2

Classes for Spatial Data in R

2.1 Introduction

Many disciplines have influenced the representation of spatial data, both in analogue and digital forms. Surveyors, navigators, and military and civil engineers refined the fundamental concepts of mathematical geography, established often centuries ago by some of the founders of science, for example by al-Khwārizmī. Digital representations came into being for practical reasons in computational geometry, in computer graphics and hardware-supported gaming, and in computer-assisted design and virtual reality. The use of spatial data as a business vehicle has been spurred in the early years of the present century by consumer broadband penetration and distributed server farms, with a prime example being Google Earth™.[1] There are often interactions between the graphics hardware required and the services offered, in particular for the fast rendering of scene views.

In addition, space and other airborne technologies have vastly increased the volumes and kinds of spatial data available. Remote sensing satellites continue to make great contributions to earth observation, with multi-spectral images supplementing visible wavelengths. The Shuttle Radar Topography Mission (SRTM) in February 2000 has provided elevation data for much of the earth. Other satellite-borne sensor technologies are now vital for timely storm warnings, amongst other things. These complement terrestrial networks monitoring, for example lightning strikes and the movement of precipitation systems by radar.

Surveying in the field has largely been replaced by aerial photogrammetry, mapping using air photographs usually exposed in pairs of stereo images. Legacy aerial photogrammetry worked with analogue images, and many research laboratories and mapping agencies have large archives of air photographs with coverage beginning from the 1930s. These images can be scanned to provide a digital representation at chosen resolutions. While

[1] `http://earth.google.com/`.

satellite imagery usually contains metadata giving the scene frame – the sensor direction in relation to the earth at scan time – air photographs need to be registered to known ground control points.

These ground control points were 'known' from terrestrial triangulation, but could be in error. The introduction of Global Positioning System (GPS) satellites has made it possible to correct the positions of existing networks of ground control points. The availability of GPS receivers has also made it possible for data capture in the field to include accurate positional information in a known coordinate reference system. This is conditioned by the requirement of direct line-of-sight to a sufficient number of satellites, not easy in mountain valleys or in city streets bounded by high buildings. Despite this limitation, around the world the introduction of earth observation satellites and revised ground control points have together caused breaks of series in published maps, to take advantage of the greater accuracy now available. This means that many older maps cannot be matched to freshly acquired position data without adjustment.

All of these sources of spatial data involve points, usually two real numbers representing position in a known coordinate reference system. It is possible to go beyond this simple basis by combining pairs of points to form line segments, combining line segments to form polylines, networks or polygons, or regular grid centres. Grids can be defined within a regular polygon, usually a rectangle, with given resolution – the size of the grid cells. All these definitions imply choices of what are known in geographical information systems (GIS) as data models, and these choices have most often been made for pragmatic reasons. All the choices also involve trade-offs between accuracy, feasibility, and cost.

Artificial objects are easiest to represent, like roads, bridges, buildings, or similar structures. They are crisply defined, and are not subject to natural change – unlike placing political borders along the centre lines or deepest channels of meandering rivers. Shorelines are most often natural and cannot be measured accurately without specifying measurement scale. Boundaries between areas of differing natural land cover are frequently indeterminate, with gradations from one land cover category to another. Say that we want to examine the spatial distribution of a species by land cover category; our data model of how to define the boundary between categories will affect the outcome, possibly strongly. Something of the same affects remote sensing, because the reported values of the observed pixels will hide sub-pixel variation.

It is unusual for spatial data to be defined in three dimensions, because of the close links between cartography and data models for spatial data. When there are multiple observations on the same attribute at varying heights or depths, they are most often treated as separate layers. GIS-based data models do not fit time series data well either, even though some environmental monitoring data series are observed in three dimensions and time. Some GIS software can handle voxels, the 3D equivalent of pixels – 2D raster cells – but the third dimension in spatial data is not handled satisfactorily, as is the case in computer-assisted design or medical imaging. On the other hand, many GIS packages do provide a 2.5D intermediate solution for viewing, by draping thematic layers, like land cover or a road network, over a digital elevation model. In this case, however, there is no 'depth' in the data model, as we can see when a road tunnel route is draped over the mountain it goes through.

2.2 Classes and Methods in R

In Chap. 1, we described R as a language and environment for data analysis. Although this is not the place to give an extended introduction to R,[2] it will be useful to highlight some of its features (see also Braun and Murdoch, 2007, for an up-to-date introduction). In this book, we will be quoting R commands in the text, showing which commands a user could give, and how the non-graphical output might be represented when printed to the console.

Of course, R can be used as a calculator to carry out simple tasks, where no values are assigned to variables, and where the results are shown without being saved, such as the area of a circle of radius 10:

```
> pi * 10^2
```

```
[1] 314.1593
```

Luckily, π is a built-in constant in R called `pi`, and so entering a rounded version is not needed. So this looks like a calculator, but appearances mislead. The first misleading impression is that the arithmetic is simply being 'done', while in fact it is being translated (parsed) into functions (operators) with arguments first, and then evaluated:

```
> "*"(pi, "^"(10, 2))
```

```
[1] 314.1593
```

When the operators or functions permit, vectors of values may be used as readily as scalar values (which are vectors of unit length) — here the '`:`' operator is used to generate an integer sequence of values:

```
> pi * (1:10)^2
```

```
 [1]   3.141593  12.566371  28.274334  50.265482  78.539816 113.097336
 [7] 153.938040 201.061930 254.469005 314.159265
```

The second misapprehension is that what is printed to the console is the 'result', when it is actually the outcome of applying the appropriate `print` method for the class of the 'result', with default arguments. If we store the value returned for the area of our circle in variable `x` using the assignment operator `<-`, we can print `x` with the default number of digits, or with more if

[2] Free documentation, including the very useful 'An Introduction to R' (Venables et al., 2008), may be downloaded from CRAN.

we so please. Just typing the variable name at the interactive prompt invokes the appropriate print method, but we can also pass it to the print method explicitly:

```
> x <- pi * 10^2
> x

[1] 314.1593

> print(x)

[1] 314.1593

> print(x, digits = 12)

[1] 314.159265359
```

We can say that the variable `x` contains an object of a particular class, in this case:

```
> class(x)

[1] "numeric"

> typeof(x)

[1] "double"
```

where `typeof` returns the storage mode of the object in variable `x`. It is the class of the object that determines the method that will be used to handle it; if there is no specific method for that class, it may be passed to a default method. These methods are also known as generic functions, often including at least `print`, `plot`, and `summary` methods. In the case of the `print` method, `numeric` is not provided for explicitly, and so the default method is used. The `plot` method, as its name suggests, will use the current graphics device to make a visual display of the object, dispatching to a specific method for the object class if provided. In comparison with the `print` method, the `summary` method provides a qualified view of the data, highlighting the key features of the object.

When the `S` language was first introduced, it did not use class/method mechanisms at all. They were introduced in Chambers and Hastie (1992) and `S` version 3, in a form that is known as `S3` classes or old-style classes. These classes were not formally defined, and 'just grew'; the vast majority of objects returned by model fitting functions belong to old-style classes. Using a non-spatial example from the standard data set `cars`, we can see that it is an object of class `data.frame`, stored in a `list`, which is a vector whose components can be arbitrary objects; `data.frame` has both names and summary methods:

```
> class(cars)

[1] "data.frame"
```

```
> typeof(cars)

[1] "list"

> names(cars)

[1] "speed" "dist"

> summary(cars)

     speed           dist       
 Min.   : 4.0   Min.   :  2.00  
 1st Qu.:12.0   1st Qu.: 26.00  
 Median :15.0   Median : 36.00  
 Mean   :15.4   Mean   : 42.98  
 3rd Qu.:19.0   3rd Qu.: 56.00  
 Max.   :25.0   Max.   :120.00  
```

The `data.frame` contains two variables, one recording the speed of the observed cars in mph, the other the stopping distance measured in feet – the observations were made in the 1920s. When uncertain about the structure of something in our R workspace, revealed for example by using the `ls` function for listing the contents of the workspace, the `str`[3] method often gives a clear digest, including the size and class:

```
> str(cars)

'data.frame': 50 obs. of 2 variables:
$ speed:num 4 4 7 7 8 ...
$ dist :num 2 10 4 22 16 ...
```

Data frames are containers for data used everywhere in S since their full introduction in Chambers and Hastie (1992, pp. 45–94). Recent and shorter introductions to data frames are given by Crawley (2005, pp. 15–22), Crawley (2007, pp. 107–133), and Dalgaard (2002, pp. 18–19) and in the online documentation (Venables et al., 2008, pp. 27–29 in the R 2.6.2 release). Data frames view the data as a rectangle of rows of observations on columns of values of variables of interest. The representation of the values of the variables of interest can include integer and floating point numeric types, logical, character, and derived classes. One very useful derived class is the factor, which is represented as integers pointing to character levels, such as `'forest'` or `'arable'`. Printed, the values look like character values, but are not – when a data frame is created, all character variables included in it are converted to factor by default. Data frames also have unique row names, represented as an integer or character vector or as an internal mechanism to signal that

[3] `str` can take additional arguments to control its output.

the sequence from 1 to the number of rows in the data frame are used. The `row.names` function is used to access and assign data frame row names.

One of the fundamental abstractions used in R is the `formula` introduced in Chambers and Hastie (1992, pp. 13–44) – an online summary may be found in Venables et al. (2008, pp. 50–52 in the R 2.6.2 release). The abstraction is intended to make statistical modelling as natural and expressive as possible, permitting the analyst to focus on the substantive problem at hand. Because the `formula` abstraction is used in very many contexts, it is worth some attention. A `formula` is most often two-sided, with a response variable to the left of the $\sim$ (tilde) operator, and in this case a determining variable on the right:

```
> class(dist ~ speed)

[1] "formula"
```

These objects are typically used as the first argument to model fitting functions, such as `lm`, which is used to fit linear models. They will usually be accompanied by a `data` argument, indicating where the variables are to be found:

```
> lm(dist ~ speed, data = cars)

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)        speed
    -17.579        3.932
```

This is a simple example, but very much more can be done with the `formula` abstraction. If we create a factor for the `speed` variable by cutting it at its quartiles, we can contrast how the `plot` method displays the relationship between two numerical variables and a numerical variable and a factor (shown in Fig. 2.1):

```
> cars$qspeed <- cut(cars$speed, breaks = quantile(cars$speed),
+     include.lowest = TRUE)
> is.factor(cars$qspeed)

[1] TRUE

> plot(dist ~ speed, data = cars)
> plot(dist ~ qspeed, data = cars)
```

Finally, let us see how the `formula` with the right-hand side factor is handled by `lm` – it is converted into ‘dummy’ variable form automatically:

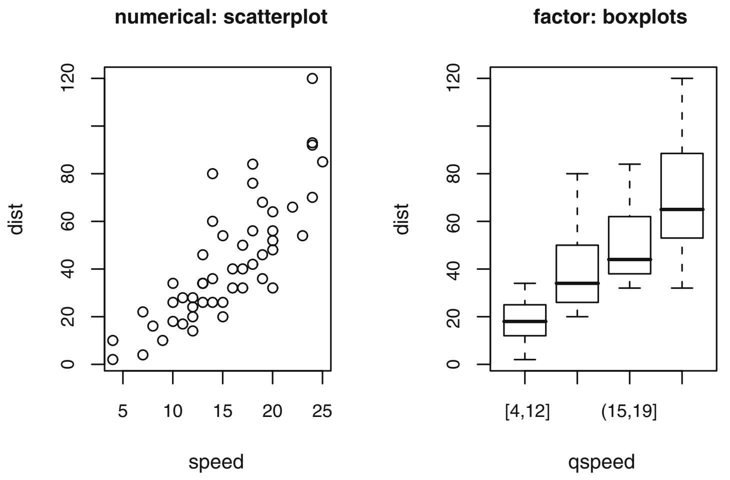

Fig. 2.1. Plot methods for a formula with numerical (*left panel*) and factor (*right panel*) right-hand side variables

```
> lm(dist ~ qspeed, data = cars)

Call:
lm(formula = dist ~ qspeed, data = cars)

Coefficients:
  (Intercept)  qspeed(12,15]  qspeed(15,19]  qspeed(19,25]
        18.20          21.98          31.97          51.13
```

Variables in the `formula` may also be transformed in different ways, for example using `log`. The `formula` is carried through into the object returned by model fitting functions to be used for prediction from new data provided in a `data.frame` with the same column names as the right-hand side variables, and the same level names if the variable is a factor.

New-style (`S4`) classes were introduced in the `S` language at release 4, and in Chambers (1998), and are described by Venables and Ripley (2000, pp. 75–121), and in subsequent documentation installed with R.[4] Old-style classes are most often simply lists with attributes; they are not defined formally. Although users usually do not change values inside old-style classes, there is nothing to stop them doing so, for example changing the representation of coordinates from floating point to integer numbers. This means that functions need to check, among other things, whether components of a class exist, and whether they are represented correctly, before they can be handled. The central advantage of new-style classes is that they have formal definitions

[4] There is little instructional material online, although this useR conference talk remains relevant: `http://www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Leisch.pdf`.

that specify the name and type of the components, called *slots*, that they contain. This simplifies the writing, maintenance, and use of the classes, because their format is known from the definition. For a further discussion of programming for classes and methods, see Sect. 6.1.

Because the classes provided by the **sp** package are new-style classes, we will be seeing how such classes work in practice below. In particular, we will be referring to the slots in class definitions; slots are specified in the definition as the representation of what the class contains. Many methods are written for the classes to be introduced in the remainder of this chapter, in particular coercion methods for changing the way an object is represented from one class to another. New-style classes can also check the validity of objects being created, for example to stop the user from filling slots with data that do not conform to the definition.

2.3 `Spatial` Objects

The foundation class is the `Spatial` class, with just two slots. The first is a bounding box, a matrix of numerical coordinates with column names `c('min', 'max')`, and at least two rows, with the first row eastings (x-axis) and the second northings (y-axis). Most often the bounding box is generated automatically from the data in subclasses of `Spatial`. The second is a `CRS` class object defining the coordinate reference system, and may be set to 'missing', represented by `NA` in R, by `CRS(as.character(NA))`, its default value. Operations on `Spatial*` objects should update or copy these values to the new `Spatial*` objects being created. We can use `getClass` to return the complete definition of a class, including its slot names and the types of their contents:

```
> library(sp)
```

```
> getClass("Spatial")
```

```
Slots:

Name:         bbox proj4string
Class:      matrix         CRS

Known Subclasses:
Class "SpatialPoints", directly
Class "SpatialLines", directly
Class "SpatialPolygons", directly
Class "SpatialPointsDataFrame", by class "SpatialPoints", distance 2
Class "SpatialPixels", by class "SpatialPoints", distance 2
Class "SpatialGrid", by class "SpatialPoints", distance 3
Class "SpatialPixelsDataFrame", by class "SpatialPoints", distance 3
Class "SpatialGridDataFrame", by class "SpatialPoints", distance 4
Class "SpatialLinesDataFrame", by class "SpatialLines", distance 2
Class "SpatialPolygonsDataFrame", by class "SpatialPolygons",
     distance 2
```

As we see, `getClass` also returns known subclasses, showing the classes that include the `Spatial` class in their definitions. This also shows where we are going in this chapter, moving from the foundation class to richer representations. But we should introduce the coordinate reference system (`CRS`) class very briefly; we will return to its description in Chap. 4.

```
> getClass("CRS")

Slots:

Name:   projargs
Class: character
```

The class has a character string as its only slot value, which may be a missing value. If it is not missing, it should be a PROJ.4-format string describing the projection (more details are given in Sect. 4.1.2). For geographical coordinates, the simplest such string is `"+proj=longlat"`, using `"longlat"`, which also shows that eastings always go before northings in **sp** classes. Let us build a simple `Spatial` object from a bounding box matrix, and a missing coordinate reference system:

```
> m <- matrix(c(0, 0, 1, 1), ncol = 2, dimnames = list(NULL,
+     c("min", "max")))
> crs <- CRS(projargs = as.character(NA))
> crs

CRS arguments: NA

> S <- Spatial(bbox = m, proj4string = crs)
> S

An object of class "Spatial"
Slot "bbox":
     min max
[1,]   0   1
[2,]   0   1

Slot "proj4string":
CRS arguments: NA
```

We could have used `new` methods to create the objects, but prefer to use helper functions with the same names as the classes that they instantiate. If the object is known not to be projected, a sanity check is carried out on the coordinate range (which here exceeds the feasible range for geographical coordinates):

```
> Spatial(matrix(c(350, 85, 370, 95), ncol = 2, dimnames = list(NULL,
+     c("min", "max"))), proj4string = CRS("+longlat"))

Error in validityMethod(object) : Geographical CRS given to
non-conformant data
```

2.4 SpatialPoints

The SpatialPoints class is the first subclass of Spatial, and a very important one. The extension of SpatialPoints to other subclasses means that explaining how this class works will yield benefits later on. In this section, we also look at methods for Spatial* objects, and at extending Spatial* objects to include attribute data, where each spatial entity, here a point, is linked to a row in a data frame. We take Spatial* objects to be subclasses of Spatial, and the best place to start is with SpatialPoints.

A two-dimensional point can be described by a pair of numbers (x, y), defined over a known region. To represent geographical phenomena, the maximum known region is the earth, and the pair of numbers measured in degrees are a geographical coordinate, showing where our point is on the globe. The pair of numbers define the location on the sphere exactly, but if we represent the globe more accurately by an ellipsoid model, such as the World Geodetic System 1984 – introduced after satellite measurements corrected our understanding of the shape of the earth – that position shifts slightly. Geographical coordinates can extend from latitude 90° to -90° in the north–south direction, and from longitude 0° to 360° or equivalently from -180° to 180° in the east–west direction. The Poles are fixed, but where the longitudes fall depends on the choice of prime meridian, most often Greenwich just east of London. This means that geographical coordinates define a point on the earth's surface unequivocally if we also know which ellipsoid model and prime meridian were used; the concept of datum, relating the ellipsoid to the distance from the centre of the earth, is introduced on p. 82.

Using the standard read.table function, we read in a data file with the positions of CRAN mirrors across the world. We extract the two columns with the longitude and latitude values into a matrix, and use str to view a digest:

```
> CRAN_df <- read.table("CRAN051001a.txt", header = TRUE)
> CRAN_mat <- cbind(CRAN_df$long, CRAN_df$lat)
> row.names(CRAN_mat) <- 1:nrow(CRAN_mat)
> str(CRAN_mat)

num [1:54, 1:2] 153 145 ...
- attr(*, "dimnames")=List of 2
..$ :chr [1:54] "1" "2" ...
..$ :NULL
```

The SpatialPoints class extends the Spatial class by adding a coords slot, into which a matrix of point coordinates can be inserted.

```
> getClass("SpatialPoints")

Slots:

Name:       coords        bbox proj4string
Class:      matrix      matrix         CRS
```

```
Extends: "Spatial"

Known Subclasses:
Class "SpatialPointsDataFrame", directly
Class "SpatialPixels", directly
Class "SpatialGrid", by class "SpatialPixels", distance 2
Class "SpatialPixelsDataFrame", by class "SpatialPixels", distance 2
Class "SpatialGridDataFrame", by class "SpatialGrid", distance 3
```

It has a `summary` method that shows the bounding box, whether the object is projected (here `FALSE`, because the string `"longlat"` is included in the projection description), and the number of rows of coordinates. Classes in **sp** are not atomic: there is no `SpatialPoint` class that is extended by `SpatialPoints`. This is because R objects are vectorised by nature, not atomic. A `SpatialPoints` object may, however, consist of a single point.

```
> llCRS <- CRS("+proj=longlat +ellps=WGS84")
> CRAN_sp <- SpatialPoints(CRAN_mat, proj4string = llCRS)
> summary(CRAN_sp)

Object of class SpatialPoints
Coordinates:
                min      max
coords.x1 -122.95000 153.0333
coords.x2  -37.81667  57.0500
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Number of points: 54
```

`SpatialPoints` objects may have more than two dimensions, but plot methods for the class use only the first two.

2.4.1 Methods

Methods are available to access the values of the slots of `Spatial` objects. The `bbox` method returns the bounding box of the object, and is used both for preparing plotting methods (see Chap. 3) and internally in handling data objects. The first row reports the west–east range and the second the south–north direction. If we want to take a subset of the points in a `SpatialPoints` object, the bounding box is reset, as we will see.

```
> bbox(CRAN_sp)

                min      max
coords.x1 -122.95000 153.0333
coords.x2  -37.81667  57.0500
```

First, the other generic method for all `Spatial` objects, `proj4string`, will be introduced. The basic method reports the projection string contained as a

CRS object in the proj4string slot of the object, but it also has an assignment form, allowing the user to alter the current value, which can also be a CRS object containing a character NA value:

```
> proj4string(CRAN_sp)

[1] "+proj=longlat +ellps=WGS84"

> proj4string(CRAN_sp) <- CRS(as.character(NA))
> proj4string(CRAN_sp)

[1] NA

> proj4string(CRAN_sp) <- llCRS
```

Extracting the coordinates from a SpatialPoints object as a numeric matrix is as simple as using the coordinates method. Like all matrices, the indices can be used to choose subsets, for example CRAN mirrors located in Brazil in 2005:

```
> brazil <- which(CRAN_df$loc == "Brazil")
> brazil

[1] 4 5 6 7 8

> coordinates(CRAN_sp)[brazil, ]

     coords.x1 coords.x2
[1,] -49.26667 -25.41667
[2,] -42.86667 -20.75000
[3,] -43.20000 -22.90000
[4,] -47.63333 -22.71667
[5,] -46.63333 -23.53333
```

In addition, a SpatialPoints object can also be accessed by index, using the "[" operator, here on the coordinate values treated as an entity. The object returned is of the same class, and retains the projection information, but has a new bounding box:

```
> summary(CRAN_sp[brazil, ])

Object of class SpatialPoints
Coordinates:
                min       max
coords.x1 -49.26667 -42.86667
coords.x2 -25.41667 -20.75000
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Number of points: 5
```

The "[" operator also works for negative indices, which remove those coordinates from the object, here by removing mirrors south of the Equator:

```
> south_of_equator <- which(coordinates(CRAN_sp)[, 2] <
+     0)
> summary(CRAN_sp[-south_of_equator, ])
```

```
Object of class SpatialPoints
Coordinates:
             min    max
coords.x1 -122.95 140.10
coords.x2   24.15  57.05
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Number of points: 45
```

Because `summary` and `print` methods are so common in R, we used them here without special mention. They are provided for **sp** classes, with `summary` reporting the number of spatial entities, the projection information, and the bounding box, and `print` gives a view of the data in the object. As usual in S, the actual underlying data and the output of the `print` method may differ, for example in the number of digits shown.

An important group of methods for visualisation of `Spatial*` objects are presented in detail in Chap. 3; each such object class has a `plot` method. Other methods will also be introduced in Chap. 5 for combining (overlaying) different `Spatial*` objects, for sampling from `Spatial` objects, and for merging spatial data objects.

2.4.2 Data Frames for Spatial Point Data

We described data frames on p. 25, and we now show how our `SpatialPoints` object can be taught to behave like a `data.frame`. Here we use numbers in sequence to index the points and the rows of our data frame, because neither the place names nor the countries are unique.

```
> str(row.names(CRAN_df))
```

```
chr [1:54] "1" "2" ...
```

What we would like to do is to associate the correct rows of our data frame object with 'their' point coordinates – it often happens that data are collected from different sources, and the two need to be merged. The `SpatialPoints-DataFrame` class is the container for this kind of spatial point information, and can be constructed in a number of ways, for example from a data frame and a matrix of coordinates. If the matrix of point coordinates has row names and the `match.ID` argument is set to its default value of `TRUE`, then the matrix row names are checked against the row names of the data frame. If they match, but are not in the same order, the data frame rows are re-ordered to suit the points. If they do not match, no `SpatialPointsDataFrame` is constructed. Note that the new object takes two indices, the first for the spatial object, the second, if given, for the column. Giving a single index number, or range of numbers, or column name or names returns a new `SpatialPointsDataFrame` with the requested columns. Using other extraction operators, especially the `$` operator, returns the data frame column referred to. These operators mimic the equivalent ones for other standard S classes as far as possible.

```
> CRAN_spdf1 <- SpatialPointsDataFrame(CRAN_mat, CRAN_df,
+     proj4string = llCRS, match.ID = TRUE)
> CRAN_spdf1[4, ]

           coordinates    place   north    east    loc      long
4 (-49.2667, -25.4167) Curitiba 25d25'S 49d16'W Brazil -49.26667
        lat
4 -25.41667

> str(CRAN_spdf1$loc)

Factor w/ 30 levels "Australia","Austria",..: 1 1 2 3 3 ...

> str(CRAN_spdf1[["loc"]])

Factor w/ 30 levels "Australia","Austria",..: 1 1 2 3 3 ...
```

If we re-order the data frame at random using `sample`, we still get the same result, because the data frame is re-ordered to match the row names of the points:

```
> s <- sample(nrow(CRAN_df))
> CRAN_spdf2 <- SpatialPointsDataFrame(CRAN_mat, CRAN_df[s,
+     ], proj4string = llCRS, match.ID = TRUE)
> all.equal(CRAN_spdf2, CRAN_spdf1)

[1] TRUE

> CRAN_spdf2[4, ]

           coordinates    place   north    east    loc      long
4 (-49.2667, -25.4167) Curitiba 25d25'S 49d16'W Brazil -49.26667
        lat
4 -25.41667
```

But if we have non-matching ID values, created by pasting pairs of letters together and sampling an appropriate number of them, the result is an error:

```
> CRAN_df1 <- CRAN_df
> row.names(CRAN_df1) <- sample(c(outer(letters, letters,
+     paste, sep = "")), nrow(CRAN_df1))

> CRAN_spdf3 <- SpatialPointsDataFrame(CRAN_mat, CRAN_df1,
+     proj4string = llCRS, match.ID = TRUE)

Error in SpatialPointsDataFrame(CRAN_mat, CRAN_df1,
proj4string = llCRS, : row.names of data and coords do not
match
```

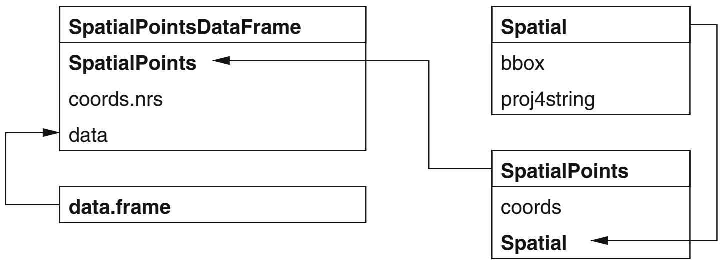

Fig. 2.2. Spatial points classes and their slots; arrows show subclass extensions

Let us examine the contents of objects of the `SpatialPointsDataFrame` class, shown in Fig. 2.2. Because the class extends `SpatialPoints`, it also inherits the information contained in the `Spatial` class object. The `data` slot is where the information from the data frame is kept, in a `data.frame` object.

```
> getClass("SpatialPointsDataFrame")

Slots:

Name:         data  coords.nrs      coords        bbox proj4string
Class:  data.frame     numeric      matrix      matrix         CRS

Extends:
Class "SpatialPoints", directly
Class "Spatial", by class "SpatialPoints", distance 2

Known Subclasses:
Class "SpatialPixelsDataFrame", directly, with explicit coerce
```

The `Spatial*DataFrame` classes have been designed to behave as far as possible like data frames, both with respect to standard methods such as `names`, and more demanding modelling functions like `model.frame` used in very many model fitting functions using `formula` and `data` arguments:

```
> names(CRAN_spdf1)

[1] "place" "north" "east"  "loc"   "long"  "lat"

> str(model.frame(lat ~ long, data = CRAN_spdf1), give.attr = FALSE)

'data.frame': 54 obs. of 2 variables:
$ lat :num -27.5 -37.8 ...
$ long:num 153 145 ...
```

Making our `SpatialPointsDataFrame` object from a matrix of coordinates and a data frame with or without ID checking is only one way to reach our goal, and others may be more convenient. We can construct the object by giving the `SpatialPointsDataFrame` function a `SpatialPoints` object as its first argument:

```
> CRAN_spdf4 <- SpatialPointsDataFrame(CRAN_sp, CRAN_df)
> all.equal(CRAN_spdf4, CRAN_spdf2)

[1] TRUE
```

We can also assign coordinates to a data frame – this approach modifies the original data frame. The coordinate assignment function can take a matrix of coordinates with the same number of rows as the data frame on the right-hand side, or an integer vector of column numbers for the coordinates, or equivalently a character vector of column names, assuming that the required columns already belong to the data frame.

```
> CRAN_df0 <- CRAN_df
> coordinates(CRAN_df0) <- CRAN_mat
> proj4string(CRAN_df0) <- llCRS
> all.equal(CRAN_df0, CRAN_spdf2)

[1] TRUE

> str(CRAN_df0, max.level = 2)

Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
..@ data :'data.frame': 54 obs. of 6 variables:
..@ coords.nrs :num(0)
..@ coords :num [1:54, 1:2] 153 145 ...
.. ..- attr(*, "dimnames")=List of 2
..@ bbox :num [1:2, 1:2] -123.0 -37.8 ...
.. ..- attr(*, "dimnames")=List of 2
..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
```

Objects created in this way differ slightly from those we have seen before, because the `coords.nrs` slot is now used, and the coordinates are moved from the `data` slot to the `coords` slot, but the objects are otherwise the same:

```
> CRAN_df1 <- CRAN_df
> names(CRAN_df1)

[1] "place" "north" "east"  "loc"   "long"  "lat"

> coordinates(CRAN_df1) <- c("long", "lat")
> proj4string(CRAN_df1) <- llCRS
> str(CRAN_df1, max.level = 2)

Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
..@ data :'data.frame': 54 obs. of 4 variables:
..@ coords.nrs :int [1:2] 5 6
..@ coords :num [1:54, 1:2] 153 145 ...
.. ..- attr(*, "dimnames")=List of 2
..@ bbox :num [1:2, 1:2] -123.0 -37.8 ...
.. ..- attr(*, "dimnames")=List of 2
..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
```

Transect and tracking data may also be represented as points, because the observation at each point contributes information that is associated with the point itself, rather than the line as a whole. Sequence numbers can be entered into the data frame to make it possible to trace the points in order, for example as part of a `SpatialLines` object as we see in the Sect. 2.5.

As an example, we use a data set[5] from satellite telemetry of a single loggerhead turtle crossing the Pacific from Mexico to Japan (Nichols et al., 2000).

```
> turtle_df <- read.csv("seamap105_mod.csv")
> summary(turtle_df)
```

```
       id               lat              lon
 Min.   :  1.00   Min.   :21.57   Min.   :-179.88
 1st Qu.: 99.25   1st Qu.:24.36   1st Qu.:-147.38
 Median :197.50   Median :25.64   Median :-119.64
 Mean   :197.50   Mean   :27.21   Mean   : -21.52
 3rd Qu.:295.75   3rd Qu.:27.41   3rd Qu.: 153.66
 Max.   :394.00   Max.   :39.84   Max.   : 179.93

                obs_date
 01/02/1997 04:16:53:  1
 01/02/1997 05:56:25:  1
 01/04/1997 17:41:54:  1
 01/05/1997 17:20:07:  1
 01/06/1997 04:31:13:  1
 01/06/1997 06:12:56:  1
 (Other)            :388
```

Before creating a `SpatialPointsDataFrame`, we will timestamp the observations, and re-order the input data frame by timestamp to make it easier to add months to Fig. 2.3, to show progress westwards across the Pacific:

```
> timestamp <- as.POSIXlt(strptime(as.character(turtle_df$obs_date),
+     "%m/%d/%Y %H:%M:%S"), "GMT")
> turtle_df1 <- data.frame(turtle_df, timestamp = timestamp)
> turtle_df1$lon <- ifelse(turtle_df1$lon < 0, turtle_df1$lon +
+     360, turtle_df1$lon)
> turtle_sp <- turtle_df1[order(turtle_df1$timestamp),
+     ]
> coordinates(turtle_sp) <- c("lon", "lat")
> proj4string(turtle_sp) <- CRS("+proj=longlat +ellps=WGS84")
```

The input data file is as downloaded, but without columns with identical values for all points, such as the number of the turtle (07667). We return to this data set in Chap. 6, examining the interesting contributed package **trip** by Michael Sumner, which proposes customised classes and methods for data of this kind.

[5] Data downloaded with permission from SEAMAP (Read et al., 2003), data set 105.

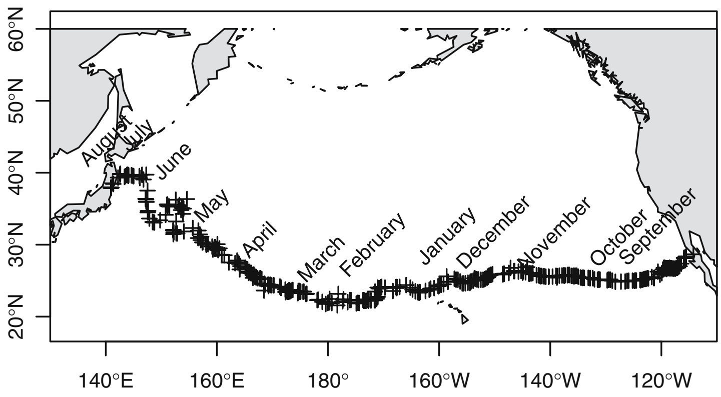

Fig. 2.3. Westward movements of a captive-raised adult loggerhead turtle (*Caretta caretta*) from 10 August 1996 to 12 August 1997

2.5 SpatialLines

Lines have been represented in `S` in a simple form as a sequence of points (see Becker et al. (1988), Murrell (2006, pp. 83–86)), based on lowering the graphic 'pen' at the first point and drawing to the successive points until an `NA` is met. Then the pen is raised and moved to the next non-`NA` value, where it is lowered, until the end of the set of points. While this is convenient for graphics output, it is less so for associating lines with data values, because the line is not subsetted into data objects in any other way than by `NA` values.

The approach adopted here is to start with a `Line` object that is a matrix of 2D coordinates, without `NA` values. A list of `Line` objects forms the `Lines` slot of a `Lines` object. An identifying character tag is also required, and will be used for constructing `SpatialLines` objects using the same approach as was used above for matching ID values for spatial points.

```
> getClass("Line")

Slots:

Name:  coords
Class: matrix

Known Subclasses: "Polygon"

> getClass("Lines")

Slots:

Name:      Lines        ID
Class:      list character
```

Neither `Line` nor `Lines` objects inherit from the `Spatial` class. It is the `SpatialLines` object that contains the bounding box and projection information for the list of `Lines` objects stored in its `lines` slot. This degree of complexity is required to be able to add observation values in a data frame, creating `SpatialLinesDataFrame` objects, and to use a range of extraction methods on these objects.

```
> getClass("SpatialLines")

Slots:

Name:        lines        bbox proj4string
Class:        list      matrix         CRS

Extends: "Spatial"

Known Subclasses: "SpatialLinesDataFrame"
```

Let us examine an example of an object of this class, created from lines retrieved from the **maps** package world database, and converted to a `SpatialLines` object using the `map2SpatialLines` function in **maptools**. We can see that the `lines` slot of the object is a list of 51 components, each of which must be a `Lines` object in a valid `SpatialLines` object.

```
> library(maps)
> japan <- map("world", "japan", plot = FALSE)
> p4s <- CRS("+proj=longlat +ellps=WGS84")
> SLjapan <- map2SpatialLines(japan, proj4string = p4s)
> str(SLjapan, max.level = 2)

Formal class 'SpatialLines' [package "sp"] with 3 slots
..@ lines :List of 51
..@ bbox :num [1:2, 1:2] 123.0 24.3 ...
.. ..- attr(*, "dimnames")=List of 2
..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
```

`SpatialLines` and `SpatialPolygons` objects are very similar, as can be seen in Fig. 2.4 – the lists of component entities stack up in a hierarchical fashion. A very typical way of exploring the contents of these objects is to use `lapply` or `sapply` in combination with `slot`. The `lapply` and `sapply` functions apply their second argument, which is a function, to each of the elements of their first argument. The command used here can be read as follows: return the length of the `Lines` slot – how many `Line` objects it contains – of each `Lines` object in the list in the `lines` slot of `SLjapan`, simplifying the result to a numeric vector. If `lapply` was used, the result would have been a list. As we see, no `Lines` object contains more than one `Line` object:

```
> Lines_len <- sapply(slot(SLjapan, "lines"),function(x) length(slot(x,
+     "Lines")))
> table(Lines_len)
```

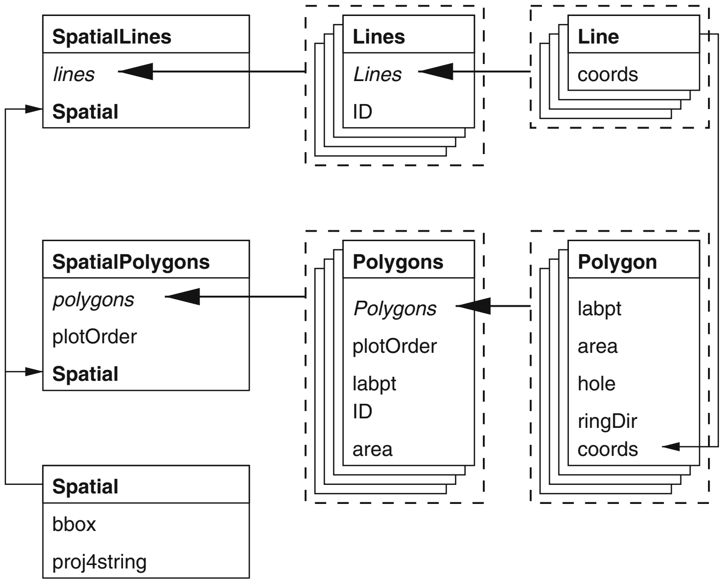

Fig. 2.4. SpatialLines and SpatialPolygons classes and slots; thin arrows show subclass extensions, thick arrows the inclusion of lists of objects

```
Lines_len
 1
51
```

We can use the `ContourLines2SLDF` function included in **maptools** in our next example, converting data returned by the base graphics function `contourLines` into a `SpatialLinesDataFrame` object; we used the volcano data set in Chap. 1, Fig. 1.3:

```
> volcano_sl <- ContourLines2SLDF(contourLines(volcano))
> t(slot(volcano_sl, "data"))
```

```
      C_1   C_2   C_3   C_4   C_5   C_6   C_7   C_8   C_9   C_10
level "100" "110" "120" "130" "140" "150" "160" "170" "180" "190"
```

We can see that there are ten separate contour level labels in the variable in the `data` slot, stored as a factor in the data frame in the object's `data` slot. As mentioned above, **sp** classes are new-style classes, and so the `slots` function can be used to look inside their slots.

To import data that we will be using shortly, we use another utility function in **maptools**, which reads shoreline data in 'Mapgen' format from the National Geophysical Data Center coastline extractor[6] into a `SpatialLines` object directly, here selected for the window shown as the object bounding box:

[6] `http://www.ngdc.noaa.gov/mgg/shorelines/shorelines.html`.

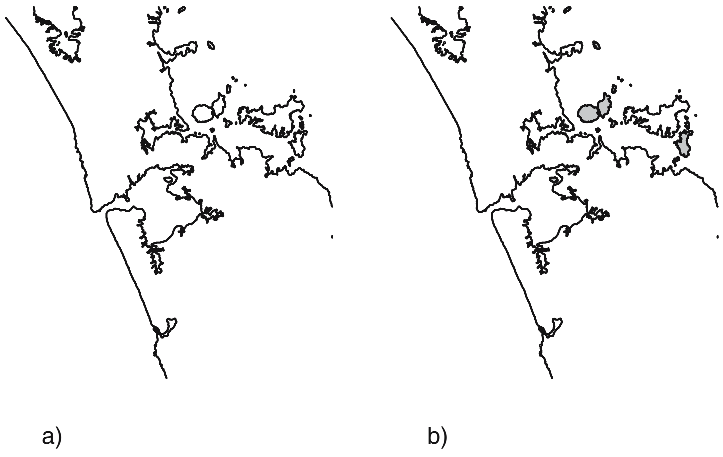

Fig. 2.5. Two maps of shorelines around Auckland: (**a**) line representation, (**b**) line representation over-plotted with islands converted to polygons and shaded. Note that Waiheke Island, the large island to the east, is not closed, and so not found as an island

```
> llCRS <- CRS("+proj=longlat +ellps=WGS84")
> auck_shore <- MapGen2SL("auckland_mapgen.dat", llCRS)
> summary(auck_shore)
```

```
Object of class SpatialLines
Coordinates:
     min   max
r1 174.2 175.3
r2 -37.5 -36.5
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
```

The shorelines are still just represented by lines, shown in Fig. 2.5, and so colour filling of apparent polygons formed by line rings is not possible. For this we need a class of polygon objects, discussed in Sect. 2.6. Lines, however, can be generalised by removing detail that is not required for analysis or visualisation – the **maps** and **RArcInfo** packages contain functions for line thinning. This operation can be performed successfully only on lines, because neighbouring polygons may have their shared boundary thinned differently. This leads to the creation of slivers, thin zones belonging to neither polygon or to both.

2.6 SpatialPolygons

The basic representation of a polygon in `S` is a closed line, a sequence of point coordinates where the first point is the same as the last point. A set of polygons is made of closed lines separated by NA points. Like lines, it is not easy to work with polygons represented this way. To have a data set to use for polygons, we first identify the lines imported above representing the shoreline around Auckland. Many are islands, and so have identical first and last coordinates.

```
> lns <- slot(auck_shore, "lines")
> table(sapply(lns, function(x) length(slot(x, "Lines"))))

 1
80

> islands_auck <- sapply(lns, function(x) {
+     crds <- slot(slot(x, "Lines")[[1]], "coords")
+     identical(crds[1, ], crds[nrow(crds), ])
+ })
> table(islands_auck)

islands_auck
FALSE  TRUE
   16    64
```

Since all the Lines in the auck_shore object contain only single Line objects, checking the equality of the first and last coordinates of the first Line object in each Lines object tells us which sets of coordinates can validly be made into polygons. The nesting of classes for polygons is the same as that for lines, but the successive objects have more slots.

```
> getClass("Polygon")

Slots:

Name:    labpt    area    hole ringDir  coords
Class: numeric numeric logical integer  matrix

Extends: "Line"
```

The Polygon class extends the Line class by adding slots needed for polygons and checking that the first and last coordinates are identical. The extra slots are a label point, taken as the centroid of the polygon, the area of the polygon in the metric of the coordinates, whether the polygon is declared as a hole or not – the default value is a logical NA, and the ring direction of the polygon (discussed later in Sect. 2.6.2). No check is made of whether lines cross or polygons have 'errors', in other words whether features are simple in the OpenGIS® (OpenGeoSpatial)[7] context; these are discussed briefly later on p. 122. GIS should do this, and we assume that data read into R can be trusted and contain only simple features.

```
> getClass("Polygons")
```

[7] http://www.opengeospatial.org/.

```
Slots:

Name:   Polygons plotOrder     labpt        ID      area
Class:      list   integer   numeric character   numeric
```

The `Polygons` class contains a list of valid `Polygon` objects, an identifying character string, a label point taken as the label point of the constituent polygon with the largest area, and two slots used as helpers in plotting using R graphics functions, given this representation of sets of polygons. These set the order in which the polygons belonging to this object should be plotted, and the gross area of the polygon, equal to the sum of all the constituent polygons. A `Polygons` object may, for example, represent an administrative district located on two sides of a river, or archipelago. Each of the parts should be seen separately, but data are only available for the larger entity.

```
> getClass("SpatialPolygons")

Slots:

Name:     polygons   plotOrder        bbox proj4string
Class:        list     integer      matrix         CRS

Extends: "Spatial"

Known Subclasses: "SpatialPolygonsDataFrame"
```

The top level representation of polygons is as a `SpatialPolygons` object, a set of `Polygons` objects with the additional slots of a `Spatial` object to contain the bounding box and projection information of the set as a whole. Like the `Polygons` object, it has a plot order slot, defined by default to plot its member polygons, stored in the `polygons` as a list of `Polygons`, in order of gross area, from largest to smallest. Choosing only the lines in the Auckland shoreline data set which are closed polygons, we can build a `SpatialPolygons` object.

```
> islands_sl <- auck_shore[islands_auck]
> list_of_Lines <- slot(islands_sl, "lines")
> islands_sp <- SpatialPolygons(lapply(list_of_Lines, function(x) {
+     Polygons(list(Polygon(slot(slot(x, "Lines")[[1]],
+         "coords"))), ID = slot(x, "ID"))
+ }), proj4string = CRS("+proj=longlat +ellps=WGS84"))
> summary(islands_sp)

Object of class SpatialPolygons
Coordinates:
         min       max
r1 174.30297 175.22791
r2 -37.43877 -36.50033
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
```

```
> slot(islands_sp, "plotOrder")

 [1] 45 54 37 28 38 27 12 11 59 53  5 25 26 46  7 55 17 34 30 16  6 43
[23] 14 40 32 19 61 42 15 50 21 18 62 23 22 29 24 44 13  2 36  9 63 58
[45] 56 64 52 39 51  1  8  3  4 20 47 35 41 48 60 31 49 57 10 33

> order(sapply(slot(islands_sp, "polygons"), function(x) slot(x,
+     "area")), decreasing = TRUE)

 [1] 45 54 37 28 38 27 12 11 59 53  5 25 26 46  7 55 17 34 30 16  6 43
[23] 14 40 32 19 61 42 15 50 21 18 62 23 22 29 24 44 13  2 36  9 63 58
[45] 56 64 52 39 51  1  8  3  4 20 47 35 41 48 60 31 49 57 10 33
```

As we saw with the construction of `SpatialLines` objects from raw coordinates, here we build a list of `Polygon` objects for each `Polygons` object, corresponding to a single identifying tag. A list of these `Polygons` objects is then passed to the `SpatialPolygons` function, with a coordinate reference system, to create the `SpatialPolygons` object. Again, like `SpatialLines` objects, `SpatialPolygons` objects are most often created by functions that import or manipulate such data objects, and seldom from scratch.

2.6.1 `SpatialPolygonsDataFrame` Objects

As with other spatial data objects, `SpatialPolygonsDataFrame` objects bring together the spatial representations of the polygons with data. The identifying tags of the `Polygons` in the `polygon` slot of a `SpatialPolygons` object are matched with the row names of the data frame to make sure that the correct data rows are associated with the correct spatial objects. The data frame is re-ordered by row to match the spatial objects if need be, provided all the objects can be matched to row names. If any differences are found, an error results. Both identifying tags and data frame row names are character strings, and so their sort order is also character, meaning that `"2"` follows `"11"` and `"111"`.[8]

As an example, we take a set of scores by US state of 1999 Scholastic Aptitude Test (SAT) used for spatial data analysis by Melanie Wall.[9] In the data source, there are also results for Alaska, Hawaii, and for the US as a whole. If we would like to associate the data with state boundary polygons provided in the **maps** package, it is convenient to convert the boundaries to a `SpatialPolygons` object – see also Chap. 4.

```
> library(maps)
> library(maptools)
> state.map <- map("state", plot = FALSE, fill = TRUE)
```

[8] Some **maptools** functions use Gregory R. Warnes' `mixedorder` sort from **gtools** to sort integer-like strings in integer order.

[9] `http://www.biostat.umn.edu/~melanie/Data/`, data here supplemented with variable names and state names as used in **maps**.

```
> IDs <- sapply(strsplit(state.map$names, ":"), function(x) x[1])
> state.sp <- map2SpatialPolygons(state.map, IDs = IDs,
+     proj4string = CRS("+proj=longlat +ellps=WGS84"))
```

Then we can use identifying tag matching to suit the rows of the data frame to the `SpatialPolygons`. Here, the rows of the data frame for which there are no matches will be dropped; all the `Polygons` objects are matched:

```
> sat <- read.table("state.sat.data_mod.txt", row.names = 5,
+     header = TRUE)
> str(sat)

'data.frame': 52 obs. of 4 variables:
$ oname :Factor w/ 52 levels "ala","alaska",..: 1 2 3 4 5 ...
$ vscore:int 561 516 524 563 497 ...
$ mscore:int 555 514 525 556 514 ...
$ pc :int 9 50 34 6 49 ...

> id <- match(row.names(sat), sapply(slot(state.sp, "polygons"),
+     function(x) slot(x, "ID")))
> row.names(sat)[is.na(id)]

[1] "alaska" "hawaii" "usa"

> state.spdf <- SpatialPolygonsDataFrame(state.sp, sat)
> str(slot(state.spdf, "data"))

'data.frame': 49 obs. of 4 variables:
$ oname :Factor w/ 52 levels "ala","alaska",..: 1 3 4 5 6 ...
$ vscore:int 561 524 563 497 536 ...
$ mscore:int 555 525 556 514 540 ...
$ pc :int 9 34 6 49 32 ...

> str(state.spdf, max.level = 2)

Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
..@ data :'data.frame': 49 obs. of 4 variables:
..@ polygons :List of 49
..@ plotOrder :int [1:49] 42 25 4 30 27 ...
..@ bbox :num [1:2, 1:2] -124.7 25.1 ...
.. ..- attr(*, "dimnames")=List of 2
..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
```

If we modify the row name of ‘`arizona`’ in the data frame to ‘`Arizona`’, there is no longer a match with a polygon identifying tag, and an error is signalled.

```
> rownames(sat)[3] <- "Arizona"
> SpatialPolygonsDataFrame(state.sp, sat)
```

```
Error in SpatialPolygonsDataFrame(state.sp, sat) : row.names
of data and Polygons IDs do not match
```

In subsequent analysis, Wall (2004) also drops District of Columbia. Rather than having to manipulate polygons and their data separately, when using a `SpatialPolygonsDataFrame` object, we can say:

```
> DC <- "district of columbia"
> not_dc <- !(row.names(slot(state.spdf, "data")) == DC)
> state.spdf1 <- state.spdf[not_dc, ]
> length(slot(state.spdf1, "polygons"))
```

```
[1] 48
```

```
> summary(state.spdf1)
```

```
Object of class SpatialPolygonsDataFrame
Coordinates:
          min       max
r1 -124.68134 -67.00742
r2   25.12993  49.38323
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Data attributes:
     oname        vscore          mscore             pc
 ala    : 1   Min.   :479.0   Min.   :475.0   Min.   : 4.00
 ariz   : 1   1st Qu.:506.2   1st Qu.:505.2   1st Qu.: 9.00
 ark    : 1   Median :530.5   Median :532.0   Median :28.50
 calif  : 1   Mean   :534.6   Mean   :534.9   Mean   :35.58
 colo   : 1   3rd Qu.:563.0   3rd Qu.:558.5   3rd Qu.:63.50
 conn   : 1   Max.   :594.0   Max.   :605.0   Max.   :80.00
 (Other):42
```

2.6.2 Holes and Ring Direction

The hole and ring direction slots are included in `Polygon` objects as heuristics to address some of the difficulties arising from `S` not being a GIS. In a traditional vector GIS, and in the underlying structure of the data stored in **maps**, boundaries between polygons are stored only once as arcs between nodes (shared vertices between three or more polygons, possibly including the external space), and the polygons are constructed on the fly from lists of directed boundary arcs, including boundaries with the external space – void – not included in any polygon. This is known as the topological representation of polygons, and is appropriate for GIS software, but arguably not for other software using spatial data. It was mentioned above that it is the user's responsibility to provide line coordinates such that the coordinates represent the line object the user requires. If the user requires, for example, that a river channel does not cross itself, the user has to impose that limitation. Other users will not need such a limitation, as for example tracking data may very well involve an animal crossing its tracks.

The approach that has been chosen in **sp** is to use two markers commonly encountered in practice, marking polygons as holes with a logical (TRUE/FALSE) flag, the `hole` slot, and using ring direction – clockwise rings are taken as not being holes, anti-clockwise as being holes. This is needed because the non-topological representation of polygons has no easy way of knowing that a polygon represents an internal boundary of an enclosing polygon, a hole, or lake.

An approach that works when the relative status of polygons is known is to set the hole slot directly. This is done in reading GSHHS shoreline data, already used in Fig. 2.3 and described in Chap. 4. The data source includes a variable for each polygon, where the levels are land: 1, lake: 2, island in lake: 3, and lake on island in lake: 4. The following example takes a region of interest on the northern, Canadian shore of Lake Huron, including Manitoulin Island, and a number of lakes on the island, including Kongawong Lake.

```
> length(slot(manitoulin_sp, "polygons"))

[1] 1

> sapply(slot(slot(manitoulin_sp, "polygons")[[1]], "Polygons"),
+     function(x) slot(x, "hole"))

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

> sapply(slot(slot(manitoulin_sp, "polygons")[[1]], "Polygons"),
+     function(x) slot(x, "ringDir"))

 [1]  1 -1  1 -1 -1  1 -1  1  1  1  1  1  1  1  1  1  1  1  1
```

In Fig. 2.6, there is only one `Polygons` object in the `polygons` slot of `manitoulin_sp`, representing the continental landmass, exposed along the northern edge, and containing the complete set of polygons. Within this is a large section covered by Lake Huron, which in turn is covered by islands and lakes on islands. Not having a full topological representation means that for plotting, we paint the land first, then paint the lake, then the islands, and finally the lakes on islands. Because the default plotting colour for holes is `'transparent'`, they can appear to be merged into the surrounding land – the same problem arises where the hole slot is wrongly assigned. The `plotOrder` slots in `Polygons` and `SpatialPolygons` objects attempt to get around this problem, but care is usually sensible if the spatial objects being handled are complicated.

2.7 SpatialGrid and SpatialPixel Objects

The point, line, and polygon objects we have considered until now have been handled one-by-one. Grids are regular objects requiring much less information to define their structure. Once the single point of origin is known, the extent

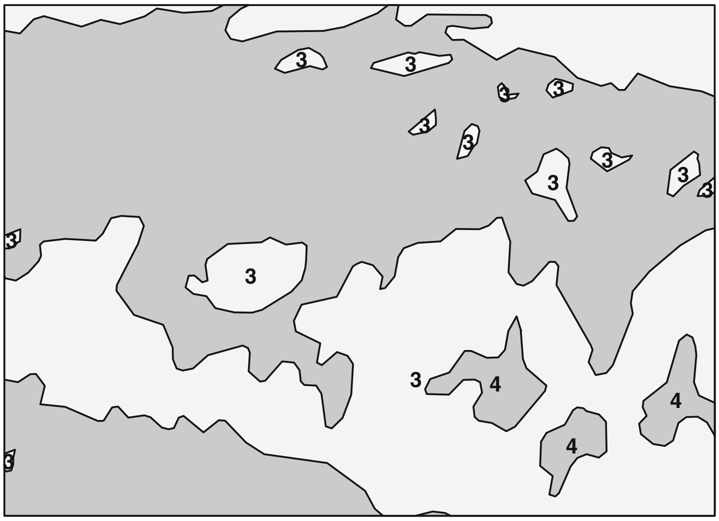

Fig. 2.6. The northern, Canadian shore of Lake Huron, including Manitoulin Island and lakes on the island; islands (*light grey*) and lakes on islands (*dark grey*) are marked with their GSHHS levels

of the grid can be given by the cell resolution and the numbers of rows and columns present in the full grid. This representation is typical for remote sensing and raster GIS, and is used widely for storing data in regular rectangular cells, such as digital elevation models, satellite imagery, and interpolated data from point measurements, as well as image processing.

```
> getClass("GridTopology")

Slots:

Name:  cellcentre.offset          cellsize         cells.dim
Class:           numeric           numeric           integer
```

As an example, we make a `GridTopology` object from the bounding box of the Manitoulin Island vector data set. If we choose a cell size of 0.01° in each direction, we can offset the south-west cell centre to make sure that at least the whole area is covered, and find a suitable number of cells in each dimension.

```
> bb <- bbox(manitoulin_sp)
> bb

     min   max
r1 277.0 278.0
r2  45.7  46.2

> cs <- c(0.01, 0.01)
> cc <- bb[, 1] + (cs/2)
> cd <- ceiling(diff(t(bb))/cs)
> manitoulin_grd <- GridTopology(cellcentre.offset = cc,
+     cellsize = cs, cells.dim = cd)
> manitoulin_grd

                       r1     r2
cellcentre.offset 277.005 45.705
cellsize            0.010  0.010
cells.dim         100.000 50.000
```

The object describes the grid completely, and can be used to construct a `SpatialGrid` object. A `SpatialGrid` object contains `GridTopology` and `Spatial` objects, together with two helper slots, `grid.index` and `coords`. These are set to zero and to the bounding box of the cell centres of the grid, respectively.

```
> getClass("SpatialGrid")

Slots:

Name:          grid   grid.index       coords         bbox
Class: GridTopology      integer       matrix       matrix

Name: proj4string
Class: CRS

Extends:
Class "SpatialPixels", directly, with explicit coerce
Class "SpatialPoints", by class "SpatialPixels", distance 2, with
     explicit coerce
Class "Spatial", by class "SpatialPixels", distance 3, with explicit
     coerce

Known Subclasses: "SpatialGridDataFrame"
```

Using the `GridTopology` object created above, and passing through the coordinate reference system of the original GSHHS data, the bounding box is created automatically, as we see from the summary of the object:

```
> p4s <- CRS(proj4string(manitoulin_sp))
> manitoulin_SG <- SpatialGrid(manitoulin_grd, proj4string = p4s)
> summary(manitoulin_SG)

Object of class SpatialGrid
Coordinates:
     min   max
r1 277.0 278.0
r2  45.7  46.2
Is projected: FALSE
proj4string : [+proj=longlat +datum=WGS84]
Number of points: 2
Grid attributes:
   cellcentre.offset cellsize cells.dim
r1           277.005     0.01       100
r2            45.705     0.01        50
```

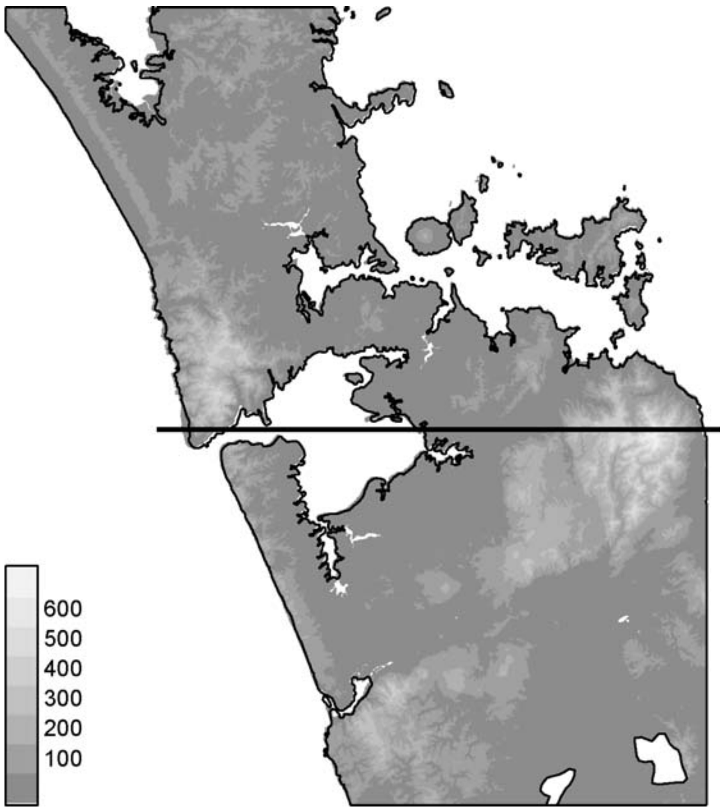

Fig. 2.7. SRTM elevation data in metres for the Auckland isthmus over-plotted with an excerpt from the GSHHS full resolution shoreline, including two lakes – there are detailed differences stemming from the very different technologies underlying the two data sources. A transect is marked for later use

As an example of using these classes with imported data, we use an excerpt from the Shuttle Radar Topography Mission (SRTM) flown in 2000, for the Auckland area[10] (Fig. 2.7). The data have been read from a Geotiff file into a `SpatialGridDataFrame` object – a `SpatialGrid` object extended with a `data` slot occupied by a `data.frame` object, filled with a single band of data representing elevation in metres.After checking the class of the data object, we examine in turn its slots. The `grid` slot contains the underlying `GridTopology` object, with the lower left cell centre coordinates, the pair of cell size resolution values, here both equal to 3 arcsec, and the numbers of columns and rows:

[10] Downloaded from the seamless data distribution system for 3 arcsec 'Finished' (90 m) data, `http://seamless.usgs.gov/`; the data can be downloaded as one degree square tiles, or cropped from a seamless raster database, as has been done here to avoid patching tiles together.

```
> class(auck_el1)

[1] "SpatialGridDataFrame"
attr(,"package")
[1] "sp"

> slot(auck_el1, "grid")

                             x             y
cellcentre.offset 1.742004e+02 -3.749958e+01
cellsize          8.333333e-04  8.333333e-04
cells.dim         1.320000e+03  1.200000e+03

> slot(auck_el1, "grid.index")

integer(0)

> slot(auck_el1, "coords")

            x         y
[1,] 174.2004 -37.49958
[2,] 175.2996 -36.50042

> slot(auck_el1, "bbox")

    min   max
x 174.2 175.3
y -37.5 -36.5

> object.size(auck_el1)

[1] 12674948

> object.size(slot(auck_el1, "data"))

[1] 12672392
```

The grid.index slot is empty, while the coords slot is as described earlier. It differs from the bounding box of the grid as a whole, contained in the bbox slot, by half a cell resolution value in each direction. The total size of the SpatialGridDataFrame object is just over 12 MB, almost all of which is made up of the data slot.

```
> is.na(auck_el1$band1) <- auck_el1$band1 <= 0 | auck_el1$band1 >
+     10000
> summary(auck_el1$band1)

     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.      NA's
     1.00     23.00     53.00     78.05    106.00    686.00 791938.00
```

Almost half of the data are at or below sea level, and other values are spikes in the radar data, and should be set to NA. Once this is done, about half of the data are missing. In other cases, even larger proportions of raster grids are missing, suggesting that an alternative representation of the same data might

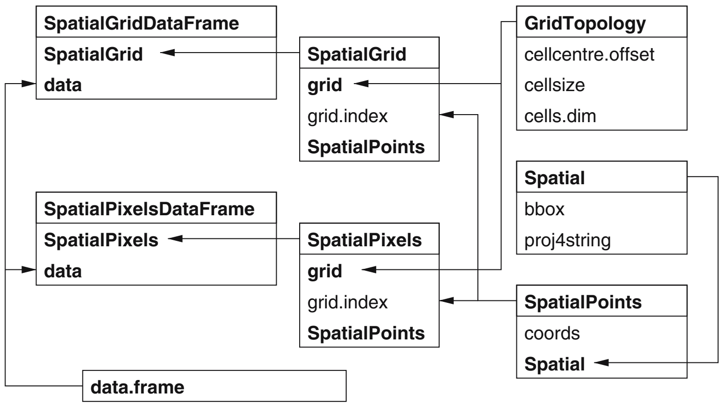

Fig. 2.8. `SpatialGrid` and `SpatialPixel` classes and their slots; arrows show subclass extensions

be attractive. One candidate from Terralib, discussed further in Chap. 4, is the cell representation of rasters, where the raster cells with data are represented by the coordinates of the cell centre, and by the sequence number of the cell among all the cells in the raster. In this representation, missing data are discarded, and savings in space and processing time can be large. It also permits cells to be stored, like points, in an external database. The class is here termed `SpatialPixels`, and has the same slots as `SpatialGrid` objects, but differently filled (Fig. 2.8). The `SpatialPixelsDataFrame` class is analogous.

```
> auck_el2 <- as(auck_el1, "SpatialPixelsDataFrame")

> object.size(auck_el2)

[1] 25349276

> object.size(slot(auck_el2, "grid.index"))

[1] 3168272

> object.size(slot(auck_el2, "coords"))

[1] 12673288

> sum(is.na(auck_el1$band1)) + nrow(slot(auck_el2, "coords"))

[1] 1584000

> prod(slot(slot(auck_el2, "grid"), "cells.dim"))

[1] 1584000
```

Returning to our example, we can coerce our `SpatialGridDataFrame` object to a `SpatialPixelsDataFrame` object. In this case, the proportion of missing to occupied cells is unfavourable, and when the `grid.index` and `coords` slots are populated with cell indices and coordinates, the output object is almost twice as large as its SpatialGridDataFrame equivalent. We can also see that the total number of cells – the product of the row and column dimensions – is equal to the number of coordinates in the output object plus the number of missing data values deleted by coercion. Had the number of attributes been 10, then the space saving relative to storing the full grid would have been 37%; with 100 attributes it would have been 48% for this particular case.

```
> auck_el_500 <- auck_el2[auck_el2$band1 > 500, ]

Warning messages:
1: grid has empty column/rows in dimension 1 in:
     points2grid(points, tolerance)
2: grid has empty column/rows in dimension 2 in:
     points2grid(points, tolerance)

> summary(auck_el_500)

Object of class SpatialPixelsDataFrame
Coordinates:
        min       max
x 175.18917 175.24333
y -37.10333 -37.01833
Is projected: FALSE
proj4string :
[+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs]
Number of points: 1114
Grid attributes:
  cellcentre.offset     cellsize cells.dim
x         175.18958 0.0008333333        65
y         -37.10292 0.0008333333       102
Data attributes:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  501.0   523.0   552.0   559.4   591.0   686.0

> object.size(auck_el_500)

[1] 38940
```

Taking just the raster cells over 500 m, of which there are very few, less than 1% of the total, yields a much smaller object. In this case it has a smaller bounding box, and gaps between the pixels present, as the warning messages indicate.

We can also create a SpatialPixels object directly from a SpatialPoints object. As our example, we use the Meuse bank data set provided with **sp**. We can pass a SpatialPoints object to the SpatialPixels function, where the Spatial object components are copied across, and the points checked to see whether they lie on a regular grid. If they do, the function will return a SpatialPixels object:

```
> data(meuse.grid)
> mg_SP <- SpatialPoints(cbind(meuse.grid$x, meuse.grid$y))
> summary(mg_SP)
```

```
Object of class SpatialPoints
Coordinates:
             min    max
coords.x1 178460 181540
coords.x2 329620 333740
Is projected: NA
proj4string : [NA]
Number of points: 3103
```

```
> mg_SPix0 <- SpatialPixels(mg_SP)
> summary(mg_SPix0)
```

```
Object of class SpatialPixels
Coordinates:
             min    max
coords.x1 178440 181560
coords.x2 329600 333760
Is projected: NA
proj4string : [NA]
Number of points: 3103
Grid attributes:
          cellcentre.offset cellsize cells.dim
coords.x1            178460       40        78
coords.x2            329620       40       104
```

```
> prod(slot(slot(mg_SPix0, "grid"), "cells.dim"))
```

```
[1] 8112
```

As we can see from the product of the cell dimensions of the underlying grid, over half of the full grid is not present in the `SpatialPixels` representation, because many grid cells lie outside the study area. Alternatively, we can coerce a `SpatialPoints` object to a `SpatialPixels` object:

```
> mg_SPix1 <- as(mg_SP, "SpatialPixels")
> summary(mg_SPix1)
```

```
Object of class SpatialPixels
Coordinates:
             min    max
coords.x1 178440 181560
coords.x2 329600 333760
Is projected: NA
proj4string : [NA]
Number of points: 3103
Grid attributes:
          cellcentre.offset cellsize cells.dim
coords.x1            178460       40        78
coords.x2            329620       40       104
```

We have now described a coherent and consistent set of classes for spatial data. Other representations are also used by R packages, and we show further ways of converting between these representations and external formats in Chap. 4. Before treating data import and export, we discuss graphical methods for **sp** classes, to show that the effort of putting the data in formal classes may be justified by the relative ease with which we can make maps.