4

PROGRAMMING IN R

4.1 OVERVIEW

As you have been working through the code and exercises in this book you have applied a number of different tools and techniques for extracting, displaying and analysing data. In places you have used some quite advanced snippets of code. However, this has all been done in a step-by-step manner, with each line of code being run individually, and the occasional function has been applied individually to a specific dataset or attribute. Quite often in spatial analysis, we would like to do the same thing repeatedly, but adjusting some of the parameters on each iteration – for example, applying the same algorithm to different data, different attributes, or using different thresholds. The aim of this chapter is to introduce some basic programming principles and routines that will allow you to do many things repeatedly in single block of code. This is the basics of writing computer programs. This chapter will:

- Describe how to combine commands into loops
- Describe how to control loops using `if`, `else`, `repeat`, etc.
- Describe logical operators to index and control
- Describe how to create functions, test them and to make them universal
- Explain how to automate short tasks in R

No previous knowledge of programming is required for you to understand the content of this chapter. Different concepts will be introduced with worked examples, supported by snippets of code, and by working through these, at the end of this chapter you should understand key principles of programming and be able to apply these to spatial information processing problems. If you have no previous experience of programming *do not worry*. By developing basic competence in R, you will get used to using code blocks or groups of commands, sometimes combined into functions. If you have experience in programming in another language, then this chapter will serve to introduce the R syntax.

4.2 INTRODUCTION

In spatial data analysis and mapping, we frequently want to apply the same set of commands over and over again, to cycle through data or lists of data and do things to data depending on whether some condition is met or not, and so on. These types of repeated actions are supported by *functions, loops* and *conditional statements*. A few simple examples serve to illustrate how R programming combines these ideas through functions with conditional commands, loops and variables.

For example, consider the following variable `tree.heights`:

```
tree.heights <- c(4.3,7.1,6.3,5.2,3.2)
```

We may wish to print out the first element of this variable if it has a value less than *6*: this is a *conditional command* as the operation (in this case to print something) is carried out conditionally (i.e. if the condition is met):

```
tree.heights

## [1] 4.3 7.1 6.3 5.2 3.2

if (tree.heights[1] < 6) { cat('Tree is small\n') } else
  { cat('Tree is large\n')}

## Tree is small
```

Alternatively, we may wish to examine all of the elements in the variable `tree.heights` and, depending on whether each individual value meets the condition, perform the same operation. We can carry out operations repeatedly using a *loop* structure as below. Notice the construction of the `for` loop in the form `for(variable in sequence) {R expression}`.

```
for (i in 1:3) {
    if (tree.heights[i] < 6) { cat('Tree',i,' is small\n') }
    else { cat('Tree',i, 'is large\n')} }

## Tree 1 is small
## Tree 2 is large
## Tree 3 is large
```

A third situation is where we wish to perform the same set of operations, group of conditional or looped commands over and over again, perhaps to different data. We can do this by grouping code and defining our own *functions*:

```
assess.tree.height <- function(tree.list, thresh)
    { for (i in 1:length(tree.list))
    { if(tree.list[i] < thresh) {cat('Tree',i, ' is small\n')}
    else { cat('Tree',i, ' is large\n')}
    }
}
assess.tree.height(tree.heights, 6)

## Tree 1 is small
## Tree 2 is large
## Tree 3 is large
## Tree 4 is small
## Tree 5 is small

tree.heights2 <- c(8,4.5,6.7,2,4)
assess.tree.height(tree.heights2, 4.5)

## Tree 1 is large
## Tree 2 is large
## Tree 3 is large
## Tree 4 is small
## Tree 5 is small
```

Notice how the code in the function `assess.tree.height` above modifies the original loop: rather than `for(i in 1:3)` it now uses the length of the variable `1:length(tree.list)` to determine how many times to loop through the data. Also a variable `thresh` was used for whatever threshold the user wishes to specify.

The sections in this chapter develop more detailed ideas around functions, loops and conditional statements, and the testing and debugging of functions, in order to support automated analyses in R.

4.3 BUILDING BLOCKS FOR PROGRAMS

In the examples above a number of programming concepts were introduced. Before we start to develop these more formally into functions, it is important to explain these *ingredients* in a bit more detail.

4.3.1 Conditional Statements

Conditional statements test to see whether some *condition* is `TRUE` or `FALSE`, and if the answer is `TRUE` then some specific actions are undertaken. Conditional statements are composed of `if` and `else`.

The `if` statement is followed by a *condition*, an expression that is evaluated, and then a *consequent*, to be executed if the condition is `TRUE`. The format of an `if` statement is:

`If`–condition–consequent

Actually this could be read as 'if the condition is true then the consequent is...'. The components of a `conditional statement` are:

- the condition, an R expression that is either `TRUE` or `FALSE`
- the consequent, any valid R statement which is only executed if the condition is TRUE

For example, consider the simple case below where the value of `x` is changed and the same condition is applied. The results are different (in the first case a statement is printed to the console, in the second it is not), because of the different values assigned to `x`.

```
x <- -7
if (x < 0) cat("x is negative")

## x is negative

x <- 8
if (x < 0) cat("x is negative")
```

Frequently `if` statements also have an *alternative consequent* that is executed when the condition is `FALSE`. Thus the format of the *conditional statement* is expanded to

`If`–condition–consequent–`else`–alternative

Again, this could be read as 'if the condition is true then do the consequent; or, if the condition is not true then do the alternative'. The components of a `conditional statement` that includes an `alternative` are:

- The *condition*, an R expression that is either `TRUE` or `FALSE`;
- The *consequent* and *alternative*, which can be any valid R statements;
- The *consequent* is executed if the *condition* is `TRUE`;
- The *alternative* is executed if the *condition* is `FALSE`.

The example is expanded below to accommodate the alternative:

```
x <- -7
if (x < 0) cat("x is negative") else cat("x is positive")

## x is negative

x <- 8
if (x < 0) cat("x is negative") else cat("x is positive")

## x is positive
```

The condition statement is composed of one or more *logical operators*, and in R these are defined as follows:

Logical operator	Description
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
!	Not (goes in front of other expressions)
&	And (combines expressions)
|	Or (combines expressions)

In addition, R contains a number of *logical functions* which can also be used to evaluate conditions. A sample of these are listed below, but many others exist.

Logical function	Description
`any(x)`	`TRUE` if any in a vector of conditions x is true
`all(x)`	`TRUE` if all of a vector of conditions x is true
`is.numeric(x)`	`TRUE` if x contains a numeric value
`is.character(x)`	`TRUE` if x contains a character value
`is.logical(x)`	`TRUE` if x contains a true or false value

There are quite a few more `is`-type functions (i.e. `logical` evaluation functions) that return `TRUE` or `FALSE` statements that can be used to develop conditional tests. To explore these enter:

```
??is.
```

The examples below illustrate how the logical tests `all` and `any` may be incorporated into conditional statements:

```
x <- c(1,3,6,8,9,5)
if (all(x > 0)) cat("All numbers are positive")

## All numbers are positive

x <- c(1,3,6,-8,9,5)
if (any(x > 0)) cat("Some numbers are positive")

## Some numbers are positive

any(x==0)

## [1] FALSE
```

4.3.2 Code Blocks

Frequently we wish to execute a group of consequent statements together if, for example, some condition is `TRUE`. Groups of statements are called *code blocks*, and in R are contained by { and }. The examples below show how code blocks can be used if a condition is `TRUE` to execute consequent statements and can be expanded to execute alternative statements if the `Condition` is `FALSE`.

```
x <- c(1,3,6,8,9,5)
if (all(x > 0)) {
    cat("All numbers are positive\n")
    total <- sum(x)
    cat("Their sum is",total) }

## All numbers are positive
## Their sum is            32
```

The curly brackets are used to group the consequent statements: that is, they contain all of the actions to be performed if the condition is met is `TRUE` and all of the alternative actions if the condition is not met (i.e. is `FALSE`):

```
if condition { consequents } else { alternatives }
```

These are illustrated in the code below:

```
x <- c(1,3,6,8,9,-5)
if (all(x > 0)) {
    cat("All numbers are positive\n")
    total <- sum(x)
    cat("Their sum is",total) } else {
    cat("Not all numbers are positive\n")
    cat("This is probably an error\n")
    cat("as numbers are rainfall levels") }

## Not all numbers are positive
## This is probably an error
## as numbers are rainfall levels
```

4.3.3 Functions

Section 4.2 included a function called `assess.tree.height`. The format of a function is:

```
function name <- function(argument list) { R
expression }
```

The R expression is usually a code block and in R the code is contained by curly brackets or braces: { and }. Wrapping the code into a `function` allows it to be used without having to retype the code each time you wish to use it. Instead, once the function has been defined and compiled, it can be called repeatedly and with different arguments or parameters. Notice in the function below that there are a number of sets of containing brackets { } that are variously related to the function, the consequent and the alternative.

```
mean.rainfall <- function(rf)
{ if (all(rf> 0))              #open Function
  { mean.value <- mean(rf)     #open Consequent
    cat("The mean is ",mean.value)
  } else                       #close Consequent
    {cat("Warning: Not all values are positive\n") #open Alternative
    }                          #close Alternative
  }                            #close Function
mean.rainfall(c(8.5,9.3,6.5,9.3,9.4))

## The mean is  8.6
```


More commonly functions are defined that do something to the input specified in the *argument list* and return the result, either to a variable or to the console window, rather than just printing something out. This is done using `return()` within the function. Its format is

```
return( R expression )
```

Essentially what this does if it is used in a function is to make `R expression` the value of the function. In the following the `mean.rainfall` function now returns the mean of the data passed to it, and this can be assigned to another variable:

```
mean.rainfall2 <- function(rf) {
if (all(rf> 0)) {
    return( mean(rf))} else {
    return(NA)}
    }
mr <- mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))
mr

## [1] 8.6
```

Notice that the code blocks used in the functions contained within the curly brackets { and } are indented. There are a number of commonly accepted protocols for doing this, but no unique one. The aim is to make the code and the nesting of sub-clauses indicated by { and } clear. In the code for `mean.rainfall` above, { is used before the first line of the code block, whereas for `mean.rainfall.2` the { is positioned immediately after the function declaration.

It is possible to declare variables inside functions, and you should note that these are distinct from external variables with the same name. Consider the internal variable `rf` in the `mean.rainfall2` function above. Because this is a variable that is *internal* to the function, it only exists *within* the function and will not alter any *external* variable of the same name. This is illustrated in the code below.

```
rf <- "Tuesday"
mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))

## [1] 8.6

rf

## [1] "Tuesday"
```

4.3.4 Loops and Repetition

Very often, we would like to run a code block a certain number of times, for example for each record in a data frame or a spatial data frame. This is done using `for` loops. The format of a loop is:

```
for( 'loop variable' in 'list of values' ) do R expression
```

Again, typically code blocks are used as in the example of a `for` loop:

```
for (i in 1:5) {
    i.cubed <- i * i * i
    cat("The cube of",i, "is",i.cubed, "\n")}

## The cube of 1 is 1
## The cube of 2 is 8
## The cube of 3 is 27
## The cube of 4 is 64
## The cube of 5 is 125
```

When working with a data frame and other tabular-like data structures, it is common to want to perform a series of R expressions on each row, on each column or on each data element. In a `for` loop the `'list of values'` can be a simple sequence of 1 to *n* (`1:n`), where *n* is related to the number of rows or columns in a dataset of the data or the length of the input variable as in the `assess.tree.height` function above.

However, there are many other situations when a different `'list of values'` is required. The function `seq` is a very useful helper function that generates number sequences. It has the following formats:

```
seq(from, to, by = step value)
```

or

```
seq(from, to, length = sequence length)
```

In the example below, it is used to generate a sequence of 0 to 1 in steps of 0.25:

```
for (val in seq(0,1,by=0.25)) {
    val.squared <- val * val
    cat("The square of",val, "is",val.squared, "\n")}

## The square of 0 is 0
## The square of 0.25 is 0.0625
```



```
## The square of 0.5 is 0.25
## The square of 0.75 is 0.5625
## The square of 1 is 1
```

`Conditional` loops are very useful when you wish to run a code block until a certain condition is met. In R these are specified using the `repeat` and `break` functions. Here is an example:

```
i <- 1; n <- 654
repeat{
    i.squared <- i * i
    if (i.squared > n) break
    i <- i + 1 }
cat("The first square number exceeding",n, "is ",i.squared,
"\n")

## The first square number exceeding 654 is 676
```

Notice in the above example that the first line of the code makes two statements separated by a semi-colon `';'`. Although it is possible to link many lines in this way, it is advisable to do this only occasionally and to link only simple snippets of code as above.

Finally, it is possible to include loops in functions as in the following example with a conditional loop:

```
first.bigger.square <- function(n) {
    i <- 1
    repeat{
      i.squared <- i * i
      if (i.squared > n) break
      i <- i + 1 }
    return(i.squared)}
first.bigger.square(76987)

## [1] 77284
```

4.3.5 Debugging

As you develop your code and compile it into functions, especially initially, you will probably encounter a few teething problems: hardly any function of reasonable size works first time! There are two general kinds of problem:

- The function crashes (i.e. it throws up an error)
- The function does not crash, but returns the wrong answer

Usually the second kind of error is the worst. *Debugging* is the process of finding the problems in the function. A typical approach to debugging is to 'step' through the function line by line and in so doing find out where a crash occurs, if one does. You should then check the values of variables to see if they have the values they are supposed to. R has tools to help with this.

To debug a function

- Enter `debug(function name)`
- Then call the function

For example, enter:

```
debug(mean.rainfall2)
```

Then just use the function you are trying to debug and R goes into 'debug mode':

```
mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))

## [1] 8.6
```

You will notice that the prompt becomes `Browse>` and the line of the function about to be executed is listed. You should note a number of features associated with `debug`:

- Entering a return executes it, and debug goes to next line
- Typing in a variable lists the value of that variable
- R can 'see' variables that are specific to the function
- Typing in any other command executes that command

When you enter `c` the return runs to the end of a loop/function/block. Typing in `Q` exits the function.

To return to normal

- Enter `undebug(function name)`

A final comment is that learning to write functions and programming is a bit like learning to drive – you may 'pass the test' but you will become a good driver by spending time behind the wheel. Similarly, the best way to learn to write functions is to practise, and the more you practise the better you will get at programming. You should try to set yourself various function writing tasks and examine the functions that are introduced throughout this book. Most of the commands that you use in R are functions that can themselves be examined: entering them without any brackets afterwards will reveal the blocks of code they use. Have a look at the `ifelse` function by entering at the R prompt:

```
ifelse
```

This allows you to examine the code blocks, the control, etc. in existing functions.

4.4 WRITING FUNCTIONS

4.4.1 Introduction

In this section you will gain some initial experience in writing functions that can be used in R, using a number of coding illustrations. You should enter the code blocks for these, compile them and then run them with some data to build up your experience. Unless you already have experience in writing code, this will be your first experience of programming. This section contains a series of specific tasks for you to complete in the form of self-test questions. The answers to the questions are provided in the final section of the chapter.

In the preceding section, the basic idea of writing functions was described. You can write functions directly by entering them at the R command line:

```
cube.root <- function(x) {
    result <- x ^ (1/3)
    return(result)}
cube.root(27)

## [1] 3
```

Note that ^ means 'raise to the power' and recall that a number to the power of one third is its cube root. The cube root of 27 is 3, since 27 = 3 × 3 × 3, hence the answer printed out for `cube.root(27)`. However, entering functions from the command line is not always very convenient:

- If you make a typing error in an early line of the definition, it is not possible to go back and correct it
- You would have to type in the definition every time you used R

A more sensible approach is to type the function definition into a text file. If you write this definition into a file – calling it, say, `functions.R` – then you can load this file when you run R, without having to type in the whole definition. Assuming you have set R to work in the directory where you have saved this file, just enter:

```
source("functions.R")
```

This has the same effect of entering the entire function at the command line. In fact any R commands in a file (not just function definitions) will be executed when the `source` function is used. Also, because the function definition is edited in a file, it is always possible to return to any typing errors and correct them – and if a function contains an error, it is easy to correct this and just redefine the function by re-entering the command above. The built-in R editor for writing and saving code was introduced in Chapter 1.

Open a text-editing window. In the new window, enter in the code for the program:

```
cube.root <- function(x) {
    result <- x ^ (1/3)
    return(result)}
```

Then use **Save As** to save the file as `functions.R` in the directory you are working in. In R you can now use `source` as described:

```
source('functions.R')
cube.root(343)
cube.root(99)
```

Note that you can type in several function definitions in the same file. For example, underneath the code for the `cube.root` function, you should define a function to compute the area of a circle. Enter:

```
circle.area <- function(r) {
    result <- pi * r ^ 2
    return(result)}
```

If you save the file, and enter `source('functions.R')` to R again then the function `circle.area` will be defined as well as `cube.root`. Enter:

```
source('functions.R')
cube.root(343)
circle.area(10)
```


4.4.2 Data Checking

One issue when writing functions is making sure that the data that have been given to the function are the right kind. For example, what happens when you try to compute the cube root of a negative number?

```
cube.root(-343)

## [1] NaN
```

That probably wasn't the answer you wanted. `NaN` stands for 'not a number', and is the value returned when a mathematical expression is numerically indeterminate. In this case, this is actually due to a shortcoming with the `^` operator in R, which only works for positive base values. In fact –7 is a perfectly valid cube root of –343, since `(-7) × (-7) × (-7) =-343`. In fact we can state a conditional rule:

- If $x \geq 0$: Calculate the cube root of x normally
- Otherwise: Use `cube.root(-x)`

That is, for cube roots of negative numbers, work out the cube root of the positive number, then change it to negative. This can be dealt with in an R function by using an `if` statement:

```
cube.root <- function(x) {
    if (x >= 0) {
      result <- x ^ (1/3) } else {
      result <- -(-x) ^ (1/3) }
    return(result)}
```

Now you should go back to the text editor and modify the code in `functions.R` to reflect this. You can do this by modifying the original `cube.root` function. You can now save this edited file, and use `source` to reload the updated function definition. The function should work with both positive and negative values.

```
cube.root(3)

## [1] 1.442

cube.root(-3)

## [1] -1.442
```

Next, try debugging the function – since it is working properly, you will not (hopefully!) find any errors, but this will demonstrate the debug facility. Enter:

```
debug(cube.root)
```

at the R command line (not in the file editor!). This tells R that you want to run `cube.root` in debug mode. Next, enter:

```
cube.root(-50)
```

at the R command line and see how repeatedly pressing the return key steps you through the function. Note particularly what happens at the `if` statement.

At any stage in the process you can type an R expression to check its value. When you get to the `if` statement enter:

```
x > 0
```

at the command line and press return to see whether it is true or false. Checking the value of expressions at various points when stepping through the code is a good way of identifying potential bugs or glitches in your code. Try running through the code for a few other cube root calculations, by replacing –50 above with different numbers, to get used to using the debugging facility. When you are finished, enter

```
undebug(cube.root)
```

at the R command line. This tells R that you are ready to return `cube.root` to running in normal mode. For further details about the debugger, at the command line enter:

```
help(debug)
```

4.4.3 More Data Checking

In the previous section, you saw how it was possible to check for negative values in the `cube.root` function. However, other things can go wrong. For example, try entering:

```
cube.root("Leicester")
```

This will cause an error to occur and to be printed out by R. This is not surprising because cube roots only make sense for numbers, not character variables. However, it might be helpful if the cube root function could spot this and print a warning explaining the problem, rather than just crashing with a fairly obscure

error message such as the one above, as it does at the moment. Again, this can be dealt with using an **if** statement. The strategy to handle this is:

- If x is numerical: Compute its cube root
- If x is not numerical: Print a warning message explaining the problem

Checking whether a variable is numerical can be done using the `is.numeric` function:

```
is.numeric(77)
is.numeric("Lex")
is.numeric("77")
v <- "Two Sevens Clash"
is.numeric(v)
```

The function could be rewritten to make use of `is.numeric` in the following way:

```
cube.root <- function(x) {
  if (is.numeric(x)) {
    if (x >= 0) { result <- x^(1/3) }
    else { result <- -(-x)^(1/3) }
    return(result) }
  else {
    cat("WARNING: Input must be numerical, not character\n")
    return(NA)}
}
```

Note that here there is an `if` statement inside another `if` statement – this is an example of a 'nested' code block. Note also that when no proper result is defined, it is possible to return the value `NA` instead of a number (NA = 'not available'). Finally, recall that the `\n` in `cat` tells R to add a carriage return (new line) when printing out the warning. Try updating your cube root function in the editor with this latest definition, and then try using it (in particular with character variables) and stepping through it using `debug`.

An alternative way of dealing with cube roots of negative numbers is to use the R functions `sign` and `abs`. The function `sign(x)` returns a value of 1 if `x` is positive, –1 if it is negative, and 0 if it is zero. The function `abs(x)` returns the value of `x` without the sign, so for example `abs(-7)` is 7, and `abs(5)` is 5. This means that you can specify the core statement in the cube root function without using an `if` statement to test for negative values, as:

```
result <- sign(x)*abs(x)^(1/3)
```

This will work for both positive and negative values of x.

Self-Test Question 1. You should define a new function `cube.root.2` that uses this way of computing cube roots – and also include a test to make sure `x` is a numerical variable, and print out a warning message if it is not.

4.4.4 Loops Revisited

In this section, you will revisit the idea of looping in function definitions. There are two main kinds of loops in R: **deterministic** and **conditional** loops. The former is executed a fixed number of times, specified at the beginning of the loop. The latter is executed until a specific condition is met.

Conditional loops

A very old example of a conditional loop is *Euclid's algorithm*. This is a method for finding the *greatest common divisor* (GCD) of a pair of numbers. The GCD of a pair of numbers is the largest number that divides exactly (i.e. with remainder zero) into each number in the pair. The algorithm is set out below:

1. Take a pair of numbers a and b – let the *dividend* be max(a, b), and the *divisor* be min(a, b).
2. Let the *remainder* be the arithmetic remainder when the dividend is divided by the divisor.
3. Replace the dividend with the divisor.
4. Replace the divisor with the remainder.
5. If the remainder is not equal to zero, repeat from step 2 to here.
6. Once the remainder is zero, the GCD is the dividend.

Without considering in depth the reasons why this algorithm works, it should be clear that it makes use of a conditional loop. The test to see whether further looping is required occurs in step 5 above. It should also be clear that the divisor, dividend and remainder are all variables. Given these observations, we can turn Euclid's algorithm into an R function:

```
gcd <- function(a,b)
{
    divisor <- min(a,b)
    dividend <- max(a,b)
    repeat
          { remainder <- dividend %% divisor
                dividend <- divisor
                divisor <- remainder
                if (remainder == 0) break
          }
    return(dividend)
}
```


The one unfamiliar thing here is the `%%` symbol. This is just the remainder operator – the value of `x %% y` is the remainder when `x` is divided by `y`.

Using the editor, create a definition of this function, and read it in to R. You can put the definition into `functions.R`. Once the function is defined, it may be tested:

```
gcd(6,15)
gcd(25,75)
gcd(31,33)
```

Self-Test Question 2. Try to match up the lines in the function definition with the lines in the description of Euclid's algorithm. You may also find it useful to step through an example of `gcd` in debug mode.

Deterministic loops

As described in earlier sections, the form of a deterministic loop is

```
for (<VAR> in <Item1>:<Item2>)
    {
    ... code in loop...
    }
```

where <VAR> refers to the looping variable. It is common practice to refer to <VAR> in the code in the loop. <Item1> and <Item2> refer to the range of values over which <VAR> loops. For example, a function to print the cube roots of numbers from 1 to n takes the form:

```
cube.root.table <- function(n)
  {
    for (x in 1:n)
    {
      cat("The cube root of ", x, " is", cube.root(x), "\n")
    }
  }
```

Self-Test Question 3. Write a function to compute and print out `GCD(x,60)` for `x` in the range 1 to `n`. (ii) Write another function to compute and print out `GCD(x,y)` for `x` in the range 1 to `n1` and `y` in the range 1 to `n2`. In this exercise you will need to nest one deterministic loop inside another one.

Self-Test Question 4. Modify the `cube.root.table` function so that the loop variable runs from 0.5 in steps of 0.5 to `n`. The key to this is provided in the descriptions of loops in the sections above.

4.4.5 Further Activity

You will notice that in the previous example, the output is rather messy, with the cube roots printing to several decimal places – it might look neater if you could print to fixed number of decimal places. In the function `cube.root.table` replace the `cat...` line with

```
cat(sprintf("The cube root of %4.0f is %8.4f \n",x,cube.root(x)))
```

Then enter `help(sprintf)` and try to work out what is happening in the code above.

4.5 WRITING FUNCTIONS FOR SPATIAL DATA

The sections on plotting and graphics in Chapter 2 outlined a number of techniques for visualising data using R, and Chapter 3 introduced some basic techniques for analysing and displaying spatial data. The exercises in this section apply some of the techniques from Chapters 2 and 3, in conjunction with writing functions and using spatial data. In so doing, these exercises show you how to create some elementary maps in R using functions rather than line by line coding. They also outline some new R commands and techniques to help put all of this together. These exercises and examples applying functions give a flavour of how R can be used to handle geographical data, and in particular how graphics can be produced.

To begin with, you will load the `GISTools` package and the `georgia` data. However, before doing this and running the code below you need to check that you are in the correct working directory. You should already be in the habit of doing this at the start of every R session. Also, if this is not a fresh R session then you should clear the workspace of any variables and functions you have created. Recall from Chapter 3 that this can be done through the menu **Misc > Remove all objects** in Windows (or **Workspace > Clear Workspace** on a Mac) or by entering:

```
rm(list = ls())
```


Then load the `GISTools` package and the `georgia` datasets:

```
library(GISTools)
data(georgia)
```

One of the variables is called `georgia.polys`. There are two ways to confirm this. A new one is to type `ls()` into R. This function tells R to list all currently defined variables:

```
ls()
## [1] "georgia"    "georgia.polys" "georgia2"
```

The other way of checking that `georgia.polys` now exists is just to type it in to R and see it printed out.

```
georgia.polys
```

What is actually printed out has been excluded here, as it would go on for pages and pages. However, the content of the variable will now be explained. `georgia.polys` is a variable of type `list`, with 159 items in the list. Each item is a matrix of k rows and 2 columns. The two columns correspond to x and y coordinates describing a polygon made from k points. Each polygon corresponds to one of the 159 counties that make up the state of Georgia in the United States. To check this quickly, enter the code below to produce Figure 4.1.

```
plot(georgia.polys[[1]],asp=1,type='l')
```

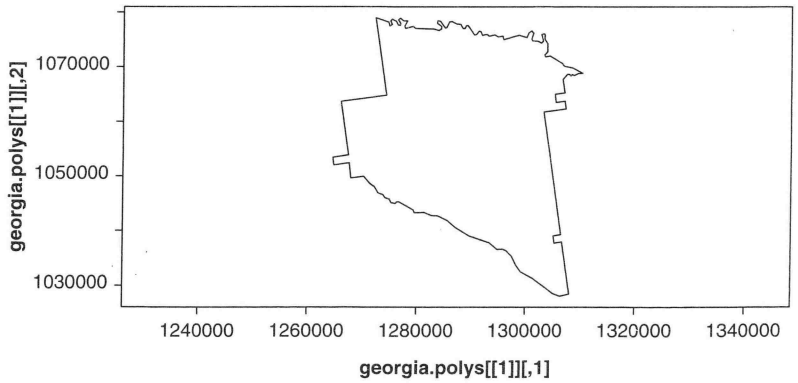

Figure 4.1 The plot produced by `plot(georgia.polys[[1]],asp=1,type='l')`

The above will not win any prizes for cartography – but it should be recognisable as Appling County, as featured in earlier chapters. In this case, the polygons are ordered alphabetically by county name and Appling happens to come first.

4.5.1 Drawing Polygons in a List

Having loaded the variable `georgia.polys`, which is a list of polygons, it would be useful to draw all of these – essentially making a map of all of the counties in Georgia. Recall that the function `polygon` draws polygons, but that it adds the polygon to an existing graph. To create the background graph, you need to use the `plot` function with the `'n'` option. A good bounding box for the whole of Georgia is

Corner	South-West	North-East
Easting	939,220 m	1,419,420 m
Northing	905,510 m	1,405,900 m

So first draw a blank plot with these limits. Then add the outlines of each of the polygons in the list. The simplest way to do this is to use `lapply` to apply the `polygon` function to each polygon in the list `georgia.polys` as in the code to below to produce Figure 4.2:

```
plot(c(939200,1419420),c(905510,1405900),asp=1,type='n')
lapply(georgia.polys,polygon)
```

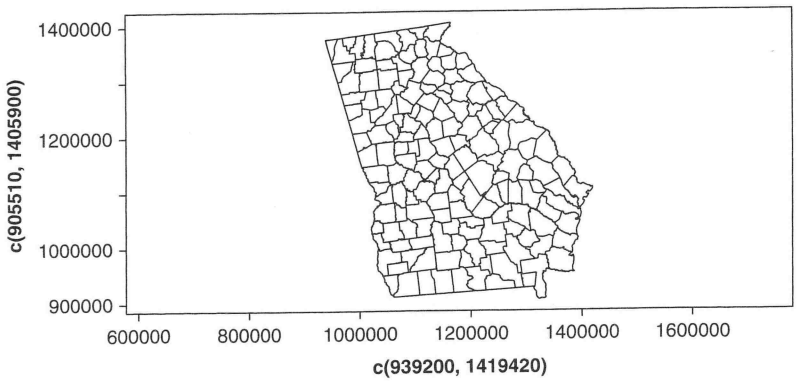

Figure 4.2 Plotting using the `lapply` function

What you will have noticed is that although this has worked, using the function caused a lot of things to be printed out. This is because `lapply` returns a list of the same length as the input list, with each element in the list corresponding to the result of applying the input function to each element in the input list. However, in this instance the function `polygon` doesn't return a value. As a result, each element in the output of `lapply` contains the value `NULL`, signifying an empty list. What you see printed out is a list of 159 `NULL` values, one for each county polygon. Since this isn't very helpful here, you can use the `invisible` function. This basically overrides the standard option of printing out the result of an expression. To do this, just enter:

```
plot(c(939200,1419420),c(905510,1405900),asp=1,type='n')
invisible(lapply(georgia.polys,polygon))
```

This has the same effect as before, but doesn't print out the result, just the map.

Self-Test Question 5. Write a function to take a polygon list, such as `georgia.polys`, and draw a map in the same way as the above example. Call it `draw.polys`. One thing you may want to adjust is the labelling on the axes. At the moment they are labelled by default with the expressions passed in the call to `plot`. In fact, it might be better to just have a blank window – basically we are trying to plot a map, not a graph!

Entering the code below will give an entirely blank window – in effect this is a graph with all of the usual annotation switched off, and the various options `xlab`, `ylab`, `xaxt`, `yaxt` and `bty` switch off displays of axes, labels and the box around the graph:

```
plot(c(939200,1419420),c(905510,1405900),asp=1,
     type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
```

Write the `draw.polys` function to use these options to create a blank window, and then plot the polygons.

4.5.2 Automatically Choosing the Bounding Box

The last result (particularly the output from the self-test question) looked more like a proper map. However, you needed to rely on a bounding box that was supplied earlier in the text. It would be useful to be able to work out the bounding box automatically given the polygon list. The R functions `min` and `max` find the largest and smallest values in a list of numbers. These can be used, on an individual polygon in the list, to find the extreme north, south, east and west coordinates. For example the code below finds the most eastern point on the boundary of polygon 1:

```
poly1 <- georgia.polys[[1]]
min(poly1[,1])

## [1] 1264520
```

The other extremes can be found by the following expressions:

Extreme point	Most northern	Most southern
R Expression	`max(poly1[,2])`	`min(poly1[,2])`
Extreme point	**Most eastern**	**Most western**
R Expression	`min(poly1[,1])`	`max(poly1[,1])`

One of these expressions could be applied to every polygon in the list to get a list of the most eastern point in each polygon. Firstly, define a `most.eastern` function to return the most eastern point of a polygon:

```
most.eastern <- function(poly) {return(min(poly[,1]))}
```

Next, use `lapply` to apply it to each polygon in the list:

```
most.eastern.list <- lapply(georgia.polys,most.eastern)
```

If you type in `most.eastern.list` you will see the result is a list of 159 items. Each one is the most eastern point of the corresponding polygon. In fact, there is a shorter way of doing this:

```
most.eastern.list <- lapply(georgia.polys,
    function(poly) {return(min(poly[,1]))} )
```

In this version, the function `most.eastern` is replaced with the definition of the function. Assuming you do not want to make use of the function again, this is a quicker way of doing things. Since the function never gets given a name, this is referred to as an *anonymous function*. In fact, you can make this even shorter. Since the function body only has one line, you don't actually need to enclose it in curly brackets { and } – and you can write the whole thing on a single line:

```
most.eastern.list <- lapply(georgia.polys,
    function(poly) return(min(poly[,1])) )
```

Now if you apply `unlist` to this list this will return a basic vector of 159 most eastern points. Finally, you can apply `min` to this – this gives you the most eastern point of *all* the polygons in the list.

```
min(unlist(most.eastern.list))

## [1] 939221
```


It is possible to combine all of these operations into a new function called `most.eastern.point` and then to test it:

```
# Function definition
most.eastern.point <- function(polys) {
        # Most eastern points
        most.eastern.list <- lapply(polys,
                function(poly) return(min(poly[,1])))
                # Return the smallest
                return(min(unlist(most.eastern.list)))}
# Test it
most.eastern.point(georgia.polys)

## [1] 939221
```

Self-Test Question 6. Write similar functions for the most western, most northern and most southern points in the polygon list.

You can test the functions you create with the code below, assuming you have used similar naming conventions:

```
c(most.eastern.point(georgia.polys),
    most.western.point(georgia.polys))
c(most.southern.point(georgia.polys),
    most.northern.point(georgia.polys))
```

Self-Test Question 7. Use these functions to update the `draw.polys` function to automatically work out the map window.

4.5.3 Shaded Maps

In this section, you will extend the methods above to produce shaded maps, rather than plain ones. To do this, you will need to create a new `factor` variable. First, make sure the `georgia` datasets are still loaded. As a reminder, three variables are loaded – `georgia`, `georgia2` and `georgia.polys`:

```
data(georgia)
```

Next, a factor variable called `classifier` will be created with two levels, `urban` and `rural`, that will be used to apply an urban/rural classification for each of the counties in Georgia. This is based on whether or not more than 50% of the population live in a rural area. Have a look at the `georgia` attributes and the rural descriptor `PctRural` by entering:

```
names(georgia)
georgia$PctRural
```

Now create the `classifier` variable:

```
classifier <- factor(ifelse(georgia$PctRural > 50,
    "rural","urban"))
```

You should examine this variable and note the use of the `factor` function and the `ifelse` function. This is new and combines both `if` and `else` statements. You should explore this in the help file.

Now, create a vector of colours, to shade in the map. To show the rural areas in dark green and the urban areas in yellow, the first step is to create a vector of appropriate colours. Define a character vector called `fill.cols` with the same length as the number of polygons, initially just containing empty strings:

```
fill.cols <- vector(mode="character",
    length=length(classifier))
```

Then set the elements in `fill.cols` that correspond to rural counties with the value `"darkgreen"`, and those corresponding to urban areas with the value `"yellow"`:

```
fill.cols[classifier=="urban"] <- "yellow"
fill.cols[classifier=="rural"] <- "darkgreen"
```

To draw the map, it is necessary to draw each polygon in the list `georgia.polys` with the colour given in the corresponding element in `fill.cols`. Note that this is possible because the `georgia` and `georgia.polys` datasets are similarly ordered. The `lapply` function can't be used here, as it can only apply functions to single elements in a list – and here we need an additional argument to give the colour. Fortunately there is also another function, `mapply` (the 'm' stands for multivariate), that handles this situation. This takes the form:

```
mapply(<function>,<1st arguments list>,<2nd
arguments list>,...)
```

Note that this is in a different order to `lapply`. In this case, the '1st argument list' is the polygon list, and the second argument is the list of colours. Assuming you have successfully defined the functions required for Self-Test Question 6, enter the code below to produce Figure 4.3.


```
# NB. ew is east/west, ns is north/south
# apply functions to determine bounding coordinates
ew <- c(most.eastern.point(georgia.polys),
      most.western.point(georgia.polys))
ns <- c(most.southern.point(georgia.polys),
      most.northern.point(georgia.polys))
# set the plot parameters
par(mar = c(0,0,0,0))
plot(ew,ns,asp=1,
       type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
invisible(mapply(polygon,georgia.polys,col=fill.cols))
```

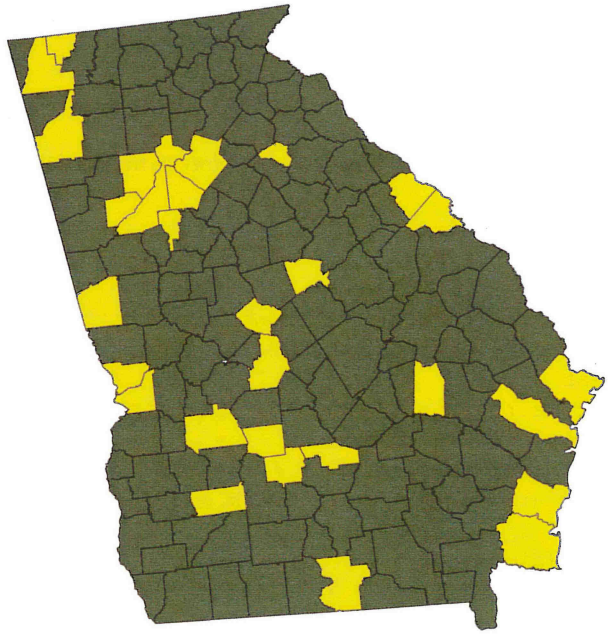

Figure 4.3 Rural/urban areas in Georgia

Self-Test Question 8. Repeat the above, but using different densities of hatching instead of colour shading, to show the rural areas. For information, you can create a vector with numeric variables, instead of characters, by using:

```
hatching <- vector(mode="numeric",
    length=length(georgia.polys))
```

You should note that a `density` of zero in the `polygon` command implies no hatching.

ANSWERS TO SELF-TEST QUESTIONS

Q1. A new `cube.root` function:

```
cube.root.2 <- function(x)
  { if (is.numeric(x))
    { result <- sign(x)*abs(x)^(1/3)
       return(result)
    } else
  { cat("WARNING: Input must be numerical, not character\n")
    return(NA) }
}
```

Q2. Match up the lines in the `gcd` function to the lines in the description of Euclid's algorithm:

```
gcd <- function(a,b)
  {
    divisor <- min(a,b) # line 1
    dividend <- max(a,b) # line 1
    repeat #line 5
      { remainder <- dividend %% divisor #line 2
        dividend <- divisor # line 3
        divisor <- remainder # line 4
        if (remainder == 0) break #line 6
      }
    return(dividend)
}
```

Q3. (i) Here is a function to compute and print out `gcd(x,60)`:

```
gcd.60 <- function(a)
 {
  for(i in 1:a)
  { divisor <- min(i,60)
     dividend <- max(i,60)
      repeat
      { remainder <- dividend %% divisor
              dividend <- divisor
```



```
              divisor <- remainder
              if (remainder == 0) break
      }
    cat(dividend, "\n")
    }
  }
```

Alternatively, you could nest the predefined `gcd` function inside the modified one:

```
gcd.60 <- function(a)
  {for(i in 1:a)
    { dividend <- gcd(i,60)
      cat(i, ":", dividend, "\n")
    }
  }
```

(ii) Here is a function to compute and print out `gcd(x,y)`:

```
gcd.all <- function(x,y)
  { for(n1 in 1:x)
    { for(n2 in 1:y)
      { dividend <- gcd(n1, n2)
        cat("when x is",n1,"& y is",n2,"dividend=",dividend,"\n")
      }
    }
  }
```

Q4. The obvious solution to this is:

```
cube.root.table <- function(n)
  { for (x in seq(0.5, n, by = 0.5))
    { cat("The cube root of ",x, " is",
        sign(x)*abs(x)^(1/3), "\n")}
}
```

However, this will not work when negative values are passed to it: `seq` cannot create the array. The function can be modified to accommodate sequences running from `0.5` to both negative and positive values of `n`:

```
cube.root.table <- function(n)
  { if (n > 0 ) by.val = 0.5
    if (n < 0 ) by.val = -0.5
```

```
    for (x in seq(0.5, n, by = by.val))
    { cat("The cube root of ",x, " is",
      sign(x)*abs(x)^(1/3), "\n") }
}
```

Q5. Write the `draw.polys` function:

```
draw.polys <- function(poly.list)
  { plot(c(939200,1419420),c(905510,1405900),asp=1,
      type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
    invisible(lapply(poly.list,polygon))
  }
# Test it
draw.polys(georgia.polys)
```

You might also want to add a test as to whether the input to the function is actually a list and report an error if it is not – you can do the test with the `is.list` function.

Q6. The function definitions and tests are given below:

```
# The function definitions
most.western.point <- function(polys) {
    most.western.list <- lapply(georgia.polys,
        function(poly) return(max(poly[,1])))
    return(max(unlist(most.western.list)))}
#
most.southern.point <- function(polys) {
    most.southern.list <- lapply(georgia.polys,
        function(poly) return(min(poly[,2])))
    return(min(unlist(most.southern.list)))}
#
most.northern.point <- function(polys) {
    most.northern.list <- lapply(georgia.polys,
        function(poly) return(max(poly[,2])))
    return(max(unlist(most.northern.list)))}
# Test the functions
c(most.eastern.point(georgia.polys),
  most.western.point(georgia.polys))

## [1] 939221 1419424

c(most.southern.point(georgia.polys),
  most.northern.point(georgia.polys))

## [1] 905508 1405900
```


Note that the last two expressions could be used as the arguments in `plot` to set the map window. This can be used in the next answer.

Q7. Combine the various functions to update the `draw.polys` function to automatically work out the map window.

```
# NB. ew = east/west ns=north/south
draw.polys <- function(poly.list) {
    ew <- c(most.eastern.point(poly.list),
            most.western.point(poly.list))
    ns <- c(most.southern.point(poly.list),
            most.northern.point(poly.list))
    plot(ew,ns,asp=1,
         type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
    invisible(lapply(poly.list,polygon)) }
#
# Test it - it should look the same as before!
#
draw.polys(georgia.polys)
```

Q8. This is one possibility – it only hatches urban counties:

```
hatch.densities <- vector(mode="numeric",length=length(georgia.polys))
hatch.densities[classifier=="urban"] <- 40
hatch.densities[classifier=="rural"] <- 0
# This assumes ew and ns were defined earlier
plot(ew,ns,asp=1,
     type='n',xlab='',ylab='',xaxt='n',yaxt='n',bty='n')
invisible(mapply(polygon,georgia.polys,density=hatch.densities))
```