

# Introduction to spatial data handling in R

Robert J. Hijmans

June 4, 2013

## 1 Introduction

This document (in its current draft form) provides a brief introduction to prepare you for spatial data handling in R. It discusses some practical aspects of working with spatial data, such as importing data and manipulating it to prepare for analysis. But, in its current form, the document mostly discusses general data handling in R and then briefly introduces spatial data. Companion documents further explain how to work with spatial data.

Spatial data refers to locations. Here these are geographical locations, that is, locations on the surface of the earth, although most the concepts can be used for locations on other planets, or other regions such as the surface of your skin.

You cannot effectively work with spatial data in R if you do not know the basics of the R language. Therefore, before getting to spatial data basic data types and data structures in R are first discussed (Chapter 2). Chapter 3 discusses some general properties of spatial data, and chapter 4 shows how you can represent spatial data with the basic data types. Chapter 5 introduces the main spatial data types that are available in R that make spatial data handling an easy task.

Don't go through this document too fast. Carefully read through the example code. Copy and paste the code to R and run it yourself. Make changes to the code and predict the result before running it to test your understanding.

## 2 Basic data types in R

This section describes, very briefly, some of the basic data types in R. Experienced R users need not read it; or can go through it quickly). It is meant as a reminder to those who have not used R much, or not recently. If the material in this section is not immediately obvious to you, you should probably consult a text book that introduces R. Good introductions to the R language are provided by Zuur et al (2009) and Adler (2009). These are both introductory texts that focus on teaching the language, not on teaching statistics (and each of these topics is hard enough to study by themselves).

## 2.1 Vectors

The most basic data type in R is the 'vector', a one-dimensional array of  $n$  values of a certain type (numeric, integer, character, logical, or factor). Even a single number is a vector (of length 1). These vectors can easily be created and manipulated:

Let's create variable `v0` that holds one number, and variables `v1` and `v2` holding three numbers. For `v1` we use the `c` (short for combine) function; for `v2` we use `:` to create a regular sequence.

```
> v0 <- 7
> v0

[1] 7

> v1 <- c(1.25, 2.9, 3.0)
> v1

[1] 1.25 2.90 3.00

> # going from 5 to 7
> v2 <- 5:7
> v2

[1] 5 6 7
```

Note that when R prints the value of `textttv0` it shows `[1] 7` because 7 is the first element in vector `v0`. That is, even a single number is a vector (of length 1). Also note that I use the "arrow" for assignment (assigning a value to a variable name) because it is the original R idiom and I think it is clearer than using `texttt=` which is now also allowed as that might be easily confused with testing for equality (`==`, see below).

Now let's create variable `x` holding a character vector of five elements. Function `class` tells us what kind of object (of which class) we have.

```
> x <- c('a', 'bc', 'def', 'gh', 'i')
> x

[1] "a"    "bc"   "def"  "gh"   "i"

> class(x)

[1] "character"

> length(x)

[1] 5
```

Remember to use quotes for character values (because character strings without quotes represent variable names), and hence to not use quotes for variables (x does not have quotes) because they would be interpreted as a character value.

```
> x <- c(a, bc, def)
Error: object 'a' not found
> 'x'
[1] "x"
```

Also remember that the character value 'a' is not equal to 'A' (double-quoted "a" is the same as single-quoted 'a', but you cannot mix the two: "a' is invalid) and that **variable** x is not the same as X. Not noting the difference between upper and lower case characters is one of the main sources of frustration for beginners with R.

A **factor** is a nominal (categorical) variable with a set of known possible values. They can be created using the `as.factor` function. In R you typically need to convert a character variable to a factor to use it in statistical modeling.

```
> f1 <- as.factor(x)
> f1
```

```
[1] a   bc  def gh  i
Levels: a bc def gh i
```

```
> f2 <- as.factor(5:7)
> f2[1]
```

```
[1] 5
Levels: 5 6 7
```

```
> as.integer(f2)
```

```
[1] 1 2 3
```

The result of `as.integer(f2)` may have been surprising. But it should not be, as there is no direct link between a category with label "5" and the number 5. In this case "5" is simply the first category and hence it gets converted to the integer 1. If you wanted the original numbers, you should do:

```
> fc2 <- as.character(f2)
> as.integer(fc2)
```

```
[1] 5 6 7
```

Elements of vectors can be obtained by indexing. Remember that brackets [ ] are used for indexing, whereas parentheses ( ) are used to call a function.

```
> # first element
> v2[1]
```

```
[1] 5
```

```
> # elements 2 to 3
> v2[2:3]
```

```
[1] 6 7
```

```
> # all elements but the first two
> v2[-c(1:2)]
```

```
[1] 7
```

Vectors can be used to compute new vectors with simple algebraic expressions.

```
> # are the elements of v1 2?
> v1 == 2
```

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

```
> # are the elements of v1 larger than 2?
> f <- v1 > 2
> f
```

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

```
> # element wise multiplication
> v3 <- v1 * v2
> v3
```

```
[1] 6.25 17.40 21.00
```

```
> # add all elements
> sum(v3)
```

```
[1] 44.65
```

In the examples above the computations used either vectors of the same length, or one of the vectors had length 1. But be careful, you can use algebraic computations with vectors of different lengths, as the shorter ones will be "recycled". R only issues a warning if the length of longer vector is not a multiple of the length of the shorter object. This is a great feature when you need it, but it may also make you overlook errors when your data are not what you think they are.

```
> a <- 1:6
> b <- 1:2
> a * b
```

```
[1] 1 4 3 8 5 12
```

The examples above illustrate a special feature of R not found in most other programming languages. This is that you do not need to 'loop' over elements in an array (vector in this case) to compute new values. It is important to use this feature as much as possible. In other programming languages you would need to do something like the 'for-loop' below to achieve the above. This is illustrated below. Note that the braces are used to open and close a "block" of code.

```
> # initialization of output variables
> s <- 0
> r <- vector(length=length(v1))
> # i goes from 1 to 3 (the length of v1)
> for (i in 1:length(v1)) {
+     s <- s + v1[i]
+     r[i] <- v1[i] * v2[i]
+ }
> s

[1] 7.15

> r

[1] 6.25 17.40 21.00

> # another example, with an if/else branch:
>
> f <- vector(length=length(v1))
> # i goes from 1 to 3 (the length of v1)
> for (i in 1:length(v1)) {
+     if (v1[i] > 2) {
+         f[i] <- TRUE
+     } else {
+         f[i] <- FALSE
+     }
+ }
> f

[1] FALSE TRUE TRUE
```

In R we avoid loops wherever we can, as they tend to be much slower than 'vectorized' computation, and because they are less concise. At first code using for-loops may seem easier to read, but after using R for a while, the reverse is true in most cases.

## 2.2 Matrices

A two-dimensional array can be represented with a matrix. Here is how you can create a matrix with missing NA values:

```
> matrix(ncol=3, nrow=3)
```

```
      [,1] [,2] [,3]
[1,]    NA    NA    NA
[2,]    NA    NA    NA
[3,]    NA    NA    NA
```

A matrix with values 1 to 9 (note that by default the values are distributed column-wise)

```
> matrix(1:6, ncol=3, nrow=2)
```

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

```
> matrix(1:6, ncol=3, nrow=2, byrow=TRUE)
```

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

```
> # the above can also be achieved using the transpose function
```

```
> # (note the reversal of ncol and nrow values)
```

```
> t(matrix(1:6, ncol=2, nrow=3))
```

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

You can also create a matrix by column-binding and/or row-binding vectors:

```
> v1 <- c(1,2,3)
```

```
> v2 <- 5:7
```

```
> # column bind
```

```
> m1 <- cbind(v1, v2)
```

```
> m1
```

```
      v1 v2
[1,]  1  5
[2,]  2  6
[3,]  3  7
```

```
> # row bind
```

```
> m2 <- rbind(v1, v2, v1*v2)
```

```
> m2
```

```
      [,1] [,2] [,3]
v1      1     2     3
v2      5     6     7
      5    12    21
```

```

> m3 <- cbind(m1, m2)
> # get the column names
> colnames(m3)

[1] "v1" "v2" ""  ""  ""

> # set the column names
> colnames(m3) <- c('ID', 'V2', 'X', 'Y', 'Z')
> m3

  ID V2 X  Y  Z
v1  1  5  1  2  3
v2  2  6  5  6  7
    3  7  5 12 21

> # dimensions of m3 (nrow, ncol)
> dim(m3)

[1] 3 5

```

Like vectors, values of matrices can be accessed through indexing. You can use a single number, in which case the cells are numbered column-wise (i.e. first the rows in the first column, then the second column, etc.), but it is often easier to use two numbers in a double index, the first for the row number(s) and the second for the column number(s).

```

> # one value
> m3[2,2]

[1] 6

> # equivalent to
> m3[5]

[1] 6

> # 2 columns and rows
> m3[1:2,1:2]

  ID V2
v1  1  5
v2  2  6

> # entire row
> m3[2, ]

  ID V2 X  Y  Z
2   6  5  6  7

```

```

> # entire column
> m3[,2]

v1 v2
5  6  7

> # you can also use column- or rownames for subsetting
> m3[c('v1', 'v2'), c('ID', 'X')]

      ID X
v1    1 1
v2    2 5

```

Computation with matrices is also 'vectorized'. For example you can do `m3 * 5` to multiply all values of `m3` with 5 or do `m3^2` or `m3 * m3` to square the values of `m3`. But often we need to compute values for the margins of a matrix, that is, a single value for each row or column. The `apply` function can be used for that:

```

> # sum values in each row
> apply(m3, 1, sum)

v1 v2
12 26 48

> # get mean for each column
> apply(m3, 2, mean)

      ID      V2      X      Y      Z
2.000000 6.000000 3.666667 6.666667 10.333333

```

Note that the `apply` uses at least three arguments: a matrix, a 1 or 2 indicating whether the computation is for rows or for columns, and a function that computes a new value (or values) for each row or column. You can read more about this in the help file of the function (type `?apply`). In most cases you will also add the argument `na.rm=TRUE` to remove NA (missing) values as any computation that includes an NA value will return NA. In this case we used existing basic functions `mean` and `sum` but we could write our own function.

Note that `apply` (and related functions such as `tapply` and `sapply`) are ways to avoid writing a loop. In the `apply` examples above you could have written a loop to do the computations row by row (or column by column) but using `apply` is more compact and efficient.

## 2.3 data.frame

In R, matrices can only contain one data type (e.g. numeric or character). In contrast, a `data.frame` can have columns (variables) of different types (each variable has to be of a single type though). A `data.frame` is what you get



when you read spread-sheet like data into R with functions like `read.table` or `read.csv`. We can also create a `data.frame` with some code, for example like this:

```
> d <- data.frame(ID=as.integer(1:4), name=c('Ana', 'Rob', 'Liu', 'Veronica'),
+               sex=as.factor(c('F','M','M', 'F')), score=c(10.2, 9, 13.5, 18),
+               stringsAsFactors=FALSE)
> d
```

	ID	name	sex	score
1	1	Ana	F	10.2
2	2	Rob	M	9.0
3	3	Liu	M	13.5
4	4	Veronica	F	18.0

```
> class(d)

[1] "data.frame"

> class(d$name)

[1] "character"

> sapply(d, class)
```

	ID	name	sex	score
	"integer"	"character"	"factor"	"numeric"

Indexing data.frames can be done as for matrices, but variables can also be accessed using the `$` sign:

```
> d$name

[1] "Ana"      "Rob"      "Liu"      "Veronica"

> d[, 'name']

[1] "Ana"      "Rob"      "Liu"      "Veronica"

> d[,2]

[1] "Ana"      "Rob"      "Liu"      "Veronica"
```

You can summarize values in a `data.frame` with functions `table`, `tapply` or `aggregate`. Have a look at the help for these functions if you are not familiar with them.

```
> # tabulate single variable
> table(d$sex)
```

```

F M
2 2

> # contingency table
> table(d[ c('name', 'sex')])

      sex
name   F M
Ana    1 0
Liu    0 1
Rob    0 1
Veronica 1 0

> # mean score by sex
> tapply(d$score, d$sex, mean)

      F      M
14.10 11.25

> aggregate(d[, 'score', drop=F], d[, 'sex', drop=FALSE], mean)

  sex score
1  F 14.10
2  M 11.25

```

## 2.4 Lists

Lists are the most flexible container to store data (a `data.frame` is in fact a special type of list). Each element of a list can contain any type of R object, e.g. a vector, matrix, `data.frame`, another list, or more complex data types such as the spatial data types described in the next chapters.

```

> e <- list(d , m3, 'abc')
> e

[[1]]
  ID   name sex score
1  1    Ana  F  10.2
2  2    Rob  M   9.0
3  3    Liu  M  13.5
4  4 Veronica F  18.0

[[2]]
  ID V2 X  Y  Z
v1  1  5 1  2  3
v2  2  6 5  6  7
    3  7 5 12 21

[[3]]
[1] "abc"

```

Indexing lists can be a bit confusing as you can both refer to the elements of the list, or the elements of the data (perhaps a matrix) in one of the list elements (note the difference that double brackets make; or ignore this if it is over your head).

```
> e[2][1]

[[1]]
      ID V2 X  Y  Z
v1    1  5  1  2  3
v2    2  6  5  6  7
      3  7  5 12 21

> e[[2]][1]

[1] 1
```

To iterate over a list, we can use `lapply` or `sapply`. The difference is that `lapply` always returns a list while `sapply` tries to simplify the result to a vector or matrix.

```
> lapply(e, NROW)

[[1]]
[1] 4

[[2]]
[1] 3

[[3]]
[1] 1

> sapply(e, length)

[1] 4 15 1
```

Note that the length of a matrix is defined as the number of cells, while the length of a data.frame is defined as the number of variables!

## 2.5 functions

We now have used many functions that come with R. But it is often important to write your own functions. Functions allow you to more clearly describe a particular task and to reuse code. Rather than repeating the same steps several times (e.g. for each of 200 species you are analysing), you write a function that gets called several times. This should lead to faster development of scripts and to fewer mistakes. Writing your own functions is easy. For example, the below function squares the sum of two numbers.

```
> sumsquare <- function(a, b) {
+   d <- a + b
+   dd <- d * d
+   return(dd)
+ }
```

We can now use the function:

```
> sumsquare(1,2)

[1] 9

> x <- 1:3
> y <- 3:5
> sumsquare(x,y)

[1] 16 36 64
```

Here is a function to compute the number of unique values in a vector:

```
> nun <- function(x)length(unique(x))
> data <- c('a', 'b', 'a', 'c', 'b')
> nun(data)

[1] 3
```

### 3 Spatial data

Spatial data always refers to a location, and it normally also includes information about these locations. For example, a spatial data set may describe the borders of the countries of the world, and also store the names of the countries and the size of their population in 2010. The location data can be referred to as the 'geometry' and the associated information can be referred to as the 'attributes' (or simply 'variables').

Vector data is commonly used to represent entities with well defined boundaries, like country boundaries, roads, or the location of weather stations. The geometry of such data structures is represented by one or multiple coordinate pairs. The simplest type is points. Points each have one coordinate pair, and 0 to n associated attributes. For example, a point might represent a place where a rat was trapped, and the attributes could include the date it was captured, the person who captured it, the species size and sex, and information about the habitat. Several points can be combined into a multi-point object, i.e. a combination of multiple spatial features with a single attribute record (this is common in point pattern analysis).

Lines and polygons are more complex. Note that the literature on spatial data analysis, and on geographic information systems, the terms 'line' and 'polygon' are used rather loosely. In this context, a 'line' in fact refers to a set of 1 or

more polylines (connected series of line segments); a polygon refers to a set of closed polylines (the last coordinates coincide with the first ones). For example the US state of Hawaii consists of several islands. Each can be represented by a single polygon, and together then can be represent a single (multi-) polygon.

Raster data is commonly used to represent continuous variables (fields). A raster divides the world into a grid of equally sized rectangles (referred to as cells or, in the context of air photos and satellite images, pixels) that all have values (or missing values) for the variables. In contrast to vector data, in raster data the geometry is not explicitly stored as coordinates. It is implicitly set by knowing the spatial extent and number of rows and columns (defining the spatial resolution (size of the raster cells)).

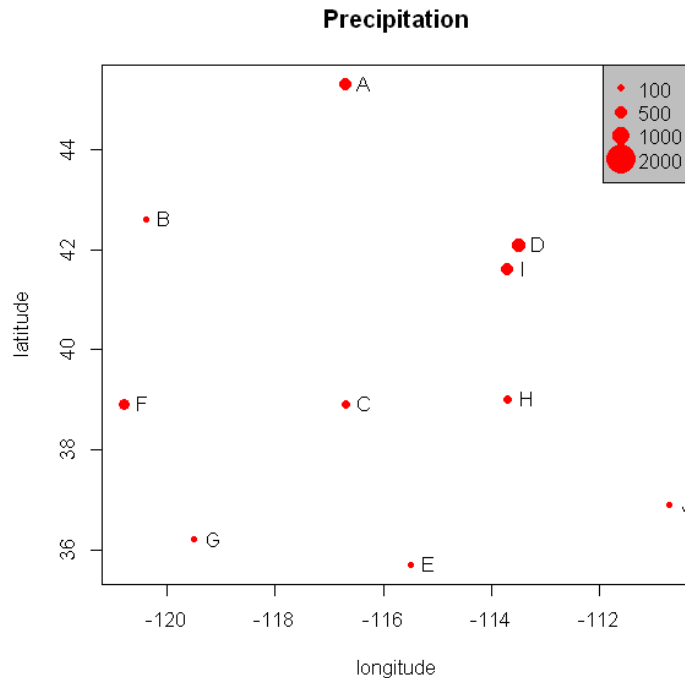
Other data types are networks (graphs) and triangulated irregular networks (TINs); these are not discussed here.

## 4 Simple representation of spatial data

We have already discussed that the basic data types in R are numbers, characters, logical (TRUE or FALSE) and factor values. Values of a single type can be combined in vectors and matrices, and variables of multiple types can be combined into a `data.frame`. This allows us to represent some basic types of spatial data. Let's say we have the location (represented by longitude and latitude) of ten weather stations (named A to J) and their annual precipitation.

In the example below we use some fake data to make a very simple map. (Note that a map is a plot of spatial data that also has labels and other graphical objects such as a scale bar or legend; the spatial data itself should not be referred to as a map).

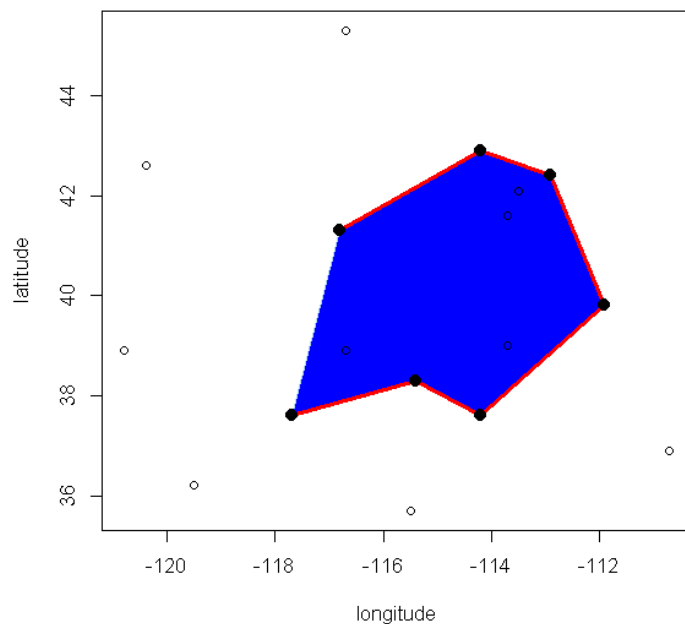
```
> name <- toupper(letters[1:10])
> longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
+               -120.8, -119.5, -113.7, -113.7, -110.7)
> latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
+              36.2, 39, 41.6, 36.9)
> precip <- (runif(length(latitude))*10)^3
> stations <- cbind(longitude, latitude)
> # plot locations, with size (cex) proportional to precip
> plot(stations, cex=1+precip/500, pch=20, col='red', main='Precipitation')
> text(stations, name, pos=4)
> # add a legend
> breaks <- c(100, 500, 1000, 2000)
> legend("topright", legend=breaks, pch=20, pt.cex=1+breaks/500, col='red', bg='gray')
```



The map shows the location of the weather stations and the size of the dots is indicative of the amount of precipitation. Note that the data are represented by "longitude, latitude", in that order, do not use "latitude, longitude" because on most maps latitude (North/South) is used for the vertical axis and longitude (East/West) for the horizontal axis. This is important to keep in mind, as it is a very common source of mistakes as most people organize their data as "latitude, longitude".

We can add multiple sets of points to the plot, and even draw lines and polygons:

```
> lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
> lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
> x <- cbind(lon, lat)
> plot(stations)
> polygon(x, col='blue', border='light blue')
> lines(x, lwd=3, col='red')
> points(x, cex=2, pch=20)
> points(stations)
```



The above illustrates how numeric vectors representing locations can be used to draw simple maps. It also shows how points can (and typically are) represented by pairs of numbers, and a line and a polygons by a number of these points. An additional criterion for polygons is that they are "closed", i.e. the first point coincides with the last point, but 'plot' takes care of that in this case.

There are cases where a simple approach like this may suffice and you may come across this in older R code or packages. Particularly when only dealing with point data such an approach may work. For example, a spatial data set representing points and attributes could be made by combining geometry and attributes in a single `data.frame`.

```
> wst <- data.frame(longitude, latitude, name, precip)
> wst
```

	longitude	latitude	name	precip
1	-116.7	45.3	A	565.67148629
2	-120.4	42.6	B	47.21965393
3	-116.7	38.9	C	182.81691351
4	-113.5	42.1	D	659.82650979
5	-115.5	35.7	E	16.01866912
6	-120.8	38.9	F	387.21148243
7	-119.5	36.2	G	0.03973569
8	-113.7	39.0	H	251.56286796

9	-113.7	41.6	I	541.24512659
10	-110.7	36.9	J	21.52768991

However, `wst` is a `data.frame` and R does not automatically understand the special meaning of the first two columns, or to what coordinate reference system it refers (longitude/latitude, or perhaps UTM zone 17S, or ....?). Moreover, it is non-trivial to do some basic spatial operations. For example, the blue polygon drawn might represent a state, and a next question might be which of the 10 stations fall within that polygon. And how about any other operation on spatial data, including reading from and writing data to files? To facilitate such operation a number of R packages have been developed that are discussed in the next chapter.

## 5 Spatial packages

### 5.1 packages

The success of R is to a large extent due to the ease of contributing new and often specialized functionality through plug-ins called "packages" or "libraries". A package is a collection of functions and/or some other R objects such as data sets, and help files. To create a package you do not need to be involved in the development of the core R software; and there is an easily accessible central repository "CRAN" from where these packages can be downloaded and installed. This means that there is relatively little need for oversight, a contributed package cannot break the code in already existing packages, and hence that the barrier to contributing code is very low. Thanks to this simple mechanism there are hundreds of active developers contributing to R. The downside is that R can be somewhat chaotic. For example, the same functionality may be implemented in slightly different ways (or not) in functions in different packages; packages doing similar things may use very different data structures and can thus be hard to use together; if newer packages improve upon older packages, the older packages still linger on, and the unsuspecting user may not be aware that and waste time learning to use obsolete software.

Despite such shortcomings, R has become the leading platform for data analysis. While R is particularly strong for particular domains of data analysis (e.g. molecular biology, ecology, spatial, finance, machine learning, time series), it is perhaps more important to note that it is **the** tool for interdisciplinary and creative data analysis, as one can integrate analytical methods and data types from virtually any domain; and it is very easy to build on existing functionality to create your own analytical methods.

To use a package, it must be downloaded to your computer. You need to do that each time you update the R software (which you should do at least once a year). Depending on how you use R you may be able to install packages via a menu, but it is easiest to use the `install.packages` function. For example to install packages `raster` and `rgdal` you do (note the quotes):

```
> install.packages(c('raster', 'rgdal'))
```



You should only install a package every couple of months or so (to get updates, e.g. by running `update.packages()`), but on a day-to-day basis you should not re-install a package every time you use it. However, you must use the `library` function to **load** a package to make its functions and data available for use in a R session. For example, to use the raster package in a script, it needs to have this line (no quotes required) before the functions can be used:

```
> library(raster)
```

## 5.2 Package sp

Package `sp` is the central package supporting spatial data analysis in R. It does not provide very many functions to modify or analyze spatial data. Rather it defines a set of "classes" to represent spatial data (See Chambers (2009) for a detailed discussion of the use of classes in R). A class defines a particular data structure such that functions (also known as 'methods') can be written for objects of that class. A `data.frame` is an example of a class. Any particular `data.frame` is an 'instantiation' of the class, or an 'object'. Package `sp` introduces a number of classes with names that start with `Spatial`. The basic types are the `SpatialPoints`, `SpatialLines`, `SpatialPolygons`, `SpatialGrid` (raster) and `SpatialPixels` (sparse raster). These classes only represent geometries. To also store attributes, classes are available with these names plus `DataFrame`, e.g., `SpatialPolygonsDataFrame`.

In most cases you will not create objects of these data types from scratch with R code. Rather you will read them from a file or database, for example from a shapefile (and it is important to recognise the difference between such a file and the object created in R to represent the data in the file). Shapefiles can be read with function `readOGR` in the `rgdal` package and with the (easier to use) function `textttshapefile` in the `raster` package.

Let's make a `Spatial` object from scratch anyway, using the same data as in the previous chapter. A `SpatialPoints` and a `SpatialPointsDataFrame` object:

```
> longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
+               -120.8, -119.5, -113.7, -113.7, -110.7)
> latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
+               36.2, 39, 41.6, 36.9)
> lonlat <- cbind(longitude, latitude)
> library(sp)
> st1 <- SpatialPoints(lonlat)
> df <- data.frame(ID=1:nrow(lonlat), precip=(latitude-30)^3)
> st2 <- SpatialPointsDataFrame(st1, data=df)
> class(st1)

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

```
> class(st2)

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

> st2
```

```
      coordinates ID    precip
1  (-116.7, 45.3)  1 3581.577
2  (-120.4, 42.6)  2 2000.376
3  (-116.7, 38.9)  3  704.969
4  (-113.5, 42.1)  4 1771.561
5  (-115.5, 35.7)  5  185.193
6  (-120.8, 38.9)  6  704.969
7  (-119.5, 36.2)  7  238.328
8   (-113.7, 39)   8  729.000
9  (-113.7, 41.6)  9 1560.896
10 (-110.7, 36.9) 10  328.509
```

Making a `SpatialPoints` object was easy. Making a `SpatialPolygons` object is more complicated:

```
> lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
> lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
> lonlat <- cbind(lon, lat)
> # close the ring of the polygon
> lonlat <- rbind(lonlat, lonlat[1,])
> polys <- SpatialPolygons( list( Polygons(list(Polygon(lonlat)), 1)))
> str(polys)
```

```
Formal class 'SpatialPolygons' [package "sp"] with 4 slots
 ..@ polygons      :List of 1
 .. ..$ :Formal class 'Polygons' [package "sp"] with 5 slots
 .. .. ..@ Polygons :List of 1
 .. .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
 .. .. .. .. ..@ labpt  : num [1:2] -114.7 40.1
 .. .. .. .. ..@ area   : num 19.7
 .. .. .. .. ..@ hole   : logi FALSE
 .. .. .. .. ..@ ringDir: int 1
 .. .. .. .. ..@ coords : num [1:8, 1:2] -117 -114 -113 -112 -114 ...
 .. .. .. .. ..- attr(*, "dimnames")=List of 2
 .. .. .. .. .. ..$ : NULL
 .. .. .. .. .. ..$ : chr [1:2] "lon" "lat"
 .. .. .. ..@ plotOrder: int 1
 .. .. .. ..@ labpt     : num [1:2] -114.7 40.1
 .. .. .. ..@ ID        : chr "1"
```

```

.. .. ..@ area      : num 19.7
..@ plotOrder  : int 1
..@ bbox      : num [1:2, 1:2] -117.7 37.6 -111.9 42.9
.. ..- attr(*, "dimnames")=List of 2
.. .. ..$ : chr [1:2] "x" "y"
.. .. ..$ : chr [1:2] "min" "max"
..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
.. .. ..@ projargs: chr NA

> class(pols)

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

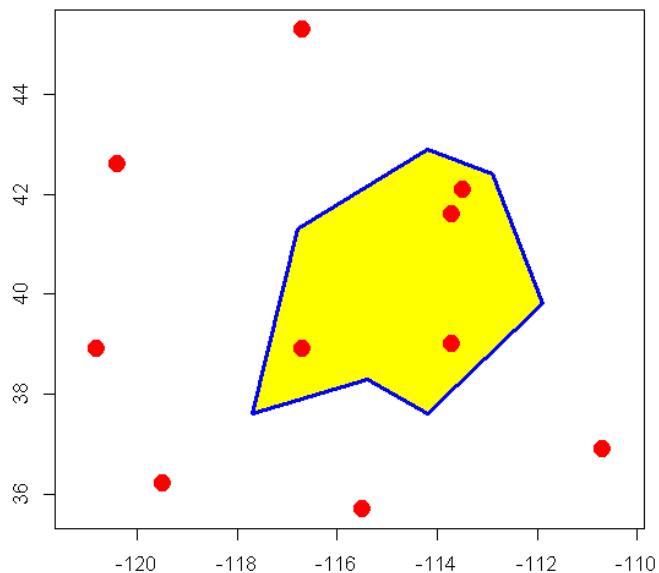
```

The structure of the SpatialPolygons class is somewhat complex as it needs to accommodate the possibility of multiple polygons, each consisting of multiple sub-polygons, some of which may be "holes". But, for most intents and purposes, you do not need to understand these structures. As these Spatial\* objects are known entities we can use generic R function plot to make a map:

```

> plot(st2, axes=TRUE)
> plot(pols, border='blue', col='yellow', lwd=3, add=TRUE)
> points(st2, col='red', pch=20, cex=3)

```



A crucially important element of a Spatial object is the coordinate reference system (CRS) of the data (sometimes referred to as the map projection). The CRS is described using the 'PROJ.4' notation system, which explains the name of the function to get or set the CRS: `proj4string`.

```
> proj4string(pols)
```

```
[1] NA
```

The CRS is undefined. If we know what it is, we can set it:

```
> proj4string(pols) <- CRS("+proj=longlat +datum=WGS84")
```

Now that it is defined with the correct CRS we can transform it to another CRS (here to the Robinson projection); using the `spTransform` function from the `rgdal` package. After the transformation, the units of the geometry are no longer in degrees, but in meters away from (longitude=0, latitude=0). The `bbox` function returns the "bounding box" (spatial extent) of the data.

```
> library(rgdal)
```

```
> polsrob <- spTransform(pols, CRS("+proj=robin +datum=WGS84"))
```

```
> proj4string(polsrob)
```

```
[1] "+proj=robin +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

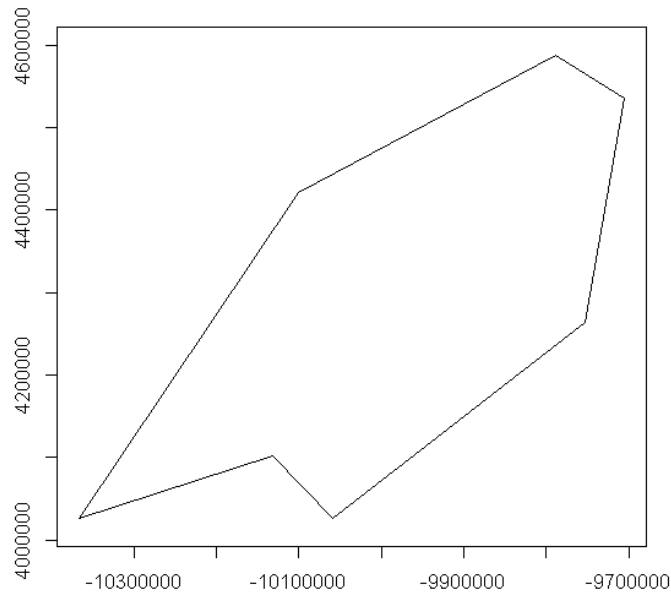
```
> bbox(pols)
```

```
      min      max
x -117.7 -111.9
y   37.6   42.9
```

```
> bbox(polsrob)
```

```
      min      max
x -10366659 -9704732
y   4026273  4587958
```

```
> plot(polsrob, axes=TRUE)
```



For much more information and examples see `vignette('intro_sp')`

### 5.3 Package raster

The raster package is built around a number of classes of which the **RasterLayer**, **RasterBrick**, and **RasterStack** classes are the most important. When discussing methods that can operate on all three of these objects, they are referred to as 'Raster\*' objects.

A **RasterLayer** object represents single-layer (variable) raster data. A **RasterLayer** object always stores a number of fundamental parameters that describe it. These include the number of columns and rows, the spatial extent, and the coordinate reference system. In addition, a **RasterLayer** can store information about the file in which the raster cell values are stored (if there is such a file). A **RasterLayer** can also hold the raster cell values in memory.

Multiple layers can be represented by a **RasterStack** and by a **RasterBrick**. These are very similar objects. The main difference is that a **RasterStack** is loose collection of **RasterLayer** objects that can refer to different files (but must all have the same extent and resolution), whereas a **RasterBrick** can only point to a single file.

Here I create a **RasterLayer** from scratch. But note that in most cases these objects are created from a file.

```
> library(raster)
```

```

> # create empty RasterLayer
> r <- raster(ncol=10, nrow=10, xmx=-80, xmn=-150, ymn=20, ymx=60)
> r

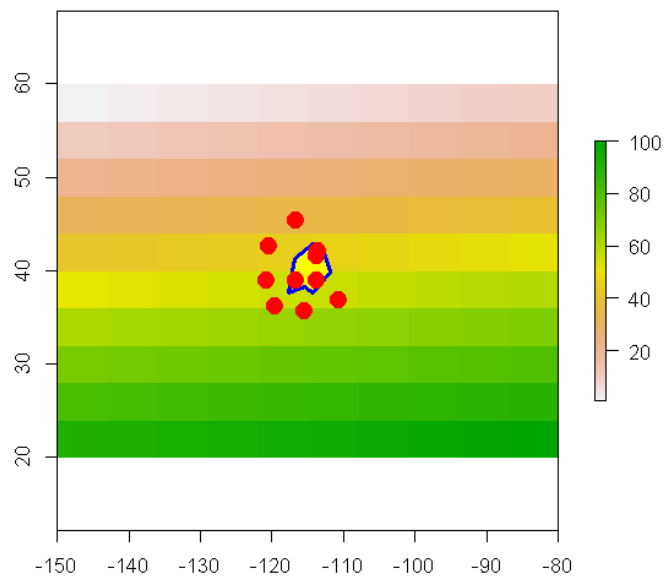
class      : RasterLayer
dimensions  : 10, 10, 100 (nrow, ncol, ncell)
resolution : 7, 4 (x, y)
extent      : -150, -80, 20, 60 (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0

> # assign values
> r[] <- 1:ncell(r)
> r

class      : RasterLayer
dimensions  : 10, 10, 100 (nrow, ncol, ncell)
resolution  : 7, 4 (x, y)
extent      : -150, -80, 20, 60 (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
data source : in memory
names       : layer
values      : 1, 100 (min, max)

> # plot
> plot(r)
> # add polygon and points
> plot(polys, border='blue', col='yellow', lwd=3, add=TRUE)
> points(st2, col='red', pch=20, cex=3)

```

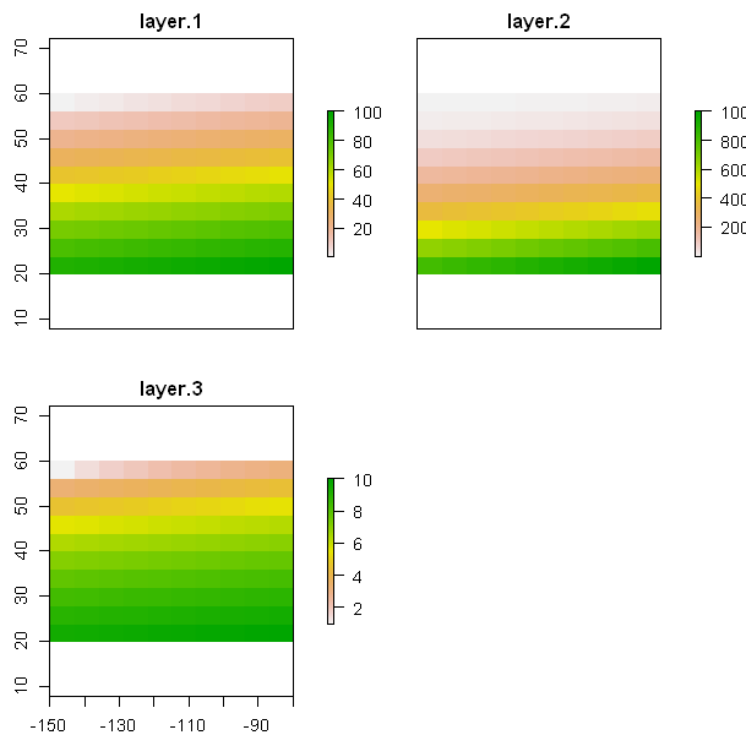


Make a RasterStack from multiple layers:

```
> r2 <- r * r
> r3 <- sqrt(r)
> s <- stack(r, r2, r3)
> s

class       : RasterStack
dimensions  : 10, 10, 100, 3  (nrow, ncol, ncell, nlayers)
resolution  : 7, 4  (x, y)
extent      : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
names       : layer.1, layer.2, layer.3
min values  :      1,      1,      1
max values  :     100, 10000,     10

> plot(s)
```



See `vignette('raster')` for more info about these objects and how they can be manipulated.

## 5.4 Other packages

Other important packages (to be discussed in more detail in the future):

**rgdal** – reading and writing spatial data (GIS data)

**rgeos** – geometric operations on vector (notably polygon) data, e.g. intersecting and merging of polygons

**dismo** – species distribution modeling

**vegan** – spatial ecology, biodiversity indices

**gdistance** – compute ecological distances (such as resistance distance or cost distance)

**spatstat** – point pattern analysis

**gstat** – geostatistics: kriging

**spdep** – inference with spatial data (detecting and correcting for spatial autocorrelation)

Most of these packages have vignettes that illustrate their use. For example, the **dismo** package has a vignette called 'sdm' that illustrates the basic steps in species distribution modeling. You can use Bivand et al, 2013 or Plant 2012 for a more general introduction to spatial statistics.

The next handouts will show how to use the spatial data classes described in R for particular types of data analysis and modeling.



## 6 References

- Adler, Joseph, 2009. R in a nutshell. A desktop quick reference. O'Reilly.
- Bivand, R.S., E.J. Pebesma and V. Gomez-Rubio, 2013. Applied spatial data analysis with R . 2nd Edition. Springer. 407p.
- Chambers, J.M., 2009. Software for data analysis: programming with R . Springer. 498p.
- Plant, R., 2012. Spatial data analysis in ecology and agriculture using R . CRC Press 648p.
- Zuur, A.F., E.N. Ieno and E. Meesters, 2009. A Beginner's Guide to R . Springer.