6

Customising Spatial Data Classes and Methods

Although the classes defined in the **sp** package cover many needs, they do not go far beyond the most typical GIS data models. In applied research, it often happens that customised classes would suit the actual data coming from the instruments better. Since S4 classes have mechanisms for inheritance, it may be attractive to build on the **sp** classes, so as to utilise their methods where appropriate. Here, we demonstrate a range of different settings in which **sp** classes can be extended. Naturally, this is only useful for researchers with specific and clear needs, so our goal is to show how (relatively) easy it may be to prototype classes extending **sp** classes for specific purposes.

6.1 Programming with Classes and Methods

This section explains the elementary basics of programming with classes and methods in R. The S language (implemented in R and S-PLUS™) contains two mechanisms for creating classes and methods: the traditional S3 system and the more recent S4 system (see Sect. 2.2, in which classes were described for the useR – here they are described for the developeR). This chapter is not a full introduction to R programming (see Braun and Murdoch (2007) for more details), but it will try to give some feel of how the Spatial classes in package **sp** can be extended to be used for wider classes of problems. For full details, the interested reader is referred to, for example, Venables and Ripley (2000) and Chambers (1998), the latter being a reference for new-style S4 classes and methods. Example code is, for example, to be found in the source code for package **sp**, available from CRAN.

Suppose we define myfun as

```
> myfun <- function(x) {
+     x + 2
+ }
```

then, calling it with the numbers 1, 2, and 3 results in

```
> myfun(1:3)

[1] 3 4 5
```

or alternatively using a named argument:

```
> myfun(x = 1:3)

[1] 3 4 5
```

The return value of the function is the last expression evaluated. Often, we want to wrap existing functions, such as a plot function:

```
> plotXplus2Yminus3 <- function(x, y, ...) {
+     plot(x = x + 2, y = y - 3, ...)
+ }
```

In this case, the ... is used to pass information to the `plot` function without explicitly anticipating what it will be: named arguments `x` and `y` or the first two arguments if they are unnamed are processed, remaining arguments are passed on. The plot function is a generic method, with an instance that depends on the class of its first (`S3`) or first n arguments (`S4`). The available instances of `plot` are shown for `S3`-type methods by

```
> methods("plot")

 [1] plot.acf*            plot.data.frame*    plot.Date*
 [4] plot.decomposed.ts*  plot.default        plot.dendrogram*
 [7] plot.density         plot.ecdf           plot.factor*
[10] plot.formula*        plot.hclust*        plot.histogram*
[13] plot.HoltWinters*    plot.isoreg*        plot.lm
[16] plot.medpolish*      plot.mlm            plot.POSIXct*
[19] plot.POSIXlt*        plot.ppr*           plot.prcomp*
[22] plot.princomp*       plot.profile.nls*   plot.spec
[25] plot.spec.coherency  plot.spec.phase     plot.stepfun
[28] plot.stl*            plot.table*         plot.ts
[31] plot.tskernel*       plot.TukeyHSD

   Non-visible functions are asterisked
```

and for `S4`-type methods by

```
> library(sp)
> showMethods("plot")

Function: plot (package graphics)
x="ANY", y="ANY"
x="SpatialLines", y="missing"
x="Spatial", y="missing"
x="SpatialPoints", y="missing"
x="SpatialPolygons", y="missing"
```

where we first loaded **sp** to make sure there are some `S4` plot methods to show.

6.1.1 S3-Style Classes and Methods

In Chap. 2, we presented R classes and methods from the perspective of a useR; here we shift perspective to that of a developeR. Building S3-style classes is simple. Suppose we want to build an object of class `foo`:

```
> x <- rnorm(10)
> class(x) <- "foo"
> x
 [1] -1.59553650 -1.17102368  0.80900393  0.63390826  0.01971040
 [6] -0.69330839 -1.56896726 -0.22350820  0.20268852  0.96951209
attr(,"class")
[1] "foo"
```

If we plot this object, for example by `plot(x)`, we get the same plot as when we would not have set the class to `foo`. If we know, however, that objects of class `foo` need to be plotted without symbols but with connected lines, we can write a plot method for this class:

```
> plot.foo <- function(x, y, ...) {
+     plot.default(x, type = "l", ...)
+ }
```

after which `plot(x)` will call this particular method, rather than a default plot method.

Class inheritance is obtained in S3 when an object is given multiple classes, as in

```
> class(x) <- c("foo", "bar")
> plot(x)
```

For this plot, first function `plot.foo` will be looked for, and if not found the second option `plot.bar` will be looked for. If none of them is found, the default `plot.default` will be used.

The S3 class mechanism is simple and powerful. Much of R works with it, including key functions such as `lm`.

```
> data(meuse)
> class(meuse)
[1] "data.frame"
> class(lm(log(zinc) ~ sqrt(dist), meuse))
[1] "lm"
```

There is, however, no checking that a class with a particular name does indeed contain the elements that a certain method for it expects. It also has design flaws, as method specification by dot separation is ambiguous in case of names such as `as.data.frame`, where one cannot tell whether it means that the method `as.data` acts on objects of class `frame`, or the method `as` acts on objects of class `data.frame`, or none of them (the answer is: none). For such reasons, S4-style classes and methods were designed.

6.1.2 S4-Style Classes and Methods

S4-style classes are formally defined using `setClass`. As an example, somewhat simplified versions of classes `CRS` and `Spatial` in **sp** are

```
> setClass("CRS", representation(projargs = "character"))
> setClass("Spatial", representation(bbox = "matrix",
+     proj4string = "CRS"), validity <- function(object) {
+     bb <- bbox(object)
+     if (!is.matrix(bb))
+         return("bbox should be a matrix")
+     n <- dimensions(object)
+     if (n < 2)
+         return("spatial.dimension should be 2 or more")
+     if (any(is.na(bb)))
+         return("bbox should never contain NA values")
+     if (any(!is.finite(bb)))
+         return("bbox should never contain infinite values")
+     if (any(bb[, "max"] < bb[, "min"]))
+         return("invalid bbox: max < min")
+     TRUE
+ })
```

The command `setClass` defines a class name as a formal class, gives the names of the class elements (called slots), and their type–type checking will happen upon construction of an instance of the class. Further checking, for example on valid dimensions and data ranges can be done in the `validity` function. Here, the validity function retrieves the bounding box using the generic `bbox` method. Generics, if not defined in the base R system, for example

```
> isGeneric("show")
```

```
[1] TRUE
```

can be defined with `setGeneric`. Defining a specific instance of a generic is done by `setMethod`:

```
> setGeneric("bbox", function(obj) standardGeneric("bbox"))
> setMethod("bbox", signature = "Spatial", function(obj) obj@bbox)
```

where the signature tells the class of the first (or first n) arguments. Here, the `@` operator is used to access the `bbox` slot in an `S4` object, not to be confused with the `$` operator to access list elements.

We now illustrate this mechanism by providing a few examples of classes, building on those available in package **sp**.

6.2 Animal Track Data in Package Trip

CRAN Package **trip**, written by Michael Sumner (Kirkwood et al., 2006; Page et al., 2006), provides a class for animal tracking data. Animal tracking data consist of sets of (x, y, t) stamps, grouped by an identifier pointing to an individual animal, sensor, or perhaps isolated period of monitoring. A strategy for this (slightly simplified from that of **trip**) is to extend the

SpatialPointsDataFrame class by a length 2 character vector carrying the names of the time column and the trip identifier column in the SpatialPointsDataFrame attribute table.

Package **trip** does a lot of work to read and analyse tracking data from data formats typical for tracking data (Argos DAT), removing duplicate observations and validating the objects, for example checking that time stamps increase and movement speeds are realistic. We ignore this and stick to the bare bones.

We now define a class called trip that extends SpatialPointsDataFrame:

```
> library(sp)
> setClass("trip", representation("SpatialPointsDataFrame",
+     TOR.columns = "character"), validity <- function(object) {
+     if (length(object@TOR.columns) != 2)
+         stop("Time/id column names must be of length 2")
+     if (!all(object@TOR.columns %in% names(object@data)))
+         stop("Time/id columns must be present in attribute table")
+     TRUE
+ })

[1] "trip"

> showClass("trip")

Slots:

Name:  TOR.columns        data  coords.nrs      coords        bbox
Class:   character  data.frame     numeric      matrix      matrix

Name:  proj4string
Class:         CRS

Extends:
Class "SpatialPointsDataFrame", directly
Class "SpatialPoints", by class "SpatialPointsDataFrame", distance 2
Class "Spatial", by class "SpatialPointsDataFrame", distance 3
```

which checks, upon creation of objects, that indeed two variable names are passed and that these names refer to variables present in the attribute table.

6.2.1 Generic and Constructor Functions

It would be nice to have a constructor function, just like data.frame or SpatialPoints, and so we now create it and set it as the generic function to be called in case the first argument is of class SpatialPointsDataFrame.

```
> trip.default <- function(obj, TORnames) {
+     if (!is(obj, "SpatialPointsDataFrame"))
+         stop("trip only supports SpatialPointsDataFrame")
+     if (is.numeric(TORnames))
```

```
+         TORnames <- names(obj)[TORnames]
+     new("trip", obj, TOR.columns = TORnames)
+ }
> if (!isGeneric("trip")) setGeneric("trip", function(obj,
+     TORnames) standardGeneric("trip"))
[1] "trip"
> setMethod("trip", signature(obj = "SpatialPointsDataFrame",
+     TORnames = "ANY"), trip.default)
[1] "trip"
```

We can now try it out, with the turtle data of Chap. 2:

```
> turtle <- read.csv("seamap105_mod.csv")
> timestamp <- as.POSIXlt(strptime(as.character(turtle$obs_date),
+     "%m/%d/%Y %H:%M:%S"), "GMT")
> turtle <- data.frame(turtle, timestamp = timestamp)
> turtle$lon <- ifelse(turtle$lon < 0, turtle$lon + 360,
+     turtle$lon)
> turtle <- turtle[order(turtle$timestamp), ]
> coordinates(turtle) <- c("lon", "lat")
> proj4string(turtle) <- CRS("+proj=longlat +ellps=WGS84")
> turtle$id <- c(rep(1, 200), rep(2, nrow(coordinates(turtle)) -
+     200))
> turtle_trip <- trip(turtle, c("timestamp", "id"))
> summary(turtle_trip)
Object of class trip
Coordinates:
        min     max
lon 140.923 245.763
lat  21.574  39.843
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Number of points: 394
Data attributes:
       id                       obs_date
 Min.   :1.000   01/02/1997 04:16:53:  1
 1st Qu.:1.000   01/02/1997 05:56:25:  1
 Median :1.000   01/04/1997 17:41:54:  1
 Mean   :1.492   01/05/1997 17:20:07:  1
 3rd Qu.:2.000   01/06/1997 04:31:13:  1
 Max.   :2.000   01/06/1997 06:12:56:  1
                 (Other)            :388
   timestamp
 Min.   :1996-08-11 01:15:00
 1st Qu.:1996-10-30 00:10:04
 Median :1997-01-24 23:31:01
 Mean   :1997-01-26 06:24:56
 3rd Qu.:1997-04-10 12:26:20
 Max.   :1997-08-13 20:19:46
```

6.2.2 Methods for Trip Objects

The summary method here is not defined for `trip`, but is the default summary inherited from class `Spatial`. As can be seen, nothing special about the trip features is mentioned, such as what the time points are and what the identifiers. We could alter this by writing a class-specific summary method

```
> summary.trip <- function(object, ...) {
+     cat("Object of class \"trip\"\nTime column: ")
+     print(object@TOR.columns[1])
+     cat("Identifier column: ")
+     print(object@TOR.columns[2])
+     print(summary(as(object, "Spatial")))
+     print(summary(object@data))
+ }
> setMethod("summary", "trip", summary.trip)

[1] "summary"

> summary(turtle_trip)

Object of class "trip"
Time column: [1] "timestamp"
Identifier column: [1] "id"
Object of class Spatial
Coordinates:
        min     max
lon 140.923 245.763
lat  21.574  39.843
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
       id                      obs_date
 Min.   :1.000   01/02/1997 04:16:53:  1
 1st Qu.:1.000   01/02/1997 05:56:25:  1
 Median :1.000   01/04/1997 17:41:54:  1
 Mean   :1.492   01/05/1997 17:20:07:  1
 3rd Qu.:2.000   01/06/1997 04:31:13:  1
 Max.   :2.000   01/06/1997 06:12:56:  1
                 (Other)            :388
   timestamp
 Min.   :1996-08-11 01:15:00
 1st Qu.:1996-10-30 00:10:04
 Median :1997-01-24 23:31:01
 Mean   :1997-01-26 06:24:56
 3rd Qu.:1997-04-10 12:26:20
 Max.   :1997-08-13 20:19:46
```

As `trip` extends `SpatialPointsDataFrame`, subsetting using `"["` and column selection or replacement using `"[["` or `"$"` all work, as these are inherited. Creating invalid trip objects can be prohibited by adding checks to the validity

function in the class definition, for example will not work because the time and/or id column are not present any more.

A custom plot method for trip could be written, for example using colour to denote a change in identifier:

```
> setGeneric("lines", function(x, ...) standardGeneric("lines"))

[1] "lines"

> setMethod("lines", signature(x = "trip"), function(x,
+     ..., col = NULL) {
+     tor <- x@TOR.columns
+     if (is.null(col)) {
+         l <- length(unique(x[[tor[2]]]))
+         col <- hsv(seq(0, 0.5, length = l))
+     }
+     coords <- coordinates(x)
+     lx <- split(1:nrow(coords), x[[tor[2]]])
+     for (i in 1:length(lx)) lines(coords[lx[[i]], ],
+         col = col[i], ...)
+ })

[1] "lines"
```

Here, the `col` argument is added to the function header so that a reasonable default can be overridden, for example for black/white plotting.

6.3 Multi-Point Data: `SpatialMultiPoints`

One of the feature types of the OpenGeospatial Consortium (OGC) simple feature specification that has not been implemented in **sp** is the `MultiPoint` object. In a `MultiPoint` object, each feature refers to a *set of* points. The **sp** classes `SpatialPointsDataFrame` only provide reference to a single point. Instead of building a new class up from scratch, we try to re-use code and build a class `SpatialMultiPoint` from the `SpatialLines` class. After all, lines are just sets of ordered points.

In fact, the `SpatialLines` class implements the `MultiLineString` simple feature, where each feature can refer to multiple lines. A special case is formed if each feature only has a single line:

```
> setClass("SpatialMultiPoints", representation("SpatialLines"),
+     validity <- function(object) {
+         if (any(unlist(lapply(object@lines,
+             function(x) length(x@Lines))) !=
+             1))
+             stop("Only Lines objects with one Line element")
+         TRUE
+     })
```

```
[1] "SpatialMultiPoints"
```

```
> SpatialMultiPoints <- function(object) new("SpatialMultiPoints",
+     object)
```

As an example, we can create an instance of this class for two MultiPoint features each having three locations:

```
> n <- 5
> set.seed(1)
> x1 <- cbind(rnorm(n), rnorm(n, 0, 0.25))
> x2 <- cbind(rnorm(n), rnorm(n, 0, 0.25))
> x3 <- cbind(rnorm(n), rnorm(n, 0, 0.25))
> L1 <- Lines(list(Line(x1)), ID = "mp1")
> L2 <- Lines(list(Line(x2)), ID = "mp2")
> L3 <- Lines(list(Line(x3)), ID = "mp3")
> s <- SpatialLines(list(L1, L2, L3))
> smp <- SpatialMultiPoints(s)
```

If we now plot object `smp`, we get the same plot as when we plot `s`, showing the two lines. The `plot` method for a `SpatialLines` object is not suitable, so we write a new one:

```
> plot.SpatialMultiPoints <- function(x, ..., pch = 1:length(x@lines),
+     col = 1, cex = 1) {
+     n <- length(x@lines)
+     if (length(pch) < n)
+         pch <- rep(pch, length.out = n)
+     if (length(col) < n)
+         col <- rep(col, length.out = n)
+     if (length(cex) < n)
+         cex <- rep(cex, length.out = n)
+     plot(as(x, "Spatial"), ...)
+     for (i in 1:n) points(x@lines[[i]]@Lines[[1]]@coords,
+         pch = pch[i], col = col[i], cex = cex[i])
+ }
> setMethod("plot", signature(x = "SpatialMultiPoints",
+     y = "missing"), function(x, y, ...) plot.SpatialMultiPoints(x,
+     ...))
```

```
[1] "plot"
```

Here we chose to pass any named ... arguments to the plot method for a `Spatial` object. This function sets up the axes and controls the margins, aspect ratio, etc. All arguments that need to be passed to `points` (`pch` for symbol type, `cex` for symbol size, and `col` for symbol colour) need explicit naming and sensible defaults, as they are passed explicitly to the consecutive calls to `points`. According to the documentation of `points`, in addition to `pch`, `cex`, and `col`, the arguments `bg` and `lwd` (symbol fill colour and symbol line width) would need a similar treatment to make this plot method completely transparent with the base `plot` method – something an end user would hope for.

Having `pch`, `cex`, and `col` arrays, the length of the number of `MultiPoints` *sets* rather than the number of points to be plotted is useful for two reasons. First, the whole point of `MultiPoints` object is to distinguish *sets* of points. Second, when we extend this class to `SpatialMultiPointsDataFrame`, for example by

```
> cName <- "SpatialMultiPointsDataFrame"
> setClass(cName, representation("SpatialLinesDataFrame"),
+     validity <- function(object) {
+         lst <- lapply(object@lines, function(x) length(x@Lines))
+         if (any(unlist(lst) != 1))
+             stop("Only Lines objects with single Line")
+         TRUE
+     })

[1] "SpatialMultiPointsDataFrame"

> SpatialMultiPointsDataFrame <- function(object) {
+     new("SpatialMultiPointsDataFrame", object)
+ }
```

then we can pass symbol characteristics by (functions of) columns in the attribute table:

```
> df <- data.frame(x1 = 1:3, x2 = c(1, 4, 2), row.names = c("mp1",
+     "mp2", "mp3"))
> smp_df <- SpatialMultiPointsDataFrame(SpatialLinesDataFrame(smp,
+     df))
> setMethod("plot", signature(x = "SpatialMultiPointsDataFrame",
+     y = "missing"), function(x, y, ...) plot.SpatialMultiPoints(x,
+     ...))

[1] "plot"

> grys <- c("grey10", "grey40", "grey80")

> plot(smp_df, col = grys[smp_df[["x1"]]], pch = smp_df[["x2"]],
+     cex = 2, axes = TRUE)
```

for which the plot is shown in Fig. 6.1.

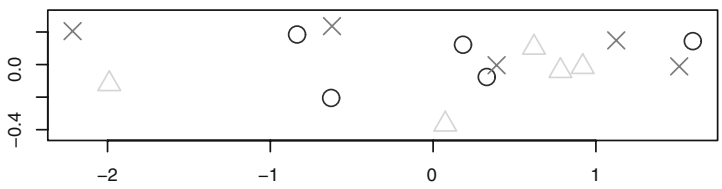

Fig. 6.1. Plot of the `SpatialMultiPointsDataFrame` object

6.4 Hexagonal Grids

Hexagonal grids are like square grids, where grid points are centres of matching hexagons, rather than squares. Package **sp** has no classes for hexagonal grids, but does have some useful functions for generating and plotting them. This could be used to build a class. Much of this code in **sp** is based on postings to the R-sig-geo mailing list by Tim Keitt, used with permission.

The spatial sampling method `spsample` has a method for sampling points on a hexagonal grid:

```
> data(meuse.grid)
> gridded(meuse.grid) = ~x + y
> xx <- spsample(meuse.grid, type = "hexagonal", cellsize = 200)
> class(xx)

[1] "SpatialPoints"
attr(,"package")
[1] "sp"
```

gives the points shown in the left side of Fig. 6.2. Note that an alternative hexagonal representation is obtained by rotating this grid 90°; we will not further consider that here.

```
> HexPts <- spsample(meuse.grid, type = "hexagonal", cellsize = 200)
> spplot(meuse.grid["dist"], sp.layout = list("sp.points",
+     HexPts, col = 1))
> HexPols <- HexPoints2SpatialPolygons(HexPts)
> df <- as.data.frame(meuse.grid)[overlay(meuse.grid, HexPts),
+     ]
> HexPolsDf <- SpatialPolygonsDataFrame(HexPols, df, match.ID = FALSE)
> spplot(HexPolsDf["dist"])
```

for which the plots are shown in Fig. 6.2.

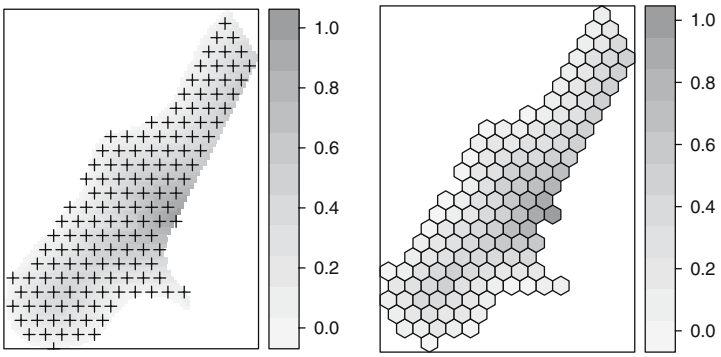

Fig. 6.2. Hexagonal points (*left*) and polygons (*right*)

We can now generate and plot hexagonal grids, but need to deal with two representations: as points and as polygons, and both representations do not tell by themselves that they represent a hexagonal grid.

For designing a hexagonal grid class we extend `SpatialPoints`, assuming that computation of the polygons can be done when needed without a prohibitive overhead.

```
> setClass("SpatialHexGrid", representation("SpatialPoints",
+     dx = "numeric"), validity <- function(object) {
+     if (object@dx <= 0)
+         stop("dx should be positive")
+     TRUE
+ })
```

```
[1] "SpatialHexGrid"
```

```
> setClass("SpatialHexGridDataFrame",
+     representation("SpatialPointsDataFrame",
+         dx = "numeric"), validity <- function(object) {
+         if (object@dx <= 0)
+             stop("dx should be positive")
+         TRUE
+     })
```

```
[1] "SpatialHexGridDataFrame"
```

Note that these class definitions do not check that instances actually do form valid hexagonal grids; a more robust implementation could provide a test that distances between points with equal y coordinate are separated by a multiple of `dx`, that the y-separations are correct and so on.

It might make sense to adapt the generic `spsample` method in package **sp** to return `SpatialHexGrid` objects; we can also add `plot` and `spsample` methods for them. Method `overlay` should work with a `SpatialHexGrid` as its first argument, by inheriting from `SpatialPoints`. Let us first see how to create the new classes. Without a constructor function we can use

```
> HexPts <- spsample(meuse.grid, type = "hexagonal", cellsize = 200)
> Hex <- new("SpatialHexGrid", HexPts, dx = 200)
> df <- as.data.frame(meuse.grid)[overlay(meuse.grid, Hex),
+     ]
> spdf <- SpatialPointsDataFrame(HexPts, df)
> HexDf <- new("SpatialHexGridDataFrame", spdf, dx = 200)
```

Because of the route taken to define both HexGrid classes, it is not obvious that the second extends the first. We can tell the S4 system this by `setIs`:

```
> is(HexDf, "SpatialHexGrid")
```

```
[1] FALSE
```

```
> setIs("SpatialHexGridDataFrame", "SpatialHexGrid")
> is(HexDf, "SpatialHexGrid")
```

```
[1] TRUE
```

to make sure that methods for `SpatialHexGrid` objects work as well for objects of class `SpatialHexGridDataFrame`.

When adding methods, several of them will need conversion to the polygon representation, so it makes sense to add the conversion function such that, for example `as(x, "SpatialPolygons")` will work:

```
> setAs("SpatialHexGrid", "SpatialPolygons",
+     function(from) HexPoints2SpatialPolygons(from,
+         from@dx))
> setAs("SpatialHexGridDataFrame", "SpatialPolygonsDataFrame",
+     function(from) SpatialPolygonsDataFrame(as(obj,
+         "SpatialPolygons"), obj@data,
+         match.ID = FALSE))
```

We can now add `plot`, `spplot`, `spsample`, and `overlay` methods for these classes:

```
> setMethod("plot", signature(x = "SpatialHexGrid", y = "missing"),
+     function(x, y, ...) plot(as(x, "SpatialPolygons"),\vspace*{-3pt}
+         ...))

[1] "plot"

> setMethod("spplot", signature(obj = "SpatialHexGridDataFrame"),
+     function(obj, ...) spplot(SpatialPolygonsDataFrame(as(obj,
+         "SpatialPolygons"), obj@data, match.ID = FALSE),
+         ...))

[1] "spplot"

> setMethod("spsample", "SpatialHexGrid", function(x, n,
+     type, ...) spsample(as(x, "SpatialPolygons"), n = n,
+     type = type, ...))

[1] "spsample"

> setMethod("overlay", c("SpatialHexGrid", "SpatialPoints"),
+     function(x, y, ...) overlay(as(x, "SpatialPolygons"),
+         y))

[1] "overlay"
```

After this, the following will work:

```
> spplot(meuse.grid["dist"], sp.layout = list("sp.points",
+     Hex, col = 1))
> spplot(HexDf["dist"])
```

Coercion to a data frame is done by

```
> as(HexDf, "data.frame")
```

Another detail not mentioned is that the bounding box of the hexgrid objects only match the grid centre points, not the hexgrid cells:

```
> bbox(Hex)
```

```
       min      max
x 178550.0 181450.0
y 329601.5 333585.3
```

```
> bbox(as(Hex, "SpatialPolygons"))
```

```
        min      max
r1 178450.0 181550.0
r2 329486.1 333700.7
```

One solution for this is to correct for this in a constructor function, and check for it in the validity test. Explicit coercion functions to the points representation would have to set the bounding box back to the points ranges. Another solution is to write a bbox method for the hexgrid classes, taking the risk that someone still looks at the incorrect bbox slot.

6.5 Spatio-Temporal Grids

Spatio-temporal data can be represented in different ways. One simple option is when observations (or model-results, or predictions) are given on a regular space–time grid.

Objects of class or extending `SpatialPoints`, `SpatialPixels`, and `SpatialGrid` do not have the constraint that they represent a two-dimensional space; they may have arbitrary dimension; an example for a three-dimensional grid is

```
> n <- 10
> x <- data.frame(expand.grid(x1 = 1:n, x2 = 1:n, x3 = 1:n),
+     z = rnorm(n^3))
> coordinates(x) <- ~x1 + x2 + x3
> gridded(x) <- TRUE
> fullgrid(x) <- TRUE
> summary(x)
```

```
Object of class SpatialGridDataFrame
Coordinates:
   min  max
x1 0.5 10.5
x2 0.5 10.5
x3 0.5 10.5
Is projected: NA
proj4string : [NA]
Number of points: 2
Grid attributes:
   cellcentre.offset cellsize cells.dim
x1                 1        1        10
x2                 1        1        10
```

```
x3                 1        1        10
Data attributes:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-3.00800 -0.70630 -0.03970 -0.02012  0.68930  3.81000
```

We might assume here that the third dimension, `x3`, represents time. If we are happy with time somehow represented by a real number (in double precision), then we are done. A simple representation is that of decimal year, with, for example 1980.5 meaning the 183rd day of 1980, or, for example relative time in seconds after the start of some event.

When we want to use the `POSIXct` or `POSIXlt` representations, we need to do some more work to see the readable version. We now devise a simple three-dimensional space–time grid with the `POSIXct` representation.

```
> setClass("SpatialTimeGrid", "SpatialGrid",
+     validity <- function(object) {
+         stopifnot(dimensions(object) ==
+             3)
+         TRUE
+     })

[1] "SpatialTimeGrid"
```

Along the same line, we can extend the `SpatialGridDataFrame` for space-time:

```
> setClass("SpatialTimeGridDataFrame", "SpatialGridDataFrame",
+     validity <- function(object) {
+         stopifnot(dimensions(object) == 3)
+         TRUE
+     })

[1] "SpatialTimeGridDataFrame"

> setIs("SpatialTimeGridDataFrame", "SpatialTimeGrid")
> x <- new("SpatialTimeGridDataFrame", x)
```

A crude summary for this class could be written along these lines:

```
> summary.SpatialTimeGridDataFrame <- function(object,
+     ...) {
+     cat("Object of class SpatialTimeGridDataFrame\n")
+     x <- gridparameters(object)
+     t0 <- ISOdate(1970, 1, 1, 0, 0, 0)
+     t1 <- t0 + x[3, 1]
+     cat(paste("first time step:", t1, "\n"))
+     t2 <- t0 + x[3, 1] + (x[3, 3] - 1) * x[3, 2]
+     cat(paste("last time step: ", t2, "\n"))
+     cat(paste("time step:      ", x[3, 2], "\n"))
+     summary(as(object, "SpatialGridDataFrame"))
+ }
```

```
> setMethod("summary", "SpatialTimeGridDataFrame",
+     summary.SpatialTimeGridDataFrame)

[1] "summary"

> summary(x)

Object of class SpatialTimeGridDataFrame
first time step: 1970-01-01 00:00:01
last time step:  1970-01-01 00:00:10
time step:       1
Object of class SpatialGridDataFrame
Coordinates:
   min  max
x1 0.5 10.5
x2 0.5 10.5
x3 0.5 10.5
Is projected: NA
proj4string : [NA]
Number of points: 2
Grid attributes:
   cellcentre.offset cellsize cells.dim
x1                 1        1        10
x2                 1        1        10
x3                 1        1        10
Data attributes:
    Min.  1st Qu.   Median     Mean  3rd Qu.
-3.00800 -0.70630 -0.03970 -0.02012  0.68930
    Max.
 3.81000
```

Next, suppose we need a subsetting method that selects on the time. When the first subset argument is allowed to be a time range, this is done by

```
> subs.SpatialTimeGridDataFrame <- function(x, i, j, ...,
+     drop = FALSE) {
+     t <- coordinates(x)[, 3] + ISOdate(1970, 1, 1, 0,
+         0, 0)
+     if (missing(j))
+         j <- TRUE
+     sel <- t %in% i
+     if (!any(sel))
+         stop("selection results in empty set")
+     fullgrid(x) <- FALSE
+     if (length(i) > 1) {
+         x <- x[i = sel, j = j, ...]
+         fullgrid(x) <- TRUE
+         as(x, "SpatialTimeGridDataFrame")
+     }
+     else {
+         gridded(x) <- FALSE
```

```
+         x <- x[i = sel, j = j, ...]
+         cc <- coordinates(x)[, 1:2]
+         p4s <- CRS(proj4string(x))
+         SpatialPixelsDataFrame(cc, x@data, proj4string = p4s)
+     }
+ }
> setMethod("[", c("SpatialTimeGridDataFrame", "POSIXct",
+     "ANY"), subs.SpatialTimeGridDataFrame)

[1] "["

> t1 <- as.POSIXct("1970-01-01 0:00:03", tz = "GMT")
> t2 <- as.POSIXct("1970-01-01 0:00:05", tz = "GMT")
> summary(x[c(t1, t2)])

Object of class SpatialTimeGridDataFrame
first time step: 1970-01-01 00:00:03
last time step:  1970-01-01 00:00:05
time step:       2
Object of class SpatialGridDataFrame
Coordinates:
   min  max
x1 0.5 10.5
x2 0.5 10.5
x3 2.0  6.0
Is projected: NA
proj4string : [NA]
Number of points: 2
Grid attributes:
   cellcentre.offset cellsize cells.dim
x1                 1        1        10
x2                 1        1        10
x3                 3        2         2
Data attributes:
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-3.0080000 -0.6764000 -0.0002298 -0.0081510  0.6546000  3.8100000

> summary(x[t1])

Object of class SpatialPixelsDataFrame
Coordinates:
   min  max
x1 0.5 10.5
x2 0.5 10.5
Is projected: NA
proj4string : [NA]
Number of points: 100
Grid attributes:
   cellcentre.offset cellsize cells.dim
x1                 1        1        10
x2                 1        1        10
```

```
Data attributes:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.8890 -0.4616  0.1353  0.0773  0.7779  2.6490
```

The reason to only convert back to `SpatialTimeGridDataFrame` when multiple time steps are present is that the time step ('cell size' in time direction) cannot be found when there is only a single step. In that case, the current selection method returns an object of class `SpatialPixelsDataFrame` for that time slice.

Plotting a set of slices could be done using levelplot or writing another `spplot` method:

```
> spplot.stgdf <- function(obj, zcol = 1, ..., format = NULL) {
+     if (length(zcol) != 1)
+         stop("can only plot a single attribute")
+     if (is.null(format))
+         format <- "%Y-%m-%d %H:%M:%S"
+     cc <- coordinates(obj)
+     df <- unstack(data.frame(obj[[zcol]], cc[, 3]))
+     ns <- as.character(coordinatevalues(getGridTopology(obj))[[3]] +
+         ISOdate(1970, 1, 1, 0, 0, 0), format = format)
+     cc2d <- cc[cc[, 3] == min(cc[, 3]), 1:2]
+     obj <- SpatialPixelsDataFrame(cc2d, df)
+     spplot(obj, names.attr = ns, ...)
+ }
> setMethod("spplot", "SpatialTimeGridDataFrame", spplot.stgdf)

[1] "spplot"
```

Now, the result of

```
> library(lattice)
> trellis.par.set(canonical.theme(color = FALSE))
> spplot(x, format = "%H:%M:%S", as.table = TRUE, cuts = 6,
+     col.regions = grey.colors(7, 0.55, 0.95, 2.2))
```

is shown in Fig. 6.3. The format argument passed controls the way time is printed; one can refer to the help of

```
> `?`(as.character.POSIXt)
```

for more details about the `format` argument.

6.6 Analysing Spatial Monte Carlo Simulations

Quite often, spatial statistical analysis results in a large number of spatial realisations of a random field, using some Monte Carlo simulation approach. Regardless whether individual values refer to points, lines, polygons, or grid cells, we would like to write some methods or functions that aggregate over these simulations, to get summary statistics such as the mean value, quantiles, or cumulative distributions values. Such aggregation can take place in two

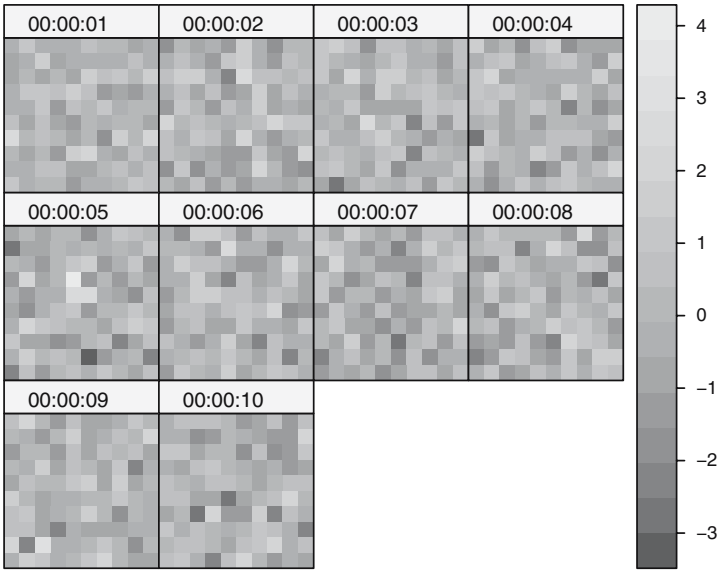

Fig. 6.3. `spplot` for an object of class `SpatialTimeGridDataFrame`, filled with random numbers

ways. Either we aggregate over the probability space and compute summary statistics for each geographical feature over the set of realisations (i.e. the rows of the attribute table), or for each realisation we aggregate over the complete geographical layer or a subset of it (i.e. aggregate over the columns of the attribute table).

Let us first generate, as an example, a set of 100 conditional Gaussian simulations for the zinc variable in the meuse data set:

```
> library(gstat)
> data(meuse)
> coordinates(meuse) <- ~x + y
> v <- vgm(0.5, "Sph", 800, 0.05)
> sim <- krige(log(zinc) ~ 1, meuse, meuse.grid, v, nsim = 100,
+     nmax = 30)

drawing 100 GLS realisations of beta...
[using conditional Gaussian simulation]

> sim@data <- exp(sim@data)
```

where the last statement back-transforms the simulations from the log scale to the observation scale. A quantile method for Spatial object attributes can be written as

```
> quantile.Spatial <- function(x, ..., byLayer = FALSE) {
+     stopifnot("data" %in% slotNames(x))
+     apply(x@data, ifelse(byLayer, 2, 1), quantile, ...)
+ }
```

after which we can find the sample below and above 95% confidence limits by

```
> sim$lower <- quantile.Spatial(sim[1:100], probs = 0.025)
> sim$upper <- quantile.Spatial(sim[1:100], probs = 0.975)
```

To get the sample distribution of the areal median, we can aggregate over layers:

```
> medians <- quantile.Spatial(sim[1:100], probs = 0.5,
+     byLayer = TRUE)

> hist(medians)
```

It should be noted that in these particular cases, the quantities computed by simulations could have been obtained faster and exactly by working analytically with ordinary (block) kriging and the normal distribution (Sect. 8.7.2).

A statistic that cannot be obtained analytically is the sample distribution of the area fraction that exceeds a threshold. Suppose that 500 is a crucial threshold, and we want to summarise the sampling distribution of the area fraction where 500 is exceeded:

```
> fractionBelow <- function(x, q, byLayer = FALSE) {
+     stopifnot(is(x, "Spatial") || !("data" %in%
+         slotNames(x)))
+     apply(x@data < q, ifelse(byLayer,
+         2, 1), function(r) sum(r)/length(r))
+ }

> over500 <- 1 - fractionBelow(sim[1:100], 200, byLayer = TRUE)
> summary(over500)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.6007  0.6768  0.6911  0.6897  0.7071  0.7448

> quantile(over500, c(0.025, 0.975))

     2.5%     97.5%
0.6460361 0.7336529
```

For space–time data, we could write methods that aggregate over space, over time, or over space and time.

6.7 Processing Massive Grids

Up to now we have made the assumption that gridded data can be completely read and are kept by R in memory. In some cases, however, we need to process grids that exceed the memory capacity of the computers available. A method for analysing grids without fully loading them into memory then seems useful. Note that package **rgdal** allows for partial reading of grids, for example

```
> x <- readGDAL("70042108.tif", output.dim = c(120, 132))
> x$band1[x$band1 <= 0] <- NA
> spplot(x, col.regions = bpy.colors())
```

reads a downsized grid, where 1% of the grid cells remained. Another option is reading certain rectangular sections of a grid, starting at some offset.

Yet another approach is to use the low-level opening routines and then subset:

```
> library(rgdal)
> x <- GDAL.open("70042108.tif")
> class(x)

[1] "GDALReadOnlyDataset"
attr(,"package")
[1] "rgdal"

> x.subs <- x[1:100, 1:100, 1]
> class(x.subs)

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

> gridparameters(x.subs)

  cellcentre.offset     cellsize cells.dim
x         174.20042 0.0008333333       100
y         -36.58292 0.0008333333       100
```

An object of class `GDALReadOnlyDataset` contains only a file handle. The subset method `"["` for it does not, as it quite often does, return an object of the same class but actually reads the data requested, with arguments interpreted as *rows*, *columns*, and raster *bands*, and returns a `SpatialGridDataFrame`. We now extend this approach to allow partial writing through `"["` as well. As the actual code is rather lengthy and involves a lot of administration, it will not all be shown and details can be found in the **rgdal** source code.

We define two classes,

```
> setClass("SpatialGDAL", representation("Spatial",
+     grid = "GridTopology", grod = "GDALReadOnlyDataset",
+     name = "character"))

[1] "SpatialGDAL"

> setClass("SpatialGDALWrite", "SpatialGDAL")

[1] "SpatialGDALWrite"
```

that derive from `Spatial`, contain a `GridTopology`, and a file handle in the `grod` slot. Next, we can define a function `open.SpatialGDAL` to open a raster file, returning a `SpatialGDAL` object and a function `copy.SpatialGDAL` that returns a writable copy of the opened raster. Note that some GDAL drivers allow only copying, some only writing, and some both.

```
> x <- open.SpatialGDAL("70042108.tif")
> nrows <- GDALinfo("70042108.tif")["rows"]
> ncols <- GDALinfo("70042108.tif")["columns"]
> xout <- copy.SpatialGDAL(x, "70042108out.tif")
> bls <- 20
> for (i in 1:(nrows/bls - 1)) {
+     r <- 1 + (i - 1) * bls
+     for (j in 1:(ncols/bls - 1)) {
+         c <- 1 + (j - 1) * bls
+         x.in <- x[r:(r + bls), c:(c + bls)]
+         xout[r:(r + bls), c:(c + bls)] <- x.in$band1 +
+             10
+     }
+     cat(paste("row-block", i, "\n"))
+ }
> close(x)
> close(xout)
```

This requires the functions `"["` and `"[<-"` to be present. They are set by

```
> setMethod("[", "SpatialGDAL", function(x, i, j, ...,
+     drop = FALSE) x@grod[i = i, j = j, ...])
> setReplaceMethod("[", "SpatialGDALWrite", function(x,
+     i, j, ..., value) {
+     ...
+ })
```

where, for the latter, the implementation details are omitted here. It should be noted that single rows or columns cannot be read this way, as they cannot be converted sensibly to a grid.

It should be noted that flat binary representations such as the Arc/Info Binary Grid allow much faster random access than ASCII representations or compressed formats such as jpeg varieties. Also, certain drivers in the GDAL library suggest an optimal block size for partial access (e.g. typically a single row), which is not used here[1].

This chapter has sketched developments beyond the base **sp** classes and methods used otherwise in this book. Although we think that the base classes cater for many standard kinds of spatial data analysis, it is clear that specific research problems will call for specific solutions, and that the R environment provides the high-level abstractions needed to help busy researchers get their work done.

[1] An attempt to use this block size is, at time of writing, found in the `blockApply` code, found in the THK CVS branch of the **rgdal** project on SourceForge.