2

The R Programming Environment

2.1 Introduction

2.1.1 Introduction to R

The analyses discussed in this book are, with a few minor exceptions, carried out in the R programming environment (R Core Development Team, 2017). R is an open source program, which means, among other things, that it is freely distributed and that in addition to the executable files, the source code is made available to all. A full description of open source software is given on the website of the Open Source Initiative at http://www.opensource.org/docs/osd. R, like some other well-known statistical software packages, is primarily *command driven*, which means that analyses in R must generally be carried out by typing a series of statements (we will throughout this book use the terms *command* and *statement* synonymously). R runs on Macintosh, Windows, and Linux platforms. This book was written using R version 3.4.3 on a Windows platform. The book is not intended to be a complete introduction to R, since many such texts are already available. Some of these are listed in Section 2.8. No prior experience with R is assumed, and indeed one of the goals of this book is to introduce R to those in the applied ecology research community who are not familiar with it. The reader who is completely unfamiliar with R is encouraged to read one or more of the introductory R books listed in Section 2.8 along with this book. Section 2.1.2 provides information on how to download R and get set up to use it.

Unlike some programming environments and statistical software packages, R is designed to be used interactively. With most programming languages and with many command-driven statistical packages, you run in *batch mode*. This means that you enter a series of statements, execute them all at once, and receive output. Although the final product of an R session is sometimes run like this, in many cases you will use R by typing one or a few lines, observing the response, and then typing more lines based on this response. This is the *interactive mode*. Moreover, R will frequently execute commands and give you no response at all. Instead, you will have to type a statement or statements requesting certain specific results from the computations.

Aside from being freely available and having freely available source code, another major feature of an open source program like R is that it is supported by a large user community. When you install R, you are installing the base package, which contains some fundamental software. Many (several thousand!) other contributed packages have been written and are maintained by people in the R community. The fact that R is open source means that anyone can write and contribute a package or modify an existing package for his or her own use. In this book, for example, we will make considerable use of the contributed

packages `sf` (Pebesma 2017)[1] and `sp` (Bivand et al., 2013b). Will see how to install contributed packages in the next section.

Finally, one feature of R, well known within the R community, is that, given any task that you want to perform, there are almost always at least a half dozen different ways to do it. There is generally no effort expended in this book to teach you more than one of them, and that one is usually the one I consider simplest and most effective.

2.1.2 Setting Yourself Up to Use This Book

Although it is possible for you to type statements into R based on the printed statements in this book, that is not the intent, and moreover, most of the code would not work because this book does not include every statement used in the analysis. Instead, the R files, as well as the data, should be downloaded from the book's companion website, http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm (don't forget the 2, or else you will get the material from the first edition, which is different). One of the great things about a programming environment like R, as opposed to a menu-driven application, is that code can be reused after slight modification. I hope that the code in this book will serve as a template that enables you to write programs to analyze your own data.

To set up the files on the disk, you need to first set up the directory structure. In the book, we will use the term "folder" and "directory" synonymously. In the Windows world they are called folders, and in the R world they are called directories. Suppose you want to store all of your materials associated with R in a folder on the *C:* drive called *rdata*. First open a window on the *C:* drive and create a new folder called *rdata*. Suppose you want to store the material associated with this book in a subfolder called *SDA2* (for "spatial data analysis, second edition"). In Windows, open the *rdata* folder and create a new folder called *SDA2*. Now you can download the data from the companion website. When you visit this site, you will see a link for data and one for R code. Clicking on each of these links will enable you to download a zip file. These zip files may be uncompressed into two folders. One contains the four data sets, each in a separate subfolder, together with auxiliary and created data. The other contains the R code. The R code is arranged in subfolders by chapter and within each folder by section within the chapter. If you wish, you may also download two other folders that contain data created during the course of working through the book as well as auxiliary data from other sources on the web. You are encouraged, however, to generate yourself as much of these data as you can.

In order to do serious R programming, you will need an editor. There are a number of editors available for R that provide improved functionality over the built-in R editor. Two very good ones that I use are TINN-R (TINN stands for This Is Not Notepad) (https://sourceforge.net/projects/tinn-r/) and RStudio (http://www.rstudio.com/). I use them both, generally TINN-R for shorter jobs and RStudio for longer projects. You can run them both on your computer at the same time, running two separate implementations of R. This is often useful for comparing, modifying, and debugging code. Also, I have found that TINN-R uses less memory than RStudio. This doesn't usually matter, but sometimes it does. Nevertheless, for purposes of consistency we will use RStudio throughout the book.

As you work though this book, you should run the code and see the results. In some cases, particularly those that involve sensitive numerical calculations where round-off

[1] We will make many multiple references to contributed packages. Therefore, to avoid overwhelming the reader with repeated citations, I will generally follow the practice of citing R packages only on their first use in the book.

errors may differ from computer to computer, or those involving sequences of pseudorandom (i.e., random appearing—these are discussed further in Section 2.3.2) numbers, your results may be somewhat different from mine.

2.2 R Basics

If you have RStudio ready to go, open it and let's begin. You don't need to open R because it will open within RStudio. The RStudio window should contain four panes. If it only contains three, click *File->New File->R Script* and the fourth will appear. Reading clockwise from the upper left, they should say something like "Unititled1," "Environment," "Files" or "Packages," and "Console." These are called, respectively, the *Script*, *Workspace*, *Plot/Help*, and *Console*.

Upon starting RStudio, in the Console window you will see some information (which is worth reading the first time you use R) followed by a prompt that, by default, looks like this:

```
>
```

The prompt signals the user to enter statements that tell R what to do. Like most programs, you can exit RStudio by selecting *File -> Exit*. You can also exit from RStudio by typing `q()` or `quit()` at the command prompt in the Console. The parentheses indicate that `q()` is a function. It is a general property of R that once a function is identified unambiguously, all further identifying letters are superfluous, and `quit()` is the only base function that begins with the letter *q*. To stop R from executing a statement, hit the *Esc* key (see Exercise 2.1) or the Stop sign in the upper right corner of the Console. Another very useful function for the beginner is `help.start()`. If you type this at the command prompt (or in the script window as discussed below), you will obtain in the RStudio Plot/Help window a list of information useful for getting started in R. Don't forget to include the parentheses.

R programs are a series of *statements*, each of which is either an *assignment* or an *evaluation*. Here is a sequence of statements.

```
> # Assign the value of 3 to w
> w <- 3
> w # Display the value of w
[1] 3
```

Like all listings in the book, this shows the actual execution of the statements in the R environment. We will display code listings using courier font, with prompts and input in normal font and R output in bold. These code listings are copied directly from the R screen. When typing statements into RStudio, you should of course not type the prompt character >. Also, to repeat, when running the R programs in this book you should use the code downloaded from the companion website http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm rather than copying the statements from the text, because not all of the statements are included in the text, only those necessary to present the concept being discussed. In fact, you should open the file 2.2.r right now so you can see the actual code. In RStudio, click *File->Open* and navigate to this file. It will appear in the Script window.

Note that in RStudio the first line and part of the third line are displayed in green. The first line in the code is a *comment*, which is any text preceded by a # symbol. Everything

in the line after the comment symbol # is ignored by R. We will emphasize comments by italicizing them. Comments are very useful in making long code sequences understandable. They can be placed anywhere in a line, so that for example in line 3 above, a short comment is placed after the statement that causes the display of the value of `w`. Line 2 of the code above is an *assignment* statement, which creates `w` and assigns to it the value 3. All assignment statements in this book will have the form `A <- B` where `A` is created by the assignment and `B` is the value assigned to `A`. The symbol in between them is a `<` followed by a – to create something that looks like an arrow. There are other ways to make an assignment, but to avoid confusion we will not even mention them. Line 3 is an *evaluation* statement. This implicitly and automatically invokes the R function `print()`, which prints the result of the evaluation. In this book we will follow the practice of indicating a function in the text body by placing parentheses after it.

Now click and drag to select the entire set of three lines in the Script window (they should turn blue) and then click on the Run icon at the top of the Script window (it looks like a box with a green arrow and should be about the fifth icon from the left). The three lines should show up in the Console (the window below the Script window), together with the fourth line showing the value of `w`. The Console contains the actual running implementation of R. The symbol **[1]** in line 4 is explained below. Also, the Workspace window (upper left right corner) should show that `w` has been assigned the value 3. Finally, the bottom line of the Console contains another command prompt, showing that it is ready to accept the next commands.

R is an *object oriented* language. This means that everything in R is an object that belongs to some class. Let's see the class to which the object `w` belongs. Repeat the process of the previous paragraph to execute the following statement.

```
> class(w) # Display the object class of w
[1] "numeric"
```

This is an evaluation statement that generates as a response the class to which the object `w` belongs, which is `numeric`. Now execute the statement `?class` to see in the Plot/Help window (lower right corner) an explanation of the `class()` function. Much of R is actually written in R. To see the source code of the function `class()`, execute `class` (without the parentheses). The source code of `class()` is not especially revealing. To see some actual source code, execute `sd` to generate the code of `sd()`, a function that computes the standard deviation. Even if you are new to R, you should recognize some things that look familiar in this function.

When executing an assignment statement, R must determine the class of the object to which the assignment is made. In the example above, R recognizes that 3 is a number and accordingly `w` is created as a member of the class `numeric`. Another commonly used class is `character`, which is indicated by single or double quotes around the symbol.

```
> # Assign z a character value
> z <- "a"
> z
[1] "a"
> class(z)
[1] "character"
```

The quotes are necessary. Although R recognizes the symbol 3 as a number, it does not recognize the symbol a without the quotes as a character. Let's try it.

```
> z <- a
Error: object 'a' not found
```

Since R is an object-oriented language, everything in R is an object that is a member of some class.

```
> class(class)
[1] "function"
```

As a programming language, R has many features that are unusual, if not unique. For this reason, experienced programmers who have worked with other languages such as C++ sometimes have problems adjusting. However, R also has much in common with these and other languages. Like many programming languages, R operates on *arrays*. These are data structures that have one or more indices. In this book, arrays will generally be *vectors* (one dimensional arrays) or *matrices* (two dimensional arrays, see Appendix A.1). We will, however, have occasion to use a three-dimensional array, which can be visualized as a stack of matrices, so that it looks like a Rubik's cube. Some programming languages, such as C++ and Fortran, require that the size of an array always be declared before that array has values assigned to it. R does not generally require this.

One way of creating an array is to use the function `c()` (for *concatenate*).

```
> # Example of the concatenate function
> z <- c(1, 7, 6.2, 4.5, -27, 1.5e2, 7251360203, w, 2*w, w^2, -27/w)
> z
 [1]     1.0           7.0     6.2     4.5    -27.0
 [6]   150.0  7251360203.0     3.0     6.0     9.0
[11]    -9.0
```

The array `z` has a total of 11 elements. The sixth is expressed in the assignment statement in exponential notation, and the last four involve the already defined object `w`, whose value is 3. When an array is displayed in R, each line begins with the index of the first element displayed in that line, so that for example the sixth element of `z` is 150. The object `w` is itself an array with one element, which is the reason that the line 4 in the code sequence at the start of this section begins with **[1]**.

If no specific indices of an array are indicated, R acts on the entire array. We can isolate one element, or a group of elements, of an array by identifying their indices using square brackets.

```
> z[6]
[1] 150
> z[c(6,8)]
[1] 150   3
```

In the second statement the concatenate function `c()` is used inside the brackets to identify two particular indices. Specifying the negative of an index removes that element.

```
> # Remove the 6th element of z
> z[-6]
 [1]          1.0   7.0   6.2   4.5   -27.0
 [6] 7251360203.0   3.0   6.0   9.0    -9.0
> z[-(6:11)]
[1] 1.0   7.0   6.2   4.5   -27.0
```

In R the expression `m:n` is interpreted as the sequence beginning at *m* and incremented by plus or minus 1 until *n* is reached or the next number would pass *n*.

If the class of the array cannot be determined from the right-hand side of an assignment statement, then the array must be constructed explicitly.

```
> # This causes an error
> v[1] <- 4
Error in v[1] <- 4: object 'v' not found
> # This works
> v <- numeric()
> v[1] <- 4
> v[2] <- 6.2
> v
[1] 4.0 6.2
```

Now we will assign the value of `v` to `z` and display the value of `z` in one statement.

```
> # Assign v to z and display the result
> print(z <- v)
[1] 4.0 6.2
```

The statement `z <- v` executes normally because the class of `z` can be determined: it is the same as the class of `v`. R allows us to display the result of the assignment statement without having to type the object on a subsequent line by including the assignment statement itself as an argument in the function `print()`.

Here are examples of some R vector and matrix operations. First we create and display a vector `w` with components 1 through 30.

```
> print(w <- 1:30)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
[21] 21 22 23 24 25 26 27 28 29 30
```

Similarly, we can execute

```
> print(z <- 30:1)
 [1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9
[23]  8  7  6  5  4  3  2  1
```

and get a decreasing sequence. Other sequences can be generated by the function `seq()`. Thus, we could have executed `w <- seq(1,30)` and gotten the same result as that above. To learn more about the function `seq()`, execute `?seq`. The resulting Help screen shows us several things. One is that `seq()` is one of a group of related functions. A second is that `seq()` has several arguments. Each of these arguments has a name and some have default values. Actually, in the case of `seq()`, *all* of the arguments have default values, but this is not always true. When an argument has a default value, if you do not specify a different value to override the default, the function will execute with the assigned default value.

Thus, if you just type `seq()`, with no arguments assigned, you get the number 1, since this is the default of the `from` and the `to` arguments. If they are not specified, arguments are assigned in the order that they appear in the help file, so that if you type `seq(1,30,2)` you will get a sequence from 1 to 30 by 2. This is the same as specifying `seq(from = 1, to = 30, by = 2)`. If you specify the name of an argument, this overrides the order in the Help screen, so that `seq(to = 30, from = 1)` gives the same result as `seq(1,30)`.

Now let's return to arrays. Indices in arrays of more than one dimension are separated by a comma. The first index of a matrix (an array of two dimensions) references the row, and the second index references the column. Appendix A.1 contains a discussion of matrix algebra. The reader unfamiliar with this subject is encouraged to consult the appendix as needed. We now create the matrix *a* and assign values.

```
> print(a <- matrix(data = 1:9, nrow = 3))
     [,1]  [,2]  [,3]
[1,]    1     4     7
[2,]    2     5     8
[3,]    3     6     9
> a[1,]
[1] 1 4 7
> 2 * a[1,]
[1] 2 8 14
```

Note that by default R assigns values to a matrix by columns. The statement `a[1,]` causes an evaluation of the first row of the matrix `a`; since no column is specified, all are evaluated. The statement `2 * a[1,]` demonstrates the application of an operation (multiplication by 2) to a set of values of an array. If no specific index is given, then an R operation is applied to all indices. The object created by the operation `2 * a[1,]` is printed but not saved. To retain it, it is necessary to use an assignment statement, for example `b <- 2 * a[1,]`. If you are typing statements like these in which there is a lot of repetition from one statement to the next, a very convenient feature of R is that if you press the *up* arrow while in the Console, R will repeat the last statement executed. Try it! As was mentioned in Section 2.1, R runs in the interactive mode. Running lines of code from the RStudio script window is much more like the batch mode, and when you are programming you might often find it convenient to work directly in the Console.

One useful function for creating and augmenting matrices is `cbind()` (short for *column bind*).

```
> w <- c(1,3,6)
> z <- (1:3)
> cbind(w, z)
     w z
[1,] 1 1
[2,] 3 2
[3,] 6 3
```

The names `w` and `z` are preserved in the matrix output. It is possible to assign names to both rows and columns of a matrix using the function `dimnames()`. R functions like `cbind()` that match pairs of vectors generally return a value even if the numbers of elements of the two vectors are not equal. The elements of the smaller vector are recycled to make up the difference. There is function `rbind()` that performs a similar function by rows.

Finally, let's try installing the contributed package `sf` mentioned in Section 2.1. If you are working in RStudio, simply go to the Plot/Help window and click the Packages tab.

Then click *Install*. You will be prompted for the packages to install. We will be needing the packages `sf`, `rgdal`, `raster`, and `maptools` very soon, so go ahead and install these. You should be able to see them in the Packages tab. However, this does not load the packages into your R session. To load, for example, the `sf` package you must type `library(sf)`. Many packages depend on the presence of other packages. Occasionally you may get an error message that reads "there is no package called 'XXX' " where "XXX" is the name of some package. This probably resulted from a problem in downloading, so all you have to do is go back and download this package. You only *install* a package once, but you have to *load* it using `library()` every time you use it. Also, you can check the dependencies of a package. Try the following.

```
> library(maptools)
> library(tools)
> package_dependencies("maptools")
$maptools
[1] "sp"      "foreign"    "methods"    "grid"     "lattice"     "stats"
[7] "utils"   "grDevices"
```

The package `maptools` depends on a lot of other packages. In particular, it depends on `sp`, which is the fundamental package for much of the spatial work in this book. Therefore, if you load `maptools`, you automatically also load, among others, `sp`.

This may be a good time to take a break. Recall that one way of exiting R and RStudio is to type and execute `q()`. This time, try typing `Q()`. It doesn't work. R is *case sensitive*. When you type `q()`, or exit in any other way, you get a message asking if you want to save the workspace image. If you say yes, then the next time you start R all of the objects to which you have assigned values (in this case `w`, `z`, and `a`) will retain these values, so that you don't have to recompute them the next time you run a program R. This can be very convenient, but it is a double-edged sword. I have found that if I retain my workspace, I soon forget what is in it and as a result I am much more likely to create a program with bugs. You can always clear your workspace in RStudio with the menu selection *Session -> Clear workspace*, and it is not a bad idea to do this whenever it is convenient.

2.3 Programming Concepts

2.3.1 Looping and Branching

In order to use R at the level of sophistication required to implement the material covered this book, it will be necessary to write computer code. Computer programs often contain two types of expression that we will use extensively, *iteration* statements and *conditional* statements. Iteration, which is also known as *looping*, is the repetition of a set of instructions, often modified by an index. As an example, we can create the same vectors `w` and `z` as in the previous section, but using a `for` loop in the same way as conventional programming languages like C++ and Fortran.

```
> w <- numeric(30)
> z <- numeric(30)
> for(i in 1:30){
+   w[i] <- i
```

```
+   z[i] <- 30 - i +1
+}
> w
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
[22] 22 23 24 25 26 27 28 29 30
```

The *index* `i` is explicitly incremented from 1 to 30 and the array elements indexed by `i` are assigned values individually, as opposed to the code in the previous section, in which the array values are all assigned in the same operation. This code also demonstrates *grouped expressions* in R. A grouped expression is an expression consisting of a collection of individual statements that are evaluated together. A grouped expression in R is delimited by brackets {}, so that no statement in the group is carried out until all statements have been entered. In the code above, line 3 ends in a left bracket. This indicates the start of the group. The + signs at the beginning of the next three lines indicates that R is withholding evaluation until the right bracket in line 6, which indicates the end of the expression. We will follow the convention of indenting grouped expressions and putting the closing brackets on separate lines. This makes it easier to distinguish the beginning and the end of each expression. RStudio and TINN-R do this automatically.

The second construct common to computer programs is the conditional statement, also known as *branching*. Conditional statements evaluate an expression as either true or false and execute a subsequent statement or statements based on the outcome of this evaluation (Hartling et al., 1983, p. 64). Here is an example in R that, although the bracket placement looks a little different from what one might expect, will otherwise be familiar to any traditional programmer.

```
> w <- 1:10
> for (i in 1:10){
+   {if(w[i] > 5)
+      w[i] <- 20
+   else
+      w[i] <- 0
+   }
+ }
> w
 [1] 0 0 0 0 0 20 20 20 20 20
```

The conditional `if–else` statement says that if the expression in brackets evaluates as `TRUE`, then the first statement or set of statements is executed; otherwise the second. The inner set of brackets is placed in front of the `if` term to ensure that the entire statement is executed correctly. This is the part that might look a little unusual to an experienced programmer, and it is not strictly necessary, but is good defensive programming. Because R operates in the interactive mode, if a line of code appears to be complete, R will execute it before going on to the next line. Therefore, R may think you are done typing an expression when you are not. In the present example, the expression evaluates correctly without the extra brackets, but I generally include them to be on the safe side.

As with looping, many branching operations in R can be carried out without the explicit use of an `if` statement. The code below carries out the same operation as that above.

```
> w <- 1:10
> w > 5
  [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
```

```
> w[w > 5]
[1]  6  7  8  9 10
> w[w > 5] <- 20
> w[w <= 5] <- 0
> w
[1]  0  0  0  0  0 20 20 20 20 20
```

Unlike many languages, R can evaluate a conditional expression contained in a subscript, so that the expression `w[w > 5]` in line 2 is interpreted as "all elements of the array *w* whose value is greater than 5." The expression `w > 5` returns a logical vector, each of whose elements evaluates to the truth or falsehood of the expression for that element of `w`. Again unlike many languages, R can correctly interpret operations involving mixtures of arrays and scalars. Thus, the assignment `w[w > 5] <- 20` assigns the value 20 to each element satisfying the conditional expression `w[i] > 5`. A word of caution is in order here. This code sequence does not work in this way for all values of the replacement quantities. See Exercise 2.5.

2.3.2 Functional Programming

Many things in R that do not look like functions really are. For example, the left square bracket in the expression `w[i]` is a function. To see an explanation, type `?"["` in the Console. In this context, the lesson of the last two code sequences of the previous section is that an operation involving looping or branching may be carried out more directly in R by using a function. This feature is not unique to those two operations. R supports *functional programming*, which permits the programmer to replace many instances of looping and branching with applications of a function or operation. This is hard on old-school programmers (like me) but frequently makes for much more concise and legible code.

Many programming operations require repetition and in other languages involve a `for` loop. In R, these operations often make use of variants of a function called `apply()`. These can be illustrated by application of one member of the `apply()` family, the function `tapply()` (short for *table apply*). We will illustrate functional programming by performing a fairly complex task: we generate a sequence of 20 random appearing numbers from a standard normal distribution, divide them into four sequential groups of five each, and then compute the mean of each group.

The first step is to generate and display a sequence of *pseudorandom numbers*, that is, a sequence of numbers that, although it is deterministic, has many statistical properties similar to those of a sequence of random numbers (Rubinstein, 1981, p. 38; Venables and Ripley, 2002, p. 111). The actual numbers generated are members of a sequence of integers, each of which determines the value of the next. The first number in the sequence is called the *seed*. Since the sequence is deterministic and since each number determines the next, setting the value of the seed establishes the entire sequence. Controlling this value, which is done by the function `set.seed()`, ensures that the same sequence of numbers is generated each time the program is executed. This is useful for debugging and comparison purposes. To control the first number, one includes a numeric argument in the function `set.seed()`, such as `set.seed(123)`. The value of the argument of `set.seed()` does not generally matter; I use 123 because it is easy to type. The sequence of pseudorandom integers is transformed by the generating function so that the resulting sequence of numbers approximates a sequence of random variables with a particular distribution. To generate an approximately normally distributed sequence with mean zero and variance 1, one uses the generating function `rnorm()` as in line 3 below. Functions `rpois()`, `runif()`, `rbinom()`, etc. are also available.

```
> # Generate a sequence of 20 pseudorandom numbers
> set.seed(123) # Set random seed
> print(w <- rnorm(20), digits = 2)
 [1] -0.56 -0.23 1.56 0.07  0.13 1.72 0.46 -1.27 -0.69 -0.45
[11]  1.22  0.36 0.40 0.11 -0.56 1.79 0.50 -1.97  0.70 -0.47
```

Sometimes `print()` gives you more precision than you want. You can limit the number of significant digits printed using the argument digits (this does not affect the computation, only the printing). In working through the code, remember that a complete description of a function can be obtained by typing that function preceded by a question mark (e.g., `?set.seed`) at the R statement prompt.

The next step is to create an index array that has a sequence of five ones, then five twos, then five threes, and then five fours. We do this initially in two steps, although ultimately we will squeeze it down to one. First, we use the function `rep()` to create a replicated sequence z of five subsequences running from 1 to 4.

```
> print(z <- rep(1:4, 5))
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

Next, we use the function `sort()` to sort these in increasing order creating the index for the four sequences.

```
> print(z <- sort(z))
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4
```

The statement `z <- sort(z)` indicates that we replace the old object `z` with a new one rather than creating a new object. We can combine these two steps into one by using the statement `z <- sort(rep(1:4, 5))`. It is more efficient to nest functions this way, but care must be taken not to overdo it or the code may become difficult to understand.

The R function `tapply(w, z, ftn,...)` applies the function whose name is `ftn` to values in array `w` as indexed by the values in array `z`. In our application, `w` is the array w of pseudorandom numbers, `z` is the array `z` of index values, and ftn is the function `mean()`. Thus the mean is computed over all elements of `w` whose index is the same as those of the of the elements of the vector `z` having the values 1, then over all `w` values whose index is the same as those of the vector `z` whose value is 2, and so forth.

```
> # Apply the mean function to w indexed by z
> tapply(w, z, mean)
          1           2          3          4
 0.19357026 -0.04431897 0.30790173 0.10934219
```

Note that the third argument is just the name of the function, without any parentheses. The actual operation takes only four lines, and is very clean and easy to understand. Here they are in sequence.

```
> set.seed(123)
> w <- rnorm(20)
> z <- sort(rep(1:4, 5))
> tapply(w, z, mean)
```

This could actually be compressed into just two lines:

```
> set.seed(123)
> tapply(rnorm(20), sort(rep(1:4, 5)), mean)
```

but this is an example of the risk of making the code less intelligible by compressing it. The earlier chapters of this book will avoid compressing code like this, but the later chapters will assume that the reader is sufficiently familiar with R syntax to put up with a bit of compression.

Code carrying out the same operation with explicit looping and branching would be much longer and more obscure. That is the upside of functional programming. The biggest downside of functional programming is that it makes use of a large number of separate functions, often housed in different contributed packages. This makes the learning curve longer and enforces continued use of R in order not to forget all the functions and their actions. With traditional languages one can stop programming for a while, take it up again, and remember with ease what a *for* loop or an *if-then-else* statement does. It is because of this downside that I decided to include an R "thesaurus" as Appendix C.

2.4 Handling Data in R

2.4.1 Data Structures in R

For our purposes, the fundamental object for storing data in R is the *data frame*. Venables et al. (2006, p. 27) state that "A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions." In describing the contents of a data frame we will use two standard terms from GIS theory: *data field* and *data record* (Lo and Yeung, 2007, p. 76). A data field is a specific attribute item such as percent clay content of a soil sample. When data are displayed in tabular form, the data fields are ordinarily the columns. A data record is the collection of the values of all the data fields for one occurrence of the data. When data are displayed in tabular form, the data records are ordinarily the rows.

Data frame properties may best be illustrated by two examples. The first is a simple data frame from Data Set 1 that contains the results of a set of surveys at 21 locations along the Sacramento River to identify the presence and estimate the number of individuals of a population of western yellow-billed cuckoos (Section 1.4.1). In the next section, we show how to read this data set into an R data frame called `data.Set1.obs`. Once this is done, one can use the function `str()` (for *structure*) to display the structure of the object.

```
> str(data.Set1.obs)
'data.frame' :    21 obs. of  5 variables:
 $ ID:         int  1 2 3 4 5 6 7 8 9 10...
 $ PresAbs   : int  1 0 0 1 1 0 0 0 0 0...
 $ Abund     : int  16 0 0 2 2 0 0 0 0 0...
 $ Easting   : num  577355 578239 578056 579635 581761...
 $ Northing  : num  4418523 4418423 4417806 4414752 4415052...
```

The components of the data frame and their first few values are displayed. These are: ID, the identification number of the location; `PresAbs`, 1 if one or more birds were observed at this location and 0 otherwise; `Abund`, the estimated total number of birds observed; `Easting`; the UTM Easting in Zone 10N of the observation site; and `Northing`, the UTM Northing.

Data frames can contain much more complex information. In the code below, the object `data.Set4.1` is a data frame containing the point sample data from Field 1 of Data Set 4, one of the two fields in the precision agriculture experiment (Section 1.4.4).

```
> class(data.Set4.1)
[1] "data.frame"
> names(data.Set4.1)
 [1] "ID"        "Row"       "Column"    "Easting"    "Northing"
 [6] "RowCol"    "Sand"      "Silt"      "Clay"       "SoilpH"
[11] "SoilTOC"   "SoilTN"    "SoilP"     "SoilK"      "CropDens"
[16] "Weeds"     "Disease"   "SPAD"      "GrainProt"  "LeafN"
[21] "FLN"       "EM38F425"  "EM38B425"  "EM38F520"   "EM38B520"
> data.Set4.1$EM38F425[20:30]
 [1] 73 NA 74 63 69 79 74 77 NA 75 79
> data.Set4.1[20:30,25]
 [1] 94 86 80 69 86 96 94 98 89 86 81
> data.Set4.1[25,25]
[1] 96
> data.Set4.1$RowCol[1:5]
[1] 1a 1b 1c 1d 1e
86 Levels: 10a 10b 10c 10d 10e 10f 11a 11b 11c 11d 11e 11f 12a... 9f
>
```

The call to `class()`, as usual, produces the class of the object. The call to the function `names()` produces the set of names of each data field in the data frame, displayed as an array. Once again, the numbers in square brackets at the beginning of each line correspond to the position in the array of the first element in the row, so that, for example, `Northing` is the fifth element and `Sand` is the seventh. Thus, if the data frame is considered as a matrix, the `Sand` data field forms the seventh column. The contents of a column may be accessed using the `$` operator, which identifies a named data field in a data frame, in the example above rows 20 through 30 of `EM38B425` are displayed. These same contents may also be accessed as in the next statement, in which the data frame is treated as a matrix. The symbol `NA`, which stands for "not available," refers to a data record with a missing or null value. The contents of the twenty-fifth row (i.e., twenty-fifth data record) of the twenty-fifth column (i.e., twenty-fifth data field) may also be accessed in matrix fashion as is done with the statement `data.Set4.1[25,25]`. A data frame can also hold non-numeric information such as the alphanumeric code for the combined row and column number of the sample points in the sample grid `data.Set4.1$RowCol[1:5]`.

The last line of R output in this code listing refers to *levels.* We can see where this comes from by applying the `class()` function to two data fields in `data.Set4.1`.

```
> class(data.Set4.1$GrainProt)
[1] "numeric"
> class(data.Set4.1$RowCol)
[1] "factor"
```

The values of the data field `RowCol` are alphanumeric. The function `read.csv()`, which was used to read the data and which is described in the next section, automatically assigns the class `factor` to any data field whose values are not purely numerical. Venables et al. (2006, p. 16) define a *factor* as "a vector object used to specify a discrete classification (grouping) of the components." The collection of all individual values is called the *levels*. Any allowable symbol, be it numerical, alphabetical, or alphanumeric, may be converted into a factor:

```
> print(w <- c("A", "A", "A", "B", "B", "B"))
[1] "A" "A" "A" "B" "B" "B"
> class(w)
[1] "character"
> print(z <- as.factor(w))
[1] A A A B B B
Levels: A B
> class(z)
[1] "factor"
```

Note that the quotation marks are absent from the elements of z; this is because these elements are not characters but factors. The use of the function `as.factor()` is an example of *coercion*, in which an object is converted from one class to another. In R, functions that begin with "as" generally involve coercion.

In addition to the data frame, a second type of data storage object that we shall have occasion to use is the *list*. Technically, a data frame is a particular type of list (Venables et al., 2006), but viewed from our perspective these two classes of object will have very little in common, so forget that you just read that. Again quoting Venables et al. (2006, p. 26), "A list is an ordered collection of objects consisting of objects known as its components." Lists can be created using *constructor functions*; here is an example:

```
> list.demo <- list(crop = "wheat", yield = "7900", fert.applied =
+   c("N", "P", "K"), fert.rate = c(150, 25, 15))
> list.demo[[1]]
[1] "wheat"
> list.demo$crop
[1] "wheat"
> list.demo[[4]]
[1] 150  25  15
> list.demo[[4]][1]
[1] 150
```

This shows the use of the simple constructor function `list()` to create a list. To access the *n*th component of a list one uses double square brackets (lines 3, 7, and 9). A list can contain character string components, vector components, and other types of components as well. The `$` operator can be used to identify named components just as it can with a data frame (line 5). If a component itself has elements, then these elements can themselves be singled out (line 9).

Finally, suppose we want to construct a data frame from scratch. The constructor function used for this purpose is, not surprisingly, `data.frame()`. We can use elements of an existing data frame, or create new values. Here is a simple example.

```
> data.ex <- data.frame(Char = data.Set4.1$RowCol[1:3],
+    Num = seq(1,5,2))
> data.ex
  Char Num
1   1a   1
2   1b   3
3   1c   5
```

2.4.2 Basic Data Input and Output

In addition to carrying out assignments and evaluating expressions, a computer program must be able to input and output data. A good deal of the data that we will be using in this book is spatial data with a very specialized format, and we will defer discussion of the specifics of spatial data input and output until later. However, we introduce at this point some of the basic concepts and expressions. This material is specific to Windows, but if you work in another environment, you should be able to translate it. One important concept is the *working directory*. This is the directory on the disk that you tell R to use as a default for input and output. Suppose, for example, that you store all of your R data for a project in a single directory. If you make this your working directory, then R will automatically go to it by default to read from or write to the disk. The function `setwd()` is used to set a particular directory (which must already exist) as the working directory. Directory names must be enclosed in quotes, and subdirectories must be indicated with a *double* backslash (or forward slash). For example, suppose you have set up the folders in Windows as described in Section 2.1.2. To establish the Windows directory *C:\rdata\SDA2\data* as the working directory you would enter the following statement.

```
> setwd("c:\\rdata\\SDA2\\data")
```

Don't forget the quotation marks! If you prefer to put your project directory in your *My Documents* folder, you can do that too. For example, on my computer the *My Documents* folder is a directory called *C:\documents and settings\replant\my documents.* To establish a subfolder of this folder as a working directory I would enter the following.

```
> setwd("c:documents and settings\\replant
+     \\my documents\\rdata\\SDA2\\data")
```

R is case sensitive, but Windows is not, so character strings passed directly to Windows, such as file and folder names, are not case sensitive. Thus, this statement would work equally well.

```
> setwd("c:documents and settings\\replant
+     \\my documents\\rdata\\sda2\\data")
```

Once a working directory has been set, R can access its subdirectories directly. This feature is convenient for our purposes in that it permits the input and output statements in this book to be independent of specific directory structure. In this book, we will not display the calls to `setwd()`, since my working directory will be different from yours.

All attribute data and point sampling data used in this book are stored in *comma separated variable* or csv files. These have the advantage that they are very simple and can be accessed and edited using both spreadsheet and word processing software. Suppose you want to input a csv-formatted file named *file1.csv* in the directory *C:\rdata\SDA2\data\newdata*, and that *C:\rdata\SDA2\data* has already been set as the working directory. You would type `read.csv("newdata\\file1.csv")` to read the file. Often the file will have a *header*, that is, the first line will contain the names of the data fields. When this is the case, the appropriate expression is `read.csv("data\\newdata\\file1.csv", header = TRUE)`.

The headers are input into R and are therefore are case sensitive.

Writing a data frame as a csv file is equally simple. Suppose we have a data frame called `my.data`, and we want to write this to the file *file2.csv* in the subdirectory *C:\rdata\SDA2\data\newdata*. Suppose again that *C:\rdata\SDA2\data* has already been set as the working directory. We would then type the statement `write.csv(my.data, "newdata\\file2.csv")`. Don't forget the double slashes and the quotation marks.

We will do all attribute data input and output using csv files because this keeps things simple. R is, however, capable of reading and writing many other file formats. The package `foreign` contains functions for some of these, and others may be found using the `help.search()` and `RSiteSearch()` functions described in Section 2.7.

2.4.3 Spatial Data Structures

There are a number of spatial data structures in contributed R packages, but among the most useful for our purposes are those in the `sf`, `raster`, and `sp` packages. We will present a highly simplified description of these data structures starting with `sf`, then moving on to `sp`, and finally to `raster`. The `sf` package includes functions that allow one to work with the vector model of spatial data (see Section 1.2.3), so that it can model point data, line data, and polygon data. A very complete description is given by Pebesma (2017). Although line data are very useful in some applications, we will have no call in this text to use them, so we will focus on point and polygon data. The term "sf" stands for "simple features," where the term "feature" is used here in the sense of a GIS representation of a spatial entity (Lo and Young, 2007, p. 74). Without going into too much detail, the geometric definition of a "simple feature" is something whose boundary, when drawn, can be represented by a set of points in a plane connected by straight lines. This is generally true of the vector data that are likely to arise from ecological applications.

Referring back to Section 2.4.1, an `sf` object combines a data frame to store the attribute data with a list to store the information for the geometric coordinates. The easiest way to introduce the `sf` concept is to construct an `sf` object that describes a set of point data. We will begin with the data frame `data.Set1.obs` that we just saw in Section 2.4.1. It contains data associated with yellow-billed cuckoo observations in Data Set 1. If you have set up your directory structure as described in Section 2.1.2, this file would be located in the file *C:\rdata\SDA2\data\set1\set1\obspts.csv*. If you have executed the statement `setwd("c:\\rdata\\SDA2\\data")`, then to load the data into R as you would execute the following statement, taken from the file *Appendix\B.1 Set1 Input.r*.

```
> data.Set1.obs <- read.csv("set1\\obspts.csv", header = TRUE)
```

Now that the data frame is loaded, let's look again at its class and structure.

```
> class(data.Set1.obs)
[1] "data.frame"
> str(data.Set1.obs)
'data.frame':    21 obs. of 5 variables:
 $ ID        : int  1 2 3 4 5 6 7 8 9 10 ...
 $ PresAbs   : int  1 0 0 1 1 0 0 0 0 0 ...
 $ Abund     : int  16 0 0 2 2 0 0 0 0 0 ...
 $ Easting   : num  577355 578239 578056 579635 581761 ...
 $ Northing  : num  4418523 4418423 4417806 4414752 4415052 ...
```

The last two lines are the data fields Easting and Northing, which contain the geometric data. To create an `sf` object, we first need to load the `sf` package. If you have not installed `sf` on your computer, you need to do so now. If you the missed instructions the first time, go back to the end of Section 2.2 to learn how to do this. Once the package is installed on your computer, you load it into R for that session by using the function `library()`.

```
> library(sf)
```

We will generally not explicitly include the call to the function `library()` in the text when we employ a function from a particular package, but you always need to load that package first. To create an `sf` object, we specify that these are the coordinates of the points.

```
> data.Set1.sf <- st_as_sf(data.Set1.obs, coords = c("Easting",
   "Northing"))
```

This uses the function `st_as _ sf()` to create the `sf` object `data.Set1.sf`.

Let's see what the object `data.Set1.sf` looks like

```
> class(data.Set1.sf)
[1] "sf"          "data.frame"

> str(data.Set1.sf)
Classes 'sf' and 'data.frame':      21 obs. of 4 variables:
 $ ID       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ PresAbs  : int 1 0 0 1 1 0 0 0 0 0 ...
 $ Abund    : int 16 0 0 2 2 0 0 0 0 0 ...
 $ geometry :sfc_POINT of length 21; first list element: Classes 'XY',
 'POINT', 'sfg' num [1:2] 577355 4418523
 - attr(*, "sf_column")= chr "geometry"
 - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA
   NA
 .. - attr(*, "names")= chr "ID" "PresAbs" "Abund"
```

The object `data.Set1.sf` is still a data frame, but it has a `geometry` object attached to it. This is represented in the `geometry` data field:

```
> str(data.Set1.sf$geometry)
sfc_POINT of length 21; first list element: Classes 'XY', 'POINT',
  'sfg' num [1:2] 577355 4418523
```

The last task remaining to create an R spatial object that we can work with is to assign a map projection. Projections are handled in the package `sf` in two ways. The first is by using the PROJ.4 Cartographic Projections library originally written by Gerald Evenden (http://trac.osgeo.org/proj/). To establish the projection or coordinate system, we use the function `st_crs()`.

```
> st_crs(data.Set1.sf) <- "+proj=utm +zone=10 +ellps=WGS84"
```

Here "crs" stands for *coordinate reference system*. The projection is specified in the argument, and therefore must be known to the investigator. In this case, it is UTM Zone 10 using the WGS84 ellipsoid. Experienced programmers may find this statement a bit odd in that

an assignment is made to a function rather than a function assigning a value to a variable. This is an example of a *replacement function*, which is described in The R Language Definition under Manuals in the R Project website, http://www.r-project.org/. If you don't see what the issue is, or if this is too much information, don't worry about it.

To see what happened, we can type the same function but without any assignment. In this case, `st_crs()` retrieves the coordinate system.

```
> st_crs(data.Set1.sf)
$epsg
[1] 32610

$proj4string
[1] "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs"

attr(,"class")
[1] "crs"
```

The data field `epsg` refers to the `epsg` code. This is a set of numeric codes created by the European Petroleum Survey Group. If you know your projection (for example UTM 10N), you can go to the site http://www.spatialreference.org/ and search for the corresponding EPSG code. The function st_crs accepts these codes as well as a PROJ4 character string, so you could use

```
> st_crs(data.Set1.sf) <- 32601
```

and get the same result.

To introduce the use of spatial features to deal with polygon data, we will create an `sf` object describing the boundary of Field 2 of Data Set 4. This is the field shown in Figures 1.2 and 1.3. (This is a simple set of polygon data because there is only one polygon; in Section 5.3.4 we create an `sf` object with multiple polygons.) The creation of a polygon describing the boundary of an area to be sampled requires a set of geographic coordinates describing that boundary. These can be obtained by walking around the site with a GPS or, possibly, by obtaining coordinates from a georeferenced remotely sensed image, either a privately obtained image or a publicly available one from a database such as Google Earth® or Terraserver®. If the boundary is not defined very precisely, one can use the function `locator()` (see Section 7.3) to obtain the coordinates quite quickly from a map or image. In the case of Field 2 the coordinates were originally obtained using a GPS and modified slightly for use in this book. They are set to an even multiple of five and so that the length of the field in the east-west direction is twice that in the north-south direction.

```
> N <- 4267873
> S <- 4267483
> E <- 592860
> W <- 592080
> N - S
[1] 390
> E - W
[1] 780
```

We first put the boundary values into a matrix `coords.mat` that holds the coordinates describing the polygon. By default, matrices in R are created by columns, so that in this

case the first four members of the sequence form the first column and the second four members form the last column. Because the region is rectangular, the coordinates of its corners are specified by four numbers. In the creation of a polygon with n vertices, a total of $n + 1$ coordinate pairs are assigned, with the first coordinate pair being repeated as the last. Thus, in this case five vertices are specified.

```
> print(coords.mat <- matrix(c(W, E,E, W,W, N,N, S,S, N),
+   ncol = 2))
        [,1]     [,2]
[1,]  592080  4267873
[2,]  592860  4267873
[3,]  592860  4267483
[4,]  592080  4267483
[5,]  592080  4267873
```

Since the geometry of an `sf` object is specified as a list, we use the constructor function `list()` to create a list containing the boundary points.

```
> coords.lst <- list(coords.mat)
```

The next step is to tell the system that we are creating a polygon rather than a set of points whose first is the same as the last. This is done by creating an `sfc` object, which is called a *list-column*.

```
> coords.pol = st_sfc(st_polygon(coords.lst))
```

Next, we once again apply the function `st_sf` to create the `sf` polygon object describing the boundary. We assign the single cell in the polygon an attribute `z` with the value 1.

```
> Set42bdry = st_sf(z = 1, coords.pol))
```

Once again, we must assign a coordinate system to the boundary using `st_crs()`. This time we will use the EPSG code.

```
> st_crs(Set42bdry) <- 32601
```

Section 2.6.2 contains a more thorough introduction to the construction of maps in R using `sf` objects. At this point, however, it is a good idea to take a look at the polygon we have created. We can do this with a simple call to the function `plot()`.

```
> plot(Set42bdry)
```

A plot of the boundary should appear in the Plot/Help window of Rstudio.

We need to save the boundary that we have created as an ESRI shapefile for later use. This is the first of many such objects that we will need to save. The output of `sf` objects is done via the function `st_write()`. As in Section 2.1.2, let us assume that you have set your working directory using `setwd()`. Suppose you have created a subfolder of this directory called *created*. The function call would then be

```
> st_write(Set42bdry, "created\\Set42bdry.shp")

Writing layer `Set42bdry' to data source `created\Set42bdry.shp' using
driver `ESRI Shapefile'
features:       1
fields:         1
geometry type:  Polygon
```

The function `st_write()` detects the extension *.shp* in the filename and appropriately creates a shapefile.

The input of data into `sf` objects is carried out using the function `st_read()`. We can illustrate this by reading the shapefile containing the land cover data shown in Figure 1.5. This is stored in the shapefile *landcover.shp*. By the way, if you are unfamiliar with ESRI shapefiles, they actually consist of at least three separate files, each with the same filename and a different extension. We will follow the usual convention of referring to only one file, but be aware that for these files to function, all of them must be present. The correct function call is

```
> data.Set1.landcover <- st_read("Set1\\landcover.shp")
```

and the R response is not reproduced here to save space. By the way, in assigning names to objects, I tend to be quite verbose, with the idea that this helps to avoid confusion as to what these objects represent. The feature of RStudio that it automatically fills in the names of variables after a few key strokes comes in handy here. Anyway, we can examine the structure of `data.Set1.landcover` in the usual way.

```
> str(data.Set1.landcover)
Classes 'sf' and 'data.frame':  1811 obs. of  7 variables:
 $ SP_ID      : Factor w/ 1811 levels "0","1","10","100",..:  1 2 924 ...
 $ Area       : num  34748 21338 183208 9626 3889...
 $ Perimeter  : num  717 834 2128 1014 335...
 $ VegType    : Factor w/ 10 levels "a","c","d","f",..:2 9 2 9 5 8 5  ...
 $ HeightClas : Factor w/ 4 levels "0","h","l","m":1 4 1 3 1 1 1 1 4  ...
 $ CoverClass : Factor w/ 5 levels "0","d","m","p",..: 1 2 1 2 1 1 1  ...
 $ geometry   :sfc_POLYGON of length 1811; first list element: List of 1
  ..$ : num [1:17, 1:2] 576493 576691 576704 576708 576697...
  ..- attr(*, "class")= chr "XY" "POLYGON" "sfg"
 - attr(*, "sf_column")= chr "geometry"
 - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA
 NA NA NA
  ..- attr(*, "names")= chr "SP_ID" "Area" "Perimeter" "VegType"...
```

We can obtain nice plots of all of the attributes of `data.Set1.landcover` with a simple call to `plot()`. In general, the results of these calls to `plot()` are not shown, but the calls are included in the code.

We now move on to the `sp` package. Although `sf` may someday replace `sp`, as of the time of the writing of this book, a complete reliance on `sf` is not possible. One reason for this is that the `raster` package, discussed below, works with `sp`. A second reason is that we will also make extensive use of the `spdep` and `maptools` packages (Bivand et al., 2013; Bivand and Piras, 2015), and these work with `sp` objects.

To begin with the `sp` package, we will use the coercion function `as()` to coerce `data.Set1.landcover` into an `sp` object.

```
> data.Set1.landcover.sp <- as(data.Set1.landcover, "Spatial")
> str(data.Set1.landcover.sp)
Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame': 1811 obs. of 6 variables:
  .. ..$ SP_ID    : Factor w/ 1811 levels "0","1","10","100",..: 1 2
924 1035 1146 1257 1368 1479 1590 1701...
  .. ..$ Area      : num [1:1811] 34748 21338 183208 9626 3889...
  .. ..$ Perimeter : num [1:1811] 717 834 2128 1014 335...
  .. ..$ VegType   : Factor w/ 10 levels "a","c","d","f",..: 2 9 2...
  .. ..$ HeightClas: Factor w/ 4 levels "0","h","l","m": 1 4 1 3 1...
  .. ..$ CoverClass: Factor w/ 5 levels "0","d","m","p",..: 1 1 2 1...
  ..@ polygons   :List of 1811
  .. ..$:Formal class 'Polygons' [package "sp"] with 5 slots
  .. .. .. ..@ Polygons:List of 1
  .. .. .. .. ..$:Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. .. .. ..@ labpt  : num [1:2] 576596 4420706
  .. .. .. .. .. .. ..@ area   : num 34748
  .. .. .. .. .. .. ..@ hole   : logi FALSE
  .. .. .. .. .. .. ..@ ringDir: int 1
  .. .. .. .. .. .. ..@ coords : num [1:17, 1:2] 576493 576691 576704
576708 576697...
  .. .. .. ..@ plotOrder: int 1
  .. .. .. ..@ labpt    : num [1:2] 576596 4420706
  .. .. .. ..@ ID       : chr "1"
  .. .. .. ..@ area     : num 34748

       *       *       *
```

You get a lot of output! Only the very beginning is shown here. The object `data.Set1.landcover.sp` is a `SpatialPointsDataFrame`. The data in this object are contained in *slots*. Without going into any more detail than is necessary, the `SpatialPointsDataFrame` itself contains five slots, only the first two of which (`data` and `polygons`) are shown here. The data slot contains the attribute data. To access the data in a slot one can use the function `slot()`. The slots themselves are objects that are members of some class.

```
> class(slot(data.Set1.landcover.sp, "data"))
[1] "data.frame"
```

To display the first ten elements of the `VegType` data field, we can type

```
> slot(data.Set1.landcover.sp, "data")$VegType[1:10]
 [1] c v c v g r g a v l
Levels: a c d f g l o r v w
```

Since `slot(data.Set1.landcover.sp,"data")` is a data frame, the `$` operator may be appended to it to specify a data field, and then square brackets may be appended to that to access particular data records. A shorthand expression of the function `slot()` as used above is the `@` operator, as in the following.

```
> data.Set1.landcover.sp@data$VegType[1:10]
 [1] c v c v g r g a v l
Levels: a c d f g l o r v w
```

The `polygons` slot contains a list with the geometrical information. It is in the form of a list, with one `list` element per polygon. Each element of the list is a `Polygon` object, which itself has slots. Most of these we can ignore, but we will be using two, the `labpt` slot and the `ID slot`. Only these are shown here.

```
> data.Set1.landcover.sp@polygons[[1]]
An object of class "Polygons"
Slot "Polygons":
[[1]]
An object of class "Polygon"
Slot "labpt":
[1] 576595.8 4420706.2
           *        *      *
Slot "ID":
[1] "1"
           *        *      *
```

We will use these in later chapters to check that attribute values are at their correct locations. Since the data are in lists, we can use `lapply()` to extract them. The function `lapply()` is one of the family of `apply()` functions discussed in Section 2.3.2. Type `?lapply` to see an succinct description of it. We can obtain the ID values as follows.

```
> lapply(data.Set1.landcover.sp@polygons, slot, "ID")
[[1]]
[1] "1"

[[2]]
[1] "2"
              *        *        *
```

In this case, the ID values are in the order of the polygons, but one cannot guarantee that this is always so. Extracting the label points is trickier because they are each a pair of coordinates.

```
> y <- lapply(data.Set1.landcover.sp@polygons, slot, "labpt")
> data.Set1.landcover.sp.loc <- matrix(0, length(y), 2)
> for (i in 1:length(y))
+    data.Set1.landcover.sp.loc[i,] <- unlist(y[[i]])
> data.Set1.landcover.sp.loc
            [,1]    [,2]
   [1,] 576595.8 4420706
   [2,] 576710.6 4420652
   [3,] 577163.5 4420675
           *        *         *
```

To complete our discussion of `sp` objects we will mention the function `coordinates()`. This is the `sp` equivalent of the function `st_crs()` described above, and, like `st_crs()`, it can work as a replacement function if it is on the left side of the assignment arrow or it can retrieve the coordinate system in place otherwise.

```
> data.Set1.sp <- data.Set1.obs
> coordinates(data.Set1.sp) <- c("Easting", "Northing")
> proj4string(data.Set1.sp) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
```

There is one subtle but important difference between the sp function coordinates() and the sf function st_crs(). The function st_crs() creates a new sf object, while the function coordinates() converts an existing data frame into an sp object.

To work with raster data, we will usually use the raster package (Hijmans, 2016). There are three fundamental objects in this package: the RasterLayer, the RasterBrick, and the RasterStack. A RasterLayer represents a single layer of raster data, such as the infrared reflectance values shown in Figure 1.2b. When working with a grid of remotely sensed values, however, often one works with data from more than one spectral band. As described in Section 1.3.4, the data use to create this figure come from a multispectral sensor (Lo and Young, 2007, p. 294) and include three bands, one at the infrared level, one at the red level, and one at the green level. For this reason, these data cannot be represented by a single RasterLayer. The RasterBrick accommodates multilevel data sets such as this by incorporating multiple layers. The data we use is stored in a *tiff* file and can be loaded into R very easily using the function brick.

```
> # Read the three bands of a tiff image as a raster stack
> library(raster)
> data.4.2.May <- brick("set4\\Set4.20596.tif")
> data.4.2.May
class       : RasterBrick
dimensions  : 621, 617, 383157, 3 (nrow, ncol, ncell, nlayers)
resolution  : 1.902529, 1.902529 (x, y)
extent      : 591965.6,  593139.5, 4267139, 4268321 (xmin, xmax, ymin, ymax)
coord. ref. : NA
data source : c:\rdata\Data\Set4\Set4.20596.tif
names       : Set4.20596.1, Set4.20596.2, Set4.20596.3
min values  :            0,            0,            0
max values  :          255,          255,          255
```

There is no coordinate reference system because none has yet been assigned. We can assign one using the function projection().

```
> projection(data.4.2.May) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
> data.4.2.May
class       : RasterBrick
dimensions  : 621, 617, 383157, 3 (nrow, ncol, ncell, nlayers)
resolution  : 1.902529, 1.902529 (x, y)
extent      : 591965.6,  593139.5, 4267139, 4268321 (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=10 +ellps=WGS84
data source : c:\aaRdata\Book\Second Edition\Data\Set4\Set4.20596.tif
names       : Set4.20596.1, Set4.20596.2, Set4.20596.3
min values  :            0,            0,            0
max values  :          255,          255,          255
```

You can take a look at all three layers of the RasterBrick using the plot() function.

```
> plot(data.4.2.May)
```

To access one layer of the brick, use double square brackets. For instance, to display only the first, infrared layer, enter the following.

```
> plot(data.4.2.May[[1]])
```

The RasterBrick object permits us to work with an object consisting of multiple layers, but it has an important property. If we look again at Figure 1.2 in its entirety, we can see that it consists of multiple layers coming from multiple data files. Each of these files contains gridded data, but the data from different files. A RasterBrick must contain data from a single file. To deal with data from multiple files we use the RasterStack object. We can build a raster stack as follows. Suppose we already have a RasterLayer object created to hold the soil electrical conductivity data shown in Figure 1.2a. This layer is created using the kriging interpolation procedure described in Section 6.3 to create an object called EC.grid. The procedure for doing this will be described in that section. This object is converted into a RasterLayer object called EC.ras using the raster function interpolate().

```
> EC.ras <- interpolate(grid.ras, EC.krig)
```

Our objective is to combine this with the infrared layer of the raster brick data.4.2.May to create a raster stack.

Although the two raster objects describe data from the same geographical area, since they were created using different methods they may not exactly overlap. We can check this quickly by determining whether they have the same extent, using the raster function of that name.

```
> extent(EC.ras)
class       : Extent
xmin        : 592077
xmax        : 592867
ymin        : 4267513
ymax        : 4267873
> extent(data.4.2.May)
class       : Extent
xmin        : 591965.6
xmax        : 593139.5
ymin        : 4267139
ymax        : 4268321
```

The extents, and thus the grids, are indeed different. In order to create a stack, we must make them line up. We do this using the function resample(), which is named after a common GIS operation in which data on one grid is "resampled" to another grid (Lo and Young, 2007, p. 154).

```
> IR.4.2.May <- resample(data.4.2.May[[1]], EC.ras)
```

Now we can create the RasterStack object.

```
> data.4.2.May.stack <- stack(IR.4.2.May, EC.ras)
> plot(data.4.2.May.stack)
```

As a contributed package, raster may change at some point in the future, so it is good to have another package as a fallback. A good one is the package rgdal (Keitt et al., 2017). For purposes of illustration, this function is occasionally used to read in image data, for example in the code in Sections 1.1 and 6.4.2.

2.5 Writing Functions in R

Like most programming languages, R provides the user with the capacity to write functions. As an example demonstrating the basics of writing an R function, we will create one that computes a simplified "trimmed standard deviation." Most statistical packages contain a function that permits the computation of a trimmed mean, that is, the mean of a data set from which a certain number or fraction of the highest and lowest values have been removed. The R function `mean()` has an optional argument `trim` that allows the user to specify a fraction of the values to be removed from each extreme before computing the mean. The function `sd()` does not have a similar argument that would permit one to compute the standard deviation of a trimmed data set. The formula used in our example is not standard for the "trimmed standard deviation" and is intended for illustrative purposes only. See Wu and Zuo (2008) for a discussion of the trimmed standard deviation.

Given a vector `z` and a fraction `trim` between 0 and 0.5, the standard deviation of the trimmed data set can be computed as follows. First we sort the data in `z`.

```
> z <- c(10,2,3,21,5,72,18,25,34,33)
> print(ztrim <- sort(z))
[1] 2 3 5 10 18 21 25 33 34 72
```

Next we compute the number to trim from either end, based on the value of `trim`. We use the coercion function `as.integer()` to obtain an integer value.

```
> trim <- 0.1
> print(tl <- as.integer(trim * length(ztrim)))
[1] 1
```

Next we trim the values, first the low end and then the high end.

```
# Trim from the low end
> print(ztrim <- ztrim[-(1:tl)])
[1] 3 5 10 18 21 25 33 34 72
```

Next we trim from the high end.

```
> # Trim from the high end
> th <- length(ztrim) - tl
> print(ztrim <- ztrim[-((th + 1):length(ztrim))])
[1] 3 5 10 18 21 25 33 34
```

Finally, we compute the standard deviation.

```
> print(zsd <- sd(ztrim))
[1] 11.91563
```

Now we will create a function `sd.trim()` that carries out the entire sequence of steps. The syntax is

```
> # Function to compute the trimmed standard deviation
> sd.trim <- function(z, trim){
```

```
+ # trim must be between 0 and 0.5 or this won't work
+ # Sort the z values
+   ztrim <- sort(z)
+ # Calculate the number to remove
+   tl <- as.integer(trim * length(ztrim))
+ # Remove the lowest values
+   ztrim <- ztrim[-(1:tl)]
+ # Remove the highest values
+   th <- length(ztrim) - tl
+   ztrim <- ztrim[-((th + 1):length(ztrim))]
+ # Compute the standard deviation
+   zsd <- sd(ztrim)
+   return(zsd)
+ }
```

The function is defined by `function(arguments) expression`. The *arguments* represent one or more objects whose values are explicitly named in the statement invoking the function. The *expression* is the step or sequence of steps that carries out the specified calculations. In this book, the last statement of any function with more than one statement will generally be the *return* statement, which specifies the vector of values to return. If no return statement exists, the last object computed is returned. In the case of a function with only one assignment statement, the return statement is superfluous. An example of such a function is given later in this section The application of a user-defined function proceeds just like the calculation of any other function.

```
> sd.trim(z, 0.1)
[1] 11.91563
```

We will be writing a large number of functions in this book. It is important to note at the very outset, however, that the code we display will deviate from good programming practice in one very important way. In an effort to keep the code simple and concise, we will generally not include the error trapping facilities that would be built into good commercial software, or into a good contributed R package. For example the code in `sd.trim()` computes the standard deviation of a data set after a fraction *trim* of the largest and smallest values have been removed. The code does not include a test to determine whether $0 \leq trim \leq 0.5$; such a test would be an absolute necessity in software that was intended for general use. Error trapping and resolution is a complex process (Goodliffe, 2007, Ch. 6) and to do it properly would dramatically increase the length and complexity of the code. Instead, to borrow a phrase from Ed Post (Post, 1983), our error trapping philosophy will be "you asked for it, you got it."

Recall that R is an object-oriented programming language. In addition to the class structure described in Section 2.2, a feature common among object-oriented programming languages is *polymorphism*. This refers to the ability of functions to accept arguments of different classes and to adjust their evaluation depending on the class of the argument or arguments. This feature can be illustrated using the function `diag()`. When the argument of this function is a matrix, `diag()` returns the diagonal of this matrix.

```
> a <- matrix(c(1,2,3,4), nrow = 2)
> a
     [,1]  [,2]
[1,]    1     3
```

```
[2,]    2    4
> diag(a)
[1] 1 4
```

When its argument is a vector, `diag()` returns a diagonal matrix whose diagonal is the components of the vector.

```
> v <- c(1,2)
> v
[1] 1 2
> diag(v)
     [,1] [,2]
[1,]    1    0
[2,]    0    2
```

Polymorphism is an incredibly useful feature in simplifying the coding of R programs involving different kinds of data structures. For example, the function `plot()` knows how to plot many kinds of objects and will generally create the most appropriate plot for a given object class (see Exercise 2.10). The first argument of a polymorphic functions like `plot()` determines which version of the function is used.

One way in which R differs from some other object-oriented programming languages is that R implements a feature called *lexical scoping* (Venables and Ripley, 2000, p. 63). Without going into details, an important feature of lexical scoping can be illustrated by creating a rather strange function called `to.the.n()` to evaluate x^n.

```
> to.the.n <- function(w) w^n
> n <- 3
> to.the.n(2)
[1] 8
```

The function is strange because it does not include the value of n as one of its arguments. Since the value of n is not available as one of the arguments of `to.the.n()`, R looks for a value outside the scope of the function. The value 3 has been assigned to a variable `n` prior to the call to the function, and R uses that value. The function `better.to.the.n()`, which does include `n` in its argument, uses this value and ignores (actually, never looks for) the value 3 that is assigned to `n` outside the function but is not used.

```
> better.to.the.n <- function(w, n) w^n
> n <- 3
> better.to.the.n(2,4)
[1] 16
```

This is a vastly oversimplified treatment of lexical scoping. We will not have much need to make use of it in this book, and a more detailed discussion is given by Venables et al. (2006, p. 46) and also in the R project FAQs (http://www.r-project.org/). The main reason for mentioning it is to warn those used to programming in some other languages of its existence. Also, if you are not careful, lexical scoping can give you some very nice rope with which to hang yourself. Specifically, if you forget to specify some variable in a function and a variable by that name happens to be hanging around in your workspace, perhaps from a completely different application, R will happily plug that value into the computation and give you a completely incorrect result. A bug like this can be very hard

to track down. For this reason, it is better to pass all values used by the function through the argument list unless there is good reason to do otherwise. In Section 5.3.1, lexical scoping is used to call a function from inside another function. This makes the code simpler.

2.6 Graphics in R

R has a very powerful set of graphics tools, and in this section we describe some of these tools using some simple examples. Our primary objective is data visualization. Data visualization, like presentation graphics, involves displaying of features of the data in graphical form. However, unlike presentation graphics, whose purpose is to convey already identified features of the data to an audience not necessarily familiar with the data themselves, the purpose of data visualization is to allow the analyst to gain insight into the data by identifying its properties and relationships (Haining, 2003, p. 189). The human brain instinctively searches for and perceives patterns in visual data. In order for the process of data visualization to be successful, the encoding of the data into a graphical object must be done in such a way that the decoding of the object by the human eye is effective (Cleveland, 1985, p. 7). This means that the visual patterns in the graphical object must not induce "optical illusions" in which the eye perceives patterns differently from the actual arrangement of the data.

The R programming environment, both in its base and contributed packages, provides an incredibly powerful and flexible array of tools for constructing graphical images. Every figure in this book except the photographs (R is not *that* good) was constructed using R. The base R graphics system embodies both the strengths and the challenge of the R programming environment. The strengths are the flexibility and the power to generate almost any type of graphical image one can desire. The challenge is that the graphics software is controlled by a function-based, command-driven programming environment that can sometimes seem discouragingly complex.

At the outset, one thing must be emphasized. As with everything else, the R programming environment provides several alternative methods for generating the graphical images displayed here. Not all of these alternatives will be presented in this book, although in some cases we will describe more than one if there is a good reason to do so. Readers who wish to learn more about other alternatives are advised to begin with Murrell (2006) and then to proceed to other sources listed in Section 2.8. One package in particular, however, deserves special mention: the package `ggplot2` (Wickham, 2009). This package, and the function `ggplot()` that it contains, provide a very powerful alternative to the function `plot()`. In particular, `ggplot()` is so simple to learn that it practically overcomes the complexity of use issue described in the previous paragraph. At least in my opinion, however, `plot()` is stronger in the creation of presentation graphics such as the figures in this book. This is partly due to the excellent description by Murrell (2006) of how to control every aspect in the presentation of a figure. Therefore I have decided to use each of these plotting functions for what I consider their strengths. The figures, which are presentation graphics, are made with traditional graphics such as `plot()`. In every case where quick and effective data visualization is appropriate, however, I have included in the code a call to `ggplot()`. Both `plot()` and `ggplot()` are described in the next section.

2.6.1 Traditional Graphics in R: Attribute Data

The R presentation graphics presented in this book are based on two fundamental packages, a "traditional" system implemented in the base package and a trellis graphics system implemented in the `lattice` package (Sarkar, 2008). Trellis graphics is a technique originated by Cleveland (1993) that seeks to organize multivariate data in a manner that simplifies the visualization of patterns and interactions. We focus on `lattice` because it is the foundation for the spatial plotting functions discussed in this book. Our discussion references figures in other chapters. The code to produce these figures is in the section containing the figure, rather than in this section. All the plots in this book were created by executing the R code using TINN-R as the editor and running R as a separate application, so that it produces a separate window that can be printed or saved. If you create these plots in RStudio, they will appear in the Plot/Help window and will generally look very different.

Functions in R that produce a graphical output directly are called *high level* graphics functions. The standard R traditional graphics system has high level functions such as `pie()`, `barplot()`, `hist()`, and `boxplot()` that produce a particular type of graph. Some of these will be described later on. The fundamental high level workhorse function of the standard presentation graphics system, at least for us, will be the function `plot()`. When applied to attribute data (i.e., with no spatial information), the output of `plot()` is a plot of an array Y against an array X. We will introduce `plot()` using Figure 3.6a in Section 3.5.1, which is one of the simplest figures in the book. It is a plot of an array called `error.rate` on the abscissa against an array called `lambda` on the ordinate. After working through this plot, you will have the basic information needed to construct a plot in R. You may wish to skim the remainder of the section, which concerns more advanced plots, on first reading. If you do so, stop skimming when it gets to the material covering the `ggplot2` package.

A simple statement to generate a scatterplot is the command

```
> plot(lambda, error.rate)
```

We have not yet discussed the *error rate*, which is a quantity that is explained in Section 3.5.1. For now, we are only interested in the plot of this quantity (again, you should open the code file *3.5.1.r* to follow this discussion). If you try this with arrays generated in Chapter 3, you will get a plot that resembles Figure 3.6a but has some important differences. There is no title, the abscissa and ordinate labels are smaller, and instead of the Greek letter λ and the words "Error Rate," the axes contain the names of the objects, `lambda` and `error.rate`, themselves. These will need adjusting. The R graphics system does, however, automatically do several things very well. The plotting region is clearly demarcated by four scale lines, with the tick marks on the outside, and, as a default, having an approximately square shape (Cleveland, 1985, p. 36).

Some of the information needed to control the appearance of the graph can be included directly in the call to the high level function `plot()`. We can specify a title. Type the following.

```
> plot(lambda, error.rate, main = "Error Rate")
```

That gets us a title. Your title may look different from that of Figure 3.6a because the publisher of the book has altered the graphics to save space. The next step is the axis labels. The ordinate is easy:

```
> plot(lambda, error.rate, main = "Error Rate", ylab = "Error Rate")
```

The abscissa requires one more step. This is to use the function `expression()` within the call to `plot()`. To see a complete list of the things `expression()` can do, type `?plotmath`. From this we see that to plot a Greek letter for the ordinate label we can type the statement

```
> plot(lambda, error.rate, main = "Error Rate", ylab = "Error Rate",
+    xlab = expression(lambda))
```

This almost gets us to where we want to be, but the labels are still too small. The general indictor in R standard graphics for size is `cex`. This is a factor by which the size of a label is multiplied when creating a graph; the default is a value of 1. The graph in Figure 3.6a is created with the statement

```
> plot(lambda, error.rate, xlab = expression(lambda),
+   ylab = "Error Rate", cex.lab = 1.5) # Fig. 3.6a
```

This solves the label size problem, but creates another problem: the abscissa label may be crowded against the side of the graphics window. We need to increase the size of the area surrounding the graph, which is called the *figure margin* (actually, this is an oversimplification; see Murrell, 2006, Section 3.1 for a full explanation). There are a number of options one can use to control this. All must be set not as an argument of `plot()` but rather as an argument of the function `par()`, which controls the R plotting environment. A very simple method is to use the argument `mai`, which gives the margin size in inches of the left, right, top, and bottom margins, respectively, in inches (yes, inches!). For our application the statement

```
> par(mai = c(1,1,1,1))
```

will do the trick. This statement is used prior to almost all traditional plots in this book and will no longer be explicitly mentioned. Entering the statement and then the call to `plot()` above produces Figure 3.6a. Settings made in a high level or low level plotting function are applied only to that instance of the function, but settings made in a call to the function `par()` are retained until they are changed or the R session is stopped without saving the workspace. Again, in the version that you see of Figure 3.6a the size and font of the title and axis labels may have been changed in the process of setting the book up for publication. For this reason, the figure may look different from the one you get as output.

Now we will move on to more advanced plots. Figure 3.6b provides another example of the use of the function `expression()`, this time for the abscissa label. The quantity plotted on the abscissa is $\bar{Y}$, the sample mean, as a function of λ. This requires two actions on the character: putting a bar over it and italicizing it. From `?plotmath` we see that the code in `expression()` to produce a bar over the Y is `bar(Y)` and the code to italicize is `italic()`, so we can combine these.

```
> plot(lambda, mean. Y, xlab = expression(lambda),
+    ylab = expression("Mean"~italic(bar(Y))),
+    ylim = c(-0.1, 0.1), cex.lab = 1.5) # Fig. 3.6b
> title(main = "Mean of Sample Means", cex.main = 2)
```

The tilde ~ character in the function `expression()` inserts a space. To separate two symbols without inserting a space, use an asterisk, as in the code for Figure 3.6c. Note that the parentheses have quotes around them.

```
> plot(lambda, sem. Y, xlab = expression(lambda),
+     ylab = expression("Mean s{"*italic(bar(Y))*"}"),
+     ylim = c(0,0.5), cex.lab = 1.5) # Fig. 3.6c
> title(main = "Mean Estimated Standard Error", cex.main = 2)
```

The code sequence introduces another concept, the *low level* plotting function. The function `title()` is a low level plotting function. It adds material to an existing graphic rather than creating a new graphic as `plot()` does. The philosophy of traditional R graphics is to start with a basic graph and then build it up by using functions to add material. A common way (but not the only way, as we will see below) to add material is by using low level plotting functions.

Now let's turn to a more complex attribute plot, that of Figure 3.1. This is a plot of a cross section of the components of the artificially generated surface shown in Figure 3.2a (which we will get to in a moment). The figure shows three line segment plots: (a) the trend $T(x,y)$, (b) the trend plus autocorrelated component $T(x,y)+\eta(x,y)$, and (c) the trend plus autocorrelated component plus uncorrelated component $Y(x,y)=T(x,y)+\eta(x,y)+\varepsilon(x,y)$. The components T, η, and ε are represented in the R code by `Yt`, `Yeta`, and `Yeps`, respectively, and Y is represented by `Y`. The initial plot is constructed with the following statement.

```
> plot(x, Y, type = "l", lwd = 1, cex.lab = 1.5, #Fig. 3.1
+     ylim = c(-0.5,2.5), xlab = expression(italic(x)),
+     ylab = expression(italic(Y)))
```

The argument `type = "l"` specifies that the graph will be a line. The argument `lwd` controls the line width. This is set at 1, which is the default value. The argument `ylim = c(-0.5,2.5)` controls the abscissa limits. A similar specification of the argument `xlim` is not needed in this particular case because the automatically generated ordinate values are adequate. The `plot()` statement produces the axes, labels, and the thin curve representing $Y(x,y)=T(x,y)+\eta(x,y)+\varepsilon(x,y)$. Now we will build the rest of the graphic by adding material. First, we add the curve representing $T(x,y)+\eta(x,y)$ with the following statement.

```
> lines(x, Yt, lwd = 3)
> lines(x, Yt + Yeta, lwd = 2)
```

The argument `lwd = 3` specifies that the line width will be three times the default width. Now it is time to add the legend. There is a function `legend()` that will do this for us (Exercise 2.8), but for this plot we will use the `text()` function. It takes a bit of trial and error to locate the correct placement, but finally we come up with the statements

```
> text(x = 12.5, y = 0.25, "T(x, y)", pos = 4)
> lines(c(10,12.5),c(0.25,0.25),lwd = 3)
```

to produce the first line of the legend, and similar statements for the rest of the legend. The argument `pos = 4` causes the text to be placed to the right of the location specified in the argument. The default is to place the center of the text at this point.

As our final, and most complex, example of an attribute graph, we will discuss Figure 5.14b. First, we issue a `plot()` statement to generate the basic graph.

```
> plot(data.samp$IRvalue, data.samp$Yield, # Fig. 5.14b
+    xlab = "Infrared Digital Number", ylab = "Yield (kg/ha)",
+    xlim = c(110, 170), ylim = c(2000, 7000),
+    main = "Transect 2", pch = 16,
+    cex.main = 2, cex.lab = 1.5, font.lab = 2)
```

Because we want the axes of this figure to match those of Figure 5.14a and 5.14c, we control the scale of the axes using the arguments `xlim = c(110, 170), ylim = c(2000, 7000)`. The argument `pch = 16` generates black dots. The next step is to plot a regression line fit to these data. The least squares fit is obtained with the statement

```
> Yield.band1 <- lm(Yield ~ IRvalue, data = data.samp)
```

The line is added to the plot with the statement

```
> abline(reg = Yield.band1)
>
```

The next step is to add data points and regression lines for the second transect. The data for this transect is stored in the data frame `data.samp2` in the data fields `IRvalue` and `Yield`. The data points are added to the graphic with the statement

```
> with(data.samp2, points(IRvalue, Yield, pch = 3))
```

The regression line is then added with the statements

```
> Yield. IR2 <- lm(Yield ~ IRvalue, data = data.samp2)
> abline(reg = Yield. IR2, lty = 3)
```

Here the argument `lty = 3` generates a dotted line (`lty` stands for "line type"). The other scatterplots and regression lines are added with similar statements. The next step is to add the mean values. The following statement produces the large black dot located on the solid line that fits all the data.

```
> points(mean(data.samp$IRvalue), Y.bar,
      + pch = 19, cex = 2)
```

Next, a symbol of the form $\bar{Y}$ is added using the statement

```
> text(144, 4600, expression(bold(bar(Y))))
```

The other means are added similarly. The subscripts are inserted using square brackets, for example in the statement

```
> text(139, 5500, expression(bold(bar(Y)[2])))
```

Finally, the legends are added. The first legend statement has this form:

```
> legend(155, 6000, c("All Data", "Transect 2"),
+   pt.cex = 1, pch = c(19,3), y.intersp = 1,
+   title = "Sample")
```

The other legend statement is similar. Once again, the axis labels and header in the figures in the book may be different from those you produce due to the publisher's saving of space in the figures.

Before taking up the application of the function `plot()` to spatial data, we briefly discuss the function `persp()`, which serves as a sort of segue between attribute plots and spatial plots. This function is used to produce the "wireframe" three-dimensional graphs of Figure 3.3. We will not have much occasion to use these wireframe graphs, so we do not spend much time discussing their use. Here is the statement that generates Figure 3.2a.

```
> x <- (1:20) + 1.5
> y <- x
> Y.persp <- matrix(4*Y, nrow = 20)
> persp(x, y, Y.persp, theta = 225, phi = 15,
+   scale = FALSE, zlab = "Y") #Fig. 3.2a
```

Here `theta` and `phi` are the angles defining the viewing direction; `theta` gives the azimuthal direction and `phi` the colatitude, that is, the angle of elevation from which the view is seen.

As you can see, the function `plot()` gives you lots of control, but at the price of lots of complexity. Now let's look at `ggplot2()`, which in some sense is the opposite. Calls to `ggplot2()` can be done in many ways, but here we will use the form described by Wickham and Grolemund (2017). The "gg" in "ggplot" stands for "grammar of graphics" (Wilkinson, 2005), an organizational structure for graphic presentation on which `ggplot()` is based. The quickest way to introduce `ggplot()` is with an example. This one comes from Section 7.3, which involves the exploration of Data Set 2, the blue oak data (you may want to go back and read Section 1.3.2 to get an idea of the data set). At this point, you should load the code file *7.3.r* and run it along with me.

We would like to get an idea of the relationships between blue oak presence/absence and environmental factors. We know for a start that blue oak is influenced directly by elevation. We can try plotting blue oak presence (*QUDO* = 1) or absence (*QUDO* = 0) against *Elevation*. So we try this.

```
> ggplot(data = data.Set2) +
+    geom_point(aes(x = Elevation, y = QUDO))
```

The first line is the call to `ggplot()` itself. It sets the coordinate system (the default is Cartesian) and identifies the data source. The next line adds a layer to the plot in the form of a "geom." A geom is a type of plot, and the function `geom _ point()` specifies a scatter plot. In this case, as you can see, the plot does not provide much information, because the values of `QUDO` are so dense. We can get a better picture of the dependence of *QUDO* on *Elevation* using a different geom.

```
> ggplot(data = data.Set2) +
+    geom_smooth(aes(x = Elevation, y = QUDO))
```

This smooths the data using a generalized additive model (Section 9.2) and provides a good indication of how blue oaks vary with elevation. What might be causing this variation? One possibility is mean annual temperature. For a nicer plot, we first convert QUDO into a character. Then we can plot mean annual temperature, *MAT*, against *Elevation* while using color to display oak presence or absence. We can also add a smooth curve summarizing the relationship between *MAT* and *Elevation*.

```
> data.Set2$QF <- as.character(data.Set2$QUDO)
> ggplot(data = data.Set2) +
+   geom_point(aes(x = Elevation, y = MAT, color = QF)) +
+   geom_smooth(aes(x = Elevation, y = MAT))
```

We can add as many geoms as we wish to the plot by adding more calls to the appropriate function.

Another climatic quantity that might influence blue oak presence or absence is annual precipitation. We can easily examine the relationship between *MAT* and *Precip.*

```
> ggplot(data = data.Set2) +
+   geom_point(aes(x = Precip, y = MAT, color = QF))
```

There are many more geoms that can be accessed (lines, boxplots, histograms, etc.), and these will be demonstrated when appropriate, especially in Chapter 7. There is much more that can be done with `ggplot()`, but with this simple introduction you should have enough to use it effectively in data visualization.

2.6.2 Traditional Graphics in R: Spatial Data

Let us now turn to the use of the function `plot()` to create maps of spatial data. Before we begin, we need to distinguish between the plotting of `sf` objects (i.e., spatial features) and that of `sp` objects (see Section 2.4.3). Because of polymorphism (see Section 2.5), the same command `plot()` can be used with objects of both classes and R will respond appropriately. For any polymorphic R function, the class is determined by the first argument. I have found that the plotting of `sf` objects is most useful for initial quick looks at the data, and that the `sp` version of `plot()` is most effective for plotting presentation graphics, such as those that are used in this book. The code, therefore, will occasionally contain, especially in Chapter 7, calls to the `sf plot()` function, but all of the figures in the book are produced using the `sp` version.

We will begin by discussing Figure 3.4a, which is relatively simple and illustrates some of the basic ideas. Once again, after this basic plot, we will discuss more complex plots, and you may want to skim this on first reading. As usual, the graphic is initiated with an application of the function `plot()`, in this case, used to generate the map of the boundary, which is contained in the `SpatialPolygonsDataFrame bdry.spdf`.

```
> plot(bdry.spdf, axes = TRUE) #Fig. 3.4a
```

When applied to an `sp` object, the function `plot()` cannot be used to generate axis labels (Bivand et al., 2013b, p. 62), but if desired, these can be added later with the function `title()`. The next step is to add the symbols for the value of the plotted quantity, in this case the detrended sand content.

```
> plot(data.Set4.1, add = TRUE, pch = 1,
+   cex = (data.Set4.1$Sand / 15))
```

Attribute values are represented in graphs created using the function `plot()` by controlling the appearance of the plotted symbols. Usually one controls either their size, through the argument `cex`; their form, through the argument `pch`; or their color, through the argument `col`. In our case, we use the argument `cex` to control the size. The argument `cex = (1 + data.Set4.1$Sand / 15` assigns a vector to `cex` with the same number of values as the number of data values to be plotted, and this is used to enable the size of the symbol to represent the data value. The value assigned is obtained by trial and error until the symbols "look right." The other elements of the map are then added with the `title()` and `legend()` functions.

```
> title(main = "Field 4.1 Sand Content", cex.main = 2,
+   xlab = "Easting", ylab = "Northing", cex.lab = 1.5)
> legend(592450, 4270600, c("15", "30", "45"), pt.cex = 1:3, pch = 1,
+   y.intersp = 1.5, title = "Percent Sand")
```

Now let's take on a more complex spatial plot, the map of the survey location of Data Set 1 displayed in Figure 1.4. This consists of a map of the state of California with the location of the study area identified and an insert that shows the study area on a smaller map scale. The map of California is obtained from the `maps` package (Becker et al., 2016).

```
> library(maps)
> data(stateMapEnv)
> cal.map <- map("state", "california",
+    fill=TRUE, col="transparent", plot = FALSE)
```

The map is converted to a `SpatialPolygons` object using the `maptools` function `map2SpatialPolygons()`.

```
> cal.map <- map("state", "california",
+    fill=TRUE, col="transparent", plot = FALSE)
> cal.poly <- map2SpatialPolygons(cal.map, "California")
> proj4string(cal.poly) <- CRS("+proj=longlat +datum=WGS84")
```

The Data Set 1 shapefile is then read. It is stored on disk in UTM Zone 10N coordinates, so the `rgdal` function `spTransform()` is used to convert it to WGS84 longitude and latitude.

```
> data.Set1.sp <- readShapePoly("set1\\set1data.shp")
> proj4string(data.Set1.sp) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
> data.Set1.wgs <- spTransform(data.Set1.sp,
+   CRS("+proj=longlat +datum=WGS84"))
```

Now we are ready to begin the plot. First, we plot the map of California and add the title and the study area map at the California scale. Because we are not going to be using axes in this map, we do not make our usual call to the function `par()`.

```
> plot(cal.poly, axes = FALSE) # Fig. 1.4
> title(main = "Set 1 Site Location",cex.main = 2)
> plot(data.Set1.wgs, add = TRUE)
```

Next, we build a small box around the study site map within California.

```
> lines(c(-122.15,-121.9), c(39.7,39.7))
> lines(c(-122.15,-122.15), c(39.7,39.95))
> lines(c(-122.15,-121.9), c(39.95,39.95))
> lines(c(-121.9,-121.9), c(39.7,39.95))
```

Then we build a large box to hold the second map of the study area, as well as an arrow to connect the two boxes.

```
> lines(c(-118,-113), c(36.3,36.3), lwd = 3)
> lines(c(-118,-113), c(41.9,41.9), lwd = 3)
> lines(c(-118,-118), c(36.3,41.9), lwd = 3)
> lines(c(-113,-113), c(36.3,41.9), lwd = 3)
> arrows(-118, 39.82, -121.85, 39.82, lwd = 3, length = 0.1)
```

Now we use the sp function `SpatialPolygonsRescale()` to add a scale bar. The radius of the earth at the equator is approximately 40,075 km, and we use this to create a scale bar of the appropriate length for the latitude at which it is displayed. The code to display the scale bar is modified from Bivand et al. (2013, p. 65).

```
> deg.per.km <- 360 / 40075
> lat <- 34
> text(-124, lat+0.35, "0")
> text(-122, lat+0.35, "200 km")
> deg.per.km <- deg.per.km * cos(pi*lat / 180)
> SpatialPolygonsRescale(layout.scale.bar(), offset = c(-124, lat),
+   scale = 200 * deg.per.km, fill=c("transparent","black"),
+   plot.grid = FALSE)
```

Bivand et al. use the function `locator()` to place the scale bar. This has the advantage of being quick, but the disadvantages that it must be repeated each time the plot is created and that the objects are never quite in the same place in two different versions of the map. The alternative, shown here, is to play around with the numbers until the map looks right. The north arrow is placed similarly.

```
> SpatialPolygonsRescale(layout.north.arrow(), offset = c(-124, 36),
+   scale = 1, plot.grid = FALSE)
```

Now it is time to insert the second map of the study area. The code for this is modified from that used to create Figure 1.5 of Murrell (2006). First, we use the `par()` function to reset the coordinates of the figure region. These are set relative to the default values 0, 1, 0, 1 (horizontal and vertical). Then we indicate that a new plot will be created with a call to the function `plot.new()`. We create the ranges of the plot based on the bounding box of the `SpatialPolygons` object, create a plot window with these specifications, and plot the map within this plot window.

```
> par(fig = c(0.5, 0.94, 0.3, 0.95), new = TRUE)
> plot.new()
> xrange <- slot(data.Set1.wgs, "bbox")[1,]
> yrange <- slot(data.Set1.wgs, "bbox")[2,]
> plot.window(xlim = xrange, ylim = yrange)
> plot(data.Set1.wgs, add = TRUE, axes = FALSE)
```

Once again, a bit of trial and error is required to get the values right in the function `par()`.

Finally, let us consider the plotting of maps that involve an image such as an aerial photograph. Figure 5.7, which depicts a map of weed infestation level in Field 4.2 over an image of the infrared band of an aerial photograph of the field, provides a good example. Creating these maps is easily done using the function `image()`, which is part of the traditional graphics package. Here is the relevant code to draw the map.

```
> image(data.May.tiff, "band1", col = greys, # Fig. 5.7
+   xlim = c(592100,592900), ylim = c(4267500,4267860),
+ axes = TRUE)
> plot(bdry.spdf, add = TRUE)
> data.samp$hiweeds <- 1
> data.samp$hiweeds[data.samp$Weeds == 5] <- 2
> plot(data.samp, add = TRUE, pch = 1, cex = data.samp$hiweeds)
> title(main = "High Weed Locations", xlab = "Easting",
+   ylab = "Northing", cex.main = 2, cex.lab = 1.5)
```

The high weed infestation level locations are depicted by larger circles, and the polygonal sampling boundary is added to the map as well, all using the standard concept of R traditional graphics: start with a basic graphic and then build on it by adding material.

2.6.3 Trellis Graphics in R, Attribute Data

Trellis graphics are based on ideas developed by Cleveland (1993). Although they can be used to generate almost any of the plots produced by traditional graphics, they are particularly useful for the exploration of multivariate data. They are important for our use because the function `spplot()`, which is the plotting workhorse of the `sp` package, works with trellis graphics, and also because many of the plots generated by the package `nlme` that we will be using for regression analysis use this system. Trellis graphics are implemented in R in the package `lattice` (Sarkar, 2008). The use of trellis graphics is illustrated by Figure 4.3. This figure displays the comparison of three Moran correlograms, one created with a distance-based spatial weights matrix, one with a path-based matrix, and one with a spatial weights matrix based on Thiessen polygons. Moran's I values are computed at five spatial lags using each of these methods in Section 4.5.1. The values are stored in three arrays, `I.d`, `I.p$res[,1]`, `I.t$res[,1]` respectively, and we start with these arrays. The three plots in Figure 4.3 are distinguished based on a *grouping variable* (also called a conditioning variable). Trellis graphics produces a separate plot for each value of the grouping variable.

The easiest way to introduce the use of the grouping variable is to display the code for Figure 4.3. First, we construct a data frame as follows. We start by concatenating the three arrays.

```
> moran.cor <- c(I.d, I.p$res[,1], I.t$res[,1])
```

Next, we create the grouping variable, an index of the method used to create the Moran's *I* values.

```
> cor.type <- sort(rep(1:3, 5))
> cor.type
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Next, we convert this into a factor and add labels identifying the method associated with each index value.

```
> cor.index <- factor(cor.type,
+   labels = c("Distance Based", "Path Based", "Thiessen Polygon"))
```

Finally, we create a data frame with the Moran's *I* and index variables, and add a data field representing the number of lags, which varies from one to five for each of the three plots.

```
> mor.frame <- data.frame(moran.cor)
> mor.frame$index <- cor.index
> mor.frame$lag <- rep(1:5, 3)
```

The final data frame looks like this.

```
> head(mor.frame, 2)
    moran.cor           index lag
1 0.20666693 Distance   Based   1
2 -0.06646138 Distance  Based   2
> tail(mor.frame, 2)
     moran.cor          index lag
14 -0.06585552 Queen's  Case    4
15 -0.16373871 Queen's  Case    5
```

Here we have used the functions `head()` and `tail()` to display the first and last records of the data frame. Now we are ready to create the plot. The trellis graphics analog of `plot()` is `xyplot()`, and that is what we will use. First, we display the code.

```
> xyplot(moran.cor ~ lag | index, data = mor.frame, type = "o",
+    layout = c(3,1), col = "black", aspect = c(1.0),
+    xlab = "Lag", ylab = "Moran's I",
+    main = "Moran Correlograms") # Fig. 4.3
```

The first argument is a *formula*. Formulas will also find use in linear regression, but they play an important role in trellis graphics. In the formula `moran.cor ~ lag | index`, `moran.cor` is the *response* and `lag | index` is the *expression* (Venables and Ripley, 2000, p. 56). As used in the `lattice` package, the response identifies the quantity to be plotted on the abscissa and the expression identifies the quantity to be plotted on the ordinate. The expression is in turn subdivided by a | symbol. The quantity to the left of this symbol is to be plotted on the ordinate. The quantity to the right is the grouping variable. The plots are grouped according to this variable as shown in Figure 4.3.

The second argument, `data = mor.frame`, identifies the data source for the plot. Among the remaining arguments, only `layout` and `aspect` are unfamiliar. The argument `layout` indicates the arrangement of the three *panels* of the lattice plot, that is, the three individual plots. The argument `aspect` indicates the aspect ratio (height to width) of each panel. A very important difference between trellis graphics and traditional graphics is that trellis graphics completely lacks the concept of creating a base graph using a high level function like `plot()` and then adding to it using low level functions like `lines()`

and `text()`. Once a plot is created, it cannot be augmented. There is a function called `update()` that permits graphs to be incrementally constructed, but they are replotted rather than augmented.

2.6.4 Trellis Graphics in R, Spatial Data

Now let us move on to the function `spplot()`. We will begin with Figure 5.1b. This is a plot of two quantities, the yield in kg ha^{-1} of the artificial population used to test various sampling schemes, and the yield data that were interpolated to produce the artificial population. The code to produce Figure 5.1b is as follows. The two data sets are contained in two `SpatialPointsDataFrame` objects, called `pop.data` and `data.Set4.2.sp`. The former is the artificial population, and the latter is the actual yield data with extreme values removed. If we were using the function `plot()`, we could create the plot with the first data set and then add the second data set using the either the function `points()` or the function `plot()` with the argument `add = TRUE`. With the `lattice`-based function `spplot()`, however, we must set everything up first and then use a single call to the plotting function. Therefore, we must combine the two data sets into one. This operation is carried out with the `sp` function `spRbind()`, which combines the two data sets and effectively does an `rbind()` operation (see Section 2.2) on the attribute data.

```
> data.Pop <- spRbind(pop.data, data.Set4.2.sp[,2])
```

In the statement above, we have taken advantage of the fact that although `data.Set4.2.sp` is a `SpatialPolygonsDataFrame`, its attribute data can be accessed as if it were an ordinary data frame (Section 2.4.3). The graph is then plotted with the following statement:

```
> greys <- grey(0:200 / 255)
> spplot(data.Pop, zcol = "Yield", col.regions = greys,
+   xlim = c(592500,592550), ylim = c(4267750,4267800),
+   scales = list(draw = TRUE), # Fig. 5.1b
+   xlab = "Easting", ylab = "Northing",
+   main = "Actual Yield and Artificial Population")
```

We get a warning message about duplicated row names that can be ignored. The color scale `greys` is restricted to the range 0–200/250 in order to avoid plotting white or nearly white dots, which would be invisible.

Now let us turn to Figure 1.5, which shows land use in a portion of the survey area of Data Set 1. This figure displays categorical data and includes a north arrow and scale bar rather than a set of axis scales. The land use types are in a code, so we first create a set of labels corresponding to this code.

```
> levels(data.Set1.cover$VegType)
 [1] "a" "c" "d" "f" "g" "l" "o" "r" "v" "w"
> data.Set1.cover@data$VegType <-
+   factor(as.character(slot(data.Set1, "data")$VegType),
+   labels = c("Grassland", "Cropland","Developed",
+   "Freshwater Wetland", "Gravel Bar", "Lacustrine", "Orchard",
+   "Riverine", "Riparian Forest", "Oak Woodland"))
```

As with the previous application of spplot(), we now create the elements that we are going to use in the final plot. These are the north arrow, the scale bar, and the text accompanying the scale bar.

```
> north <- list("SpatialPolygonsRescale", layout.north.arrow(),
+     offset = c(579800,4420000), scale = 600)
> scale <- list("SpatialPolygonsRescale", layout.scale.bar(),
+     offset = c(579400,4419000), scale = 1000,
+     fill = c("transparent", "black"))
> text1 <- list("sp.text", c(579400, 4419200), "0")
> text2 <- list("sp.text", c(580300, 4419200), "1 km")
> map.layout = list(north, text1, text2, scale)
```

The shades of gray are chosen so that water is displayed as white, unsuitable habitat as light gray, and suitable habitat as dark gray. The actual call to function spplot() is then quite short.

```
> greys <-
+     grey(c(180, 190, 140, 240, 150, 250, 130, 170, 30, 250) / 255)
> spplot(data.Set1.cover, "VegType", col.regions = greys,
+     main = "Land Use, Northern End, 1997",
+     xlim = c(576000,580500), ylim = c(4417400,4421000),
+     sp.layout = map.layout)
```

The function spplot() can also be used to create multiple panel plots. Figure 3.9 is an example. This figure displays the distribution of values of the solution Y of the equation $Y = \mu + \lambda W(Y - \mu) + \varepsilon$, where W is the spatial weights matrix, for three values of the autocorrelation coefficient λ. The plot is created using the following code, which is based on code in the sp Gallery https://edzer.github.io/sp/.

```
> greys <- grey(0:255 / 255)
> spplot(Y.data, c("Y0", "Y4", "Y8"),
+      names.attr = c("0", "0.4", "0.8"),
+      col.regions = greys, scales = list(draw = TRUE),
+      layout = c(3,1), xlim = c(1,23)) #Fig. 3.9
```

The second argument of spplot() is the name(s) of the data field(s) of the SpatialPolygonsDataFrame object Y.data that are to be plotted; since there is more than one data field in this case, spplot() plots them each in a separate panel. The third argument provides a means for labeling the individual panels. The value of xlim is set to leave empty space on either side of the plots so that they do not run together.

So which is better, plot() or spplot()? Before taking this on, I should also note that you can also plot spatial objects with ggplot(). Again, I have found this to be best for quick maps in data exploration (along with the sf version of plot()). As far as the question that begins the paragraph, I don't think that there is a single correct answer, so the book provides numerous examples that you can use to learn both. The function spplot() is better for figures that show multiple panels, such as Figure 3.9. In some cases I have found that the capacity of plot() to build graphs by successive applications of low level functions can be used to advantage.

2.6.5 Using Color in R

In order to keep the cost of the book as low as possible, all of the figures are rendered in gray scale. Color versions of a selection of the figures are provided on the book's companion website, http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm. The figures on the website are examples where interpretation is enhanced through the use of color, and code is provided for all figures. It is important to have a facility with color images, both because this format is often available for publications and because even when it is not, color images are often much more useful both for exploration and presentation than gray scale ones. Fortunately, R makes it very easy to create attractive color images. Before discussing these, it is useful to review some concepts of color cartography. Although we commonly create colors using the red—green—blue (RGB) color model, in which each color is a mixture of intensities of these colors, an alternative color model called the hue- value- saturation model is also useful (Lo and Yeung, 2007, p. 260; Tufte, 1990, p. 92). In this model, hue is the color property associated with the spectral wavelength or mixture of wavelengths (the "color"), value is the degree of lightness or darkness (the "shade"), and saturation is the degree of richness. Hue is the most visually "interesting" property, and differences in hue are easily perceived when placed next to each other. For this reason, hue is generally considered the best means with which to represent categorical quantities. Value is considered the most effective means to represent numerical quantities because we can more easily perceive ordinal differences in values than in hues. These are not hard-and-fast rules but rather guidelines, and indeed we will have occasion to violate them below.

The base R package provides a number of color palettes in the form of functions: `rainbow()`, `heat.colors()`, `terrain.colors()`, `topo.colors()`, and `cm.colors()` (for cyan-magenta). Each of these requires an argument `n`, which indicates the number of colors in the palette. Use `?rainbow` to see information on all of them. As an example, Figure 1.1 includes a topographical map of California, so one is tempted to try either `topo.colors()` or `terrain.colors()` on it. The statement

```
> image(ca.elev, col = topo.colors(10), axes = TRUE)
```

produces a map that is not particularly attractive, at least to my eye. The statement

```
> image(ca.elev, col = terrain.colors(10), axes = TRUE)
```

is better, but the lower elevations are brown and the upper elevations are green, which, although it accurately reflects the color of vegetation in California during the summer, is not the form in which elevation is typically displayed. We can reverse the colors by creating a vector `col.terrain` using this statement.

```
> col.terrain <- terrain.colors(10)[10:1]
> image(ca.elev, col = col.terrain, axes = TRUE)
```

This renders the lower elevations as green and the upper elevations as brown, but creates a green background. The background is the highest color value, so we can obtain a white background with the statement

```
> col.terrain[10] <- "white"
```

This creates an intuitive elevation map. We can then locate the survey points with a nice, contrasting color like red.

```
> plot(Set2.plot, pch = 16, cex = 0.5, add = TRUE, col = "red")
```

Note here that our use of brown and green to represent altitude violates the guideline that an interval scale quantity (elevation) should be represented by color value and not hue. The very use of the term "terrain colors" indicates that our eye is used to seeing maps in which green and brown represent terrain, and therefore can easily interpret an elevation map using this color scale.

We would usually like to plot "quantitative" quantities using different values of the same hue. We can create these different values by mixing hues with white using the function `colorRampPalette()` as described by Bivand et al. (2013, p. 79). The code to do this for Figure 3.9 is a very simple modification of that given in the previous section.

```
> rw.colors <- colorRampPalette(c("red", "white"))
> spplot(Y.data, c("Y0", "Y4", "Y8"),
+     names.attr = c("0", "0.4", "0.8"),
+     col.regions = rw.colors(20), scales = list(draw = TRUE),
+     layout = c(3,1))
```

The function `rainbow()` produces different hues, which may be used for categorical variables. Let's try them on Figure 1.4, which displays land-use categories.

```
spplot(data.Set1.cover, "VegType", col.regions = rainbow(10),
   main = "Land Use, Northern End, 1997",
   xlim = c(576000,580500), ylim = c(4417400,4421000),
   sp.layout = map.layout)
```

This produces a map that satisfies the theory of different hues for different categories. There are, however, two problems. The first is that the map looks pretty garish. The second is that, although by coincidence the river water is blue, none of the other classes have colors that correspond to what we think they should look like. Moreover, there is nothing to distinguish the suitable class, riparian, from the others. We can make a more attractive and intuitive map by using hues that correspond to our expectations and distinguishing between them using values. One of the most popular R color systems is the package `RColorBrewer` (Neuwirth, 2014). This provides a set of palettes, and the paper of Brewer et al. (2003) provides a discussion of their appropriate use. We can use the package to modify the map of Figure 1.4 as follows. We will make the water shades of blue, the "natural" vegetation shades of green (except for riparian forest, the cuckoo habitat), the cropped vegetation and developed areas shades of purple, the gravel bar tan, and the riparian forest yellow (to make it stand out). To do this we will use two of the `RColorBrewer` palettes, `BrBG`, which ranges from brown to blue to blue-green, and `PiYG`, which ranges from purple to green.

```
> brown.bg <- brewer.pal(11,"BrBG")
> purple.green <- brewer.pal(11,"PiYG")
> color.array <- c(
+    purple.green[8], #Grassland
```

```
+     purple.green[4], #Cropland
+     purple.green[3], #Developed
+     brown.bg[7], #Freshwater Wetland
+     brown.bg[4], #Gravel Bar
+     brown.bg[8], #Lacustrine
+     purple.green[2], #Orchard
+     brown.bg[7], #Riverine
+     "yellow", #Riparian Forest
+     purple.green[11]) #Oak Woodland
> spplot(data.Set1.cover, "VegType", col.regions = color.array,
+     main = "Land Use, Northern End, 1997",
+     xlim = c(576000,580500), ylim = c(4417400,4421000),
+     sp.layout = map.layout)
```

Good color selection in cartography is, of course, an art.

2.7 Continuing on from Here with R

As you continue to work with R, you will find two things to be true: first, there is always more to learn; and second, if you stand still you will fall behind. To deal with the first issue, we must address the question of where one goes to continue learning. Of course, Google is always a good option. When I have a specific question, I can almost always find the answer there or on the specialized site https://rseek.org/. Beyond that, when you are starting out with R one of the best things you can do is to subscribe to the R-help mailing list, together with any of the special interest mailing lists that may be appropriate (e.g., R-SIG-Geo for spatial data analysis and R-SIG ecology for ecological data analysis). These are available via the website https://www.r-project.org/mail.html. You will see that they are all quite active. At first you may feel a bit overwhelmed by technical questions that you don't understand, but don't despair! You will gradually find these to be a great source for answers to questions that you didn't realize you had. Another good source is the R Bloggers site https://www.r-bloggers.com/.

A good deal of information is available directly from R. In addition to the `help.start()` function, the function `help.search()` uses fuzzy matching to search R for documentation matching the character string entered in the argument of the function (in quotes); use `?help.search` to obtain details. The function `R.SiteSearch()` searches for matching words or phrases in the R-help mailing list archives, R manuals, and help pages. Finally, many R packages contain one or more *vignettes*, which provide documentation and examples. A list of available vignettes is accessible by typing `vignette()`, and specific vignettes can then be displayed by typing `vignette("name")` where `name` is the name of the vignette.

Now for the challenges. R is very dynamic. It is driven entirely by its users, and these users are constantly making changes an improvements in the software that they have contributed. New packages appear all the time, and old ones are modified. The discussion lists described in the previous paragraphs are a good way to keep up with these changes. One consequence of this dynamism is that you will occasionally make a call to a function and get a warning that it has been *deprecated*. This means that the maintainer of the package has stopped supporting the function, generally because it has been replaced by something better, but that the maintainer is still making it available to give users a time to switch. Ordinarily the announcement will contain a reference to the new function, and you will just have to learn how to use it.

2.8 Further Reading

There are many excellent books describing the R programming environment and its use in statistical analysis; indeed, the number of such books is increasing exponentially, and we will only mention a subset. Historically, R was created as an open source version of the language S, developed by Bell Labs (Becker et al., 1988). The S language evolved into a commercial product called S-Plus. Good entry points to R are Venables et al. (2006, 2008). Spector (1994) also provides a good introduction to the R and S-Plus languages, and Crawley (2007) provides almost encyclopedic coverage. Venables and Ripley (2002) is the standard text for an introduction to the use of R in statistical analysis, and Heiberger and Holland (2004) also provide an excellent, and somewhat more detailed, discussion. For those moving from other languages, Marques de Sá (2007) can be useful as it provides a sort of Rosetta stone between R and other statistical packages. Rizzo (2008) discusses a broad range of topics at a slightly higher level. Leisch (2004) provides a discussion of S4 classes, used in the `sp` package, in R.

As an open source project, R makes all of its source code available for inspection. Some R functions are written in R, and these can be accessed directly by typing the name of the function, as was illustrated in Section 2.2 with the function `sd()`. Many R functions, however, are programmed in a compiled language such as C++. Ligges (2006) provides an extensive discussion on how to retrieve source code for any R function.

The best source for R graphics is Murrell (2006). Sarkar (2008) is an excellent source for trellis graphics. Bivand et al. (2013) provide an excellent introduction to graphics with the `sp` functions. Wickham and Grolemund (2017) is the standard printed reference for `ggplot()`, although you can learn most of what you need to know just by googling "ggplot."

For those unfamiliar with GIS concepts, Lo and Yeung (2007) provide an excellent introduction. In addition to Lo and Yeung (2007), which this book uses as the primary GIS reference, there are many others that provide excellent discussions of issues such as map projections, Thiessen polygons, spatial resampling, and so forth. Prominent among these are de Smith et al. (2007), Bonham-Carter (1994), Burrough and McDonnell (1998), Clarke (1990), and Johnston (1998). Good GIS texts at a more introductory level include Chang (2008), Demers (2009), and Longley et al. (2001).

Exercises

All of the exercises are assumed to be carried out in the R programming environment, running under Windows.

2.1 Venables and Ripley (2002) make the very cogent point that one of the first things to learn about a programming environment is how to stop it from running away. First type `i <- 1:20` to create an index. Then type the statement `while (i < 5) i <- 2`. What happens? Why does this happen? To stop the execution, either hit the <Esc> key, or, if you are running RStudio, click on the little Stop sign that appears in the upper right corner of the Console. This is how you "bail out" of any R execution in Windows.

2.2 Like all programming languages, mathematical operators in R have an *order of precedence*, which can be overridden by appropriate use of parentheses. (a) Type the following expressions and observe the result:

```
3 * 4 ^ 2 + 2
(3 * 4) ^ 2 + 2
-1:2
-(1:2)
```

(b) Type `1:3 - 1` and observe the result.

(c) Type the statement `w <- 1:10` to create a vector `w`. Type the statement `w > 5`. The R symbol for equality used in a test is the double equal sign ==. Type w == 5. The R symbol for inequality used in a test is `!=`. Type `w != 5`.

2.3 (a) Using a single R statement, create a matrix `A` with three rows and four columns whose elements are the row number (i.e., every element of row 1 is 1, every element of row 2 is 2, and every element of row 3 is 3) (hint: make sure you read the Help file `?matrix`). Display the matrix, then display the second row of the matrix, then display every row but the first.

(b) In a single statement using the function `cbind()`, create a matrix `B` whose first column is equal to the first column of `A` and whose second column is five times the first column of `A`.

(c) Create a data frame `C` from the matrix `B` and then give the fields of `C` the names "`C1`" and "`C2`."

(d) Create a list `L` whose first element `L1` is the vector `w`, whose second element `L2` is the matrix `B`, and whose third element `L3` is the data frame `C`. Display L$L3$C2. Now display the same object using the `[[]]` and `[]`operators (*note*: in R nomenclature, these operators are denoted `[[` and `[`. To look them up in Help, type `?"["` and `?"[["`.

2.4 (a) Use the function `c()` to create a character vector `w` whose elements are the number 6, 2.4, and 3. Then create a vector `z` whose elements are the characters "*a*," "*b*," and "*c*." Note that in R characters are always enclosed in quotes.

(b) Use the function `c()` to create a vector `z` that is the concatenation of the vectors `w` and *z* created in part (a). Evaluate the vector `z`. What has happened to the elements of the vector `w`?

(c) Apply the function `as.numeric()` to the vector *z* to recover the numerical values of the vector *w* of part (a). What happens to the components of the vector *z*?

2.5 In Section 2.3.1, we created a vector *w* whose elements were 1:10, and then we used two code sequences to convert the first five of these elements to 0 and the second five to 20. Repeat this code exactly, but instead of converting the second five elements to 20, attempt to convert them to 3. Do the two code sequences give the same result? If not, create a second code sequence that gives the same results as the first for and values of the two replacement quantities 20 and 3.

2.6 (a) In Section 2.3.2, we used the functions `rep()` and `sort()` to create a sequence of five ones, five twos, five threes, and five fours. In this exercise we will use another approach. First, use the constructor function `numeric()` to create a vector *z* of 20 zeroes.

(b) Use the *modulo* operator `%%` to assign a value of 1 to every fifth component of the vector *z*. The modulo operation `u %% v` evaluates to *u* mod *v*, which, for integer values of *u* and *v* is the remainder of the division of *u* by *v*. Thus, for

example 6 mod 5 = 1. The only numbers i such that i mod 5 = 0 are 0, 5, 10, etc., and therefore the components of the vector z with these indices are assigned the value 1. To read about an operator using special characters, you must place it in quotes, so you type `?"%%"`.

(c) Next, use the R function `cumsum()` to compute the index vector. For a vector z, the ith component of *cumsum*(z) is $\sum_{k=1}^{i} z_k$.

2.7 (a) R has some built-in numbers, including `pi`. Read about the function `print()`, and then evaluate pi to 20 significant digits.

(b) Read about the function `sin()`. Read about the function `seq()` and create a vector w whose components run from 0 to 6 in steps of 0.25 (i.e., 0, 0.25, 0.5,0.75, …, 6.0) Now set `n` equal to 0.5 and create a vector `z1` by evaluating $\sin(n\pi w)$.

(c) Create a new function `sin.npi(w)` defined as `sin(n * pi * w)`. Do *not* pass n in the argument of the function; instead, use lexical scoping to determine it. Create a vector `z2` by the statement `z2 <- sin(n * pi * w)`. Use the function `cbind()` to create an array `z` whose columns are `z1` and `z2`, and visually compare them to see that they are equal.

(d) Read about the function `all.equal()` and use this to test the equality of `z1` and `z2`.

(e) Create a function `e.m.sin(w)` defined as `exp(-sin.npi(w))` (note that because of lexical scoping you do not have to pass the function `sin.npi(w)` as an argument). Repeat parts (b) and (c) with this new function.

2.8 (a) R contains many built-in data sets. These are accessed using the function `data()`. Read about this function in the usual way by typing `?data`. Then type `data` to read the list of available data sets. One of the data sets, `UKgas`, which gives quarterly gas consumption in the UK between 1960 and 1986. Type `help(UKgas)` to read about this data set.

(b) We would like to compute and plot the mean UK gas consumption by year. The `UKgas` data set is a member of the `ts` (time series) object class. Although we will occasionally deal with time series in this book, we will not use this object class. Therefore we will convert `UKgas` to a matrix `UKg`. Use the function `matrix()` to accomplish this (hint: you can ignore the year column in the `ts` object in this conversion).

(c) Try typing `mean(UKg)` to compute the yearly mean. That clearly doesn't work. Instead, we will use the function `apply()`. Type `?apply` to read about this function and then use it to compute a vector `UKgm` of mean annual gas consumption. Check your results by computing the mean consumption in one particular year using a different method.

(d) Create a data frame `UKg.df` from the matrix `UKg`. Then add a field `mean` from `UKgm` that contains the mean consumption. Finally, add a field `year` that contains the year. Give these fields the names *Winter*, *Spring*, *Summer*, *Fall*, *Mean*, and *Year*.

(e) Repeat part (c) computing the mean gas consumption over years by quarter.

2.9 (a) Use the function `plot()` to plot a graph of mean UK gas consumption against year for the years 1960 through 1986 as computed in Problem 2.8.

(b) For these data, `plot()` automatically plots points. Type the correct statement to plot a line graph instead, with the title "Mean UK Gas Consumption," the abscissa label "Year," and the ordinate label "Mean Consumption."

(c) Read about the function `points()` and then add points to indicate the gas consumption in the winter of each year. This causes a problem. Adjust the scale of the ordinate, find the maximum consumption over all quarters, and replot the graph with an ordinate scale a bit larger than this.

(d) Plot the data points for all the seasons. Use the argument `pch` to distinguish seasons.

(e) R has a function `legend()` that we will discuss in part (f), but for now we will make a key by hand. In the upper left, at about $w = 1965$, $z = 1000$, plot a single point using the same value of `pch` as you used for the `Winter` data. Next to this, use the function `text()` to insert the word "Winter." Make sure you read about the argument `pos` of the function `text()` and specify it correctly to locate the text to the right of the symbol. Repeat this process to insert the appropriate symbol and text for the other seasons at appropriate locations.

(f) Now read about the function `legend()` and redraw the plot using this function in place of the handmade legend of part (e).

2.10 One of the consequences of polymorphism is that there may be more than one version of the same function available. To see how you can find all available versions of a function, first load the `raster` package by typing `library(raster)`. Next type `getAnwhere("plot")`. Next, type `raster::plot` and `graphics::plot`. The double colon is used to specify membership of a function in a class. Now type `library(sf)`. The plotting function in the sf package is called `plot _ sf()`, but the package will also understand the command `plot()`. Type `?plot`. Try typing `?plot.sf` and `?plot.raster` to see another way of seeing the help file for a specific instance of a function.

2.11 Field 1 of Data Set 4 has a trapezoidal shape. The UTM coordinates of the western, southern, and northern boundaries are approximately 592025, 4270452, and 4271132, respectively. The Easting at the northern boundary is 592470, and at the southern boundary 592404. Create an `sf` polygon object describing the boundary of this field and save it as an ESRI shapefile named *Set4.19697bdry.shp*.