I_turnout,
  listw=lw_q_W, mlvar=FALSE)[,1], check.attributes=FALSE)
# [1] TRUE

```

15.4 Exercises

1. Why are join-count measures on a chessboard so different between `rook` and `queen` neighbours?
2. Please repeat the simulation shown in section 15.1 using the chessboard polygons and the row-standardized `queen` contiguity neighbours. Why is it important to understand that spatial autocorrelation usually signals (unavoidable) mis-specification in our data?
3. Do we need to use conditional permutation in inference for local measures of spatial autocorrelation?
4. Why is false discovery rate adjustment recommended for local measures of spatial autocorrelation?
5. Compare the local Moran's I_i standard deviate values for the simulated data from exercise 15.2 for the analytical conditional approach, and Saddlepoint approximation. Consider the advantages and disadvantages of the Saddlepoint approximation approach.

Chapter 16

Spatial Regression

Even though it may be tempting to focus on interpreting the map pattern of an areal support response variable of interest, the pattern may largely derive from covariates (and their functional forms), as well as the respective spatial footprints of the variables in play. Spatial autoregressive models in two dimensions began without covariates and with clear links to time series (Whittle, 1954). Extensions included tests for spatial autocorrelation in linear model residuals, and models applying the autoregressive component to the response or the residuals, where the latter matched the tests for residuals (Cliff and Ord, 1972, 1973). These “lattice” models of areal data typically express the dependence between observations using a graph of neighbours in the form of a contiguity matrix.

Of course, handling a spatial correlation structure in a generalised least squares model or a (generalized) linear or nonlinear mixed effects model such as those provided in the **nlme** and many other packages does not have to use a graph of neighbours (Pinheiro and Bates, 2000). These models are also spatial regression models, using functions of the distance between observations, and fitted variograms to model the spatial autocorrelation present; such models have been held to yield a clearer picture of the underlying processes (Wall, 2004), building on geostatistics. For example, the **glmmTMB** package successfully uses this approach to spatial regression (Brooks et al., 2017). Here we will only consider spatial regression using spatial weights matrices.

16.1 Markov random field and multilevel models with spatial weights

There is a large literature in disease mapping using conditional autoregressive (CAR) and intrinsic CAR (ICAR) models in spatially structured random

effects. These extend to multilevel models, in which the spatially structured random effects may apply at different levels of the model (Bivand et al., 2017). In order to try out some of the variants, we need to remove the no-neighbour observations from the tract level, and from the model output zone aggregated level, in two steps as reducing the tract level induces a no-neighbour outcome at the model output zone level. Many of the model estimating functions take

`family=` arguments, and fit generalized linear mixed effects models with per-observation spatial random effects structured using a Markov random field representation of relationships between neighbours. In the multilevel case, the random effects may be modelled at the group level, which is the case presented in the following examples.

We follow Gómez-Rubio (2019) in summarizing Pinheiro and Bates (2000) and McCulloch and Searle (2001) to describe the mixed-effects model representation of spatial regression models. In a Gaussian linear mixed model setting, a random effect u is added to the model, with response Y , fixed covariates X , their coefficients β and error term $\varepsilon_i \sim N(0, \sigma^2)$, $i = 1, \dots, n$:

$$Y = X\beta + Zu + \varepsilon$$

Z is a fixed design matrix for the random effects. If there are n random effects, it will be an $n \times n$ identity matrix, if instead the observations are aggregated into m groups, so with $m < n$ random effects, it will be an $n \times m$ matrix showing which group each observation belongs to. The random effects are modelled as a multivariate Normal distribution $u \sim N(0, \sigma_u^2 \Sigma)$, and Σ is the square variance-covariance matrix of the random effects.

A division has grown up, possibly unhelpfully, between scientific fields using CAR models (Besag, 1974), and simultaneous autoregressive models (SAR) (Ord, 1975; Hepple, 1976). Although CAR and SAR models are closely related, these fields have found it difficult to share experience of applying similar models, often despite referring to key work summarising the models (Ripley, 1981, 1988; Cressie, 1993). Ripley gives the SAR variance as (1981, page 89), here shown as the inverse Σ^{-1} (also known as the precision matrix):

$$\Sigma^{-1} = [(I - \rho W)'(I - \rho W)]$$

where ρ is a spatial autocorrelation parameter and W is a nonsingular spatial weights matrix that represents spatial dependence. The CAR variance is:

$$\Sigma^{-1} = (I - \rho W)$$

where W is a symmetric and strictly positive definite spatial weights matrix. In the case of the intrinsic CAR model, avoiding the estimation of a spatial autocorrelation parameter, we have:

$$\Sigma^{-1} = M = \text{diag}(n_i) - W$$

where W is a symmetric and strictly positive definite spatial weights matrix as before and n_i are the row sums of W . The Besag-York-Mollié model includes intrinsic CAR spatially structured random effects and an unstructured random effects. The Leroux model combines matrix components for unstructured and spatially structured random effects, where the spatially structured random effects are taken as following an intrinsic CAR specification:

$$\Sigma^{-1} = [(1 - \rho)I_n + \rho M]$$

References to the definitions of these models may be found in Gómez-Rubio (2020b), and estimation issues affecting the Besag-York-Mollié and Leroux models are reviewed by Gerber and Furrer (2015).

More recent books expounding the theoretical bases for modelling with areal data simply point out the similarities between SAR and CAR models in relevant chapters (Gaetan and Guyon, 2010; van Lieshout, 2019); the interested reader is invited to consult these sources for background information.

16.1.1 Boston house value data set

Here we shall use the Boston housing data set, which has been restructured and furnished with census tract boundaries (Bivand, 2017). The original data set used 506 census tracts and a hedonic model to try to estimate willingness to pay for clean air. The response was constructed from counts of ordinal answers to a 1970 census question about house value. The response is left and right censored in the census source and has been treated as Gaussian. The key covariate was created from a calibrated meteorological model showing the annual nitrogen oxides (NOX) level for a smaller number of model output zones. The numbers of houses responding also varies by tract and model output zone. There are several other covariates, some measured at the tract level, some by town only, where towns broadly correspond to the air pollution model output zones.

We can start by reading in the 506 tract data set from **spData** (Bivand et al., 2021b), and creating a contiguity neighbour object and from that again a row standardized spatial weights object.

```
library(sf)
library(spData)
boston_506 <- st_read(system.file("shapes/boston_tracts.shp", package="spData")[1])
# Reading layer `boston_tracts' from data source
```

```
#   `/home/edzer/R/x86_64-pc-linux-gnu-library/4.0/spData/shapes/boston_tracts.shp'
#   using driver `ESRI Shapefile'
# Simple feature collection with 506 features and 36 fields
# Geometry type: POLYGON
# Dimension:      XY
# Bounding box:  xmin: -71.5 ymin: 42 xmax: -70.6 ymax: 42.7
# Geodetic CRS:  NAD27

nb_q <- spdep::poly2nb(boston_506)
lw_q <- spdep::nb2listw(nb_q, style="W")
```

If we examine the median house values, we find that those for censored values have been assigned as missing, and that 17 tracts are affected.

```
table(boston_506$censored)
#
#   left     no right
#       2      489      15

summary(boston_506$median)
#   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# 5600 16800 21000 21749 24700 50000    17
```

Next, we can subset to the remaining 489 tracts with non-censored house values, and the neighbour object to match. The neighbour object now has one observation with no neighbours.

```
boston_506$CHAS <- as.factor(boston_506$CHAS)
boston_489 <- boston_506[!is.na(boston_506$median),]
nb_q_489 <- spdep::poly2nb(boston_489)
lw_q_489 <- spdep::nb2listw(nb_q_489, style="W", zero.policy=TRUE)
```

The NOX_ID variable specifies the upper level aggregation, letting us aggregate the tracts to air pollution model output zones. We can create aggregate neighbour and row standardized spatial weights objects, and aggregate the NOX variable taking means, and the CHAS Charles River dummy variable for observations on the river. Here we follow the principles outlined in section 5.3.1 for spatially extensive and intensive variables; neither NOX nor CHAS can be summed as they are not count variables.

```
agg_96 <- list(as.character(boston_506$NOX_ID))
boston_96 <- aggregate(boston_506[, "NOX_ID"], by=agg_96, unique)
nb_q_96 <- spdep::poly2nb(boston_96)
```

```
lw_q_96 <- spdep::nb2listw(nb_q_96)
boston_96$NOX <- aggregate(boston_506$NOX, agg_96, mean)$x
boston_96$CHAS <- aggregate(as.integer(boston_506$CHAS)-1, agg_96, max)$x
```

The response is aggregated using the `weightedMedian()` function in `matrixStats`, and midpoint values for the house value classes. Counts of houses by value class were punched to check the published census values, which can be replicated using `weightedMedian()` at the tract level. Here we find two output zones with calculated weighted medians over the upper census question limit of USD 50,000, and remove them subsequently as they also are affected by not knowing the appropriate value to insert for the top class by value. This is a case of spatially extensive aggregation, for which the summation of counts is appropriate:

```
nms <- names(boston_506)
ccounts <- 23:31
for (nm in nms[c(22, ccounts, 36)]) {
  boston_96[[nm]] <- aggregate(boston_506[[nm]], agg_96, sum)$x
}
br2 <- c(3.50, 6.25, 8.75, 12.50, 17.50, 22.50, 30.00, 42.50, 60.00)*1000
counts <- as.data.frame(boston_96)[, nms[ccounts]]
f <- function(x) matrixStats::weightedMedian(x=br2, w=x, interpolate=TRUE)
boston_96$median <- apply(counts, 1, f)
is.na(boston_96$median) <- boston_96$median > 50000
summary(boston_96$median)
#   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
# 9009 20417 23523 25263 30073 49496 2
```

Before subsetting, we aggregate the remaining covariates by weighted mean using the tract population counts punched from the census (Bivand, 2017); these are spatially intensive variables, not count data.

```
boston_94 <- boston_96[!is.na(boston_96$median),]
nb_q_94 <- spdep::subset.nb(nb_q_96, !is.na(boston_96$median))
lw_q_94 <- spdep::nb2listw(nb_q_94, style="W")
```

We now have two data sets at each level, at the lower, census tract level, and at the upper, air pollution model output zone level, one including the censored observations, the other excluding them.

```
boston_94a <- aggregate(boston_489[, "NOX_ID"], list(boston_489$NOX_ID), unique)
nb_q_94a <- spdep::poly2nb(boston_94a)
NOX_ID_no_neighs <- boston_94a$NOX_ID[which(spdep::card(nb_q_94a) == 0)]
boston_487 <- boston_489[is.na(match(boston_489$NOX_ID, NOX_ID_no_neighs)),]
```

```
boston_93 <- aggregate(boston_487[, "NOX_ID"], list(ids = boston_487$NOX_ID), unique)
row.names(boston_93) <- as.character(boston_93$NOX_ID)
nb_q_93 <- spdep::poly2nb(boston_93, row.names=unique(as.character(boston_93$NOX_ID)))
```

The original model related the log of median house values by tract to the square of NOX values, including other covariates usually related to house value by tract, such as aggregate room counts, aggregate age, ethnicity, social status, distance to downtown and to the nearest radial road, a crime rate, and town-level variables reflecting land use (zoning, industry), taxation and education (Bivand, 2017). This structure will be used here to exercise issues raised in fitting spatial regression models, including the presence of multiple levels.

16.2 Multilevel models of the Boston data set

The ZN, INDUS, NOX, RAD, TAX and PTRATIO variables show effectively no variability within the TASSIM zones, so in a multilevel model the random effect may absorb their influence.

```
form <- formula(log(median) ~ CRIM + ZN + INDUS + CHAS + I((NOX*10)^2) + I(RM^2) +
  AGE + log(DIS) + log(RAD) + TAX + PTRATIO + I(BB/100) +
  log(I(LSTAT/100)))
```

16.2.1 IID random effects with lme4

The **lme4** package (Bates et al., 2021) lets us add an independent and identically distributed (IID) unstructured random effect at the model output zone level by updating the model formula with a random effects term:

```
library(Matrix)
library(lme4)
#
# Attaching package: 'lme4'
# The following object is masked from 'package:nlme':
#
#      lmList
MLM <- lmer(update(form, . ~ . + (1 | NOX_ID)), data=boston_487, REML=FALSE)
```

copying the random effect into the "sf" object for mapping below.

```
boston_93$MLM_re <- ranef(MLM)[[1]][,1]
```

16.2.2 IID and CAR random effects with hglm

The same model may be estimated using the **hglm** package (Alam et al., 2019), which also permits the modelling of discrete responses, this time using an extra one-sided formula to express the random effects term:

```
suppressPackageStartupMessages(library(hglm))
suppressWarnings(HGLM_iid <- hglm(fixed=form, random= ~1 | NOX_ID, data=boston_487,
                                    family=gaussian()))
boston_93$HGLM_re <- unname(HGLM_iid$ranef)
```

The same package has been extended to spatially structured SAR and CAR random effects, for which a sparse spatial weights matrix is required (Alam et al., 2015); we choose binary spatial weights:

```
W <- as(spdep::nb2listw(nb_q_93, style="B"), "CsparseMatrix")
```

We fit a CAR model at the upper level, using the `rand.family=` argument, where the values of the indexing variable `NOX_ID` match the row names of W :

```
suppressWarnings(HGLM_car <- hglm(fixed=form, random= ~ 1 | NOX_ID, data=boston_487,
                                       family=gaussian(), rand.family=CAR(D=W)))
boston_93$HGLM_ss <- HGLM_car$ranef[,1]
```

16.2.3 SAR random effects with HSAR

The **HSAR** package (Dong et al., 2020) is restricted to the Gaussian response case (Dong and Harris, 2015; Dong et al., 2015), and requires the specification of a sparse design matrix mapping the upper level entities onto lower level entities:

```
library(HSAR)
library(MatrixModels, warn.conflicts=FALSE)
Z <- as(model.Matrix(~ -1 + as.factor(NOX_ID), data=boston_487, sparse=TRUE),
        "dgCMatrix")
```

`hsar()` fits an upper level SAR random effect using MCMC; if `W=` is a lower level weights matrix rather than `NULL`, it will also fit a lower level spatial econometrics-style spatial lag model, adding the lower level spatially lagged response to the model:

```
set.seed(123)
suppressWarnings(HSAR_ss <- hsar(form, data=boston_487, W=NULL, M=W, Delta=Z,
                                     burnin=2000, Nsim=12000, thinning=2))

boston_93$HSAR_ss <- HSAR_ss$Mus[,]
```

16.2.4 IID and ICAR random effects with R2BayesX

The **R2BayesX** package (Umlauf et al., 2017) provides flexible support for structured additive regression models, including spatial multilevel models. The models include an IID unstructured random effect at the upper level using the "re" specification in the `sx()` model term (Umlauf et al., 2015); we choose the "MCMC" method:

```
suppressPackageStartupMessages(library(R2BayesX))

BX_iid <- bayesx(update(form, . ~ . + sx(NOX_ID, bs="re")), family="gaussian",
                    data=boston_487, method="MCMC", iterations=12000, burnin=2000, step=2, seed=123)

boston_93$BX_re <- BX_iid$effects["sx(NOX_ID):re"][[1]]$Mean
```

and the "mrf" (Markov Random Field) spatially structured intrinsic CAR random effect specification based on a graph derived from converting a suitable "nb" object for the upper level. The "region.id" attribute of the "nb" object needs to contain values corresponding to the indexing variable in the `sx()` effects term, to facilitate the internal construction of design matrix Z :

```
RBX_gra <- nb2gra(nb_q_93)
all.equal(row.names(RBX_gra), attr(nb_q_93, "region.id"))
# [1] TRUE
```

As we saw above in the intrinsic CAR model definition, the counts of neighbours are entered on the diagonal, but the current implementation uses a dense, not sparse, matrix:

```
all.equal(unname(diag(RBX_gra)), spdep::card(nb_q_93))
# [1] TRUE
```

The `sx()` model term continues to include the indexing variable, and now passes through the intrinsic CAR precision matrix:

```
BX_mrf <- bayesx(update(form, . ~ . + sx(NOX_ID, bs="mrf", map=RBX_gra)),
  family="gaussian", data=boston_487, method="MCMC",
  iterations=12000, burnin=2000, step=2, seed=123)

boston_93$BX_ss <- BX_mrf$effects["sx(NOX_ID):mrf"][[1]]$Mean
```

16.2.5 IID, ICAR and Leroux random effects with INLA

Bivand et al. (2015) and Gómez-Rubio (2020b) present the use of the **INLA** package (Rue et al., 2021) and the **inla()** model fitting function with spatial regression models:

```
suppressPackageStartupMessages(library(INLA))
```

Although differing in details, the approach by updating the fixed model formula with an unstructured random effects term is very similar to that seen above:

```
INLA_iid <- inla(update(form, . ~ . + f(NOX_ID, model="iid")), family="gaussian",
  data=boston_487)

boston_93$INLA_re <- INLA_iid$summary.random$NOX_ID$mean
```

As with most implementations, care is needed to match the indexing variable with the spatial weights; in this case using indices 1, …, 93 rather than the NOX_ID variable directly:

```
ID2 <- as.integer(as.factor(boston_487$NOX_ID))
```

The same sparse binary spatial weights matrix is used, and the intrinsic CAR representation is constructed internally:

```
INLA_ss <- inla(update(form, . ~ . + f(ID2, model="besag", graph=W)), family="gaussian",
  data=boston_487)

boston_93$INLA_ss <- INLA_ss$summary.random$ID2$mean
```

The sparse Leroux representation as given by Gómez-Rubio (2020b) can be constructed in the following way:

```
M <- Diagonal(nrow(W), rowSums(W)) - W
Cmatrix <- Diagonal(nrow(M), 1) - M
```

This model can be estimated using the "generic1" model with the specified precision matrix:

```
INLA_lr <- inla(update(form, . ~ . + f(ID2, model = "generic1", Cmatrix = Cmatrix)),
  family="gaussian", data=boston_487)

boston_93$INLA_lr <- INLA_lr$summary.random$ID2$mean
```

16.2.6 ICAR random effects with mgcv::gam()

In a very similar way, the `gam()` function in the `mgcv` package (Wood, 2021) can take an "mrf" term using a suitable "nb" object for the upper level. In this case the "nb" object needs to have the contents of the "region.id" attribute copied as the names of the neighbour list components, and the indexing variable needs to be a factor (Wood, 2017):

```
library(mgcv)
names(nb_q_93) <- attr(nb_q_93, "region.id")
boston_487$NOX_ID <- as.factor(boston_487$NOX_ID)
```

The specification of the spatially structured term again differs in details from those above, but achieves the same purpose. The "REML" method of `bayesx()` gives the same results as `gam()` using "REML" in this case:

```
GAM_MRF <- gam(update(form, . ~ . + s(NOX_ID, bs="mrf", xt=list(nb=nb_q_93))),
  data=boston_487, method="REML")
```

The upper level random effects may be extracted by predicting terms; as we can see, the values in all lower-level tracts belonging to the same upper-level air pollution model output zones are identical:

```
ssre <- predict(GAM_MRF, type="terms", se=FALSE)[, "s(NOX_ID)"]
all(sapply(tapply(ssre, list(boston_487$NOX_ID), c), function(x) length(unique(x)) == 1))
# [1] TRUE
```

so we can return the first value for each upper-level unit:

```
boston_93$GAM_ss <- aggregate(ssre, list(boston_487$NOX_ID), head, n=1)$x
```

16.2.7 Upper level random effects: summary

In the cases of `hglm()`, `bayesx()`, `inla()` and `gam()`, we could also model discrete responses without further major difficulty, and `bayesx()`, `inla()` and `gam()` also facilitate the generalization of functional form fitting for included covariates.

Unfortunately, the coefficient estimates for the air pollution variable for these multilevel models are not helpful. All are negative as expected, but the inclusion of the model output zone level effects, IID or spatially structured, makes it is hard to disentangle the influence of the scale of observation from that of covariates observed at that scale rather than at the tract level.

Figure 16.1 shows that the air pollution model output zone level IID random effects are very similar across the four model fitting functions reported. In all the maps, the central downtown zones have stronger negative random effect values, but strong positive values are also found in close proximity; suburban areas take values closer to zero.

```
library(tmap, warn.conflicts=FALSE)
tm_shape(boston_93) + tm_fill(c("MLM_re", "HGLM_re", "INLA_re", "BX_re"), midpoint=0,
  title="IID") + tm_facets(free.scales=FALSE) + tm_borders(lwd=0.3, alpha=0.4) +
  tm_layout(panel.labels=c("lmer", "hglm", "inla", "bayesx"))
```

Figure 16.2 shows that the spatially structured random effects are also very similar to each other, with the "SAR" spatial smooth being perhaps a little smoother than the "CAR" smooths when considering the range of values taken by the random effect term.

```
tm_shape(boston_93) + tm_fill(c("HGLM_ss", "HSAR_ss", "INLA_lr", "INLA_ss", "BX_ss",
  "GAM_ss"), midpoint=0, title="SSRE") + tm_facets(free.scales=FALSE) +
  tm_borders(lwd=0.3, alpha=0.4) + tm_layout(panel.labels=c("hglm CAR", "hsar SAR",
  "inla Leroux", "inla ICAR", "bayesx ICAR", "gam ICAR"))
```

Although there is still a great need for more thorough comparative studies of model fitting functions for spatial regression including multilevel capabilities, there has been much progress over recent years. Vranckx et al. (2019) offer a recent comparative survey of disease mapping spatial regression, typically set in a Poisson regression framework offset by an expected count. In Bivand and Gómez-Rubio (2021), methods for estimating spatial survival models using spatial weights matrices are compared with spatial probit models.

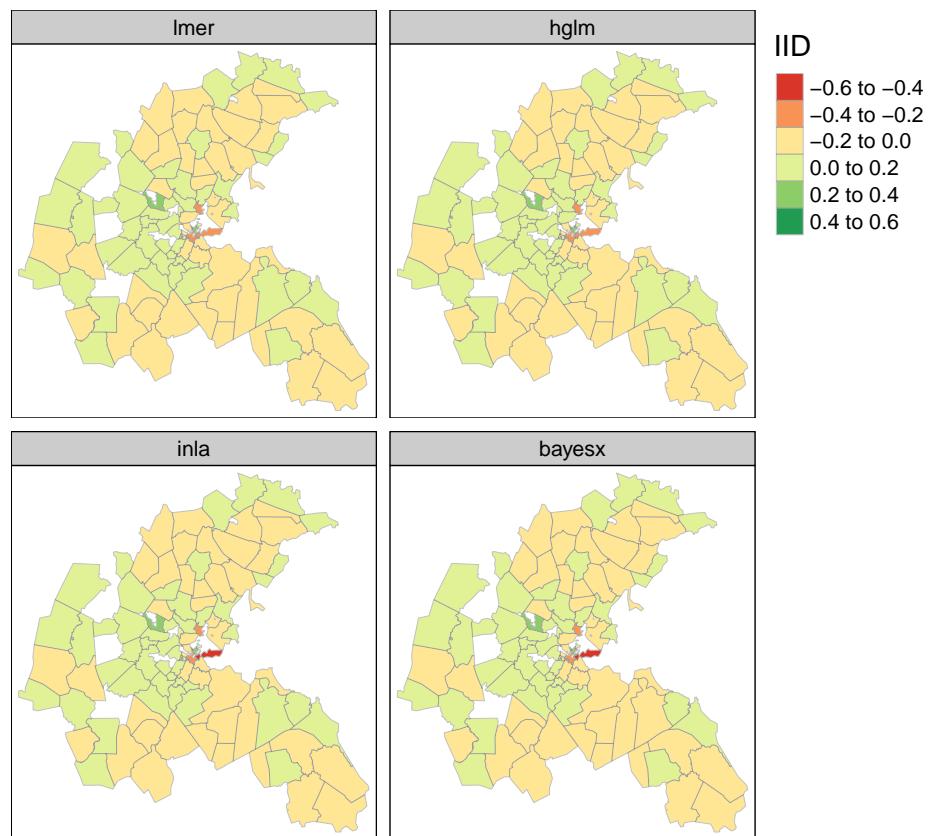


Figure 16.1: Air pollution model output zone level IID random effects estimated using **lme4**, **hglm**, **INLA** and **R2BayesX**; the range of the response, `log(median)` is 2.1893.

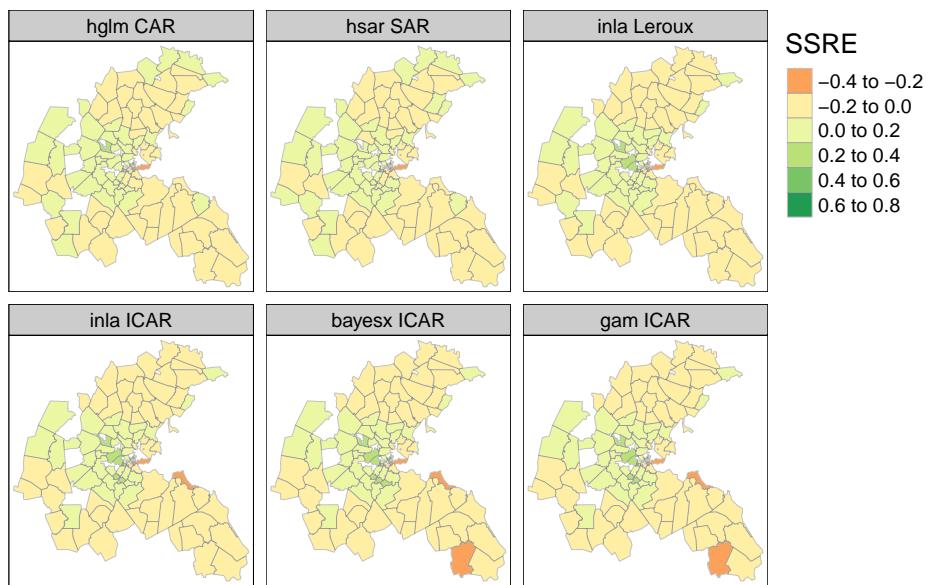


Figure 16.2: Air pollution model output zone level spatially structured random effects estimated using **hglm**, **HSAR**, **INLA**, **R2BayesX** and **mgcv**.

Chapter 17

Spatial econometrics models

Spatial autoregression models using spatial weights matrices were described in some detail using maximum likelihood estimation some time ago (Cliff and Ord, 1973, 1981). A family of models was elaborated in spatial econometric terms extending earlier work, and in many cases using the simultaneous autoregressive framework and row standardization of spatial weights (Anselin, 1988). The simultaneous and conditional autoregressive frameworks can be compared, and both can be supplemented using case weights to reflect the relative importance of different observations (Waller and Gotway, 2004).

Before moving to presentations of issues raised in fitting spatial regression models, it is worth making a few further points. A recent review of spatial regression in a spatial econometrics setting is given by Kelejian and Piras (2017); note that their usage is to call the spatial coefficient of the lagged response λ and that of the lagged residuals ρ , the reverse of other usage (Anselin, 1988; LeSage and Pace, 2009); here we use ρ_{Lag} for the spatial coefficient in the spatial lag model, and ρ_{Err} for the spatial error model. One interesting finding is that relatively dense spatial weights matrices may downweight model estimates, suggesting that sparser weights are preferable (Smith, 2009). Another useful finding is that the presence of residual spatial autocorrelation need not bias the estimates of variance of regression coefficients, provided that the covariates themselves do not exhibit spatial autocorrelation (Smith and Lee, 2012). In general, however, the footprints of the spatial processes of the response and covariates may not be aligned, and if covariates and the residual are autocorrelated, it is likely that the estimates of variance of regression coefficients will be biased downwards if attempts are not made to model the spatial processes.

17.1 Spatial econometric models: definitions

In trying to model spatial processes, one of the earliest spatial econometric representations is to model the spatial autocorrelation in the residual (spatial error model, SEM):

$$\mathbf{y} = \mathbf{X} + \mathbf{u}, \quad \mathbf{u} = \rho_{\text{Err}} \mathbf{W} \mathbf{u} + ,$$

where \mathbf{y} is an $(N \times 1)$ vector of observations on a response variable taken at each of N locations, \mathbf{X} is an $(N \times k)$ matrix of covariates, β is a $(k \times 1)$ vector of parameters, \mathbf{u} is an $(N \times 1)$ spatially autocorrelated disturbance vector, ϵ is an $(N \times 1)$ vector of independent and identically distributed disturbances and ρ_{Err} is a scalar spatial parameter.

This model, and other spatial econometric models, do not fit into the mixed models framework. Here the modelled spatial process interacts directly with the response, covariates, and their coefficients. This modelling framework appears to draw on an older tradition extending time series to two dimensions:

$$\mathbf{u} = (\mathbf{I} - \rho_{\text{Err}} \mathbf{W})^{-1} , \quad \mathbf{y} = \mathbf{X} + (\mathbf{I} - \rho_{\text{Err}} \mathbf{W})^{-1} , \quad (\mathbf{I} - \rho_{\text{Err}} \mathbf{W})\mathbf{y} = (\mathbf{I} - \rho_{\text{Err}} \mathbf{W})\mathbf{X} + .$$

If the processes in the covariates and the response match, we should find little difference between the coefficients of a least squares and a SEM, but very often they diverge, suggesting that a Hausman test for this condition should be employed (Pace and LeSage, 2008). This may be related to earlier discussions of a spatial equivalent to the unit root and cointegration where spatial processes match (Fingleton, 1999).

A model with a spatial process in the response only is termed a spatial lag model (SLM, often SAR - spatial autoregressive) (LeSage and Pace, 2009). Durbin models add the spatially lagged covariates to the covariates included in the spatial model; spatial Durbin models are reviewed by Mur and Angulo (2006). If it is chosen to admit a spatial process in the residuals in addition to a spatial process in the response, again two models are formed, a general nested model (GNM) nesting all the others, and a model without spatially lagged covariates (SAC, also known as SARAR - Spatial

AutoRegressive-AutoRegressive model). If neither the residuals nor the response are modelled with spatial processes, spatially lagged covariates may be added to a linear model, as a spatially lagged X model (SLX) (Elhorst, 2010; Bivand, 2012; LeSage, 2014; Halleck Vega and Elhorst, 2015). We can write the general nested model (GNM) as:

$$\mathbf{y} = \rho_{\text{Lag}} \mathbf{W} \mathbf{y} + \mathbf{X} + \mathbf{W} \mathbf{X} + \mathbf{u}, \quad \mathbf{u} = \rho_{\text{Err}} \mathbf{W} \mathbf{u} + ,$$

where β is a $(k' \times 1)$ vector of parameters. k' defines the subset of the intercept and covariates, often $k' = k - 1$ when using row standardised spatial weights and omitting the spatially lagged intercept.

This may be constrained to the double spatial coefficient model SAC/SARAR by setting $\rho_{Lag} = 0$, to the spatial Durbin (SDM) by setting $\rho_{Err} = 0$, and to the error Durbin model (SDEM) by setting $\rho_{Lag} = 0$. Imposing more conditions gives the spatial lag model (SLM) with $\rho_{Lag} = 0$ and $\rho_{Err} = 0$, the spatial error model (SEM) with $\rho_{Lag} = 0$ and $\rho_{Err} = 0$, and the spatially lagged X model (SLX) with $\rho_{Lag} = 0$ and $\rho_{Err} = 0$.

Although making predictions for new locations for which covariates are observed was raised as an issue some time ago, it has many years to make progress in reviewing the possibilities (Bivand, 2002; Goulard et al., 2017; Laurent and Margaretic, 2021). The prediction methods for SLM, SDM, SEM, SDEM, SAC and GNM models fitted with maximum likelihood were contributed as a Google Summer of Coding project by Martin Gubri. This work, and work on similar models with missing data (Suesse, 2018) is also relevant for exploring censored median house values in the Boston data set. Work on prediction also exposed the importance of the reduced form of these models, in which the spatial process in the response interacts with the regression coefficients in the SLM, SDM, SAC and GNM models.

The consequence of these interactions is that a unit change in a covariate will only impact the response as the value of the regression coefficient if the spatial coefficient of the lagged response is zero. Where it is non-zero, global spillovers, impacts, come into play, and these impacts should be reported rather than the regression coefficients (LeSage and Pace, 2009; Elhorst, 2010; Bivand, 2012; LeSage, 2014; Halleck Vega and Elhorst, 2015). Local impacts may be reported for SDEM and SLX models, using linear combination to calculate standard errors for the total impacts of each covariate (sums of coefficients on the covariates and their spatial lags).

This can be seen from the GNM data generation process:

$$(\mathbf{I} - \rho_{Err} \mathbf{W})(\mathbf{I} - \rho_{Lag} \mathbf{W})\mathbf{y} = (\mathbf{I} - \rho_{Err} \mathbf{W})(\mathbf{X} + \mathbf{WX}) + ,$$

re-writing:

$$\mathbf{y} = (\mathbf{I} - \rho_{Lag} \mathbf{W})^{-1}(\mathbf{X} + \mathbf{WX}) + (\mathbf{I} - \rho_{Lag} \mathbf{W})^{-1}(\mathbf{I} - \rho_{Err} \mathbf{W})^{-1} .$$

There is interaction between the ρ_{Lag} and ρ_{Err} (and ρ_{Lag} if present) coefficients. This can be seen from the partial derivatives:

$\partial y_i / \partial x_{jr} = ((\mathbf{I} - \rho_{Lag} \mathbf{W})^{-1}(\mathbf{I}\beta_r + \mathbf{W}\gamma_r))_{ij}$. This dense matrix $S_r(\mathbf{W}) = ((\mathbf{I} - \rho_{Lag} \mathbf{W})^{-1}(\mathbf{I}\beta_r + \mathbf{W}\gamma_r))$ expresses the direct impacts (effects) on its principal diagonal, and indirect impacts in off-diagonal elements.

Piras and Prucha (2014) revisit and correct Florax et al. (2003) (see also comments by Hendry (2006) and Florax et al. (2006)), finding that the common use of pre-test strategies for model selection probably ought to be replaced by the estimation of the most general model appropriate for the relationships being modelled. In the light of this finding, pre-test model selection will not be used here.

Current work in the **spatialreg** package is focused on refining the handling of spatially lagged covariates using a consistent **Durbin=** argument taking either a logical value or a formula giving the subset of covariates to add in spatially lagged form. There is a speculation that some covariates, for example some dummy variables, should not be added in spatially lagged form. This then extends to handling these included spatially lagged covariates appropriately in calculating impacts. This work applies to cross-sectional models fitted using MCMC or maximum likelihood, and will offer facilities to spatial panel models.

It is worth mentioning the almost unexplored issues of functional form assumptions, for which flexible structures are useful, including spatial quantile regression presented in the **McSpatial** package (McMillen, 2013). There are further issues with discrete response variables, covered by some functions in **McSpatial**, and in the **spatialprobit** and **ProbitSpatial** packages (Wilhelm and de Matos, 2013; Martinetti and Geniaux, 2017); the MCMC implementations of the former are based on LeSage and Pace (2009). Finally, Wagner and Zeileis (2019) show how an SLM model may be used in the setting of recursive partitioning, with an implementation using **spatialreg::lagsarlm()** in the **lagsarlmtree** package.

The review of cross-sectional maximum likelihood and generalized method of moments (GMM) estimators in **spatialreg** (Bivand and Piras, 2021) and **sphet** for spatial econometrics style spatial regression models by Bivand and Piras (2015) is still largely valid. In the review, estimators in these R packages were compared with alternative implementations available in other programming languages elsewhere. The review did not cover Bayesian spatial econometrics style spatial regression. More has changed with respect to spatial panel estimators described in Millo and Piras (2012), but will not be covered here.

Because Bivand et al. (2021a) covers many of the features of R packages for spatial econometrics, updating Bivand and Piras (2015), and including recent advances in General Method of Moments and spatial panel modelling, this chapter will be restricted to a small number of examples drawing on Bivand (2017) using the Boston house value data set.

17.2 Maximum likelihood estimation in **spatialreg**

For models with single spatial coefficients (SEM and SDEM using **errorsarlm()**, SLM and SDM using **lagsarlm()**), the methods initially described by Ord (1975) are used. The following table shows the functions that can be used to estimate the models described above using maximum likelihood.

model	model name	maximum likelihood estimation function
SEM	spatial error	errorsarlm(..., Durbin=FALSE, ...)
SEM	spatial error	spautolm(..., family="SAR", ...)
SDEM	spatial Durbin error	errorsarlm(..., Durbin=TRUE, ...)
SLM	spatial lag	lagsarlm(..., Durbin=FALSE, ...)
SDM	spatial Durbin	lagsarlm(..., Durbin=TRUE, ...)
SAC	spatial autoregressive combined	sacsarlm(..., Durbin=FALSE, ...)
GNM	general nested	sacsarlm(..., Durbin=TRUE, ...)

The estimating functions **errorsarlm()** and **lagsarlm()** take similar arguments, where the first two, **formula=** and **data=** are shared by most model estimating functions. The third argument is a **listw** spatial weights object, while **na.action=** behaves as in other model estimating functions if the spatial weights can reasonably be subsetted to avoid observations with missing values. The **weights=** argument may be used to provide weights indicating the known degree of per-observation variability in the variance term - this is not available for **lagsarlm()**.

The **Durbin=** argument replaces the earlier **type=** and **etype=** arguments, and if not given is taken as **FALSE**. If given, it may be **FALSE**, **TRUE** in which case all spatially lagged covariates are included, or a one-sided formula specifying which spatially lagged covariates should be included. The **method=** argument gives the method for calculating the log determinant term in the log likelihood function, and defaults to **"eigen"**, suitable for moderately sized data sets. The **interval=** argument gives the bounds of the domain for the line search using **stats::optimize()** used for finding the spatial coefficient. The **tol.solve()** argument, passed through to **base::solve()**, was needed to handle data sets with differing numerical scales among the coefficients which hindered inversion of the variance-covariance matrix; the default value in **base::solve()** used to be much larger. The **control=** argument takes a list of control values to permit more careful adjustment of the running of the estimation function.

The **sacsarlm()** function may take second spatial weights and interval arguments if the spatial weights used to model the two spatial processes in the SAC and GNM specifications differ. By default, the same spatial weights are used. By default, **stats::nlminb()** is used for numerical optimization, using

a heuristic to choose starting values. Like `lagsarlm()`, this function does not take a `weights=` argument.

Where larger data sets are used, a numerical Hessian approach is used to calculate the variance-covariance matrix of coefficients, rather than an analytical asymptotic approach.

17.2.1 Boston house value data set examples

The examples use the objects read and created in chapter 16, based on Bivand (2017).

```
library(spatialreg)
#
# Attaching package: 'spatialreg'
# The following objects are masked from 'package:spdep':
#
#     as_dgRMatrix_listw, as_dsCMatrix_I, as_dsCMatrix_IrW,
#     as_dstMatrix_listw, as.spam.listw, can.be.simmed,
#     cheb_setup, create_WX, do_ldet, eigen_pre_setup,
#     eigen_setup, eigenw, errorsarlm, get.ClusterOption,
#     get.coresOption, get.mcOption, get.VerboseOption,
#     get.ZeroPolicyOption, GMargminImage, GMerrorsar,
#     griffith_sone, gstsls, Hausman.test, impacts,
#     intImpacts, Jacobian_W, jacobianSetup, l_max, lagmess,
#     lagsarlm, lextrB, lextrS, lextrW, lmSLX,
#     LU_prepermute_setup, LU_setup, Matrix_J_setup,
#     Matrix_setup, mcdet_setup, MCMCsamp, ME, mom_calc,
#     mom_calc_int2, moments_setup, powerWeights, sacsarlm,
#     SE_classic_setup, SE_interp_setup, SE_whichMin_setup,
#     set.ClusterOption, set.coresOption, set.mcOption,
#     set.VerboseOption, set.ZeroPolicyOption,
#     similar.listw, spam_setup, spam_update_setup,
#     SpatialFiltering, spautolm, spBreg_err, spBreg_lag,
#     spBreg_sac, stsls, subgraph_eigenw, trW
eigs_489 <- eigenw(lw_q_489)
SDEM_489 <- errorsarlm(form, data=boston_489, listw=lw_q_489, Durbin=TRUE,
                        zero.policy=TRUE, control=list(pre_eig=eigs_489))
SEM_489 <- errorsarlm(form, data=boston_489, listw=lw_q_489,
                        zero.policy=TRUE, control=list(pre_eig=eigs_489))
```

Here we are using the `control=` list argument to pass through pre-computed eigenvalues for the default "eigen" method.

```
cbind(data.frame(model=c("SEM", "SDEM")),
      rbind(broom::tidy(Hausman.test(SEM_489)),
            broom::tidy(Hausman.test(SDEM_489))))[,1:4]
#   model statistic p.value parameter
# 1   SEM      52.0 2.83e-06      14
# 2   SDEM     48.7 6.48e-03      27
```

Both Hausman test results for the 489 tract data set suggest that the regression coefficients do differ from their non-spatial counterparts, perhaps indicating that the footprints of the spatial processes do not match.

```
eigs_94 <- eigenw(lw_q_94)
SDEM_94 <- errorsarlm(form, data=boston_94, listw=lw_q_94, Durbin=TRUE,
                      control=list(pre_eig=eigs_94))
SEM_94 <- errorsarlm(form, data=boston_94, listw=lw_q_94, control=list(pre_eig=eigs_94))
```

For the 94 air pollution model output zones, the Hausman tests find little difference between coefficients:

```
cbind(data.frame(model=c("SEM", "SDEM")),
      rbind(broom::tidy(Hausman.test(SEM_94)),
            broom::tidy(Hausman.test(SDEM_94))))[, 1:4]
#   model statistic p.value parameter
# 1   SEM      15.66 0.335      14
# 2   SDEM      9.21 0.999      27
```

This is related to the fact that the SEM and SDEM models add little to least squares or SLX at the air pollution model output zone level, using likelihood ratio tests:

```
cbind(data.frame(model=c("SEM", "SDEM")), rbind(broom::tidy(LR1.Sarlm(SEM_94)),
                                                 broom::tidy(LR1.Sarlm(SDEM_94))))[,c(1, 4:6)]
#   model statistic p.value parameter
# 1   SEM      2.593 0.107      1
# 2   SDEM      0.216 0.642      1
```

We can use `spatialreg::LR.Sarlm()` to apply a likelihood ratio test between nested models, but here choose `lmtest::lrtest()`, which gives the same results, preferring models including spatially lagged covariates both for tracts and model output zones:

```
broom::tidy(lmtest::lrtest(SEM_489, SDEM_489))
# # A tibble: 2 x 5
```

```
#   X.Df LogLik   df statistic  p.value
#   <dbl> <dbl> <dbl>      <dbl>    <dbl>
# 1    16  273.    NA       NA     NA
# 2    29  311.    13      74.4  1.23e-10
```

```
broom::tidy(lmtest::lrtest(SEM_94, SDEM_94))
# # A tibble: 2 x 5
#   X.Df LogLik   df statistic  p.value
#   <dbl> <dbl> <dbl>      <dbl>    <dbl>
# 1    16  59.7    NA       NA     NA
# 2    29  81.3    13      43.2  0.0000421
```

The SLX model is fitted using least squares, and also returns a log likelihood value, letting us test whether we need a spatial process in the residuals. In the tract data set we obviously do:

```
SLX_489 <- lmSLX(form, data=boston_489, listw=lw_q_489, zero.policy=TRUE)
broom::tidy(lmtest::lrtest(SLX_489, SDEM_489))
# # A tibble: 2 x 5
#   X.Df LogLik   df statistic  p.value
#   <dbl> <dbl> <dbl>      <dbl>    <dbl>
# 1    28  231.    NA       NA     NA
# 2    29  311.    1       159.  1.55e-36
```

but in the output zone case we do not.

```
SLX_94 <- lmSLX(form, data=boston_94, listw=lw_q_94)
broom::tidy(lmtest::lrtest(SLX_94, SDEM_94))
# # A tibble: 2 x 5
#   X.Df LogLik   df statistic p.value
#   <dbl> <dbl> <dbl>      <dbl>    <dbl>
# 1    28  81.2    NA       NA     NA
# 2    29  81.3    1       0.216   0.642
```

These outcomes are sustained also when we use the counts of house units by tract and output zones as case weights:

```
SLX_489w <- lmSLX(form, data=boston_489, listw=lw_q_489, weights=units, zero.policy=TRUE,
SDEM_489w <- errorsarlm(form, data=boston_489, listw=lw_q_489, Durbin=TRUE,
weights=units, zero.policy=TRUE, control=list(pre_eig=eigs_489))
broom::tidy(lmtest::lrtest(SLX_489w, SDEM_489w))
# # A tibble: 2 x 5
#   X.Df LogLik   df statistic  p.value
#   <dbl> <dbl> <dbl>      <dbl>    <dbl>
```

```

# 1    28   311.     NA      NA  NA
# 2    29   379.     1     136.  1.70e-31

SLX_94w <- lmSLX(form, data=boston_94, listw=lw_q_94, weights=units)
SDEM_94w <- errorsarlm(form, data=boston_94, listw=lw_q_94, Durbin=TRUE, weights=units,
                         control=list(pre_eig=eigs_94))
broom::tidy(lmtest::lrtest(SLX_94w, SDEM_94w))
# # A tibble: 2 x 5
#   X.Df LogLik   df statistic p.value
#   <dbl> <dbl> <dbl>     <dbl>    <dbl>
# 1    28   97.5    NA     NA      NA
# 2    29   98.0     1     0.917   0.338

```

In this case and based on arguments advanced in Bivand (2017), the use of weights is justified because tract counts of reported housing units underlyingng the weighted median values vary from 5 to 3,031, and air pollution model output zone counts vary from 25 to 12,411. Because of this, and because a weighted general nested model has not been developed, we cannot take the GNM as the starting point for general-to-simpler testing, but start rather from the SDEM model, and use the Hausman test to guide the choice of units of observation.

17.3 Impacts

Global impacts have been seen as crucial for reporting results from fitting models including the spatially lagged response (SLM, SDM, SAC, GNM) for over ten years (LeSage and Pace, 2009). Extension to other models including spatially lagged covariates (SLX, SDEM) has followed (Elhorst, 2010; Bivand, 2012; Halleck Vega and Elhorst, 2015). For SLM, SDM, SAC and GNM models fitted with maximum likelihood or GMM, the variance-covariance matrix of the coefficients is available, and can be used to make random draws from a multivariate Normal distribution with mean set to coefficient values and variance to the estimated variance-covariance matrix. For these models fitted using Bayesian methods, draws are already available. In the SDEM case, the draws on the regression coefficients of the unlagged covariates represent direct impacts, and draws on the coefficients of the spatially lagged covariates represent indirect impacts, and their by-draw sums the total impacts.

Since sampling is not required for inference for SLX and SDEM models, linear combination is used for models fitted using maximum likelihood; results are shown here for the air pollution variable only. The literature has not yet resolved the question of how to report model output, as each covariate is now represented by three impacts. Where spatially lagged covariates are included,

two coefficients are replaced by three impacts, here for the air pollution variable of interest.

```
sum_imp_94_SDEM <- summary(impacts(SDEM_94))
rbind(Impacts=sum_imp_94_SDEM$mat[5,], SE=sum_imp_94_SDEM$semat[5,])
#           Direct Indirect   Total
# Impacts -0.01276 -0.01845 -0.0312
# SE       0.00235  0.00472  0.0053
```

In the SLX and SDEM models, the direct impacts are the consequences for the response of changes in air pollution in the same observational entity, and the indirect (local) impacts are the consequences for the response of changes in air pollution in neighbouring observational entities.

```
sum_imp_94_SLX <- summary(impacts(SLX_94))
rbind(Impacts=sum_imp_94_SLX$mat[5,], SE=sum_imp_94_SLX$semat[5,])
#           Direct Indirect   Total
# Impacts -0.0128 -0.01874 -0.03151
# SE       0.0028  0.00556  0.00611
```

Applying the same approaches to the weighted spatial regressions, the total impacts of air pollution on house values are reduced, but remain significant:

```
sum_imp_94_SDEMw <- summary(impacts(SDEM_94w))
rbind(Impacts=sum_imp_94_SDEMw$mat[5,], SE=sum_imp_94_SDEMw$semat[5,])
#           Direct Indirect   Total
# Impacts -0.00592 -0.01076 -0.01668
# SE       0.00269  0.00531  0.00559
```

On balance, using a weighted spatial regression representation including only the spatially lagged covariates aggregated to the air pollution model output zone level seems to clear most of the mis-specification issues, and as Bivand (2017) discusses in more detail, gives a willingness to pay for pollution abatement that is much larger than mis-specified alternative models:

```
sum_imp_94_SLXw <- summary(impacts(SLX_94w))
rbind(Impacts=sum_imp_94_SLXw$mat[5,], SE=sum_imp_94_SLXw$semat[5,])
#           Direct Indirect   Total
# Impacts -0.00620 -0.01221 -0.01842
# SE       0.00326  0.00628  0.00629
```

17.4 Predictions

In the Boston tracts data set, 17 observations of median house values, the response, are censored. We will use the `predict()` method for "Sarlm" objects

to fill in these values; the method was re-written by Martin Gubri based on Goulard et al. (2017; see also Laurent and Margaretic, 2021). The `pred.type=` argument specifies the prediction strategy among those presented in the article.

Using these as an example and comparing some `pred.type=` variants for the SDEM model and predicting out-of-sample, we can see that there are differences, suggesting that this is a fruitful area for study. There have been a number of alternative proposals for handling missing variables (Gómez-Rubio et al., 2015; Suesse, 2018). Another reason for increasing attention on prediction is that it is fundamental for machine learning approaches, in which prediction for validation and test data sets drives model specification choice. The choice of training and other data sets with dependent spatial data remains an open question, and is certainly not as simple as with independent data.

Here, we'll list the predictions for the censored tract observations using three different prediction types, taking the exponent to get back to the USD median house values. Note that the `row.names()` of the `newdata=` object are matched with the whole-data spatial weights matrix "`region.id`" attribute to make out-of-sample prediction possible:

```
nd <- boston_506[is.na(boston_506$median),]
t0 <- exp(predict(SDEM_489, newdata=nd, listw=lw_q, pred.type="TS", zero.policy=TRUE))
suppressWarnings(t1 <- exp(predict(SDEM_489, newdata=nd, listw=lw_q, pred.type="KP2",
                                    zero.policy=TRUE)))
suppressWarnings(t2 <- exp(predict(SDEM_489, newdata=nd, listw=lw_q, pred.type="KP5",
                                    zero.policy=TRUE)))
```

We can also use the "`slm`" model in INLA to predict missing response values as part of the model fitting function call. A certain amount of set-up code is required as the "`srm`" model is still experimental:

```
library(INLA)
W <- as(lw_q, "CsparseMatrix")
n <- nrow(W)
e <- eigenw(lw_q)
re.idx <- which(abs(Im(e)) < 1e-6)
rho.max <- 1/max(Re(e[re.idx]))
rho.min <- 1/min(Re(e[re.idx]))
rho <- mean(c(rho.min, rho.max))
boston_506$idx <- 1:n
zero.variance = list(prec=list(initial = 25, fixed=TRUE))
args.srm <- list(rho.min = rho.min, rho.max = rho.max, W = W, X = matrix(0, n, 0),
                 Q.beta = matrix(1,0,0))
hyper.srm <- list(prec = list(prior = "loggamma", param = c(0.01, 0.01)),
                   rho = list(initial=0, prior = "logitbeta", param = c(1,1)))
WX <- create_WX(model.matrix(update(form, CMEDV ~ .), data=boston_506), lw_q)
```

```
SDEM_506_slm <- inla(update(form, . ~ . + WX + f(idx, model="slm", args.slm=args.slm,
  hyper=hyper.slm)), data=boston_506, family="gaussian", control.family=
  list(hyper=zero.variance), control.compute=list(dic=TRUE, cpo=TRUE))
mvs <- SDEM_506_slm$marginals.fitted.values[which(is.na(boston_506$median))]
mv_mean <- sapply(mvs, function(x) mean(exp(x[, 1])), )
mv_sd <- sapply(mvs, function(x) sd(exp(x[, 1])))
```

INLA also provide gridded estimates of the marginal distributions of the predictions, offering a way to assess the uncertainty associated with the predicted values:

	fit_TS	fit_KP2	fit_KP5	INLA_slm	INLA_slm_sd	censored
# 13	23912	29477	28147	33941	15901	right
# 14	28126	27001	28516	34512	17148	right
# 15	30553	36184	32476	45061	21574	right
# 17	18518	19621	18878	22842	10045	right
# 43	9564	6817	7561	7420	3332	left
# 50	8371	7196	7383	7508	3518	left
# 312	51477	53301	54173	62103	31468	right
# 313	45921	45823	47095	51123	25328	right
# 314	44196	44586	45361	46778	22165	right
# 317	43427	45707	45442	52252	24140	right
# 337	39879	42072	41127	44763	19567	right
# 346	44708	46694	46108	49153	20254	right
# 355	48188	49068	48911	53177	23605	right
# 376	42881	45883	44966	51685	23040	right
# 408	44294	44615	45670	50515	23959	right
# 418	38211	43375	41914	47610	21486	right
# 434	41647	41690	42398	45045	20239	right

The spatial regression toolbox remains incomplete, and it will take time to fill in blanks. It remains unfortunate that the several traditions in spatial regression seldom seem to draw on each others' understandings and advances.

Chapter 18

Older R Spatial Packages

18.1 Retiring `rgdal` and `rgeos`

R users who have been around a bit longer, in particular before packages like `sf` and `stars` were developed, may be more familiar with older packages like `maptools`, `sp`, `rgeos`, and `rgdal`. A fair question is whether they should migrate existing code and/or existing R packages depending on these packages.

The answer is: yes.

Unless someone steps up to volunteer maintaining packages `maptools`, `rgdal` and `rgeos`, the plan is to retire packages by the end of 2023. Retirement means that maintenance will halt, and that as a consequence the packages will sooner or later disappear from CRAN. One reason for retirement is that their maintainer has retired, another that their role has been superseded by the newer packages. We hold it not very likely that a new maintainer will take over, in part because much of the code of these packages has over a few decades gradually evolved along with developments in the GEOS, GDAL and PROJ libraries, and now contains numerous constructs that are no longer necessary and make it hard to read.

Before `rgeos` and `rgdal` retire, existing ties that package `sp` has to `rgdal` and `rgeos` can and will be replaced by ties to package `sf`. This only involves validation of coordinate reference system identifiers, and checking whether rings are holes or exterior rings. Theoretically one could replace `rgdal` and `rgeos` with packages that would call into `sf` for their ties to the GEOS, GDAL and PROJ libraries but that would involve a major effort.

18.2 links and differences between sf and sp

There are a number of differences between `sf` and `sp`. The most notable is that `sp` classes are formal, S4 classes where `sf` uses the (more) informal S3 class hierarchy. `sf` objects derive from data.frames, or from tibbles, and as such can be used easier with much of the other infrastructure of R and e.g. with the tidyverse package family. `sf` objects keep geometry in a list-column, meaning that a geometry is *always* a list element. Package `sp` used data structures much less strictly, and for instance all coordinates of `SpatialPoints` or `SpatialPixels` are kept in matrices, which is much more performant for certain problems but is not possible with a list-column.

Conversion from an `sf` object `x` to its `sp` equivalent is done by

```
library(sp)
y = as(x, "Spatial")
```

and the conversion the other way around is done by

```
x0 = st_as_sf(y)
```

There are some limitations to conversions like this:

- `sp` does not distinguish between `LINESTRING` and `MULTILINESTRING` geometries, or between `POLYGON` or `MULTIPOLYGON`, so e.g. a `LINESTRING` will after conversion to `sp` come back as a `MULTILINESTRING`
- `sp` does have no representation for `GEOMETRYCOLLECTION` geometries, or `sf` objects with geometries *not* in the “big seven” (section 3.1.1)
- `sf` or `sfc` objects of geometry type `GEOMETRY`, with mixed geometry types, cannot be converted into `sp` objects
- attribute-geometry relationship attributes get lost when converting to `sp`
- `sf` objects with more than one geometry list-column will, when converting to `sp`, lose their secondary list-column(s).

18.3 migration code and packages

The wiki page of the GitHub site for `sf`, found at

<https://github.com/r-spatial/sf/wiki/Migrating>

contains a list of methods and functions in `rgeos`, `rgdal` and `sp` and the corresponding `sf` method or function. This may help converting existing code or packages.

A simple approach to migrate code is when only `rgdal::readOGR` is used to read `file`. As an alternative, one might use

```
x = as(sf::read_sf("file"), "Spatial")
```

however possible arguments to `readOGR`, when used, would need more care.

An effort by us is underway to convert all code of our earlier book “Applied Spatial Data Analysis with R” (with Virgilio Gomez-Rubio, Bivand et al. (2013)) to run entirely without `rgdal`, `rgeos` and `maptools` and where possible without `sp`. The scripts are found at
https://github.com/rsbivand/sf_asdar2ed.

18.4 Package raster and terra

Package `raster` has been a workhorse package for analysing raster data with R since 2010, and has since then grown into a package for “Geographic Data Analysis and Modeling” (Hijmans, 2021a), indicating that it is used for all kinds of raster data. The `raster` package uses `sp` objects for vector data, and `rgdal` to read and write data to formats served by the GDAL library. A follow-up package, `terra`, for “Spatial Data Analysis” (Hijmans, 2021b), “is very similar to the ‘raster’ package; but [...] can do more, is easier to use, and [...] is faster”. The `terra` package comes with its own classes for vector data, but accepts many `sf` objects, with similar restrictions as listed above for conversion to `sp`.

Raster maps, or stacks of them from package `raster` or `terra` can be converted to `stars` objects using `st_as_stars()`. Package `sf` contains an `st_as_sf()` method for `SpatVector` objects from package `terra`. Package `terra` has its own direct links to GDAL, GEOS and PROJ so no longer needs other packages for that. Migration from `raster` to `terra` may become more important once `rgdal` is no longer easily installable (section 18.1).

The online book “Spatial Data Science with R”, written by Robert Hijmans and found at <https://rspatial.org/terra> details the `terra` approach to spatial data analysis. Package `sf` and `stars` and several other r-spatial packages discussed in this book reside on the `r-spatial` github organisation (note the hyphen between `r` and `spatial`, which is absent on Hijmans’ organisation), which has a blog site, with links to this book, found at <https://r-spatial.org/>.

Packages `sf` and `stars` on one hand and `terra` on the other have a lot of goals in common, but try to reach them in different ways, emphasizing different aspects of data analysis, software engineering, and community management. Although this may confuse some users, we believe that these differences are beneficial, encourage diversity and choice, and hopefully work as an encouragement for others to continue trying out new ideas when using R for spatial data problems.

R basics

This chapter provides some minimal set of R basics that may make it easier to read this book. A more comprehensive book on R basics is given in (Wickham, 2014a), chapter 2.

Pipes

The `%>%` (pipe) symbols should be read as *then*: we read

```
a %>% b() %>% c() %>% d(n = 10)
```

as *with a do b then c then d with n being 10*, and that is just alternative syntax for

```
d(c(b(a)), n = 10)
```

or

```
tmp1 <- b(a)
tmp2 <- c(tmp1)
tmp3 <- d(tmp2, n = 10)
```

To many, the pipe-form is easier to read because execution order follows reading order, from left to right. Like nested function calls, it avoids the need to choose names for intermediate results, but like nested function calls, it is hard to debug intermediate results that diverge from out expectations. Note that the intermediate results do exist in memory, so neither form saves memory allocation. The pipe that appeared natively in R 4.1.0, `|>`, can be used anywhere in this book where `%>%` is used. The reason we kept to the `%>%` pipe is to not exclude users of older R version to use the code sections in this book.

Data structures

As pointed out by (Chambers, 2016), *everything that exists in R is an object*. This includes objects that make things happen, such as language objects or functions, but also the more basic “things”, such as data objects. Some basic R data structures will now be discussed.

Homogeneous vectors

Data objects contain data, and possibly metadata. Data is always in the form of a vector, which can have different type. We can find the type by `typeof`, and vector length by `length`. Vectors are created by `c`, which combines individual elements:

```
typeof(1:10)
# [1] "integer"
length(1:10)
# [1] 10
typeof(1.0)
# [1] "double"
length(1.0)
# [1] 1
typeof(c("foo", "bar"))
# [1] "character"
length(c("foo", "bar"))
# [1] 2
typeof(c(TRUE, FALSE))
# [1] "logical"
```

Vectors of this kind can only have a single type.

Note that vectors can have zero length, e.g. in,

```
i = integer(0)
typeof(i)
# [1] "integer"
i
# integer(0)
length(i)
# [1] 0
```

We can retrieve (or in assignments: replace) elements in a vector using `[` or `[:`:

```
a = c(1,2,3)
a[2]
# [1] 2
a[[2]]
# [1] 2
a[2:3]
# [1] 2 3
a[2:3] = c(5,6)
a
# [1] 1 5 6
a[[3]] = 10
a
# [1] 1 5 10
```

where the difference is that `[` can operate on an index *range* (or multiple indexes), and `[[` operates on a single vector value.

Heterogeneous vectors: list

An additional vector type is the `list`, which can combine any types in its elements:

```
l <- list(3, TRUE, "foo")
typeof(l)
# [1] "list"
length(l)
# [1] 3
```

For lists, there is a further distinction between `[` and `[[`: the single `[` returns always a list, and `[[` returns the *contents* of a list element:

```
l[1]
# [[1]]
# [1] 3
l[[1]]
# [1] 3
```

For replacement, one case use `[` when providing a list, and `[[` when providing a new value:

```
l[1:2] = list(4, FALSE)
l
# [[1]]
```

```
# [1] 4
#
# [[2]]
# [1] FALSE
#
# [[3]]
# [1] "foo"
l[[3]] = "bar"
l
# [[1]]
# [1] 4
#
# [[2]]
# [1] FALSE
#
# [[3]]
# [1] "bar"
```

In case list elements are *named*, as in

```
l = list(first = 3, second = TRUE, third = "foo")
l
# $first
# [1] 3
#
# $second
# [1] TRUE
#
# $third
# [1] "foo"
```

we can use names as in `l[["second"]]` and this can be abbreviated to

```
l$second
# [1] TRUE
l$second = FALSE
l
# $first
# [1] 3
#
# $second
# [1] FALSE
#
# $third
# [1] "foo"
```

This is convenient, but also requires name look-up in the names attribute (see below).

NULL and removing list elements

NULL is the null value in R; it is special in the sense that it doesn't work in simple comparisons:

```
3 == NULL # not FALSE!
# logical(0)
NULL == NULL # not even TRUE!
# logical(0)
```

but has to be treated specially, using `is.null`:

```
is.null(NULL)
# [1] TRUE
```

When we want to remove one or more list elements, we can do so by creating a new list that does not contain the elements that needed removal, as in

```
l = l[c(1,3)] # remove second, implicitly
l
# $first
# [1] 3
#
# $third
# [1] "foo"
```

but we can also assign NULL to the element we want to eliminate:

```
l$second = NULL
l
# $first
# [1] 3
#
# $third
# [1] "foo"
```

Attributes

We can glue arbitrary metadata objects to data objects, as in

```
a = 1:3
attr(a, "some_meta_data") = "foo"
a
# [1] 1 2 3
# attr(,"some_meta_data")
# [1] "foo"
```

and this can be retrieved, or replaced by

```
attr(a, "some_meta_data")
# [1] "foo"
attr(a, "some_meta_data") = "bar"
attr(a, "some_meta_data")
# [1] "bar"
```

In essence, the attribute of an object is a named list, and we can get or set the complete list by

```
attributes(a)
# $some_meta_data
# [1] "bar"
attributes(a) = list(some_meta_data = "foo")
attributes(a)
# $some_meta_data
# [1] "foo"
```

A number of attributes are treated specially by R, see e.g. `?attributes`.

object class and class attribute

Every object in R “has a class”, meaning that `class(obj)` returns a character vector with the class of `obj`. Some objects have an *implicit* class, e.g. vectors

```
class(1:3)
# [1] "integer"
class(c(TRUE, FALSE))
# [1] "logical"
class(c("TRUE", "FALSE"))
# [1] "character"
```

but we can also set the class explicit, either by using `attr` or by using `class` in the left-hand side of an expression:

```
a = 1:3
class(a) = "foo"
a
# [1] 1 2 3
# attr("class")
# [1] "foo"
class(a)
# [1] "foo"
attributes(a)
# $class
# [1] "foo"
```

in which case the newly set class overrides the earlier implicit class. This way, we can add methods for class `foo`, e.g. by

```
print.foo = function(x, ...) print(paste("an object of class foo with length", length(x)))
print(a)
# [1] "an object of class foo with length 3"
```

Providing such methods are generally intended to create more usable software, but at the same time they may make the objects more opaque. It is sometimes useful to see what an object “is made of” by printing it after the class attribute is removed, as in

```
unclass(a)
# [1] 1 2 3
```

As a more elaborate example, consider the case where a polygon is made using package `sf`:

```
library(sf)
p = st_polygon(list(rbind(c(0,0), c(1,0), c(1,1), c(0,0))))
p
# POLYGON ((0 0, 1 0, 1 1, 0 0))
```

which prints the well-known-text form; to understand what the data structure is like, we can use

```
unclass(p)
# [[1]]
#      [,1] [,2]
# [1,]    0    0
# [2,]    1    0
# [3,]    1    1
# [4,]    0    0
```

the dim attribute

The `dim` attribute sets the matrix or array dimensions:

```
a = 1:8
class(a)
# [1] "integer"
attr(a, "dim") = c(2,4) # or: dim(a) = c(2,4)
class(a)
# [1] "matrix" "array"
a
#      [,1] [,2] [,3] [,4]
# [1,]    1    3    5    7
# [2,]    2    4    6    8
attr(a, "dim") = c(2,2,2) # or: dim(a) = c(2,2,2)
class(a)
# [1] "array"
a
# , , 1
#
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4
#
# , , 2
#
#      [,1] [,2]
# [1,]    5    7
# [2,]    6    8
```

various names attributes

Named vectors carry their names in a `names` attribute. We saw examples for lists above, an example for a numeric vector is:

```
a = c(first = 3, second = 4, last = 5)
a["second"]
# second
#      4
attributes(a)
# $names
# [1] "first"  "second" "last"
```

More name attributes are e.g. `dimnames` of matrices or arrays, which not only names dimensions, but also the labels associated with each of the dimensions:

```
a = matrix(1:4, 2, 2)
dimnames(a) = list(rows = c("row1", "row2"), cols = c("col1", "col2"))
a
#      cols
# rows   col1 col2
# row1    1    3
# row2    2    4
attributes(a)
# $dim
# [1] 2 2
#
# $dimnames
# $dimnames$rows
# [1] "row1" "row2"
#
# $dimnames$cols
# [1] "col1" "col2"
```

Data.frame objects have rows and columns, and each have names:

```
df = data.frame(a = 1:3, b = c(TRUE, FALSE, TRUE))
attributes(df)
# $names
# [1] "a" "b"
#
# $class
# [1] "data.frame"
#
# $row.names
# [1] 1 2 3
```

using structure

When programming, the pattern of adding or modifying attributes before returning an object is extremely common, an example being:

```
f = function(x) {
  a = create_obj(x) # call some other function
  attributes(a) = list(class = "foo", meta = 33)
  a
}
```

The last two statements can be contracted in

```
f = function(x) {
  a = create_obj(x) # call some other function
  structure(a, class = "foo", meta = 33)
}
```

where function `structure` adds, replaces, or (in case of value `NULL`) removes attributes from the object in its first argument.

dissecting a MULTIPOLYGON

We can use the above examples to dissect an `sf` object with MULTIPOLYGONS into pieces. Suppose we use the `nc` dataset,

```
library(sf)
system.file("gpkg/nc.gpkg", package = "sf") %>%
  read_sf() -> nc
```

we can see from the attributes of `nc`,

```
attributes(nc)
# $names
# [1] "AREA"      "PERIMETER"   "CNTY_"       "CNTY_ID"     "NAME"
# [6] "FIPS"       "FIPSNO"      "CRESS_ID"    "BIR74"      "SID74"
# [11] "NWBIR74"   "BIR79"       "SID79"       "NWBIR79"   "geom"
#
# $row.names
#  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
# [16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
# [31] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
# [46] 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
# [61] 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
# [76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
# [91] 91 92 93 94 95 96 97 98 99 100
#
# $class
# [1] "sf"        "tbl_df"     "tbl"        "data.frame"
#
# $sf_column
# [1] "geom"
#
# $agr
#   AREA PERIMETER      CNTY_      CNTY_ID      NAME      FIPS
# <NA>  <NA>      <NA>      <NA>      <NA>      <NA>
```

```

#   FIPSNO  CRESS_ID      BIR74      SID74      NWBIR74      BIR79
#   <NA>     <NA>      <NA>      <NA>      <NA>      <NA>
#   SID79    NWBIR79
#   <NA>     <NA>
# Levels: constant aggregate identity

```

that the geometry column is named `geom`. When we take out this column,

```

nc$geom
# Geometry set for 100 features
# Geometry type: MULTIPOLYGON
# Dimension: XY
# Bounding box: xmin: -84.3 ymin: 33.9 xmax: -75.5 ymax: 36.6
# Geodetic CRS: NAD27
# First 5 geometries:
# MULTIPOLYGON (((-81.5 36.2, -81.5 36.3, -81.6 3...
# MULTIPOLYGON (((-81.2 36.4, -81.2 36.4, -81.3 3...
# MULTIPOLYGON (((-80.5 36.2, -80.5 36.3, -80.5 3...
# MULTIPOLYGON (((-76 36.3, -76 36.3, -76 36.3, -...
# MULTIPOLYGON (((-77.2 36.2, -77.2 36.2, -77.3 3...

```

we see an object that has the following attributes

```

attributes(nc$geom)
# $n_empty
# [1] 0
#
# $crs
# Coordinate Reference System:
#   User input: NAD27
#   wkt:
# GEOGCRS["NAD27",
#   DATUM["North American Datum 1927",
#     ELLIPSOID["Clarke 1866",6378206.4,294.978698213898,
#     LENGTHUNIT["metre",1]],
#   PRIMEM["Greenwich",0,
#     ANGLEUNIT["degree",0.0174532925199433]],
#   CS[ellipsoidal,2],
#     AXIS["geodetic latitude (Lat)",north,
#       ORDER[1],
#       ANGLEUNIT["degree",0.0174532925199433]],
#     AXIS["geodetic longitude (Lon)",east,
#       ORDER[2],
#       ANGLEUNIT["degree",0.0174532925199433]],
#   
```

```

#      USAGE[
#          SCOPE["Geodesy."],
#          AREA["North and central America: Antigua and Barbuda - onshore. Bahamas - on",
#              BBOX[7.15,167.65,83.17,-47.74]],
#          ID["EPSG",4267]]
#
# $class
# [1] "sfc_MULTIPOLYGON" "sfc"
#
# $precision
# [1] 0
#
# $bbox
# xmin  ymin  xmax  ymax
# -84.3 33.9 -75.5 36.6

```

When we take the *contents* of the fourth list element, we obtain

which is a list,

```
typeof(nc$geom[[4]])  
# [1] "list"
```

with attributes

```
attributes(nc$geom[[4]])  
# $class  
# [1] "XY"                      "MULTIPOLYGON" "sfq"
```

and length

```
length(nc$geom[[4]])  
# [1] 3
```

The length indicates the number of outer rings: a multi-polygon can consist of more than one polygon. We see that most counties only have a single polygon:

A multi-polygon is a list with polygons,

```
typeof(nc$geom[[4]])
# [1] "list"
```

and the *first* polygon of the fourth multi-polygon is again a list, because polygons have an outer ring *possibly* followed by multiple inner rings (holes)

```
typeof(nc$geom[[4]][[1]])
# [1] "list"
```

we see that it contains only one ring, the exterior ring:

```
length(nc$geom[[4]][[1]])
# [1] 1
```

and we can print type, the dimension and the first set of coordinates by

```
typeof(nc$geom[[4]][[1]][[1]])
# [1] "double"
dim(nc$geom[[4]][[1]][[1]])
# [1] 26 2
head(nc$geom[[4]][[1]][[1]])
#      [,1] [,2]
# [1,] -76.0 36.3
# [2,] -76.0 36.3
# [3,] -76.0 36.3
# [4,] -76.0 36.4
# [5,] -76.1 36.3
# [6,] -76.2 36.4
```

and we can now for instance change the latitude of the third coordinate by

```
nc$geom[[4]][[1]][[1]][3,2] = 36.5
```


Bibliography

- Alam, M., Rønnegård, L., and Shen, X. (2015). Fitting conditional and simultaneous autoregressive spatial models in *hglm*. *The R Journal*, 7(2):5–18.
- Alam, M., Ronnegard, L., and Shen, X. (2019). *hglm: Hierarchical Generalized Linear Models*. R package version 2.2-1.
- Anselin, L. (1988). *Spatial econometrics: methods and models*. Kluwer Academic Publishers.
- Anselin, L. (1995). Local indicators of spatial association - LISA. *Geographical Analysis*, 27(2):93–115.
- Anselin, L. (1996). The Moran scatterplot as an ESDA tool to assess local instability in spatial association. In Fischer, M. M., Scholten, H. J., and Unwin, D., editors, *Spatial Analytical Perspectives on GIS*, pages 111–125. Taylor & Francis, London.
- Appel, M. (2020). *gdalcubes: Earth Observation Data Cubes from Satellite Image Collections*. R package version 0.3.1.
- Appel, M. and Pebesma, E. (2019). On-demand processing of data cubes from satellite image collections with the *gdalcubes* library. *Data*, 4(3):92.
- Appelhans, T., Detsch, F., Reudenbach, C., and Woellauer, S. (2021). *mapview: Interactive Viewing of Spatial Data in R*. R package version 2.10.0.
- Assunção, R. and Reis, E. A. (1999). A new proposal to adjust Moran’s I for population density. *Statistics in Medicine*, 18:2147–2162.
- Avis, D. and Horton, J. (1985). Remarks on the sphere of influence graph. In Goodman, J. E., editor, *Discrete Geometry and Convexity*, pages 323–327. New York Academy of Sciences, New York, New York.
- Aybar, C. (2021). *rgee: R Bindings for Calling the Earth Engine API*. R package version 1.0.9.
- Baddeley, A., Rubak, E., and Turner, R. (2015). *Spatial point patterns: methodology and applications with R*. Chapman and Hall/CRC.

- Baddeley, A., Turner, R., and Rubak, E. (2021). *spatstat: Spatial Point Pattern Analysis, Model-Fitting, Simulation, Tests*. R package version 2.2-0.
- Bates, D. and Maechler, M. (2021). *Matrix: Sparse and Dense Matrix Classes and Methods*. R package version 1.3-4.
- Bates, D., Maechler, M., Bolker, B., and Walker, S. (2021). *lme4: Linear Mixed-Effects Models using Eigen and S4*. R package version 1.1-27.1.
- Baumann, P., Hirschorn, E., and Masó, J. (2017). Ogc coverage implementation schema. *OGC Implementation Standard*.
- Bavaud, F. (1998). Models for spatial weights: a systematic look. *Geographical Analysis*, 30:153–171.
- Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300.
- Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *The Annals of Statistics*, 29(4):1165 – 1188.
- Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 36:pp. 192–236.
- BIPM, I., IFCC, I., IUPAC, I., and ISO, O. (2012). The international vocabulary of metrology—basic and general concepts and associated terms (vim), 3rd edn. jcgm 200: 2012. *JCGM (Joint Committee for Guides in Metrology)*.
- Bivand, R. (2017). Revisiting the Boston data set — changing the units of observation affects estimated willingness to pay for clean air. *REGION*, 4(1):109–127.
- Bivand, R. (2020a). *classInt: Choose Univariate Class Intervals*. R package version 0.4-3.
- Bivand, R. (2020b). *Why have CRS, projections and transformations changed?*
- Bivand, R. (2021a). *CRAN Task View: Analysis of Spatial Data*.
- Bivand, R. (2021b). *spdep: Spatial Dependence: Weighting Schemes, Statistics*. R package version 1.1-8.
- Bivand, R., Gómez-Rubio, V., and Rue, H. (2015). Spatial data analysis with r-inla with some extensions. *Journal of Statistical Software, Articles*, 63(20):1–31.
- Bivand, R., Millo, G., and Piras, G. (2021a). A review of software for spatial econometrics in R. *Mathematics*, 9(11).

- Bivand, R., Nowosad, J., and Lovelace, R. (2021b). *spData: Datasets for Spatial Analysis*. R package version 0.3.10.
- Bivand, R. and Piras, G. (2021). *spatialreg: Spatial Regression Analysis*. R package version 1.1-8.
- Bivand, R. S. (2002). Spatial econometrics functions in R: Classes and methods. *Journal of Geographical Systems*, 4:405–421.
- Bivand, R. S. (2012). After 'Raising the Bar': Applied Maximum Likelihood Estimation of Families of Models in Spatial Econometrics. *Estadística Española*, 54:71–88.
- Bivand, R. S. and Gómez-Rubio, V. (2021). Spatial survival modelling of business re-opening after katrina: Survival modelling compared to spatial probit modelling of re-opening within 3, 6 or 12 months. *Statistical Modelling*, 21(1–2):137–160.
- Bivand, R. S., Müller, W., and Reder, M. (2009). Power calculations for global and local Moran's *I*. *Computational Statistics and Data Analysis*, 53:2859–2872.
- Bivand, R. S., Pebesma, E., and Gomez-Rubio, V. (2013). *Applied spatial data analysis with R, Second edition*. Springer, NY.
- Bivand, R. S. and Piras, G. (2015). Comparing implementations of estimation methods for spatial econometrics. *Journal of Statistical Software*, 63(1):1–36.
- Bivand, R. S. and Portnov, B. A. (2004). Exploring spatial data analysis techniques using R: the case of observations with no neighbours. In Anselin, L., Florax, R. J. G. M., and Rey, S. J., editors, *Advances in Spatial Econometrics: Methodology, Tools, Applications*, pages 121–142. Springer, Berlin.
- Bivand, R. S., Sha, Z., Osland, L., and Thorsen, I. S. (2017). A comparison of estimation methods for multilevel models of spatially structured data. *Spatial Statistics*.
- Bivand, R. S. and Wong, D. W. S. (2018). Comparing implementations of global and local indicators of spatial association. *TEST*, 27(3):716–748.
- Blangiardo, M. and Cameletti, M. (2015). *Spatial and spatio-temporal Bayesian models with R-INLA*. John Wiley & Sons.
- Blangiardo, M., Cameletti, M., Baio, G., and Rue, H. (2013). Spatial and spatio-temporal models with r-inla. *Spatial and Spatio-temporal Epidemiology*, 4:33 – 49.
- Boots, B. and Okabe, A. (2007). Local statistical spatial analysis: Inventory and prospect. *International Journal of Geographical Information Science*, 21(4):355–375.

- Brazil Data Cube Team (2021). *rstac: Client Library for SpatioTemporal Asset Catalog*. R package version 0.9.1.
- Breidt, F. J., Opsomer, J. D., et al. (2017). Model-assisted survey estimation with modern prediction techniques. *Statistical Science*, 32(2):190–205.
- Brody, H., Rip, M. R., Vinten-Johansen, P., Paneth, N., and Rachman, S. (2000). Map-making and myth-making in Broad Street: the London cholera epidemic, 1854. *The Lancet*, 356(9223):64–68.
- Brooks, M. E., Kristensen, K., van Benthem, K. J., Magnusson, A., Berg, C. W., Nielsen, A., Skaug, H. J., Maechler, M., and Bolker, B. M. (2017). glmmTMB balances speed and flexibility among packages for zero-inflated generalized linear mixed modeling. *The R Journal*, 9(2):378–400.
- Brown, P. G. (2010). Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM.
- Brus, D. J. (2021a). Statistical approaches for spatial sample survey: Persistent misconceptions and new developments. *European Journal of Soil Science*, 72(2):686–703.
- Brus, D. J. (2021b). Statistical approaches for spatial sample survey: Persistent misconceptions and new developments. *European Journal of Soil Science*, 72(2):686–703.
- Bureau International des Poids et Mesures (2006). *The International System of Units (SI), 8th edition*. Organisation Intergouvernementale de la Convention du Mètre.
- Chambers, J. (2016). *Extending R*. CRC Press.
- Chrisman, N. (2012). A deflationary approach to fundamental principles in GIScience. In *Francis Harvey (ed.) Are there fundamental principles in Geographic Information Science?*, pages 42–64. CreateSpace, United States.
- Clementini, E., Di Felice, P., and van Oosterom, P. (1993). A small set of formal topological relationships suitable for end-user interaction. In Abel, D. and Chin Ooi, B., editors, *Advances in Spatial Databases*, pages 277–295, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Cliff, A. and Ord, J. (1972). Testing for spatial autocorrelation among regression residuals. *Geographical Analysis*, 4:267–284.
- Cliff, A. D. and Ord, J. K. (1973). *Spatial Autocorrelation*. Pion, London.
- Cliff, A. D. and Ord, J. K. (1981). *Spatial Processes*. Pion, London.
- Cobb, G. W. and Moore, D. S. (1997). Mathematics, statistics and teaching. *The American Mathematical Monthly*, 104:801–823.

- Cressie, N. A. C. (1993). *Statistics for Spatial Data*. New York:Wiley.
- Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*:1695.
- Davies, T. and Bryant, D. (2013). On circulant embedding for gaussian random fields in r. *Journal of Statistical Software, Articles*, 55(9):1–21.
- De Gruijter, J., Brus, D. J., Bierkens, M. F., and Knotters, M. (2006). *Sampling for natural resource monitoring*. Springer Science & Business Media.
- De Gruijter, J. and Ter Braak, C. (1990). Model-free estimation from spatial samples: a reappraisal of classical sampling theory. *Mathematical geology*, 22(4):407–415.
- Diggle, P. J. and Ribeiro Jr., P. J. (2007). *Model-Based Geostatistics*. Springer, New York.
- Diggle, P. J., Tawn, J. A., and Moyeed, R. A. (1998). Model-based geostatistics. *Applied Statistics*, pages 299–350.
- Do, V. H., Laurent, T., and Vanhems, A. (2021). *Guidelines on Areal Interpolation Methods*, pages 385–407. Springer International Publishing, Cham.
- Do, V. H., Thomas-Agnan, C., and Vanhems, A. (2015a). Accuracy of areal interpolation methods for count data. *Spatial Statistics*, 14:412 – 438.
- Do, V. H., Thomas-Agnan, C., and Vanhems, A. (2015b). Spatial reallocation of areal data: a review. *Rev. Econ. Rég. Urbaine*, 1/2:27–58.
- Dong, G. and Harris, R. (2015). Spatial autoregressive models for geographically hierarchical data structures. *Geographical Analysis*, 47(2):173–191.
- Dong, G., Harris, R., Jones, K., and Yu, J. (2015). Multilevel modeling with spatial interaction effects with application to an emerging land market in beijing, china. *PLOS One*, 10(6).
- Dong, G., Harris, R., and Mimis, A. (2020). *HSAR: Hierarchical Spatial Autoregressive Model*. R package version 0.5.1.
- Duncan, O. D., Cuzzort, R. P., and Duncan, B. (1961). *Statistical geography: Problems in analyzing areal data*. Free Press, Glencoe, IL.
- Dunnington, D. (2021). *ggspatial: Spatial Data Framework for ggplot2*. R package version 1.1.5.
- Dunnington, D., Pebesma, E., and Rubak, E. (2021). *s2: Spherical Geometry Operators Using the S2 Geometry Library*. R package version 1.0.6.
- Eddelbuettel, D. (2013). *Seamless R and C++ integration with Rcpp*. Springer.

- Egenhofer, M. J. and Franzosa, R. D. (1991). Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161–174.
- Elhorst, J. P. (2010). Applied spatial econometrics: Raising the bar. *Spatial Economic Analysis*, 5:9–28.
- Evenden, G. I. (1990). *Cartographic Projection Procedures for the UNIX Environment — A User's Manual*. Open-File Report 90-284. U.S. Geological Survey.
- Evers, K. and Knudsen, T. (2017). *Transformation pipelines for PROJ.4*. FIG Working Week 2017 Proceedings.
- file., S. A. (2020). *igraph: Network Analysis and Visualization*. R package version 1.2.6.
- Fingleton, B. (1999). Spurious spatial regression: Some Monte Carlo results with a spatial unit root and spatial cointegration. *Journal of Regional Science*, 9:1–19.
- Fisher, R. A. et al. (1937). *The design of experiments*. Number 2nd Ed. Oliver & Boyd, Edinburgh & London.
- Florax, R. J., Folmer, H., and Rey, S. J. (2003). Specification searches in spatial econometrics: the relevance of hendry's methodology. *Regional Science and Urban Economics*, 33(5):557–579.
- Florax, R. J., Folmer, H., and Rey, S. J. (2006). A comment on specification searches in spatial econometrics: The relevance of hendry's methodology: A reply. *Regional Science and Urban Economics*, 36(2):300–308.
- Freni-Sterrantino, A., Ventrucci, M., and Rue, H. (2018). A note on intrinsic conditional autoregressive models for disconnected graphs. *Spatial and Spatio-temporal Epidemiology*, 26:25–34.
- Gabriel, E., Rowlingson, B., and Diggle, P. (2013). stpp: An r package for plotting, simulating and analyzing spatio-temporal point patterns. *Journal of Statistical Software, Articles*, 53(2):1–29.
- Gaetan, C. and Guyon, X. (2010). *Spatial Statistics and Modeling*. Springer, New York.
- Galton, A. (2004). Fields and objects in space, time and space-time. *Spatial cognition and computation*, 4.
- Garnier, S. (2021). *viridis: Colorblind-Friendly Color Maps for R*. R package version 0.6.1.
- Geary, R. C. (1954). The contiguity ratio and statistical mapping. *The Incorporated Statistician*, 5:115–145.

- Gerber, F. and Furrer, R. (2015). Pitfalls in the implementation of bayesian hierarchical modeling of areal count data: An illustration using bym and leroux models. *Journal of Statistical Software, Code Snippets*, 63(1):1–32.
- Getis, A. and Ord, J. K. (1992). The analysis of spatial association by the use of distance statistics. *Geographical Analysis*, 24(2):189–206.
- Getis, A. and Ord, J. K. (1996). Local spatial statistics: an overview. In Longley, P. and Batty, M., editors, *Spatial Analysis: Modelling in a GIS Environment*, pages 261–277. GeoInformation International, Cambridge.
- Giraud, T. (2021). *mapsf: Thematic Cartography*. R package version 0.2.0.
- Gómez-Rubio, V. (2019). Spatial data analysis with inla. coding club uc3m tutorial series. universidad carlos iii de madrid.
- Gómez-Rubio, V. (2020a). *Bayesian inference with INLA*. CRC Press.
- Gómez-Rubio, V. (2020b). *Bayesian Inference with INLA*. CRC Press, Boca Raton, FL.
- Goodchild, M. F. and Lam, N. S. N. (1980). *Areal interpolation: a variant of the traditional spatial problem*. Department of Geography, University of Western Ontario London, ON, Canada.
- Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Thau, D., and Moore, R. (2017). Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202:18 – 27. Big Remotely Sensed Data: tools, applications and experiences.
- Goulard, M., Laurent, T., and Thomas-Agnan, C. (2017). About predictions in spatial autoregressive models: optimal and almost optimal strategies. *Spatial Economic Analysis*, 12(2-3):304–325.
- Gräler, B., Pebesma, E., and Heuvelink, G. (2016). Spatio-Temporal Interpolation using gstat. *The R Journal*, 8(1):204–218.
- Gómez-Rubio, V., Bivand, R., and Rue, H. (2015). A new latent class to fit spatial econometrics models with integrated nested laplace approximations. *Procedia Environmental Sciences*, 27:116 – 118. Spatial Statistics conference 2015.
- Hahsler, M. and Piekenbrock, M. (2021). *dbSCAN: Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms*. R package version 1.1-8.
- Halleck Vega, S. and Elhorst, J. P. (2015). The SLX model. *Journal of Regional Science*, 55(3):339–363.
- Hand, D. J. (2004). *Measurement: theory and practice*. A Hodder Arnold Publication.

- Healy, K. (2018). *Data Visualization, a practical introduction*. Princeton University Press.
- Heaton, M. J., Datta, A., Finley, A. O., Furrer, R., Guinness, J., Guhaniyogi, R., Gerber, F., Gramacy, R. B., Hammerling, D., Katzfuss, M., Lindgren, F., Nychka, D. W., Sun, F., and Zammit-Mangion, A. (2018). A case study competition among methods for analyzing large spatial data. *Journal of Agricultural, Biological and Environmental Statistics*.
- Hendry, D. F. (2006). A comment on “specification searches in spatial econometrics: The relevance of hendry’s methodology”. *Regional Science and Urban Economics*, 36(2):309–312.
- Hepple, L. W. (1976). A maximum likelihood model for econometric estimation with spatial series. In Masser, I., editor, *Theory and Practice in Regional Science*, London Papers in Regional Science, pages 90–104, London. Pion.
- Herring, J. et al. (2011). Opengis® implementation standard for geographic information-simple feature access-part 1: Common architecture [corrigendum].
- Herring, J. R. (2010). Opengis implementation standard for geographic information-simple feature access-part 2: Sql option. *Open Geospatial Consortium Inc.*
- Herring, J. R. (2011). Opengis implementation standard for geographic information-simple feature access-part 1: Common architecture. *Open Geospatial Consortium Inc*, page 111.
- Hersbach, H., Bell, B., Berrisford, P., Hirahara, S., Horányi, A., Muñoz-Sabater, J., Nicolas, J., Peubey, C., Radu, R., Schepers, D., Simmons, A., Soci, C., Abdalla, S., Abellan, X., Balsamo, G., Bechtold, P., Biavati, G., Bidlot, J., Bonavita, M., De Chiara, G., Dahlgren, P., Dee, D., Diamantakis, M., Dragani, R., Flemming, J., Forbes, R., Fuentes, M., Geer, A., Haimberger, L., Healy, S., Hogan, R. J., Hólm, E., Janisková, M., Keeley, S., Laloyaux, P., Lopez, P., Lupu, C., Radnoti, G., de Rosnay, P., Rozum, I., Vamborg, F., Villaume, S., and Thépaut, J.-N. (2020). The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, 146(730):1999–2049.
- Hijmans, R. J. (2021a). *raster: Geographic Data Analysis and Modeling*. R package version 3.4-13.
- Hijmans, R. J. (2021b). *terra: Spatial Data Analysis*. R package version 1.3-4.
- Hufkens, K. (2020). *ecmwfr: Interface to ECMWF and CDS Data Web Services*. R package version 1.3.0.
- Ihaka, R., Murrell, P., Hornik, K., Fisher, J. C., Stauffer, R., Wilke, C. O., McWhite, C. D., and Zeileis, A. (2021). *colorspace: A Toolbox for Manipulating and Assessing Colors and Palettes*. R package version 2.0-2.

- Hilfie, J. and Lott, R. (2008). *Datums and map projections for remote sensing, GIS, and surveying*. Whittles Pub. CRC Press, Scotland, UK.
- ISO (2004). *Geographic information – Simple feature access – Part 1: Common architecture*. ISO 19125-1:2004.
- Joo, R., Boone, M. E., Clay, T. A., Patrick, S. C., Clusella-Trullas, S., and Basille, M. (2020). Navigating through the R packages for movement. *Journal of Animal Ecology*, 89(1):248–267.
- Joo, R., Boone, M. E., Sumner, M., and Basille, M. (2021). *CRAN Task View: Processing and Analysis of Tracking Data*.
- Journel, A. G. and Huijbregts, C. J. (1978). *Mining geostatistics*. Academic press London.
- Karney, C. F. (2013). Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55.
- Kelejian, H. and Piras, G. (2017). *Spatial Econometrics*. Academic Press, London.
- Knudsen, T. and Evers, K. (2017). *Transformation pipelines for PROJ.4*. Geological Research Abstracts, Vol. 19, EGU2017-8050.
- Krainski, E. T., Gómez-Rubio, V., Bakka, H., Lenzi, A., Castro-Camilo, D., Simpson, D., Lindgren, F., and Rue, H. (2018). *Advanced spatial modeling with stochastic partial differential equations using R and INLA*. CRC Press.
- Kyriakidis, P. (2004). A geostatistical framework for areal-to-point spatial interpolation. *Geographical Analysis*, 36:259–289.
- Laurent, T. and Margaretic, P. (2021). *Predictions in Spatial Econometric Models: Application to Unemployment Data*, pages 409–426. Springer International Publishing, Cham.
- LeSage, J. P. (2014). What regional scientists need to know about spatial econometrics. *Review of Regional Studies*, 44:13–32.
- LeSage, J. P. and Pace, K. R. (2009). *Introduction to Spatial Econometrics*. CRC Press, Boca Raton, FL.
- Li, X. and Anselin, L. (2021). *rgeoda: R Library for Spatial Data Analysis*. R package version 0.0.8-1.
- Lott, R. (2015). Geographic information-well-known text representation of coordinate reference systems.
- Lovelace, R., Ellison, R., and Morgan, M. (2021). *stplanr: Sustainable Transport Planning*. R package version 0.8.2.

- Lovelace, R., Nowosad, J., and Muenchow, J. (2019). *Geocomputation with R*. Chapman and Hall/CRC.
- Lu, M., Appel, M., and Pebesma, E. (2018). Multidimensional arrays for analysing geoscientific data. *ISPRS International Journal of Geo-Information*, 7(8):313.
- Marius Appel, Edzer Pebesma, M. M. (2021). *Cloud-based processing of satellite image collections in R using STAC, COGs, and on-demand data cubes*.
- Martin, D. (1989). Mapping population data from zone centroid locations. *Transactions of the Institute of British Geographers, New Series*, 14:90–97.
- Martinetti, D. and Geniaux, G. (2017). Approximate likelihood estimation of spatial probit models. *Regional Science and Urban Economics*, 64:30 – 45.
- McCulloch, C. E. and Searle, S. R. (2001). *Generalized, Linear, and Mixed Models*. Wiley, New York.
- McMillen, D. P. (2003). Spatial autocorrelation or model misspecification? *International Regional Science Review*, 26:208–217.
- McMillen, D. P. (2013). *Quantile Regression for Spatial Data*. Springer-Verlag, Heidelberg.
- Mennis, J. (2003). Generating surface models of population using dasymetric mapping. *The Professional Geographer*, 55(1):31–42.
- Meyer, H. and Pebesma, E. (2020). Predicting into unknown space? estimating the area of applicability of spatial prediction models.
- Millo, G. and Piras, G. (2012). splm: Spatial panel data models in R. *Journal of Statistical Software*, 47(1):1–38.
- Moran, P. A. P. (1948). The interpretation of statistical maps. *Journal of the Royal Statistical Society, Series B (Methodological)*, 10(2):243–251.
- Moreno, M. and Basille, M. (2018). *Drawing beautiful maps programmatically with R, sf and ggplot2 — Part 1: Basics*.
- Mur, J. and Angulo, A. (2006). The spatial durbin model and the common factor tests. *Spatial Economic Analysis*, 1(2):207–226.
- Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes*. R package version 1.1-2.
- Obe, R. O. and Hsu, L. S. (2015). *PostGIS in action*. Manning Publications Co.

- Okabe, A., Satoh, T., Furuta, T., Suzuki, A., and Okano, K. (2008). Generalized network voronoi diagrams: Concepts, computational methods, and applications. *International Journal of Geographical Information Science*, 22(9):965–994.
- Olsson, G. (1970). Explanation, prediction, and meaning variance: An assessment of distance interaction models. *Economic Geography*, 46:223–233.
- Ord, J. K. (1975). Estimation Methods for Models of Spatial Interaction. *Journal of the American Statistical Association*, 70(349):120–126.
- Ord, J. K. and Getis, A. (2001). Testing for local spatial autocorrelation in the presence of global autocorrelation. *Journal of Regional Science*, 41(3):411–432.
- Pace, R. and LeSage, J. (2008). A spatial Hausman test. *Economics Letters*, 101:282–284.
- Papadopoulos, S., Datta, K., Madden, S., and Mattson, T. (2016). The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360.
- Pebesma, E. (2012). spacetime: Spatio-temporal data in R. *Journal of Statistical Software*, 51(7):1–30.
- Pebesma, E. (2018). Simple Features for R: Standardized Support for Spatial Vector Data. *The R Journal*, 10(1):439–446.
- Pebesma, E. (2021a). CRAN Task View: Handling and Analyzing Spatio-Temporal Data.
- Pebesma, E. (2021b). *lwgeom: Bindings to Selected liblwgeom Functions for Simple Features*. R package version 0.2-6.
- Pebesma, E. (2021c). *sf: Simple Features for R*. <https://r-spatial.github.io/sf/>, <https://github.com/r-spatial/sf/>.
- Pebesma, E. (2021d). *stars: Spatiotemporal Arrays, Raster and Vector Data Cubes*. <https://r-spatial.github.io/stars/>, <https://github.com/r-spatial/stars/>.
- Pebesma, E. and Graeler, B. (2021). *gstat: Spatial and Spatio-Temporal Geostatistical Modelling, Prediction and Simulation*. R package version 2.0-8.
- Pebesma, E., Mailund, T., Kalinowski, T., and Ucar, I. (2021). *units: Measurement Units for R Vectors*. R package version 0.7-2.
- Pebesma, E. J. (2004). Multivariable geostatistics in S: the gstat package. *Computers & Geosciences*, 30:683–691.

- Pinheiro, J. C. and Bates, D. M. (2000). *Mixed-Effects Models in S and S-Plus*. Springer, New York.
- Piras, G. and Prucha, I. R. (2014). On the finite sample properties of pre-test estimators of spatial models. *Regional Science and Urban Economics*, 46:103–115.
- Plate, T. and Heiberger, R. (2016). *abind: Combine Multidimensional Arrays*. R package version 1.4-5.
- Raim, A., Holan, S., Bradley, J., and Wikle, C. (2021). Spatio-temporal change of support modeling with r. *Computational Statistics*, 36:749–780.
- Raim, A. M., Holan, S. H., Bradley, J. R., and Wikle, C. K. (2020). *stcos: Space-Time Change of Support*. R package version 0.3.0.
- Raoult, B., Bergeron, C., Alós, A. L., Thépaut, J.-N., and Dee, D. (2017). Climate service develops user-friendly data store. *ECMWF newsletter*, 151:22–27.
- Ripley, B. D. (1981). *Spatial Statistics*. Wiley, New York.
- Ripley, B. D. (1988). *Statistical Inference for Spatial Processes*. Cambridge University Press, Cambridge.
- Rue, H., Lindgren, F., Simpson, D., Martino, S., Teixeira Krainski, E., Bakka, H., Riebler, A., and Fuglstad, G.-A. (2021). *INLA: Full Bayesian Analysis of Latent Gaussian Models using Integrated Nested Laplace Approximations*. R package version 21.02.23.
- Sauer, J., Oshan, T. M., Rey, S., and Wolf, L. J. (2021). On null hypotheses and heteroskedasticity.
- Schabenberger, O. and Gotway, C. A. (2005). *Statistical Methods for Spatial Data Analysis*. Chapman & Hall/CRC, Boca Raton/London.
- Scheider, S., Gräler, B., Pebesma, E., and Stasch, C. (2016). Modeling spatiotemporal information generation. *International Journal of Geographical Information Science*, 30(10):1980–2008. (open access).
- Schlather, M. (2011). Construction of covariance functions and unconditional simulation of random fields. In *Porcu, E., Montero, J.M. and Schlather, M., Space-Time Processes and Challenges Related to Environmental Problems*. New York: Springer.
- Schlesinger, T. and Eugster, M. J. A. (2013). *osmar: OpenStreetMap and R*. R package version 1.1-7.

- Schramm, M., Pebesma, E., Milenković, M., Foresta, L., Dries, J., Jacob, A., Wagner, W., Mohr, M., Neteler, M., Kadunc, M., Miksa, T., Kempeneers, P., Verbesselt, J., Gößwein, B., Navacchi, C., Lippens, S., and Reiche, J. (2021). The openeo api—harmonising the use of earth observation cloud services using virtual data cube functionalities. *Remote Sensing*, 13(6).
- She, B., Zhu, X., Ye, X., Guo, W., Su, K., and Lee, J. (2015). Weighted network voronoi diagrams for local spatial analysis. *Computers, Environment and Urban Systems*, 52:70 – 80.
- Smith, T. E. (2009). Estimation bias in spatial models with strongly connected weight matrices. *Geographical Analysis*, 41(3):307–332.
- Smith, T. E. and Lee, K. L. (2012). The effects of spatial autoregressive dependencies on inference in ordinary least squares: a geometric approach. *Journal of Geographical Systems*, 14:91–124.
- Sokal, R. R., Oden, N. L., and Thomson, B. A. (1998). Local spatial autocorrelation in a biological model. *Geographical Analysis*, 30:331–354.
- Stasch, C., Scheider, S., Pebesma, E., and Kuhn, W. (2014). Meaningful spatial prediction and aggregation. *Environmental Modelling & Software*, 51:149–165. (open access).
- Stoyan, D., Rodríguez-Cortés, F. J., Mateu, J., and Gille, W. (2017). Mark variograms for spatio-temporal point processes. *Spatial Statistics*, 20:125 – 147.
- Suesse, T. (2018). Marginal maximum likelihood estimation of sar models with missing data. *Computational Statistics & Data Analysis*, 120:98 – 110.
- Tennekes, M. (2018). tmap: Thematic maps in R. *Journal of Statistical Software*, 84(6):1–39.
- Tennekes, M. (2021). *tmap: Thematic Maps*. R package version 3.3-2.
- Tiefelsdorf, M. (2002). The saddlepoint approximation of Moran's I and local Moran's I_i reference distributions and their numerical evaluation. *Geographical Analysis*, 34:187–206.
- Tobler, W. (1979). Smooth pycnophylactic interpolation for geographical regions. *Journal of the American Statistical Association*, 74:519–530.
- Tobler, W. R. (1970). A computer movie simulating urban growth in the detroit region. *Economic Geography*, 46:234–240.
- UCAR (2014). *UDUNITS 2.2.26 Manual*.
- UCAR (2020). *The NetCDF User's Guide*.

- Umlauf, N., Adler, D., Kneib, T., Lang, S., and Zeileis, A. (2015). Structured additive regression models: An R interface to BayesX. *Journal of Statistical Software*, 63(21):1–46.
- Umlauf, N., Kneib, T., Lang, S., and Zeileis, A. (2017). *R2BayesX: Estimate Structured Additive Regression Models with BayesX*. R package version 1.1-1.
- Upton, G. and Fingleton, B. (1985). *Spatial data analysis by example: point pattern and qualitative data*. Wiley, New York.
- van der Meer, L., Abad, L., Gilardi, A., and Lovelace, R. (2021). *sfnetworks: Tidy Geospatial Networks*. R package version 0.5.2.
- van Lieshout, M. N. M. (2019). *Theory of Spatial Statistics*. Chapman and Hall/CRC, Boca Raton, FL.
- Veach, E., Rosenstock, J., Engle, E., Snedegar, R., Basch, J., and Manshreck, T. (2020). S2 geometry. *Website*.
- Ver Hoef, J. M. and Cressie, N. (1993). Multivariable spatial prediction. *Mathematical Geology*, 25(2):219–240.
- Vranckx, M., Neyens, T., and Faes, C. (2019). Comparison of different software implementations for spatial disease mapping. *Spatial and Spatio-temporal Epidemiology*, 31:100302.
- Wagner, M. and Zeileis, A. (2019). Heterogeneity and spatial dependence of regional growth in the EU: A recursive partitioning approach. *German Economic Review*, 20(1):67–82.
- Wall, M. M. (2004). A close look at the spatial structure implied by the CAR and SAR models. *Journal of Statistical Planning and Inference*, 121:311–324.
- Waller, L. A. and Gotway, C. A. (2004). *Applied Spatial Statistics for Public Health Data*. John Wiley & Sons, Hoboken, NJ.
- Whittle, P. (1954). On Stationary Processes in the Plane. *Biometrika*, 41(3-4):434–449.
- Wickham, H. (2014a). *Advanced R, Second Edition*. CRC Press.
- Wickham, H. (2014b). Tidy data. *Journal of Statistical Software*, 59(1).
- Wickham, H. (2016). *ggplot2: elegant graphics for data analysis*. Springer.
- Wickham, H. (2021). *tidyverse: Easily Install and Load the Tidyverse*. R package version 1.3.1.
- Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., et al. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

- Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2021). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.3.5.
- Wickham, H. and Grolemund, G. (2017). *R for Data Science*. O'Reilly.
- Wikle, C. K., Zammit-Mangion, A., and Cressie, N. (2019). *Spatio-temporal Statistics with R*. CRC Press.
- Wilhelm, S. and de Matos, M. G. (2013). Estimating Spatial Probit Models in R. *The R Journal*, 5(1):130–143.
- Wilke, C. O. (2019). *Fundamentals of Data Visualization*. O'Reilly Media, Inc.
- Wood, S. (2017). *Generalized Additive Models: An Introduction with R*. Chapman and Hall/CRC, 2 edition.
- Wood, S. (2021). *mgcv: Mixed GAM Computation Vehicle with Automatic Smoothness Estimation*. R package version 1.8-36.
- Zeileis, A., Fisher, J. C., Hornik, K., Ihaka, R., McWhite, C. D., Murrell, P., Stauffer, R., and Wilke, C. O. (2020). colorspace: A toolbox for manipulating and assessing colors and palettes. *Journal of Statistical Software*, 96(1):1–49.