UNIT 4 CONTROL STATEMENTS, FUNCTIONS, R GRAPHS

Control statements Arithmetic and Boolean operators and values Default values for arguments - Returning Boolean values functions are objects Environment and Scope issues Writing Upstairs - Recursion Replacement functions Tools for composing function code Math and Simulations in R Creating Graphs Customizing Graphs Saving graphs to files Creating three-dimensional plots.

Control Statements in R Programming

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable. In this article, we'll discuss all the control statements with the examples.

In R programming, there are 8 types of control statements as follows:

- if condition
- if-else condition
- for loop
- nested loops
- while loop
- repeat and break statement
- return statement
- next statement

if condition

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

Syntax:

```
if(expression){
   statements
   ....
   ....
}
```

Example:

```
x <- 100

if(x > 10){
print(paste(x, "is greater than 10"))
}
```

if-else condition

It is similar to if condition but when the test expression in if condition fails, then statements in else condition are executed.

Syntax:

```
if(expression){
   statements
   ....
   ....
}
else{
   statements
   ....
   ....
}
```

Example:

```
x <- 5


# Check value is less than or greater than 10
if(x > 10){
  print(paste(x, "is greater than 10"))
}else{
  print(paste(x, "is less than 10"))
}
```

for loop

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

Syntax:

```
for(value in vector){
   statements
```

```
    ....
    ....
}
```

Example:

```
x <- letters[4:10]


for(i in x){
  print(i)
}
```

Nested loops

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

Example:

```
# Defining matrix
m <- matrix(2:15, 2)


for (r in seq(nrow(m))) {
  for (c in seq(ncol(m))) {
    print(m[r, c])
  }
}
```

while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

Syntax:

```
while(expression){
    statement
    ....
    ....
}
```

Example:

```
x = 1


# Print 1 to 5
while(x <= 5){
  print(x)
  x = x + 1
}
```

repeat loop and break statement

repeat is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. **break** statement can be used in any type of loop to exit from the loop.

Syntax:

```
repeat {
   statements
   ....
   ....
   if(expression) {
      break
   }
}
```

Example:

```
x = 1


# Print 1 to 5
repeat{
  print(x)
  x = x + 1
  if(x > 5){
    break
   }
}
```

return statement

return statement is used to return the result of an executed function and returns control to the calling function.

Syntax:

return(expression)

Example:

```
# Checks value is either positive, negative or zero

func <- function(x){
  if(x > 0){
    return("Positive")
  }else if(x < 0){
    return("Negative")
  }else{
    return("Zero")
  }
}


func(1)

func(0)

func(-1)
```

next statement

next statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

Example:

```
# Defining vector

x <- 1:10


# Print even numbers

for(i in x){
  if(i%%2 != 0){
    next #Jumps to next loop
```

```
  }
  print(i)
}
```

R-Operators

Operators are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numericals as input operands.

R supports majorly four kinds of binary operators between a set of operands. In this article, we will see various types of operators in R Programming language and their usage.

Arithmetic Operators

Arithmetic Operators modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors.

1. Addition operator (+)

The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:

```
a <- c (1, 0.1)

b <- c (2.33, 4)

print (a+b)
```

Output:

[1] 3.33 4.10

2. Subtraction Operator (-)

The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:

```
 a <- 6

 b <- 8.4

 print (a-b)
```

Output:

[1] -2.4

3. Multiplication Operator (*)

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

```
b= c(4,4)
```

c= c(5,5)

print (b*c)

Output:

[1] 20 20

4. Division Operator (/)

The first operand is divided by the second operand with the use of the '/' operator.

```
a <- 10
b <- 5
print (a/b)
```

Output:

[1] 2

5. Power Operator (^)

The first operand is raised to the power of the second operand.

```
a <- 4
b <- 5
print(a^b)
```

Output:

[1] 1024

6. Modulo Operator (%%)

It returns the remainder after dividing the first operand by the second operand.

```
a<- c(2, 22)
b<-c(2,4)
print(a %% b)
```

Output:

[1] 0 2

Boolean

x && y Boolean AND for scalars

x || y Boolean OR for scalars

x&y Boolean AND for vectors (vector x,y,result)

x|y Boolean OR for vectors (vector x,y,result)

!x Boolean negation

```
> x
[1]  TRUE FALSE  TRUE
> y
[1]  TRUE  TRUE FALSE
> x & y
[1]  TRUE FALSE FALSE
> x[1] && y[1]
[1] TRUE
> x && y  # looks at just the first elements of each vector
[1] TRUE
> if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
> if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
  the condition has length > 1 and only the first element will be used
```

The central point is that in evaluating an if, we need a single Boolean, not a vector of Booleans, hence the warning seen in the preceding example, as well as the need for having both the & and && operators.

The Boolean values TRUE and FALSE can be abbreviated as T and F (both must be capitalized). These values change to 1 and 0 in arithmetic expressions:

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
```

> (1 < 2) == TRUE

[1] TRUE

> (1 < 2) == 1

[1] TRUE

Default Values for Arguments

we read in a data set from a file named exams:

> testscores <- read.table("exams",header=TRUE)

The argument header=TRUE tells R that we have a header line, so R should not count that first line in the file as data. This is an example of the use of named arguments. Here are the first few lines of the function:

> read.table

function (file, header = FALSE, sep = "", quote = "\"'", dec = ".", row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA", colClasses = NA, nrows = -1, skip = 0, check.names = TRUE, fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE, comment.char = "#", allowEscapes = FALSE, flush = FALSE, stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")

{

if (is.character(file)) {

file <- file(file, "r")

on.exit(close(file))

….

….

The second formal argument is named header. The = FALSE field means that this argument is optional, and if we don't specify it, the default value will be FALSE. If we don't want the default value, we must name the argument in our call:

> testscores <- read.table("exams",header=TRUE)

Hence the terminology named argument. Note, though, that because R uses lazy evaluation—it does not evaluate an expression until/unless it needs to—the named argument may not actually be used

Functions are objects

R functions are first-class objects (of the class "function", of course), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

```
> g <- function(x) {
+ return(x+1)
 + }
```

Here, function() is a built-in R function whose job is to create functions! On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function we're creating—here, just x—and the second is the body of that function—here, just the single statement return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g. By the way, even the "{" is a function, as you can verify by typing this:

```
> ?"{"
```

Its job is the make a single unit of what could be several statements. These two arguments to function() can later be accessed via the R functions formals() and body(), as follows:

```
> formals(g)
$x
> body(g)
{
return(x + 1)
 }
```

Recall that when using R in interactive mode, simply typing the name of an object results in printing that object to the screen. Functions are no exception, since they are objects just like anything else.

```
> g
function(x) {
return(x+1)
}
```

This is handy if you're using a function that you wrote but which you've forgotten the details of. Printing out a function is also useful if you are not quite sure what an R library function does. By looking at the code, you may understand it better. For example, if you are not sure as to the exact behavior of the graphics function abline(), you could browse through its code to better understand how to use it.

```
> abline
function (a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
coef = NULL, untf = FALSE, ...)
{
```

```
int_abline <- function(a, b, h, v, untf, col = par("col"), lty = par("lty"), lwd = par("lwd"), ...)
.Internal(abline(a, b, h, v, untf, col, lty, lwd, ...))

 if (!is.null(reg)) {

if (!is.null(a))

warning("'a' is overridden by 'reg'")

a <- reg }

if (is.object(a) || is.list(a)) {

p <- length(coefa <- as.vector(coef(a)))
```

Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

```
> f1 <- function(a,b) return(a+b)

> f2 <- function(a,b) return(a-b)

> f <- f1

> f(3,2)

 [1] 5

> f <- f2

> f(3,2)

[1] 1

> g <- function(h,a,b) h(a,b)

> g(f1,3,2)

 [1] 5

> g(f2,3,2)

 [1] 1
```

Environment and Scope Issues

A function—formally referred to as a closure in the R documentation— consists not only of its arguments and body but also of its environment. The latter is made up of the collection of objects present at the time the function is created. An understanding of how environments work in R is essential for writing effective R functions.

1. The Top-Level Environment

```
> w <- 12

> f <- function(y) {
```

```
+ d <- 8
+ h <- function() {
+ return(d*(w+y))
+ }
+ return(h())
 + } > environment(f)
```

Here, the function f() is created at the top level—that is, at the interpreter command prompt—and thus has the top-level environment, which in R output is referred to as R_GlobalEnv but which confusingly you refer to in R code as .GlobalEnv. If you run an R program as a batch file, that is considered top level, too. The function ls() lists the objects of an environment. If you call it at the top level, you get the top-level environment. Let's try it with our example code:

```
> ls()
[1] "f" "w"
```

2. The Scope Hierarchy

We have h() being local to f(), just like d. In such a situation, it makes sense for scope to be hierarchical. Thus, R is set up so that d, which is local to f(), is in turn global to h(). The same is true for y, as arguments are considered locals in R. Similarly, the hierarchical nature of scope implies that since w is global to f(), it is global to h() as well. Indeed, we do use w within h(). In terms of environments then, h()'s environment consists of whatever objects are defined at the time h() comes into existence; that is, at the time that this assignment is executed:

```
h <- function() {
return(d*(w+y))
}
```

3. More on ls()

Without arguments, a call to ls() from within a function returns the names of the current local variables (including arguments). With the envir argument, it will print the names of the locals of any frame in the call chain.

Here's an example:

```
> f
function(y) {
   d <- 8
   return(h(d,y))
}
> h
function(dee,yyy) {
   print(ls())
   print(ls(envir=parent.frame(n=1)))
   return(dee*(w+yyy))
}



> f(2)
[1] "dee" "yyy"
[1] "d" "y"
[1] 112
```

4. Functions Have (Almost) No Side Effects

Yet another influence of the functional programming philosophy is that functions do not change nonlocal variables; that is, generally, there are no side effects. Roughly speaking, the code in a function has read access to its nonlocal variables, but it does not have write access to them. Our code can appear to reassign those variables, but the action will affect only copies, not the variables themselves. Let's demonstrate this by adding some more code to our previous example.

```
> w <- 12
> f
function(y) {
   d <- 8
   w <- w + 1
   y <- y - 2
   print(w)
   h <- function() {
      return(d*(w+y))
   }
   return(h())
}



   > t <- 4
   > f(t)
   [1] 13
   [1] 120
   > w
   [1] 12
   > t
   [1] 4
```

Writing Upstairs

On the other hand, direct write access to variables at higher levels via the standard <- operator is not possible. If you wish to write to a global variable—or more generally, to any variable higher in the environment hierarchy than the level at which your write statement exists—you can use the superassignment operator, <<-, or the assign() function. Let's discuss the superassignment operator first.

7.8.1 Writing to Nonlocals with the Superassignment Operator

Consider the following code:

```
> two <- function(u) {
+    u <<- 2*u
+    z <- 2*z
+ }
> x <- 1
> z <- 3
> u
Error: object "u" not found
> two(x)
> x


[1] 1
> z
[1] 3
> u
[1] 2
```

Let's look at the impact (or not) on the three top-level variables x, z, and u:

• x: Even though x was the actual argument to two() in the example, it retained the value 1 after the call. This is because its value 1 was copied to the formal argument u, which is treated as a local variable within the function. Thus, when u changed, x did not change with it.

• z: The two z values are entirely unrelated to each other—one is top level, and the other is local to two(). The change in the local variable has no effect on the global variable. Of course, having two variables with the same name is probably not good programming practice.

• u: The u value did not even exist as a top-level variable prior to our calling two(), hence the "not found" error message. However, it was created as a top-level variable by the superassignment operator within two(), as confirmed after the call.

2. Writing to Nonlocals with assign()

You can also use the assign() function to write to upper-level variables. Here's an altered version of the previous example:

```
> two
function(u) {
   assign("u",2*u,pos=.GlobalEnv)
   z <- 2*z
}
> two(x)
> x
[1] 1
> u
[1] 2
```

Here, we replaced the superassignment operator with a call to assign(). That call instructs R to assign the value 2*u (this is the local u) to a variable u further up the call stack, specifically in the top-level environment. In this case, that environment is only one call level higher, but if we had a chain of calls, it could be much further up. The fact that you reference variables using character strings in assign() can come in handy

7.8.3 Extended Example: Discrete-Event Simulation in R

Discrete-event simulation (DES) is widely used in business, industry, and government. The term discrete event refers to the fact that the state of the system changes only in discrete quantities, rather than changing continuously.

A typical example would involve a queuing system, say people lining up to use an ATM. Let's define the state of our system at time t to be the number of people in the queue at that time. The state changes only by +1, when someone arrives, or by −1, when a person finishes an ATM transaction. This is in contrast to, for instance, a simulation of weather, in which temperature, barometric pressure, and so on change continuously.

This will be one of the longer, more involved examples in this book. But it exemplifies a number of important issues in R, especially concerning global variables, and will serve as an example when we discuss appropriate use global variables in the next section. Your patience will turn out to be a good investment of time. (It is not assumed here that the reader has any prior background in DES.)

Central to DES operation is maintenance of the event list, which is simply a list of scheduled events. This is a general DES term, so the word list here does not refer to the R data type. In fact, we'll represent the event list by a data frame.

In the ATM example, for instance, the event list might at some point in the simulation look like this:

customer 1 arrives at time 23.12

customer 2 arrives at time 25.88

customer 3 arrives at time 25.97

customer 1 finishes service at time 26.02

Since the earliest event must always be handled next, the simplest form of coding the event list is to store it in time order, as in the example. (Readers with computer science background might notice that a more efficient approach might be to use some kind of binary tree for storage.) Here, we will implement it as a data frame, with the first row containing the earliest scheduled event, the second row containing the second earliest, and so on.

The main loop of the simulation repeatedly iterates. Each iteration pulls the earliest event off of the event list, updates the simulated time to reflect the occurrence of that event, and reacts to this event. The latter action will typically result in the creation of new events. For example, if a customer arrival occurs when the queue is empty, that customer's service will begin—one event triggers setting up another. Our code must determine the customer's service time, and then it will know the time at which service will be finished, which is another event that must be added to the event list.

One of the oldest approaches to writing DES code is the event-oriented paradigm. Here, the code to handle the occurrence of one event directly sets up another event, reflecting our preceding discussion

4. When Should You Use Global Variables?

Use of global variables is a subject of controversy in the programming community. Obviously, the question raised by the title of this section cannot be answered in any formulaic way, as it is a matter of personal taste and style. Nevertheless, most programmers would probably consider the outright banning of global variables, which is encouraged by many teachers of programming, to be overly rigid. In this section, we will explore the possible value of globals in the context of the structures of R. Here, the term global variable, or just global, will be used to include any variable located higher in the environment hierarchy than the level of the given code of interest.

The use of global variables in R is more common than you may have guessed. You might be surprised to learn that R itself makes very substantial use of globals internally, both in its C code and in its R routines. The superassignment operator <<-, for instance, is used in many of the R library functions (albeit typically in writing to a variable just one level up in the environment hierarchy). Threaded code and GPU code, which are used for writing fast programs tend to make heavy use of global variables, which provide the main avenue of communication between parallel actors

5. Closures

Recall that an R closure consists of a function's arguments and body together with its environment at the time of the call. The fact that the environment is included is exploited in a type of programming that uses a feature also known (in a slight overloading of terminology) as a closure. A closure consists of a function that sets up a local variable and then creates another function that accesses that variable.

```
> counter
 function () {
 ctr <- 0
```

```
 f <- function() {
 ctr <<- ctr + 1
 cat("this count currently has value",ctr,"\n")
 }
return(f)
}
```

7.9 Recursion

Once a mathematics PhD student whom I knew to be quite bright, but who had little programming background, sought my advice on how to write a certain function. I quickly said, "You don't even need to tell me what the function is supposed to do. The answer is to use recursion." Startled, he asked what recursion is. I advised him to read about the famous Towers of Hanoi problem. Sure enough, he returned the next day, reporting that he was able to solve his problem in just a few lines of code, using recursion. Obviously, recursion can be a powerful tool. Well then, what is it?

A *recursive* function calls itself. If you have not encountered this concept before, it may sound odd, but the idea is actually simple. In rough terms, the idea is this:

> To solve a problem of type X by writing a recursive function `f()`:
>
> 1. Break the original problem of type X into one or more smaller problems of type X.
> 2. Within `f()`, call `f()` on each of the smaller problems.
> 3. Within `f()`, piece together the results of (b) to solve the original problem.

7.9.1 A Quicksort Implementation

A classic example is Quicksort, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88). We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired.

R's vector-filtering capability and its `c()` function make implementation of Quicksort quite easy.

NOTE *This example is for the purpose of demonstrating recursion. R's own sort function, `sort()`, is much faster, as it is written in C.*

```
qs <- function(x) {
   if (length(x) <= 1) return(x)
   pivot <- x[1]
   therest <- x[-1]
   sv1 <- therest[therest < pivot]
   sv2 <- therest[therest >= pivot]
   sv1 <- qs(sv1)
   sv2 <- qs(sv2)
   return(c(sv1,pivot,sv2))
}
```

Note carefully the *termination condition*:

```
if (length(x) <= 1) return(x)
```

Without this, the function would keep calling itself repeatedly on empty vectors, executing forever. (Actually, the R interpreter would eventually refuse to go any further, but you get the idea.)

Sounds like magic? Recursion certainly is an elegant way to solve many problems. But recursion has two potential drawbacks:

- It's fairly abstract. I knew that the graduate student, as a fine mathematician, would take to recursion like a fish to water, because recursion is really just the inverse of proof by mathematical induction. But many programmers find it tough.
- Recursion is very lavish in its use of memory, which may be an issue in R if applied to large problems.

7.9.2 Extended Example: A Binary Search Tree

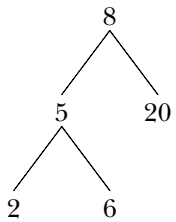
Treelike data structures are common in both computer science and statistics. In R, for example, the `rpart` library for a recursive partioning approach to regression and classification is very popular. Trees obviously have applications in genealogy, and more generally, graphs form the basis of analysis of social networks.

However, there are real issues with tree structures in R, many of them related to the fact that R does not have pointer-style references, as discussed in Section 7.7. Indeed, for this reason and for performance purposes, a better option is often to write the core code in C with an R wrapper, as we'll discuss in Chapter 15. Yet trees can be implemented in R itself, and if performance is not an issue, using this approach may be more convenient.

For the sake of simplicity, our example here will be a binary search tree, a classic computer science data structure that has the following property:

> In each node of the tree, the value at the left link, if any, is less than or equal to that of the parent, while the value at the right link, if any, is greater than that of the parent.

Here is an example:



We've stored 8 in the *root*—that is, the head—of the tree. Its two child nodes contain 5 and 20, and the former itself has two child nodes, which store 2 and 6.

Note that the nature of binary search trees implies that at any node, all of the elements in the node's left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree—5, 2 and 6—are less than 8, while 20 is greater than 8.

If implemented in C, a tree node would be represented by a C struct, similar to an R list, whose contents are the stored value, a pointer to the left child, and a pointer to the right child. But since R lacks pointer variables, what can we do?

Our solution is to go back to the basics. In the old prepointer days in FORTRAN, linked data structures were implemented in long arrays. A pointer, which in C is a memory address, was an array index instead.

Specifically, we'll represent each node by a row in a three-column matrix. The node's stored value will be in the third element of that row, while the first and second elements will be the left and right links. For instance, if the first element in a row is 29, it means that this node's left link points to the node stored in row 29 of the matrix.

Remember that allocating space for a matrix in R is a time-consuming activity. In an effort to amortize the memory-allocation time, we allocate new space for a tree's matrix several rows at a time, instead of row by row. The number of rows allocated each time will be given in the variable `inc`. As is common with tree traversal, we implement our algorithm with recursion.

NOTE *If you anticipate that the matrix will become quite large, you may wish to double its size at each allocation, rather than grow it linearly as we have here. This would further reduce the number of time-consuming disruptions.*

Before discussing the code, let's run through a quick session of tree building using its routines.

```
> x <- newtree(8,3)
> x
$mat
     [,1] [,2] [,3]
[1,]   NA   NA    8
[2,]   NA   NA   NA
[3,]   NA   NA   NA

$nxt
[1] 2

$inc
[1] 3

> x <- ins(1,x,5)
> x
$mat
```

```
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA   NA    5
[3,]   NA   NA   NA


$nxt
[1] 3

$inc
[1] 3

> x <- ins(1,x,6)
> x
$mat
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA    3    5
[3,]   NA   NA    6

$nxt
[1] 4

$inc
[1] 3

> x <- ins(1,x,2)
> x
$mat
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]    4    3    5
[3,]   NA   NA    6
[4,]   NA   NA    2
[5,]   NA   NA   NA
[6,]   NA   NA   NA

$nxt
[1] 5

$inc
[1] 3

> x <- ins(1,x,20)
> x
$mat
     [,1] [,2] [,3]
```

```
[1,]    2    5    8
[2,]    4    3    5
[3,]   NA   NA    6
[4,]   NA   NA    2
[5,]   NA   NA   20
[6,]   NA   NA   NA

$nxt
[1] 6

$inc
[1] 3
```

What happened here? First, the command containing our call `newtree(8,3)` creates a new tree, assigned to `x`, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time. The result is that the matrix component of the list `x` is now as follows:

```
     [,1] [,2] [,3]
[1,]   NA   NA    8
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children.

We then make the call `ins(1,x,5)` to insert a second value, 5, into the tree `x`. The argument 1 specifies the root. In other words, the call says, "Insert 5 in the subtree of `x` whose root is in row 1." Note that we need to reassign the return value of this call back to `x`. Again, this is due to the lack of pointer variables in R. The matrix now looks like this:

```
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA   NA    5
[3,]   NA   NA   NA
```

The element 2 means that the left link out of the node containing 8 is meant to point to row 2, where our new element 5 is stored.

The session continues in this manner. Note that when our initial allotment of three rows is full, `ins()` allocates three new rows, for a total of six. In the end, the matrix is as follows:

```
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    4    3    5
[3,]   NA   NA    6
```

```
[4,]   NA   NA    2
[5,]   NA   NA   20
[6,]   NA   NA   NA
```

This represents the tree we graphed for this example.

The code follows. Note that it includes only routines to insert new items and to traverse the tree. The code for deleting a node is somewhat more complex, but it follows a similar pattern.

```
# routines to create trees and insert items into them are included
# below; a deletion routine is left to the reader as an exercise

# storage is in a matrix, say m, one row per node of the tree; if row
# i contains (u,v,w), then node i stores the value w, and has left and
# right links to rows u and v; null links have the value NA

# the tree is represented as a list (mat,nxt,inc), where mat is the
# matrix, nxt is the next empty row to be used, and inc is the number of
# rows of expansion to be allocated whenever the matrix becomes full

# print sorted tree via in-order traversal
printtree <- function(hdidx,tr) {
   left <- tr$mat[hdidx,1]
   if (!is.na(left)) printtree(left,tr)
   print(tr$mat[hdidx,3])  # print root
   right <- tr$mat[hdidx,2]
   if (!is.na(right)) printtree(right,tr)
}

# initializes a storage matrix, with initial stored value firstval
newtree <- function(firstval,inc) {
   m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
   m[1,3] <- firstval
   return(list(mat=m,nxt=2,inc=inc))
}

# inserts newval into the subtree of tr, with the subtree's root being
# at index hdidx; note that return value must be reassigned to tr by the
# caller (including ins() itself, due to recursion)
ins <- function(hdidx,tr,newval) {
   # which direction will this new node go, left or right?
   dir <- if (newval <= tr$mat[hdidx,3]) 1 else 2
   # if null link in that direction, place the new node here, otherwise
   # recurse
   if (is.na(tr$mat[hdidx,dir])) {
      newidx <- tr$nxt  # where new node goes
      # check for room to add a new element
      if (tr$nxt == nrow(tr$mat) + 1) {
```

```
40            tr$mat <-
41               rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
42         }
43         # insert new tree node
44         tr$mat[newidx,3] <- newval
45         # link to the new node
46         tr$mat[hdidx,dir] <- newidx
47         tr$nxt <- tr$nxt + 1  # ready for next insert
48         return(tr)
49      } else tr <- ins(tr$mat[hdidx,dir],tr,newval)
50   }
```

There is recursion in both `printtree()` and `ins()`. The former is definitely the easier of the two, so let's look at that first. It prints out the tree, in sorted order.

Recall our description of a recursive function `f()` that solves a problem of category X: We have `f()` split the original X problem into one or more smaller X problems, call `f()` on them, and combine the results. In this case, our problem's category X is to print a tree, which could be a subtree of a larger one. The role of the function on line 13 is to print the given tree, which it does by calling itself in lines 15 and 18. There, it prints first the left subtree and then the right subtree, pausing in between to print the root.

This thinking—print the left subtree, then the root, then the right subtree—forms the intuition in writing the code, but again we must make sure to have a proper termination mechanism. This mechanism is seen in the `if()` statements in lines 15 and 18. When we come to a null link, we do not continue to recurse.

The recursion in `ins()` follows the same principles but is considerably more delicate. Here, our "category X" is an insertion of a value into a subtree. We start at the root of a tree, determine whether our new value must go into the left or right subtree (line 33), and then call the function again on that subtree. Again, this is not hard in principle, but a number of details must be attended to, including the expansion of the matrix if we run out of room (lines 40–41).

One difference between the recursive code in `printtree()` and `ins()` is that the former includes two calls to itself, while the latter has only one. This implies that it may not be difficult to write the latter in a nonrecursive form.

7.10 Replacement Functions

Recall the following example from Chapter 2:

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
```

```
[1] "a"  "b"  "ab"
> x
 a  b ab
 1  2  4
```

Consider one line in particular:

```
> names(x) <- c("a","b","ab")
```

Looks totally innocuous, eh? Well, no. In fact, it's outrageous! How on Earth can we possibly assign a value to the result of a function call? The resolution to this odd state of affairs lies in the R notion of *replacement functions*.

The preceding line of R code actually is the result of executing the following:

```
x <- "names<-"(x,value=c("a","b","ab"))
```

No, this isn't a typo. The call here is indeed to a function named `names<-()`. (We need to insert the quotation marks due to the special characters involved.)

7.10.1 What's Considered a Replacement Function?

Any assignment statement in which the left side is not just an identifier (meaning a variable name) is considered a replacement function. When encountering this:

```
g(u) <- v
```

R will try to execute this:

```
u <- "g<-"(u,value=v)
```

Note the "try" in the preceding sentence. The statement will fail if you have not previously defined `g<-()`. Note that the replacement function has one more argument than the original function `g()`, a named argument `value`, for reasons explained in this section.

In earlier chapters, you've seen this innocent-looking statement:

```
x[3] <- 8
```

The left side is not a variable name, so it must be a replacement function, and indeed it is, as follows.

Subscripting operations are functions. The function `"["()` is for reading vector elements, and `"[<-"()` is used to write. Here's an example:

```
> x <- c(8,88,5,12,13)
> x
```

```
[1]  8 88  5 12 13
> x[3]
[1] 5


> "["(x,3)
[1] 5
> x <- "[<-"(x,2:3,value=99:100)
> x
[1]   8  99 100  12  13
```

Again, that complicated call in this line:

```
> x <- "[<-"(x,2:3,value=99:100)
```

is simply performing what happens behind the scenes when we execute this:

```
x[2:3] <- 99:100
```

We can easily verify what's occurring like so:

```
> x <- c(8,88,5,12,13)
> x[2:3] <- 99:100
> x
[1]   8  99 100  12  13
```

7.10.2 Extended Example: A Self-Bookkeeping Vector Class

Suppose we have vectors on which we need to keep track of writes. In other words, when we execute the following:

```
x[2] <- 8
```

we would like not only to change the value in `x[2]` to 8 but also increment a count of the number of times `x[2]` has been written to. We can do this by writing class-specific versions of the generic replacement functions for vector subscripting.

NOTE *This code uses classes, which we'll discuss in detail in Chapter 9. For now, all you need to know is that S3 classes are constructed by creating a list and then anointing it as a class by calling the `class()` function.*

```
1  # class "bookvec" of vectors that count writes of their elements
2
3  # each instance of the class consists of a list whose components are the
4  # vector values and a vector of counts
5
```

```
# construct a new object of class bookvec
newbookvec <- function(x) {
   tmp <- list()
   tmp$vec <- x # the vector itself
   tmp$wrts <- rep(0,length(x)) # counts of the writes, one for each element
   class(tmp) <- "bookvec"
   return(tmp)
}

# function to read
"[.bookvec" <- function(bv,subs) {
   return(bv$vec[subs])
}

# function to write
"[<-.bookvec" <- function(bv,subs,value) {
   bv$wrts[subs] <- bv$wrts[subs] + 1 # note the recycling
   bv$vec[subs] <- value
   return(bv)
}
\end{Code}

Let's test it.

\begin{Code}
> b <- newbookvec(c(3,4,5,5,12,13))
> b
$vec
[1] 3 4 5 5 12 13

$wrts
[1] 0 0 0 0 0 0

attr(,"class")
[1] "bookvec"
> b[2]
[1] 4
> b[2] <- 88 # try writing
> b[2] # worked?
[1] 88
> b$wrts # write count incremented?
[1] 0 1 0 0 0 0
```

We have named our class `"bookvec"`, because these vectors will do their own bookkeeping—that is, keep track of write counts. So, the subscripting functions will be `[.bookvec()` and `[<-.bookvec()`.

Our function `newbookvec()` (line 7) does the construction for this class. In it, you can see the structure of the class: An object will consist of the vector itself, `vec` (line 9), and a vector of write counts, `wrts` (line 10).

By the way, note in line 11 that the function `class()` itself is a replacement function!

The functions `[.bookvec()` and `[<-.bookvec()` are fairly straightforward. Just remember to return the entire object in the latter.

7.11 Tools for Composing Function Code

If you are writing a short function that's needed only temporarily, a quick-and-dirty way to do this is to write it on the spot, right there in your interactive terminal session. Here's an example:

```
> g <- function(x) {
+    return(x+1)
+ }
```

This approach obviously is infeasible for longer, more complex functions. Now, let's look at some better ways to compose R code.

7.11.1 Text Editors and Integrated Development Environments

You can use a text editor such as Vim, Emacs, or even Notepad, or an editor within an integrated development environment (IDE) to write your code in a file and then read it into R from the file. To do the latter, you can use R's `source()` function.

For instance, suppose we have functions `f()` and `g()` in a file *xyz.R*. In R, we give this command:

```
> source("xyz.R")
```

This reads `f()` and `g()` into R as if we had typed them using the quick-and-dirty way shown at the beginning of this section.

If you don't have much code, you can cut and paste from your editor window to your R window.

Some general-purpose editors have special plug-ins available for R, such as ESS for Emacs and Vim-R for Vim. There are also IDEs for R, such as the commercial one by Revolution Analytics, and open source products such as StatET, JGR, Rcmdr, and RStudio.

7.11.2 The edit() Function

A nice implication of the fact that functions are objects is that you can edit functions from within R's interactive mode. Most R programmers do their code editing with a text editor in a separate window, but for a small, quick change, the `edit()` function can be handy.

For instance, we could edit the function `f1()` by typing this:

```
> f1 <- edit(f1)
```

This opens the default editor on the code for `f1`, which we could then edit and assign back to `f1`.

Or, we might be interested in having a function `f2()` very similar to `f1()` and thus could execute the following:

```
> f2 <- edit(f1)
```

This gives us a copy of `f1()` to start from. We would do a little editing and then save to `f2()`, as seen in the preceding command.

The editor involved will depend on R's internal options variable `editor`. In UNIX-class systems, R will set this from your shell's `EDITOR` or `VISUAL` environment variable, or you can set it yourself, as follows:

```
> options(editor="/usr/bin/vim")
```

For more details on using options, see the online documentation by typing the following:

```
> ?options
```

You can use `edit()` to edit data structures, too.

7.12 Writing Your Own Binary Operations

You can invent your own operations! Just write a function whose name begins and ends with %, with two arguments of a certain type, and a return value of that type.

For example, here's a binary operation that adds double the second operand to the first:

```
> "%a2b%" <- function(a,b) return(a+2*b)
> 3 %a2b% 5
[1] 13
```

A less trivial example is given in the section about set operations in Section 8.5.

7.13 Anonymous Functions

As remarked at several points in this book, the purpose of the R function `function()` is to create functions. For instance, consider this code:

```
inc <- function(x) return(x+1)
```

It instructs R to create a function that adds 1 to its argument and then assigns that function to `inc`. However, that last step—the assignment—is not always taken. We can simply use the function object created by our call to `function()` without naming that object. The functions in that context are called *anonymous*, since they have no name. (That is somewhat misleading, since even nonanonymous functions only have a name in the sense that a variable is pointing to them.)

Anonymous functions can be convenient if they are short one-liners and are called by another function. Let's go back to our example of using `apply` in Section 3.3:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1] [,2] [,3]
[1,] 0.5 1.000 1.50
[2,] 0.5 0.625 0.75
```

Let's bypass the middleman—that is, skip the assignment to `f`—by using an anonymous function within our call to `apply()`, as follows:

```
> y <- apply(z,1,function(x) x/c(2,8))
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

What really happened here? The third formal argument to `apply()` must be a function, which is exactly what we supplied here, since the return value of `function()` is a function!

Doing things this way is often clearer than defining the function externally. Of course, if the function is more complicated, that clarity is not attained.

8

DOING MATH AND SIMULATIONS IN R

R contains built-in functions for your favorite math operations and, of course, for statistical distributions. This chapter provides an overview of using these functions. Given the mathematical nature of this chapter, the examples assume a slightly higher-level knowledge than those in other chapters. You should be familiar with calculus and linear algebra to get the most out of these examples.

8.1 Math Functions

R includes an extensive set of built-in math functions. Here is a partial list:

- `exp()`: Exponential function, base `e`
- `log()`: Natural logarithm
- `log10()`: Logarithm base 10
- `sqrt()`: Square root
- `abs()`: Absolute value

Math functions in R Programming

The math function denotes the mathematical or numeric calculations with some built-in functions. The set of instructions for particular functions is predefined in the R program, the user just needs to call the function for accomplishing their task of execution. The following table highlights some of the numeric or mathematical built-in functions in R.

Serial No	Built-in function	Description	Example
1	**abs(x)**	It returns the absolute value of input x	x<- -2 print(abs(x)) Output [1] 2
2	**sqrt(x)**	It returns the square root of input x	x<- 2 print(sqrt(x)) Output [1] 1.414214
3	**ceiling(x)**	It returns the smallest integer which is larger than or equal to x.	x<- 2.8 print(ceiling(x)) Output [1] 3
4	**floor(x)**	It returns the largest integer, which is smaller than or equal to x.	x<- 2.8 print(floor(x)) Output [1] 2
5	**trunc(x)**	It returns the truncate value of input x.	x<- c(2.2,6.56,10.11) print(trunc(x)) Output [1] 2 6 10

6	**round(x, digits=n)**	It returns round value of input x.	x=2.456 print(round(x,digits=2)) x=2.4568 print(round(x,digits=3)) Output [1] 2.46 [1] 2.457
7	**cos(x), sin(x), tan(x)**	It returns cos(x), sin(x) , tan(x) value of input x	x<- 2 print(cos(x)) print(sin(x)) print(tan(x)) Output [1] -0.4161468 [1] 0.9092974 [1] -2.18504
8	**log(x)**	It returns natural logarithm of input x	x<- 2 print(log(x)) Output [1] 0.6931472
9	**log10(x)**	It returns common logarithm of input x	x<- 2 print(log10(x))

			Output [1] 0.30103
10	**exp(x)**	It returns exponent	x<- 2 print(exp(x)) Output [1] 7.389056

Simulation Using R Programming

Simulation is a powerful technique in statistics and data analysis, used to model complex systems, understand random processes, and predict outcomes. In R, various packages and functions facilitate simulation studies.

Introduction to Simulation in R

Simulating scenarios is a powerful tool for making informed decisions and exploring potential outcomes without the need for real-world experimentation. This article delves into the world of simulation using R Programming Language versatile programming language widely used for statistical computing and graphics. We'll equip you with the knowledge and code examples to craft effective simulations in R, empowering you to:

- **Predict the Unpredictable:** Explore "what-if" scenarios by simulating various conditions within your system or process.
- **Test Hypotheses with Confidence:** Analyze the behavior of your system under different circumstances to validate or challenge your assumptions.
- **Estimate Parameters with Precision:** Evaluate the impact of changing variables on outcomes, allowing for more accurate parameter estimation.
- **Forecast Trends for Informed Decisions:** Leverage simulated data to predict future behavior and make data-driven choices for your system or process.
-

Types of Simulations

This article will walk you through the basics of simulation in R, covering different types of simulations and practical examples.

1. **Monte Carlo Simulation**: Uses random sampling to compute results and is commonly used for numerical integration and risk analysis.
2. **Discrete Event Simulation**: Models the operation of systems as a sequence of events in time.
3. **Agent-Based Simulation**: Simulates the actions and interactions of autonomous agents to assess their effects on the system.

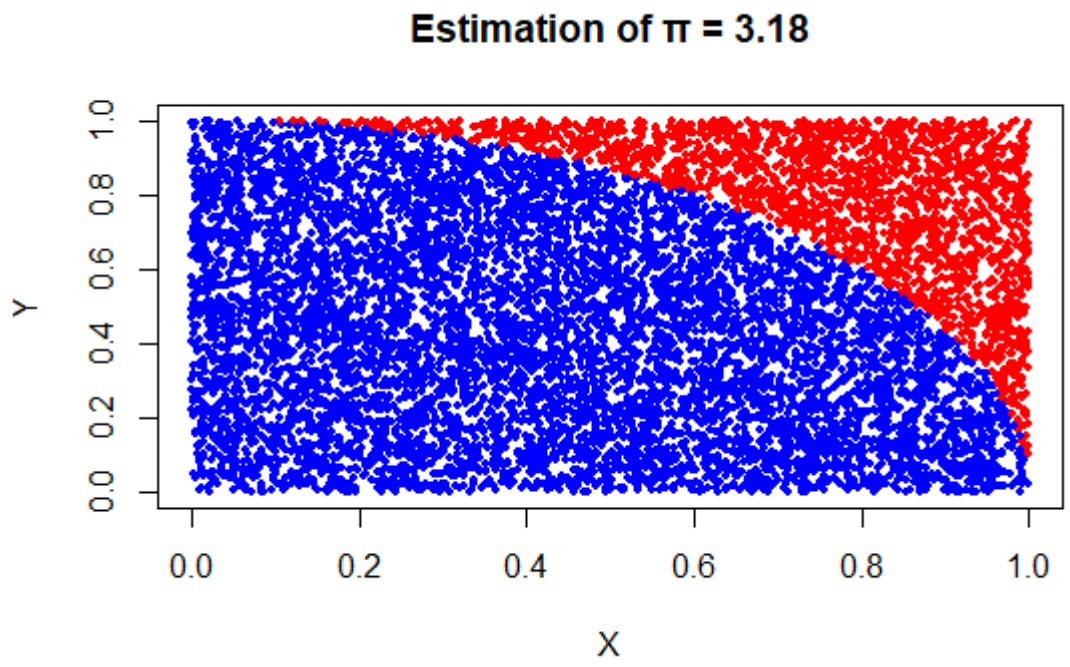
Let's start with a simple Monte Carlo simulation to estimate the value of π.

Estimating π Using Monte Carlo Simulation

The idea is to randomly generate points in a unit square and count how many fall inside the unit circle. The ratio of points inside the circle to the total number of points approximates π/4.

```r
# Number of points to generate
n <- 10000

# Generate random points
set.seed(123)
x <- runif(n)
y <- runif(n)

# Calculate distance from (0,0) and check if inside the unit circle
inside <- x^2 + y^2 <= 1

# Estimate π
pi_estimate <- (sum(inside) / n) * 4
pi_estimate

# Plot the points
plot(x, y, col = ifelse(inside, 'blue', 'red'), pch = 19, cex = 0.5,
     main = paste("Estimation of π =", round(pi_estimate, 4)),
     xlab = "X", ylab = "Y")
```

Output:

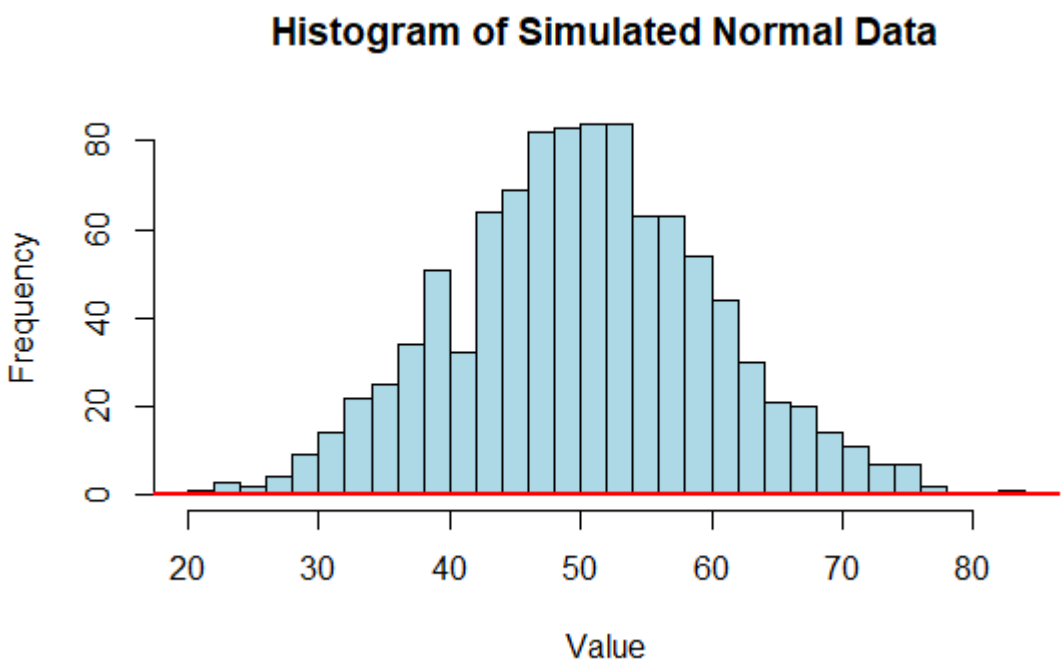


- **runif(n):** Generates n random numbers uniformly distributed between 0 and 1.
- **inside:** Logical vector indicating whether each point lies inside the unit circle.
- **sum(inside) / n * 4:** Estimates π using the ratio of points inside the circle to total points.

Simulating a Normal Distribution

Simulating data from a normal distribution is straightforward with R's rnorm() function. Let's simulate 1000 data points from a normal distribution with a mean of 50 and a standard deviation of 10.

```
# Parameters
mean <- 50
sd <- 10
n <- 1000

# Simulate data
set.seed(123)
data <- rnorm(n, mean = mean, sd = sd)

# Plot the histogram
hist(data, breaks = 30, col = "lightblue", main = "Histogram of Simulated Normal Data",
     xlab = "Value", ylab = "Frequency")

# Add a density curve
lines(density(data), col = "red", lwd = 2)
```



- **rnorm(n, mean, sd):** Generates n random numbers from a normal distribution with specified mean and sd.
- **hist():** Plots a histogram of the simulated data.
- **lines(density(data)):** Adds a kernel density estimate to the histogram.

Discrete Event Simulation

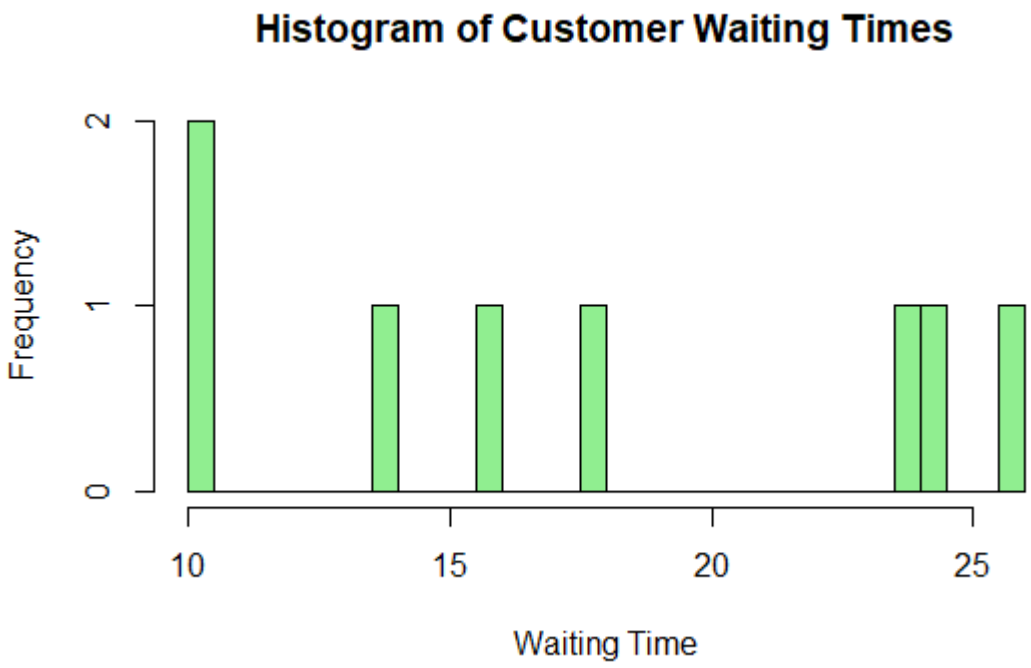
Let's simulate the operation of a simple queue system using the simmer package.

```
# Install and load simmer package
```

```
install.packages("simmer")
library(simmer)

# Define a simple queueing system
env <- simmer("queueing_system")

# Define arrival and service processes
arrival <- trajectory("arrival") %>%
  seize("server", 1) %>%
  timeout(function() rexp(1, 1/10)) %>%
  release("server", 1)

# Add resources and arrivals to the environment
env %>%
  add_resource("server", 1) %>%
  add_generator("customer", arrival, function() rexp(1, 1/5))

# Run the simulation for a specified period
env %>%
  run(until = 100)

# Extract and plot results
arrivals <- get_mon_arrivals(env)
hist(arrivals$end_time - arrivals$start_time, breaks = 30, col = "lightgreen",
     main = "Histogram of Customer Waiting Times",
     xlab = "Waiting Time", ylab = "Frequency")
```

Output:

```
simmer environment: queueing_system | now: 100 | next: 100.308581297463
{ Monitor: in memory }
{ Resource: server | monitored: TRUE | server status: 1(1) | queue status: 7(Inf) }
{ Source: customer | monitored: 1 | n_generated: 17 }
```



- **simmer("queueing_system"):** Creates a new simulation environment.
- **trajectory():** Defines the sequence of operations for arriving customers.
- **seize(), timeout(), release():** Define the customer actions (seizing a server, spending time being served, and releasing the server).
- **add_resource(), add_generator():** Add resources (servers) and customer arrival processes to the environment.
- **run(until = 100):** Runs the simulation for 100 time units.
- **get_mon_arrivals():** Extracts arrival data for analysis.

Probabilities using R

Probability theory is a fundamental concept in mathematics and statistics that plays a crucial role in various fields such as finance, engineering, medicine, and more. Understanding probabilities allows us to make informed decisions in uncertain situations. In this comprehensive guide, we'll delve into the basics of probabilities using R Programming Language.

Basic Concepts of Probability in R

Probability in R is the measure of the likelihood that an event will occur. The probability of an event A, denoted as P(A), lies between 0 and 1, where 0 indicates impossibility and 1 indicates certainty. Some key concepts include:

- **Sample Space (S):** The set of all possible outcomes of a random experiment.
- **Event:** Any subset of the sample space.
- **Probability of an Event:** The likelihood of occurrence of an event, calculated as the ratio of favorable outcomes to the total number of outcomes.

12

GRAPHICS

R has a very rich set of graphics facilities. The R home page (*http://www.r-project.org/*) has a few colorful examples, but to really appreciate R's graphical power, browse through the R Graph Gallery at *http://addictedtor.free.fr/graphiques*.

In this chapter, we cover the basics of using R's base, or traditional, graphics package. This will give you enough foundation to start working with graphics in R. If you're interested in pursuing R graphics further, you may want to refer to the excellent books on the subject.[1]

12.1 Creating Graphs

To begin, we'll look at the foundational function for creating graphs: `plot()`. Then we'll explore how to build a graph, from adding lines and points to attaching a legend.

---

[1] These include Hadley Wickham, *ggplot2: Elegant Graphics for Data Analysis* (New York: Springer-Verlag, 2009); Dianne Cook and Deborah F. Swayne, *Interactive and Dynamic Graphics for Data Analysis: With R and GGobi* (New York: Springer-Verlag, 2007); Deepayan Sarkar, *Lattice: Multivariate Data Visualization with R* (New York: Springer-Verlag, 2008); and Paul Murrell, *R Graphics* (Boca Raton, FL: Chapman and Hall/CRC, 2011).

12.1.1 The Workhorse of R Base Graphics: The plot() Function

The `plot()` function forms the foundation for much of R's base graphing operations, serving as the vehicle for producing many different kinds of graphs. As mentioned in Section 9.1.1, `plot()` is a generic function, or a placeholder for a family of functions. The function that is actually called depends on the class of the object on which it is called.

Let's see what happens when we call `plot()` with an X vector and a Y vector, which are interpreted as a set of pairs in the (*x*,*y*) plane.

```
> plot(c(1,2,3), c(1,2,4))
```

This will cause a window to pop up, plotting the points (1,1), (2,2), and (3,4), as shown in Figure 12-1. As you can see, this is a very plain-Jane graph. We'll discuss adding some of the fancy bells and whistles later in the chapter.

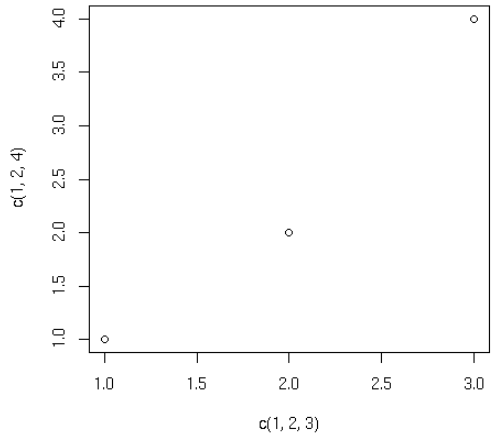


Figure 12-1: Simple point plot

NOTE *The points in the graph in Figure 12-1 are denoted by empty circles. If you want to use a different character type, specify a value for the named argument* `pch` *(for* point character*).*

The `plot()` function works in stages, which means you can build up a graph in stages by issuing a series of commands. For example, as a base, we might first draw an empty graph, with only axes, like this:

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

This draws axes labeled *x* and *y*. The horizontal (*x*) axis ranges from −3 to 3. The vertical (*y*) axis ranges from −1 to 5. The argument `type="n"` means that there is nothing in the graph itself.

12.1.2 Adding Lines: The abline() Function

We now have an empty graph, ready for the next stage, which is adding a line:

```
> x <- c(1,2,3)
> y <- c(1,3,8)
> plot(x,y)
> lmout <- lm(y ~ x)
> abline(lmout)
```

After the call to `plot()`, the graph will simply show the three points, along with the *x*- and *y*- axes with hash marks. The call to `abline()` then adds a line to the current graph. Now, which line is this?

As you learned in Section 1.5, the result of the call to the linear-regression function `lm()` is a class instance containing the slope and intercept of the fitted line, as well as various other quantities that don't concern us here. We've assigned that class instance to `lmout`. The slope and intercept will now be in `lmout$coefficients`.

So, what happens when we call `abline()`? This function simply draws a straight line, with the function's arguments treated as the intercept and slope of the line. For instance, the call `abline(c(2,1))` draws this line on whatever graph you've built up so far:

$$y = 2 + 1 \cdot x$$

But `abline()` is written to take special action if it is called on a regression object (though, surprisingly, it is not a generic function). Thus, it will pick up the slope and intercept it needs from `lmout$coefficients` and plot that line. It superimposes this line onto the current graph, the one that graphs the three points. In other words, the new graph will show both the points and the line, as in Figure 12-2.

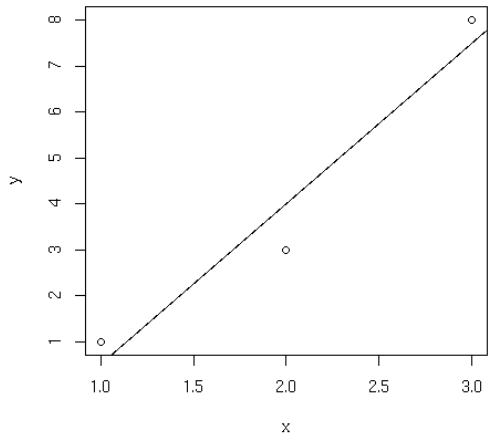


Figure 12-2: Using `abline()`

You can add more lines using the `lines()` function. Though there are many options, the two basic arguments to `lines()` are a vector of *x*-values and a vector of *y*-values. These are interpreted as (*x*,*y*) pairs representing points to be added to the current graph, with lines connecting the points. For instance, if X and Y are the vectors (1.5,2.5) and (3,3), you could use this call to add a line from (1.5,3) to (2.5,3) to the present graph:

```
> lines(c(1.5,2.5),c(3,3))
```

If you want the lines to "connect the dots," but don't want the dots themselves, include `type="l"` in your call to `lines()` or to `plot()`, as follows:

```
> plot(x,y,type="l")
```

You can use the `lty` parameter in `plot()` to specify the type of line, such as solid or dashed. To see the types available and their codes, enter this command:

```
> help(par)
```

12.1.3 Starting a New Graph While Keeping the Old Ones

Each time you call `plot()`, directly or indirectly, the current graph window will be replaced by the new one. If you don't want that to happen, use the command for your operating system:

- On Linux systems, call `X11()`.
- On a Mac, call `macintosh()`.
- On Windows, call `windows()`.

For instance, suppose you wish to plot two histograms of vectors X and Y and view them side by side. On a Linux system, you would type the following:

```
> hist(x)
> x11()
> hist(y)
```

12.1.4 Extended Example: Two Density Estimates on the Same Graph

Let's plot nonparametric density estimates (these are basically smoothed histograms) for two sets of examination scores in the same graph. We use the function `density()` to generate the estimates. Here are the commands we issue:

```
> d1 = density(testscores$Exam1,from=0,to=100)
> d2 = density(testscores$Exam2,from=0,to=100)
```

```
> plot(d1,main="",xlab="")
> lines(d2)
```

First, we compute nonparametric density estimates from the two variables, saving them in objects `d1` and `d2` for later use. We then call `plot()` to draw the curve for exam 1, at which point the plot looks like Figure 12-3. We then call `lines()` to add exam 2's curve to the graph, producing Figure 12-4.

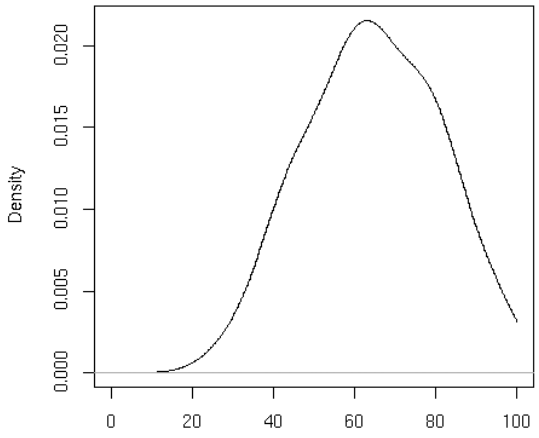


Figure 12-3: Plot of first density

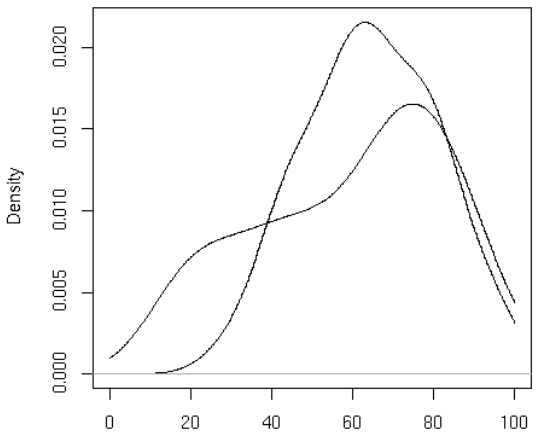


Figure 12-4: Addition of second density

Note that we asked R to use blank labels for the figure as a whole and for the *x*-axis. Otherwise, R would have gotten such labels from `d1`, which would have been specific to exam 1.

Also note that we needed to plot exam 1 first. The scores there were less diverse, so the density estimate was narrower and taller. Had we plotted exam 2, with its shorter curve, first, exam 1's curve would have been too tall for the plot window. Here, we first ran the two plots separately to see which was taller, but let's consider a more general situation.

Say we wish to write a broadly usable function that will plot several density estimates on the same graph. For this, we would need to automate the process of determining which density estimate is tallest. To do so, we would use the fact that the estimated density values are contained in the `y` component of the return value from the call to `density()`. We would then call `max()` on each density estimate and use `which.max()` to determine which density estimate is the tallest.

The call to `plot()` both initiates the plot and draws the first curve. (Without specifying `type="l"`, only the points would have been plotted.) The call to `lines()` then adds the second curve.

12.1.5 Extended Example: More on the Polynomial Regression Example

In Section 9.1.7, we defined a class `"polyreg"` that facilitates fitting polynomial regression models. Our code there included an implementation of the generic `print()` function. Let's now add one for the generic `plot()` function:

```
# polyfit(x,maxdeg) fits all polynomials up to degree maxdeg; y is
# vector for response variable, x for predictor; creates an object of
# class "polyreg", consisting of outputs from the various regression
# models, plus the original data
polyfit <- function(y,x,maxdeg) {
   pwrs <- powers(x,maxdeg)  # form powers of predictor variable
   lmout <- list()  # start to build class
   class(lmout) <- "polyreg"  # create a new class
   for (i in 1:maxdeg) {
      lmo <- lm(y ~ pwrs[,1:i])
      # extend the lm class here, with the cross-validated predictions
      lmo$fitted.xvvalues <- lvoneout(y,pwrs[,1:i,drop=F])
      lmout[[i]] <- lmo
   }
   lmout$x <- x
   lmout$y <- y
   return(lmout)
}

# generic print() for an object fits of class "polyreg":  print
# cross-validated mean-squared prediction errors
print.polyreg <- function(fits) {
   maxdeg <- length(fits) - 2  # count lm() outputs only, not $x and $y
```

```
   n <- length(fits$y)
   tbl <- matrix(nrow=maxdeg,ncol=1)
   cat("mean squared prediction errors, by degree\n")
   colnames(tbl) <- "MSPE"
   for (i in 1:maxdeg) {
      fi <- fits[[i]]
      errs <- fits$y - fi$fitted.xvvalues
      spe <- sum(errs^2)
      tbl[i,1] <- spe/n
   }
   print(tbl)
}

# generic plot(); plots fits against raw data
plot.polyreg <- function(fits) {
   plot(fits$x,fits$y,xlab="X",ylab="Y")  # plot data points as background
   maxdg <- length(fits) - 2
   cols <- c("red","green","blue")
   dg <- curvecount <- 1
   while (dg < maxdg) {
      prompt <- paste("RETURN for XV fit for degree",dg,"or type degree",
         "or q for quit ")
      rl <- readline(prompt)
      dg <- if (rl == "") dg else if (rl != "q") as.integer(rl) else break
      lines(fits$x,fits[[dg]]$fitted.values,col=cols[curvecount%%3 + 1])
      dg <- dg + 1
      curvecount <- curvecount + 1
   }
}

# forms matrix of powers of the vector x, through degree dg
powers <- function(x,dg) {
   pw <- matrix(x,nrow=length(x))
   prod <- x
   for (i in 2:dg) {
      prod <- prod * x
      pw <- cbind(pw,prod)
   }
   return(pw)
}

# finds cross-validated predicted values; could be made much faster via
# matrix-update methods
lvoneout <- function(y,xmat) {
   n <- length(y)
   predy <- vector(length=n)
   for (i in 1:n) {
```

```
      # regress, leaving out ith observation
      lmo <- lm(y[-i] ~ xmat[-i,])
      betahat <- as.vector(lmo$coef)
      # the 1 accommodates the constant term
      predy[i] <- betahat %*% c(1,xmat[i,])
   }
   return(predy)
}

# polynomial function of x, coefficients cfs
poly <- function(x,cfs) {
   val <- cfs[1]
   prod <- 1
   dg <- length(cfs) - 1
   for (i in 1:dg) {
      prod <- prod * x
      val <- val + cfs[i+1] * prod
   }
}
```

As noted, the only new code is `plot.polyreg()`. For convenience, the code is reproduced here:

```
# generic plot(); plots fits against raw data
plot.polyreg <- function(fits) {
   plot(fits$x,fits$y,xlab="X",ylab="Y")  # plot data points as background
   maxdg <- length(fits) - 2
   cols <- c("red","green","blue")
   dg <- curvecount <- 1
   while (dg < maxdg) {
      prompt <- paste("RETURN for XV fit for degree",dg,"or type degree",
         "or q for quit ")
      rl <- readline(prompt)
      dg <- if (rl == "") dg else if (rl != "q") as.integer(rl) else break
      lines(fits$x,fits[[dg]]$fitted.values,col=cols[curvecount%%3 + 1])
      dg <- dg + 1
      curvecount <- curvecount + 1
   }
}
```

As before, our implementation of the generic function takes the name of the class, which is `plot.polyreg()` here.

The `while` loop iterates through the various polynomial degrees. We cycle through three colors, by setting the vector `cols`; note the expression `curvecount %%3` for this purpose.

The user can choose either to plot the next sequential degree or select a different one. The query, both user prompt and reading of the user's reply, is done in this line:

```
rl <- readline(prompt)
```

We use the R string function `paste()` to assemble a prompt, offering the user a choice of plotting the next fitted polynomial, plotting one of a different degree, or quitting. The prompt appears in the interactive R window in which we issued the `plot()` call. For instance, after taking the default choice twice, the command window looks like this:

```
> plot(lmo)
RETURN for XV fit for degree 1 or type degree or q for quit
RETURN for XV fit for degree 2 or type degree or q for quit
RETURN for XV fit for degree 3 or type degree or q for quit
```

The plot window looks like Figure 12-5.

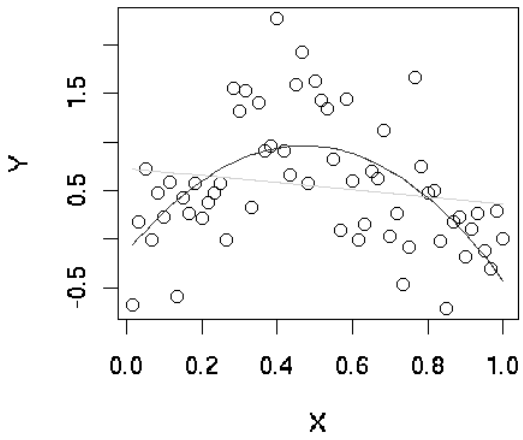


Figure 12-5: Plotting a polynomial fit

12.1.6 Adding Points: The points() Function

The `points()` function adds a set of (x,y) points, with labels for each, to the currently displayed graph. For instance, in our first example, suppose we entered this command:

```
points(testscores$Exam1,testscores$Exam3,pch="+")
```

The result would be to superimpose onto the current graph the points of the exam scores from that example, using plus signs (+) to mark them.

As with most of the other graphics functions, there are many options, such as point color and background color. For instance, if you want a yellow background, type this command:

```
> par(bg="yellow")
```

Now your graphs will have a yellow background, until you specify otherwise.

As with other functions, to explore the myriad of options, type this:

```
> help(par)
```

12.1.7 Adding a Legend: The legend() Function

The `legend()` function is used, not surprisingly, to add a legend to a multi-curve graph. This could tell the viewer something like, "The green curve is for the men, and the red curve displays the data for the women." Type the following to see some nice examples:

```
> example(legend)
```

12.1.8 Adding Text: The text() Function

Use the `text()` function to place some text anywhere in the current graph. Here's an example:

```
text(2.5,4,"abc")
```

This writes the text "abc" at the point (2.5,4) in the graph. The center of the string, in this case "b," would go at that point.

To see a more practical example, let's add some labels to the curves in our exam scores graph, as follows:

```
> text(46.7,0.02,"Exam 1")
> text(12.3,0.008,"Exam 2")
```

The result is shown in Figure 12-6.

In order to get a certain string placed exactly where you want it, you may need to engage in some trial and error. Or you may find the `locator()` function to be a much quicker way to go, as detailed in the next section.

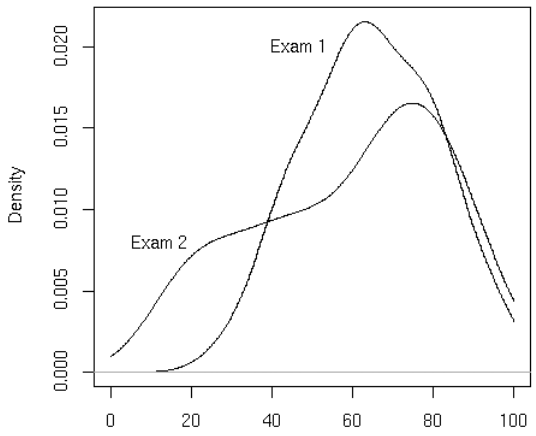


Figure 12-6: Placing text

12.1.9 Pinpointing Locations: The locator() Function

Placing text exactly where you wish can be tricky. You could repeatedly try different *x*- and *y*-coordinates until you find a good position, but the `locator()` function can save you a lot of trouble. You simply call the function and then click the mouse at the desired spot in the graph. The function returns the *x*- and *y*-coordinates of your click point. Specifically, typing the following will tell R that you will click in one place in the graph:

```
locator(1)
```

Once you click, R will tell you the exact coordinates of the point you clicked. Call `locator(2)` to get the locations of two places, and so on. (Warning: Make sure to include the argument.)

Here is a simple example:

```
> hist(c(12,5,13,25,16))
> locator(1)
$x
[1] 6.239237

$y
[1] 1.221038
```

This has R draw a histogram and then calls `locator()` with the argument `1`, indicating we will click the mouse once. After the click, the function returns a list with components `x` and `y`, the *x*- and *y*-coordinates of the point where we clicked.

To use this information to place text, combine it with `text()`:

```
> text(locator(1),"nv=75")
```

Here, `text()` was expecting an *x*-coordinate and a *y*-coordinate, specifying the point at which to draw the text "nv=75." The return value of `locator()` supplied those coordinates.

12.1.10 Restoring a Plot

R has no "undo" command. However, if you suspect you may need to undo your next step when building a graph, you can save it using `recordPlot()` and then later restore it with `replayPlot()`.

Less formally but more conveniently, you can put all the commands you're using to build up a graph in a file and then use `source()`, or cut and paste with the mouse, to execute them. If you change one command, you can redo the whole graph by sourcing or copying and pasting your file.

For our current graph, for instance, we could create file named `examplot.R` with the following contents:

```
d1 = density(testscores$Exam1,from=0,to=100)
d2 = density(testscores$Exam2,from=0,to=100)
plot(d1,main="",xlab="")
lines(d2)
text(46.7,0.02,"Exam 1")
text(12.3,0.008,"Exam 2")
```

If we decide that the label for exam 1 was a bit too far to the right, we can edit the file and then either do the copy-and-paste or execute the following:

```
> source("examplot.R")
```

12.2 Customizing Graphs

You've seen how easy it is to build simple graphs in stages, starting with plot(). Now you can begin to enhance those graphs, using the many options R provides.

12.2.1 Changing Character Sizes: The cex Option

The `cex` (for *character expand*) function allows you to expand or shrink characters within a graph, which can be very useful. You can use it as a named

parameter in various graphing functions. For instance, you may wish to draw the text "abc" at some point, say (2.5,4), in your graph but with a larger font, in order to call attention to this particular text. You could do this by typing the following:

```
text(2.5,4,"abc",cex = 1.5)
```

This prints the same text as in our earlier example but with characters 1.5 times the normal size.

12.2.2 Changing the Range of Axes: The xlim and ylim Options

You may wish to have the ranges on the *x*- and *y*-axes of your plot be broader or narrower than the default. This is especially useful if you will be displaying several curves in the same graph.

You can adjust the axes by specifying the `xlim` and/or `ylim` parameters in your call to `plot()` or `points()`. For example, `ylim=c(0,90000)` specifies a range on the *y*-axis of 0 to 90,000.

If you have several curves and do not specify `xlim` and/or `ylim`, you should draw the tallest curve first so there is room for all of them. Otherwise, R will fit the plot to the first one your draw and then cut off taller ones at the top! We took this approach earlier, when we plotted two density estimates on the same graph (Figures 12-3 and 12-4). Instead, we could have first found the highest values of the two density estimates. For `d1`, we find the following:

```
> d1

Call:
        density.default(x = testscores$Exam1, from = 0, to = 100)

Data: testscores$Exam1 (39 obs.);        Bandwidth 'bw' = 6.967

       x             y
 Min.   :  0   Min.   :1.423e-07
 1st Qu.: 25   1st Qu.:1.629e-03
 Median : 50   Median :9.442e-03
 Mean   : 50   Mean   :9.844e-03
 3rd Qu.: 75   3rd Qu.:1.756e-02
 Max.   :100   Max.   :2.156e-02
```

So, the largest y-value is 0.022. For `d2`, it was only 0.017. That means we should have plenty of room if we set `ylim` at 0.03. Here is how we could draw the two plots on the same picture:

```
> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")
> lines(d2)
> lines(d1)
```

First we drew the bare-bones plot—just axes without innards, as shown in Figure 12-7. The first two arguments to `plot()` give `xlim` and `ylim`, so that the lower and upper limits on the Y axis will be 0 and 0.03. Calling `lines()` twice then fills in the graph, yielding Figures 12-8 and 12-9. (Either of the two `lines()` calls could come first, as we've left enough room.)

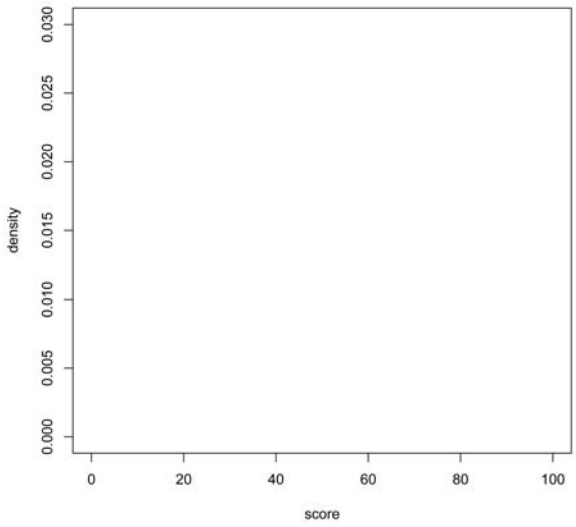


Figure 12-7: Axes only

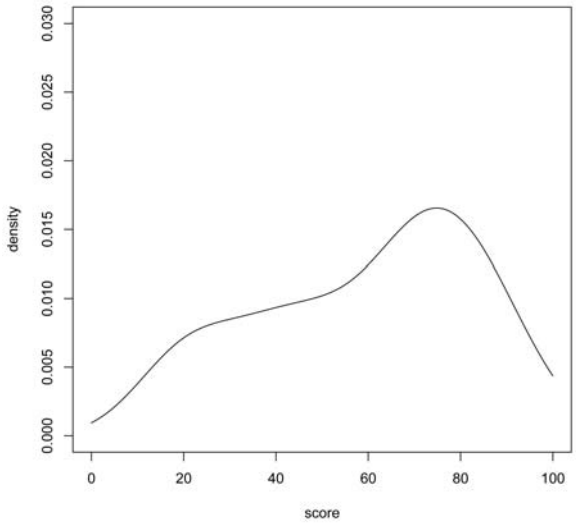


Figure 12-8: Addition of `d2`

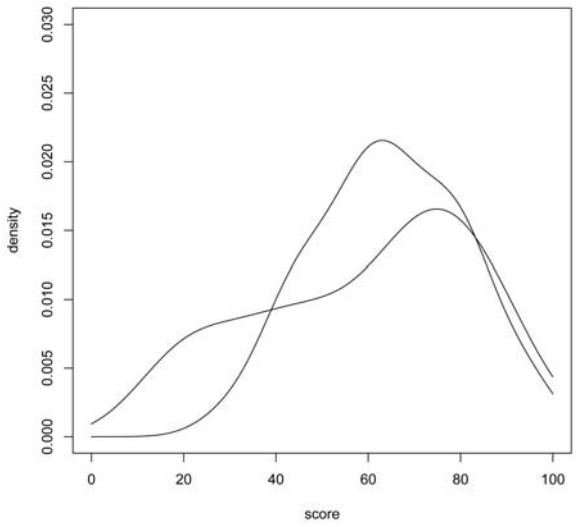


Figure 12-9: Addition of `d1`

12.2.3 Adding a Polygon: The polygon() Function

You can use `polygon()` to draw arbitrary polygonal objects. For example, the following code draws the graph of the function $f(x) = 1 - e^{-x}$ and then adds a rectangle that approximates the area under the curve from $x = 1.2$ to $x = 1.4$.

```
> f <- function(x) return(1-exp(-x))
> curve(f,0,2)
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")
```

The result is shown in Figure 12-10.

In the call to `polygon()` here, the first argument is the set of *x*-coordinates for the rectangle, and the second argument specifies the *y*-coordinates. The third argument specifies that the rectangle in this case should be shaded in solid gray.

As another example, we could use the `density` argument to fill the rectangle with striping. This call specifies 10 lines per inch:

```
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),density=10)
```

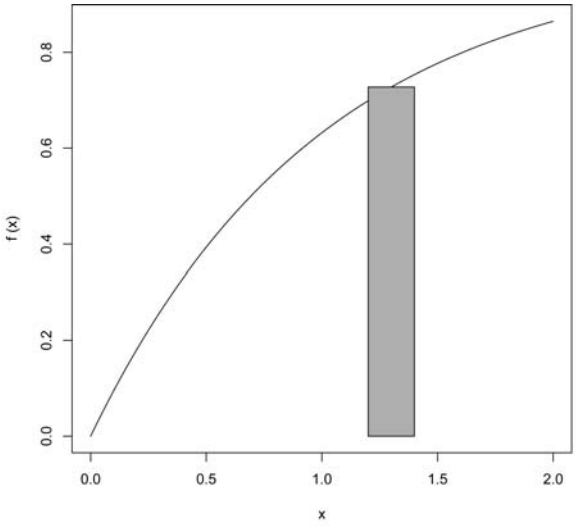


Figure 12-10: Rectangular area strip

12.2.4 Smoothing Points: The lowess() and loess() Functions

Just plotting a cloud of points, connected or not, may give you nothing but an uninformative mess. In many cases, it is better to smooth out the data by fitting a nonparametric regression estimator such as `lowess()`.

Let's do that for our test score data. We'll plot the scores of exam 2 against those of exam 1:

```
> plot(testscores)
> lines(lowess(testscores))
```

The result is shown in Figure 12-11.

A newer alternative to `lowess()` is `loess()`. The two functions are similar but have different defaults and other options. You need some advanced knowledge of statistics to appreciate the differences. Use whichever you find gives better smoothing.

12.2.5 Graphing Explicit Functions

Say you want to plot the function $g(t) = (t^2 + 1)^{0.5}$ for `t` between 0 and 5. You could use the following R code:

```
g <- function(t) { return (t^2+1)^0.5 }  # define g()
x <- seq(0,5,length=10000)  # x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x)  # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]
plot(x,y,type="l")
```

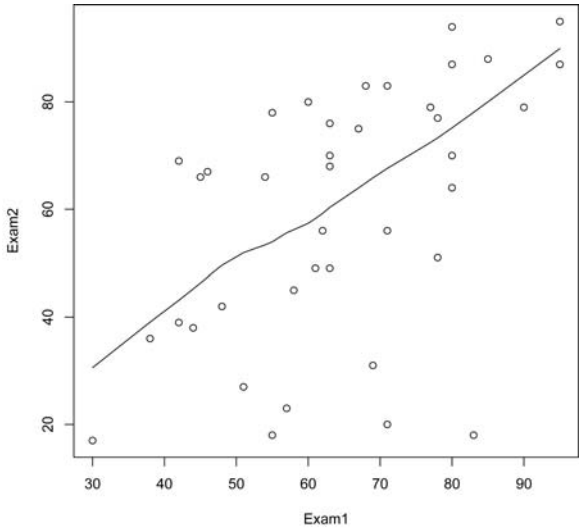


Figure 12-11: Smoothing the exam score relation

But you could avoid some work by using the `curve()` function, which basically uses the same method:

```
> curve((x^2+1)^0.5,0,5)
```

If you are adding this curve to an existing plot, use the `add` argument:

```
> curve((x^2+1)^0.5,0,5,add=T)
```

The optional argument `n` has the default value 101, meaning that the function will be evaluated at 101 equally spaced points in the specified range of `x`.

Use just enough points for visual smoothness. If you find 101 is not enough, experiment with higher values of `n`.

You can also use `plot()`, as follows:

```
> f <- function(x) return((x^2+1)^0.5)
> plot(f,0,5)  # the argument must be a function name
```

Here, the call `plot()` leads to calling `plot.function()`, the implementation of the generic `plot()` function for the `function` class.

Again, the approach is your choice; use whichever one you prefer.

12.2.6 Extended Example: Magnifying a Portion of a Curve

After you use `curve()` to graph a function, you may want to "zoom in" on one portion of the curve. You could do this by simply calling `curve()` again on

the same function but with a restricted *x* range. But suppose you wish to display the original plot and the close-up one in the same picture. Here, we will develop a function, which we'll name `inset()`, to do this.

In order to avoid redoing the work that `curve()` did in plotting the original graph, we will modify its code slightly to save that work, via a return value. We can do this by taking advantage of the fact that you can easily inspect the code of R functions written in R (as opposed to the fundamental R functions written in C), as follows:

```
> curve
function (expr, from = NULL, to = NULL, n = 101, add = FALSE,
    type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
{
    sexpr <- substitute(expr)
    if (is.name(sexpr)) {
# ...lots of lines omitted here...
    x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
        if (any(c(from, to) <= 0))
            stop("'from' and 'to' must be > 0 with log=\"x\"")
        exp(seq.int(log(from), log(to), length.out = n))
    }
    else seq.int(from, to, length.out = n)
    y <- eval(expr, envir = list(x = x), enclos = parent.frame())
    if (add)
        lines(x, y, type = type, ...)
    else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
}
```

The code forms vectors `x` and `y`, consisting of the *x*- and *y*-coordinates of the curve to be plotted, at `n` equally spaced points in the range of *x*. Since we'll make use of those in `inset()`, let's modify this code to return `x` and `y`. Here's the modified version, which we've named `crv()`:

```
> crv
function (expr, from = NULL, to = NULL, n = 101, add = FALSE,
    type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
{
    sexpr <- substitute(expr)
    if (is.name(sexpr)) {
# ...lots of lines omitted here...
    x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
        if (any(c(from, to) <= 0))
            stop("'from' and 'to' must be > 0 with log=\"x\"")
        exp(seq.int(log(from), log(to), length.out = n))
    }
    else seq.int(from, to, length.out = n)
    y <- eval(expr, envir = list(x = x), enclos = parent.frame())
    if (add)
```

```
        lines(x, y, type = type, ...)
    else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
    return(list(x=x,y=y))  # this is the only modification
}
```

Now we can get to our inset() function.

```
# savexy:  list consisting of x and y vectors returned by crv()
# x1,y1,x2,y2:  coordinates of rectangular region to be magnified
# x3,y3,x4,y4:  coordinates of inset region
inset <- function(savexy,x1,y1,x2,y2,x3,y3,x4,y4) {
   rect(x1,y1,x2,y2)  # draw rectangle around region to be magnified
   rect(x3,y3,x4,y4)  # draw rectangle around the inset
   # get vectors of coordinates of previously plotted points
   savex <- savexy$x
   savey <- savexy$y
   # get subscripts of xi our range to be magnified
   n <- length(savex)
   xvalsinrange <- which(savex >= x1 & savex <= x2)
   yvalsforthosex <- savey[xvalsinrange]
   # check that our first box contains the entire curve for that X range
   if (any(yvalsforthosex < y1 | yvalsforthosex > y2)) {
      print("Y value outside first box")
      return()
   }
   # record some differences
   x2mnsx1 <- x2 - x1
   x4mnsx3 <- x4 - x3
   y2mnsy1 <- y2 - y1
   y4mnsy3 <- y4 - y3
   # for the ith point in the original curve, the function plotpt() will
   # calculate the position of this point in the inset curve
   plotpt <- function(i) {
      newx <- x3 + ((savex[i] - x1)/x2mnsx1) * x4mnsx3
      newy <- y3 + ((savey[i] - y1)/y2mnsy1) * y4mnsy3
      return(c(newx,newy))
   }
   newxy <- sapply(xvalsinrange,plotpt)
   lines(newxy[1,],newxy[2,])
}
```

Let's try it out.

```
xyout <- crv(exp(-x)*sin(1/(x-1.5)),0.1,4,n=5001)
inset(xyout,1.3,-0.3,1.47,0.3,  2.5,-0.3,4,-0.1)
```

The resulting plot looks like Figure 12-12.

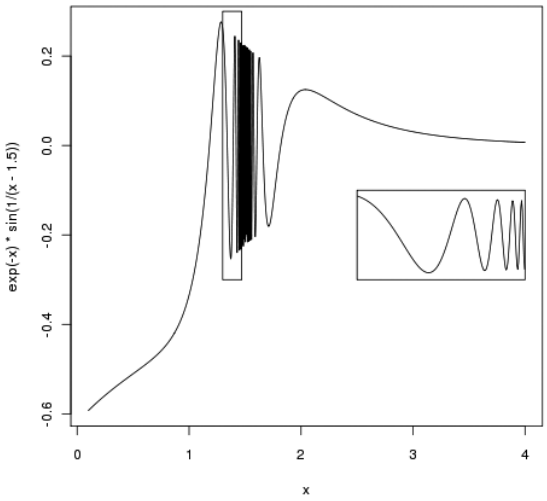


Figure 12-12: Adding an inset graph

12.3 Saving Graphs to Files

The R graphics display can consist of various graphics devices. The default device is the screen. If you want to save a graph to a file, you must set up another device.

Let's go through the basics of R graphics devices first to introduce R graphics device concepts, and then discuss a second approach that is much more direct and convenient.

12.3.1 R Graphics Devices

Let's open a file:

```
> pdf("d12.pdf")
```

This opens the file *d12.pdf*. We now have two devices open, as we can confirm:

```
> dev.list()
X11 pdf
  2   3
```

The screen is named `X11` when R runs on Linux. (It's named `windows` on Windows systems.) It is device number 2 here. Our PDF file is device number 3. Our active device is the PDF file:

```
> dev.cur()
pdf
  3
```

All graphics output will now go to this file instead of to the screen. But what if we wish to save what's already on the screen?

12.3.2 Saving the Displayed Graph

One way to save the graph currently displayed on the screen is to reestablish the screen as the current device and then copy it to the PDF device, which is 3 in our example, as follows:

```
> dev.set(2)
X11
  2
> dev.copy(which=3)
pdf
  3
```

But actually, it is best to set up a PDF device as shown earlier and then rerun whatever analyses led to the current screen. This is because the copy operation can result in distortions due to mismatches between screen devices and file devices.

12.3.3 Closing an R Graphics Device

Note that the PDF file we create is not usable until we close it, which we do as follows:

```
> dev.set(3)
pdf
  3
> dev.off()
X11
  2
```

You can also close the device by exiting R, if you're finished working with it. But in future versions of R, this behavior may not exist, so it's probably better to proactively close.

12.4 Creating Three-Dimensional Plots

R offers a number of functions to plot data in three dimensions such as `persp()` and `wireframe()`, which draw surfaces, and `cloud()`, which draws three-dimensional scatter plots. Here, we'll look at a simple example that uses `wireframe()`.

```
> library(lattice)
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg)
```

First, we load the `lattice` library. Then the call to `expand.grid()` creates a data frame, consisting of two columns named `x` and `y`, in all possible combinations of the values of the two inputs. Here, `a` and `b` had 10 and 15 values, respectively, so the resulting data frame will have 150 rows. (Note that the data frame that is input to `wireframe()` does not need to be created by `expand.grid()`.)

We then added a third column, named `z`, as a function of the first two columns. Our call to `wireframe()` creates the graph. The arguments, given in regression model form, specify that `z` is to be graphed against `x` and `y`. Of course, `z`, `x`, and `y` refer to names of columns in `eg`. The result is shown in Figure 12-13.

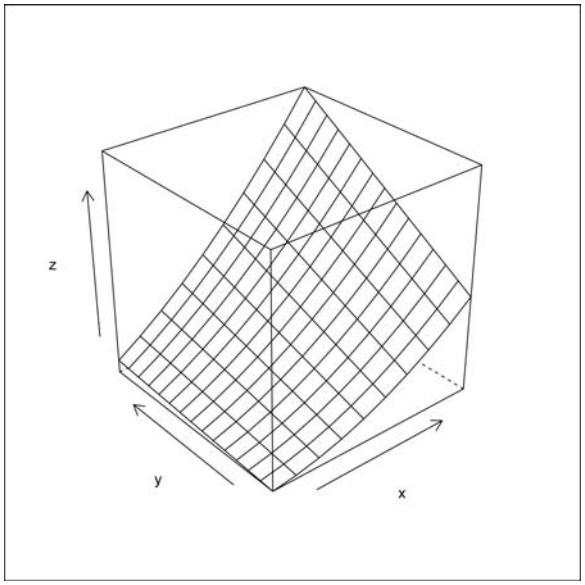


Figure 12-13: Example of using `wireframe()`

All the points are connected as a surface (like connecting points by lines in two dimensions). In contrast, with `cloud()`, the points are isolated.

For `wireframe()`, the (*x*,*y*) pairs must form a rectangular grid, though not necessarily be evenly spaced.

The three-dimensional plotting functions have many different options. For instance, a nice one for `wireframe()` is `shade=T`, which makes the data easier to see. Many functions, some with elaborate options, and whole new graphics packages work at a higher (read "more convenient and powerful") level of abstraction than R's base graphics package. For more information, refer to the books cited in footnote 1 at the beginning of this chapter.

15

INTERFACING R TO OTHER LANGUAGES

R is a great language, but it can't do everything well. Thus, it is sometimes desirable to call code written in other languages from R. Conversely, when working in other great languages, you may encounter tasks that could be better done in R.

R interfaces have been developed for a number of other languages, from ubiquitous languages like C to esoteric ones like the Yacas computer algebra system. This chapter will cover two interfaces: one for calling C/C++ from R and the other for calling R from Python.

15.1 Writing C/C++ Functions to Be Called from R

You may wish to write your own C/C++ functions to be called from R. Typically, the goal is performance enhancement, since C/C++ code may run much faster than R, even if you use vectorization and other R optimization techniques to speed things up.

Another possible goal in dropping down to the C/C++ level is specialized I/O. For example, R uses the TCP protocol in layer 3 of the standard Internet communication system, but UDP can be faster in some settings.

To work in UDP, you need C/C++, which requires an interface to R for those languages.

R actually offers two C/C++ interfaces via the functions `.C()` and `.Call()`. The latter is more versatile but requires some knowledge of R's internal structures, so we'll stick with `.C()` here.

15.1.1 Some R-to-C/C++ Preliminaries

In C, two-dimensional arrays are stored in row-major order, in contrast to R's column-major order. For instance, if you have a 3-by-4 array, the element in the second row and second column is element number 5 of the array when viewed linearly, since there are three elements in the first column and this is the second element in the second column. Also keep in mind that C subscripts begin at 0, rather than at 1, as with R.

All the arguments passed from R to C are received by C as pointers. Note that the C function itself must return `void`. Values that you would ordinarily return must be communicated through the function's arguments, such as `result` in the following example.

15.1.2 Example: Extracting Subdiagonals from a Square Matrix

Here, we will write C code to extract subdiagonals from a square matrix. (Thanks to my former graduate assistant, Min-Yu Huang, who wrote an earlier version of this function.) Here's the code for the file *sd.c*:

```
#include <R.h>  // required

// arguments:
//    m:  a square matrix
//    n:  number of rows/columns of m
//    k:  the subdiagonal index--0 for main diagonal, 1 for first
//        subdiagonal, 2 for the second, etc.
//    result:  space for the requested subdiagonal, returned here

void subdiag(double *m, int *n, int *k, double *result)
{
  int nval = *n, kval = *k;
  int stride = nval + 1;
  for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
     result[i] = m[j];
}
```

The variable `stride` alludes to a concept from the parallel-processing community. Say we have a matrix in 1,000 columns and our C code is looping through all the elements in a given column, from top to bottom. Again, since C uses row-major order, consecutive elements in the column are 1,000 elements apart from each other if the matrix is viewed as one long vector.

Here, we would say that we are traversing that long vector with a stride of 1,000—that is, accessing every thousandth element.

15.1.3 Compiling and Running Code

You compile your code using R. For example, in a Linux terminal window, we could compile our file like this:

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include      -fpic  -g -O2 -c sd.c -o sd.o
gcc -std=gnu99 -shared  -o sd.so sd.o   -L/usr/lib/R/lib -lR
```

This would produce the dynamic shared library file *sd.so*.

Note that R has reported how it invoked GCC in the output of the example. You can also run these commands by hand if you have special requirements, such as special libraries to be linked in. Also note that the locations of the *include* and *lib* directories may be system-dependent.

NOTE *GCC is easily downloadable for Linux systems. For Windows, it is included in Cygwin, an open source package available from* http://www.cygwin.com/.

We can then load our library into R and call our C function like this:

```
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
result=double(dim(m)[1]-k))
[[1]]
 [1]  1  6 11 16 21  2  7 12 17 22  3  8 13 18 23  4  9 14 19 24  5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

For convenience here, we've given the name `result` to both the formal argument (in the C code) and the actual argument (in the R code). Note that we needed to allocate space for `result` in our R code.

As you can see from the example, the return value takes on the form of a list consisting of the arguments in the R call. In this case, the call had four arguments (in addition to the function name), so the returned list has four components. Typically, some of the arguments will be changed during execution of the C code, as was the case here with `result`.

15.1.4 Debugging R/C Code

Chapter 13 discussed a number of tools and methods for debugging R code. However, the R/C interface presents an extra challenge. The problem in using a debugging tool such as GDB here is that you must first apply it to R itself.

The following is a walk-through of the R/C debugging steps using GDB on our previous *sd.c* code as the example.

```
$ R -d gdb
GNU gdb 6.8-debian
...
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")
>   # hit ctrl-c here
Program received signal SIGINT, Interrupt.
0xb7ffa430 in __kernel_vsyscall ()
(gdb) b subdiag
Breakpoint 1 at 0xb77683f3: file sd.c, line 3.
(gdb) continue
Continuing.

Breakpoint 1, subdiag (m=0x92b9480, n=0x9482328, k=0x9482348, result=0x9817148)
    at sd.c:3
3          int nval = *n, kval = *k;
(gdb)
```

So, what happened in this debugging session?

1. We launched the debugger, GDB, with R loaded into it, from a command line in a terminal window:

   ```
   R -d gdb
   ```

2. We told GDB to run R:

   ```
   (gdb) run
   ```

3. We loaded our compiled C code into R as usual:

   ```
   > dyn.load("sd.so")
   ```

4. We hit the CTRL-C interrupt key pair to pause R and put us back at the GDB prompt.
5. We set a breakpoint at the entry to `subdiag()`:

   ```
   (gdb) b subdiag
   ```

6. We told GDB to resume executing R (we needed to hit the ENTER key a second time in order to get the R prompt):

```
(gdb) continue
```

We then executed our C code:

```
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
+ result=double(dim(m)[1]-k))

Breakpoint 1, subdiag (m=0x942f270, n=0x96c3328, k=0x96c3348, result=0x9a58148)
    at subdiag.c:46
46 if (*n < 1) error("n < 1\n");
```

At this point, we can use GDB to debug as usual. If you're not familiar with GDB, you may want to try one of the many quick tutorials on the Web. Table 15-1 lists some of the most useful commands.

Table 15-1: Common GDB Commands

Command	Description
`l`	List code lines
`b`	Set breakpoint
`r`	Run/rerun
`n`	Step to next statement
`s`	Step into function call
`p`	Print variable or expression
`c`	Continue
`h`	Help
`q`	Quit

15.1.5 Extended Example: Prediction of Discrete-Valued Time Series

Recall our example in Section 2.5.2 where we observed 0- and 1-valued data, one per time period, and attempted to predict the value in any period from the previous k values, using majority rule. We developed two competing functions for the job, `preda()` and `predb()`, as follows:

```
# prediction in discrete time series; 0s and 1s; use k consecutive
# observations to predict the next, using majority rule; calculate the
# error rate
preda <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   # the vector pred will contain our predicted values
   pred <- vector(length=n-k)
```

```
   for (i in 1:(n-k)) {
      if (sum(x[i:(i+(k-1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
   }
   return(mean(abs(pred-x[(k+1):n])))
}

predb <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   pred <- vector(length=n-k)
   sm <- sum(x[1:k])
   if (sm >= k2) pred[1] <- 1 else pred[1] <- 0
   if (n-k >= 2) {
      for (i in 2:(n-k)) {
         sm <- sm + x[i+k-1] - x[i-1]
         if (sm >= k2) pred[i] <- 1 else pred[i] <- 0
      }
   }
   return(mean(abs(pred-x[(k+1):n])))
}
```

Since the latter avoids duplicate computation, we speculated it would be faster. Now is the time to check that.

```
> y <- sample(0:1,100000,replace=T)
> system.time(preda(y,1000))
   user  system elapsed
  3.816   0.016   3.873
> system.time(predb(y,1000))
   user  system elapsed
  1.392   0.008   1.427
```

Hey, not bad! That's quite an improvement.

However, you should always ask whether R already has a fine-tuned function that will suit your needs. Since we're basically computing a moving average, we might try the `filter()` function, with a constant coefficient vector, as follows:

```
predc <- function(x,k) {
   n <- length(x)
   f <- filter(x,rep(1,k),sides=1)[k:(n-1)]
   k2 <- k/2
   pred <- as.integer(f >= k2)
   return(mean(abs(pred-x[(k+1):n])))
}
```

That's even more compact than our first version. But it's a lot harder to read, and for reasons we will explore soon, it may not be so fast. Let's check.

```
> system.time(predc(y,1000))
   user  system elapsed
  3.872   0.016   3.945
```

Well, our second version remains the champion so far. This actually should be expected, as a look at the source code shows. Typing the following shows the source for that function:

```
> filter
```

This reveals (not shown here) that `filter1()` is called. The latter is written in C, which should give us some speedup, but it still suffers from the duplicate computation problem—hence the slowness.

So, let's write our own C code.

```
#include <R.h>

void predd(int *x, int *n, int *k, double *errrate)
{
   int nval = *n, kval = *k, nk = nval - kval, i;
   int sm = 0;  // moving sum
   int errs = 0;  // error count
   int pred;  // predicted value
   double k2 = kval/2.0;
   // initialize by computing the initial window
   for (i = 0; i < kval; i++) sm += x[i];
   if (sm >= k2) pred = 1; else pred = 0;
   errs = abs(pred-x[kval]);
   for (i = 1; i < nk; i++) {
      sm = sm + x[i+kval-1] - x[i-1];
      if (sm >= k2) pred = 1; else pred = 0;
      errs += abs(pred-x[i+kval]);
   }
   *errrate = (double) errs / nk;
}
```

This is basically `predb()` from before, "hand translated" into C. Let's see if it will outdo `predb()`.

```
> system.time(.C("predd",as.integer(y),as.integer(length(y)),as.integer(1000),
+   errrate=double(1)))
   user  system elapsed
  0.004   0.000   0.003
```

The speedup is breathtaking.

You can see that writing certain functions in C can be worth the effort. This is especially true for functions that involve iteration, as R's own iteration constructs, such as `for()`, are slow.

15.2 Using R from Python

Python is an elegant and powerful language, but it lacks built-in facilities for statistical and data manipulation, two areas in which R excels. This section demonstrates how to call R from Python, using RPy, one of the most popular interfaces between the two languages.

15.2.1 Installing RPy

RPy is a Python module that allows access to R from Python. For extra efficiency, it can be used in conjunction with NumPy.

You can build the module from the source, available from *http://rpy.sourceforge.net*, or download a prebuilt version. If you are running Ubuntu, simply type this:

```
sudo apt-get install python-rpy
```

To load RPy from Python (whether in Python interactive mode or from code), execute the following:

```
from rpy import *
```

This will load a variable `r`, which is a Python class instance.

15.2.2 RPy Syntax

Running R from Python is in principle quite simple. Here is an example of a command you might run from the >>> Python prompt:

```
>>> r.hist(r.rnorm(100))
```

This will call the R function `rnorm()` to produce 100 standard normal variates and then input those values into R's histogram function, `hist()`.

As you can see, R names are prefixed by `r.`, reflecting the fact that Python wrappers for R functions are members of the class instance `r`.

The preceding code will, if not refined, produce ugly output, with your (possibly voluminous!) data appearing as the graph title and the *x*-axis label. You can avoid this by supplying a title and label, as in this example:

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

RPy syntax is sometimes less simple than these examples would lead you to believe. The problem is that R and Python syntax may clash. For instance,

consider a call to the R linear model function `lm()`. In our example, we will predict `b` from `a`.

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm('v2 ~ v1',data=r.data_frame(v1=a,v2=b))
```

This is somewhat more complex than it would have been if done directly in R. What are the issues here?

First, since Python syntax does not include the tilde character, we needed to specify the model formula via a string. Since this is done in R anyway, this is not a major departure.

Second, we needed a data frame to contain our data. We created one using R's `data.frame()` function. In order to form a period in an R function name, we need to use an underscore on the Python end. Thus we called `r.data_frame()`. Note that in this call, we named the columns of our data frame `v1` and `v2` and then used these in our model formula.

The output object is a Python dictionary (analog of R's `list` type), as you can see here (in part):

```
>>> lmout
{'qr': {'pivot': [1, 2], 'qr': array([[ -1.73205081, -17.32050808],
       [  0.57735027,  -6.164414  ],
       [  0.57735027,   0.78355007]]), 'qraux':
```

You should recognize the various attributes of `lm()` objects here. For example, the coefficients of the fitted regression line, which would be contained in `lmout$coefficients` if this were done in R, are here in Python as `lmout['coefficients']`. So, you can access those coefficients accordingly, for example like this:

```
>>> lmout['coefficients']
{'v1': 2.5263157894736841, '(Intercept)': -2.5964912280701729}
>>> lmout['coefficients']['v1']
2.5263157894736841
```

You can also submit R commands to work on variables in R's namespace, using the function `r()`. This is convenient if there are many syntax clashes. Here is how we could run the `wireframe()` example in Section 12.4 in RPy:

```
>>> r.library('lattice')
>>> r.assign('a',a)
>>> r.assign('b',b)
>>> r('g <- expand.grid(a,b)')
>>> r('g$Var3 <- g$Var1^2 + g$Var1 * g$Var2')
>>> r('wireframe(Var3 ~ Var1+Var2,g)')
>>> r('plot(wireframe(Var3 ~ Var1+Var2,g))')
```

First, we used `r.assign()` to copy a variable from Python's namespace to R's. We then ran `expand.grid()` (with a period in the name instead of an underscore, since we are running in R's namespace), assigning the result to `g`. Again, the latter is in R's namespace. Note that the call to `wireframe()` did not automatically display the plot, so we needed to call `plot()`.

The official documentation for RPy is at *http://rpy.sourceforge.net/rpy/doc/rpy.pdf*. Also, you can find a useful presentation, "RPy—R from Python," at *http://www.daimi.au.dk/~besen/TBiB2007/lecture-notes/rpy.html*.

16

PARALLEL R

Since many R users have very large computational needs, various tools for some kind of parallel operation of R have been devised. This chapter is devoted to parallel R.

Many a novice in parallel processing has, with great anticipation, written parallel code for some application only to find that the parallel version actually ran more slowly than the serial one. For reasons to be discussed in this chapter, this problem is especially acute with R.

Accordingly, understanding the nature of parallel-processing hardware and software is crucial to success in the parallel world. These issues will be discussed here in the context of common platforms for parallel R.

We'll start with a few code examples and then move to general performance issues.

16.1 The Mutual Outlinks Problem

Consider a network graph of some kind, such as web links or links in a social network. Let `A` be the *adjacency matrix* of the graph, meaning that, say, `A[3,8]` is 1 or 0, depending on whether there is a link from node 3 to node 8.

For any two vertices, say any two websites, we might be interested in mutual outlinks—that is, outbound links that are common to two sites. Suppose that we want to find the mean number of mutual outlinks, averaged

over all pairs of websites in our data set. This mean can be found using the following outline, for an *n*-by-*n* matrix:

```
1  sum = 0
2  for i = 0...n-1
3     for j = i+1...n-1
4        for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5  mean = sum / (n*(n-1)/2)
```

Given that our graph could contain thousands—even millions—of websites, our task could entail quite large amounts of computation. A common approach to dealing with this problem is to divide the computation into smaller chunks and then process each of the chunks simultaneously, say on separate computers.

Let's say that we have two computers at our disposal. We might have one computer handle all the odd values of `i` in the `for i` loop in line 2 and have the second computer handle the even values. Or, since dual-core computers are fairly standard these days, we could take this same approach on a single computer. This may sound simple, but a number of major issues can arise, as you'll learn in this chapter.

16.2 Introducing the snow Package

Luke Tierney's `snow` (Simple Network of Workstations) package, available from the CRAN R code repository, is arguably the simplest, easiest-to-use form of parallel R and one of the most popular.

NOTE *The CRAN Task View page on parallel R,* http://cran.r-project.org/web/views/HighPerformanceComputing.html*, has a fairly up-to-date list of available parallel R packages.*

To see how `snow` works, here's code for the mutual outlinks problem described in the previous section:

```
1  # snow version of mutual links problem
2
3  mtl <- function(ichunk,m) {
4     n <- ncol(m)
5     matches <- 0
6     for (i in ichunk) {
7        if (i < n) {
8           rowi <- m[i,]
9           matches <- matches +
10             sum(m[(i+1):n,] %*% rowi)
11       }
12    }
13    matches
14 }
```

```
15
16 mutlinks <- function(cls,m) {
17    n <- nrow(m)
18    nc <- length(cls)
19    # determine which worker gets which chunk of i
20    options(warn=-1)
21    ichunks <- split(1:n,1:nc)
22    options(warn=0)
23    counts <- clusterApply(cls,ichunks,mtl,m)
24    do.call(sum,counts) / (n*(n-1)/2)
25 }
```

Suppose we have this code in the file *SnowMutLinks.R*. Let's first discuss how to run it.

16.2.1 Running snow Code

Running the above `snow` code involves the following steps:

1. Load the code.
2. Load the `snow` library.
3. Form a `snow` cluster.
4. Set up the adjacency matrix of interest.
5. Run your code on that matrix on the cluster you formed.

Assuming we are running on a dual-core machine, we issue the following commands to R:

```
> source("SnowMutLinks.R")
> library(snow)
> cl <- makeCluster(type="SOCK",c("localhost","localhost"))
> testm <- matrix(sample(0:1,16,replace=T),nrow=4)
> mutlinks(cl,testm)
[1] 0.6666667
```

Here, we are instructing `snow` to start two new R processes on our machine (`localhost` is a standard network name for the local machine), which I will refer to here as *workers*. I'll refer to the original R process—the one in which we type the preceding commands—as the *manager*. So, at this point, three instances of R will be running on the machine (visible by running the `ps` command if you are in a Linux environment, for example).

The workers form a *cluster* in `snow` parlance, which we have named `cl`. The `snow` package uses what is known in the parallel-processing world as a *scatter/gather* paradigm, which works as follows:

1. The manager partitions the data into chunks and parcels them out to the workers (scatter phase).

2. The workers process their chunks.
3. The manager collects the results from the workers (gather phase) and combines them as appropriate to the application.

We have specified that communication between the manager and workers will be via network sockets (covered in Chapter 10).

Here's a test matrix to check the code:

```
> testm
  [,1] [,2] [,3] [,4]
[1,]    1    0    0    1
[2,]    0    0    0    0
[3,]    1    0    1    1
[4,]    0    1    0    1
```

Row 1 has zero outlinks in common with row 2, two in common with row 3, and one in common with row 4. Row 2 has zero outlinks in common with the rest, but row 3 has one in common with row 4. That is a total of four mutual outlinks out of $4 \times 3/2 = 6$ pairs—hence, the mean value of $4/6 = 0.6666667$, as you saw earlier.

You can make clusters of any size, as long as you have the machines. In my department, for instance, I have machines whose network names are `pc28`, `pc29`, and `pc30`. Each machine is dual core, so I could create a six-worker cluster as follows:

```
> cl6 <- makeCluster(type="SOCK",c("pc28","pc28","pc29","pc29","pc30","pc30"))
```

16.2.2 Analyzing the snow Code

Now let's see how the `mutlinks()` function works. First, we sense how many rows the matrix `m` has, in line 17, and the number of workers in our cluster, in line 18.

Next, we need to determine which worker will handle which values of `i` in the `for i` loop in our outline code shown earlier in Section 16.1. R's `split()` function is well suited for this. For instance, in the case of a 4-row matrix and a 2-worker cluster, that call produces the following:

```
> split(1:4,1:2)
$`1`
[1] 1 3

$`2`
[1] 2 4
```

An R list is returned whose first element is the vector (1,3) and the second is (2,4). This will set up having one R process work on the odd values of `i` and the other work on the even values, as we discussed earlier. We ward off the

warnings that split() would give us ("data length is not a multiple of split variable") by calling options().

The real work is done in line 23, where we call the snow function clusterApply(). This function initiates a call to the same specified function (mtl() here), with some arguments specific to each worker and some optional arguments common to all. So, here's what the call in line 23 does:

1. Worker 1 will be directed to call the function mtl() with the arguments ichunks[[1]] and m.
2. Worker 2 will call mtl() with the arguments ichunks[[2]] and m, and so on for all workers.
3. Each worker will perform its assigned task and then return the result to the manager.
4. The manager will collect all such results into an R list, which we have assigned here to counts.

At this point, we merely need to sum all the elements of counts. Well, I shouldn't say "merely," because there is a little wrinkle to iron out in line 24.

R's sum() function is capable of acting on several vector arguments, like this:

```
> sum(1:2,c(4,10))
[1] 17
```

But here, counts is an R list, not a (numeric) vector. So we rely on do.call() to extract the vectors from counts, and then we call sum() on them.

Note lines 9 and 10. As you know, in R, we try to vectorize our computation wherever possible for better performance. By casting things in matrix-times-vector terms, we replace the for j and for k loops in the outline in Section 16.1 by a single vector-based expression.

16.2.3 How Much Speedup Can Be Attained?

I tried this code on a 1000-by-1000 matrix m1000. I first ran it on a 4-worker cluster and then on a 12-worker cluster. In principle, I should have had speedups of 4 and 12, respectively. But the actual elapsed times were 6.2 seconds and 5.0 seconds. Compare these figures to the 16.9 seconds runtime in nonparallel form. (The latter consisted of the call mtl(1:1000,m1000).) So, I attained a speedup of about 2.7 instead of a theoretical 4.0 for a 4-worker cluster and 3.4 rather than 12.0 on the 12-node system. (Note that some timing variation occurs from run to run.) What went wrong?

In almost any parallel-processing application, you encounter *overhead*, or "wasted" time spent on noncomputational activity. In our example, there is overhead in the form of the time needed to send our matrix from the manager to the workers. We also encountered a bit of overhead in sending the function mtl() itself to the workers. And when the workers finish their tasks, returning their results to the manager causes some overhead, too. We'll

discuss this in detail when we talk about general performance considerations in in Section 16.4.1.

16.2.4 Extended Example: K-Means Clustering

To learn more about the capabilities of snow, we'll look at another example, this one involving k-means clustering (KMC).

KMC is a technique for exporatory data analysis. In looking at scatter plots of your data, you may have the perception that the observations tend to cluster into groups, and KMC is a method for finding such groups. The output consists of the centroids of the groups.

The following is an outline of the algorithm:

```
for iter = 1,2,...,niters
   set vector and count totals to 0
   for i = 1,...,nrow(m)
      set j = index of the closest group center to m[i,]
      add m[i,] to the vector total for group j, v[j]
      add 1 to the count total for group j, c[j]
   for j = 1,...,ngrps
      set new center of group j = v[j] / c[j]
```

Here, we specify niters iterations, with initcenters as our initial guesses for the centers of the groups. Our data is in the matrix m, and there are ngrps groups.

The following is the snow code to compute KMC in parallel:

```
# snow version of k-means clustering problem

library(snow)

# returns distances from x to each vector in y;
# here x is a single vector and y is a bunch of them;
# define distance between 2 points to be the sum of the absolute values
# of their componentwise differences; e.g., distance between (5,4.2) and
# (3,5.6) is 2 + 1.4 = 3.4
dst <- function(x,y) {
   tmpmat <- matrix(abs(x-y),byrow=T,ncol=length(x)) # note recycling
   rowSums(tmpmat)
}

# will check this worker's mchunk matrix against currctrs, the current
# centers of the groups, returning a matrix; row j of the matrix will
# consist of the vector sum of the points in mchunk closest to jth
# current center, and the count of such points
findnewgrps <- function(currctrs) {
   ngrps <- nrow(currctrs)
   spacedim <- ncol(currctrs) # what dimension space are we in?
```

```
22    # set up the return matrix
23    sumcounts <- matrix(rep(0,ngrps*(spacedim+1)),nrow=ngrps)
24    for (i in 1:nrow(mchunk)) {
25       dsts <- dst(mchunk[i,],t(currctrs))
26       j <- which.min(dsts)
27       sumcounts[j,] <- sumcounts[j,] + c(mchunk[i,],1)
28    }
29    sumcounts
30 }
31
32 parkm <- function(cls,m,niters,initcenters) {
33    n <- nrow(m)
34    spacedim <- ncol(m) # what dimension space are we in?
35    # determine which worker gets which chunk of rows of m
36    options(warn=-1)
37    ichunks <- split(1:n,1:length(cls))
38    options(warn=0)
39    # form row chunks
40    mchunks <- lapply(ichunks,function(ichunk) m[ichunk,])
41    mcf <- function(mchunk) mchunk <<- mchunk
42    # send row chunks to workers; each chunk will be a global variable at
43    # the worker, named mchunk
44    invisible(clusterApply(cls,mchunks,mcf))
45    # send dst() to workers
46    clusterExport(cls,"dst")
47    # start iterations
48    centers <- initcenters
49    for (i in 1:niters) {
50       sumcounts <- clusterCall(cls,findnewgrps,centers)
51       tmp <- Reduce("+",sumcounts)
52       centers <- tmp[,1:spacedim] / tmp[,spacedim+1]
53       # if a group is empty, let's set its center to 0s
54       centers[is.nan(centers)] <- 0
55    }
56    centers
57 }
```

The code here is largely similar to our earlier mutual outlinks example. However, there are a couple of new `snow` calls and a different kind of usage of an old call.

Let's start with lines 39 through 44. Since our matrix `m` does not change from one iteration to the next, we definitely do not want to resend it to the workers repeatedly, exacerbating the overhead problem. Thus, first we need to send each worker its assigned chunk of `m`, just once. This is done in line 44 via `snow`'s `clusterApply()` function, which we used earlier but need to get creative with here. In line 41, we define the function `mcf()`, which will, running

on a worker, accept the worker's chunk from the manager and then keep it as a global variable `mchunk` on the worker.

Line 46 makes use of a new `snow` function, `clusterExport()`, whose job it is to make copies of the manager's global variables at the workers. The variable in question here is actually a function, `dst()`. Here is why we need to send it separately: The call in line 50 will send the function `findnewgrps()` to the workers, but although that function calls `dst()`, `snow` will not know to send the latter as well. Therefore we send it ourselves.

Line 50 itself uses another new `snow` call, `clusterCall()`. This instructs each worker to call `findnewgrps()`, with `centers` as argument.

Recall that each worker has a different matrix chunk, so this call will work on different data for each worker. This once again brings up the controversy regarding the use of global variables, discussed in Section 7.8.4. Some software developers may be troubled by the use of a hidden argument in `findnewgrps()`. On the other hand, as mentioned earlier, using `mchunk` as an argument would mean sending it to the workers repeatedly, compromising performance.

Finally, take a look at line 51. The `snow` function `clusterApply()` always returns an R list. In this case, the return value is in `sumcounts`, each element of which is a matrix. We need to sum the matrices, producing a totals matrix. Using R's `sum()` function wouldn't work, as it would total all the elements of the matrices into a single number. Matrix addition is what we need.

Calling R's `Reduce()` function will do the matrix addition. Recall that any arithmetic operation in R is implemented as a function; in this case, it is implemented as the function `"+"`. The recall to `Reduce()` then successively applies `"+"` to the elements of the list `sumcounts`. Of course, we could just write a loop to do this, but using `Reduce()` may give us a small performance boost.

16.3 Resorting to C

As you've seen, using parallel R may greatly speed up your R code. This allows you to retain the convenience and expressive power of R, while still ameliorating large runtimes in big applications. If the parallelized R gives you sufficiently good performance, then all is well.

Nevertheless, parallel R is still R and thus still subject to the performance issues covered in Chapter 14. Recall that one solution offered in that chapter was to write a performance-critical portion of your code in C and then call that code from your main R program. (The references to C here mean C or C++.) We will explore this from a parallel-processing viewpoint. Here, instead of writing parallel R, we write ordinary R code that calls parallel C. (I assume a knowledge of C.)

16.3.1 Using Multicore Machines

The C code covered here runs only on multicore systems, so we must discuss the nature of such systems.

You are probably familiar with dual-core machines. Any computer includes a CPU, which is the part that actually runs your program. In essence, a dual-core machine has two CPUs, a quad-core system has four, and so on. With multiple cores, you can do parallel computation!

This parallel computation is done with *threads*, which are analogous to `snow`'s workers. In computationally intensive applications, you generally set up as many threads as there are cores, for example two threads in a dual-core machine. Ideally, these threads run simultaneously, though overhead issues do arise, as will be explained when we look at general performance issues in Section 16.4.1.

If your machine has multiple cores, it is structured as a *shared-memory* system. All cores access the same RAM. The shared nature of the memory makes communication between the cores easy to program. If a thread writes to a memory location, the change is visible to the other threads, without the programmer needing to insert code to make that happen.

16.3.2 Extended Example: Mutual Outlinks Problem in OpenMP

OpenMP is a very popular package for programming on multicore machines. To see how it works, here is the mutual outlinks example again, this time in R-callable OpenMP code:

```
#include <omp.h>
#include <R.h>

int tot;  // grand total of matches, over all threads

// processes row pairs (i,i+1), (i,i+2), ...
int procpairs(int i, int *m, int n)
{  int j,k,sum=0;
   for (j = i+1; j < n; j++) {
      for (k = 0; k < n; k++)
         // find m[i][k]*m[j][k] but remember R uses col-major order
         sum += m[n*k+i] * m[n*k+j];
   }
   return sum;
}

void mutlinks(int *m, int *n, double *mlmean)
{  int nval = *n;
   tot = 0;
   #pragma omp parallel
   {  int i,mysum=0,
          me = omp_get_thread_num(),
          nth = omp_get_num_threads();
      // in checking all (i,j) pairs, partition the work according to i;
      // this thread me will handle all i that equal me mod nth
      for (i = me; i < nval; i += nth) {
```

```
            mysum += procpairs(i,m,nval);
        }
        #pragma omp atomic
        tot += mysum;
    }
    int divisor = nval * (nval-1) / 2;
    *mlmean = ((float) tot)/divisor;
}
```

16.3.3 Running the OpenMP Code

Again, compilation follows the recipe in Chapter 15. We do need to link in the OpenMP library, though, by using the `-fopenmp` and `-lgomp` options. Suppose our source file is *romp.c*. Then we use the following commands to run the code:

```
gcc -std=gnu99 -fopenmp -I/usr/share/R/include -fpic -g -O2 -c romp.c -o romp.o
gcc -std=gnu99 -shared -o romp.so romp.o -L/usr/lib/R/lib -lR -lgomp
```

Here's an R test:

```
> dyn.load("romp.so")
> Sys.setenv(OMP_NUM_THREADS=4)
> n <- 1000
> m <- matrix(sample(0:1,n^2,replace=T),nrow=n)
> system.time(z <- .C("mutlinks",as.integer(m),as.integer(n),result=double(1)))
   user  system elapsed
  0.830   0.000   0.218
> z$result
[1] 249.9471
```

The typical way to specify the number of threads in OpenMP is through an operating system environment variable, `OMP_NUM_THREADS`. R is capable of setting operating system environment variables with the `Sys.setenv()` function. Here, I set the number of threads to 4, because I was running on a quad-core machine.

Note the runtime—only 0.2 seconds! This compares to the 5.0-second time we saw earlier for a 12-node `snow` system. This might be surprising to some readers, as our code in the `snow` version was vectorized to a fair degree, as mentioned earlier. Vectorizing is good, but again, R has many hidden sources of overhead, so C might do even better.

NOTE *I tried R's new byte-compilation function `cmpfun()`, but `mtl()` actually became slower.*

Thus, if you are willing to write part of your code in parallel C, dramatic speedups may be possible.

16.3.4 OpenMP Code Analysis

OpenMP code is C, with the addition of *pragmas* that instruct the compiler to insert some library code to perform OpenMP operations. Look at line 20, for instance. When execution reaches this point, the threads will be activated. Each thread then executes the block that follows—lines 21 through 31—in parallel.

A key point is variable scope. All the variables within the block starting on line 21 are local to their specific threads. For example, we've named the total variable in line 21 `mysum` because each thread will maintain its own sum. By contrast, the global variable `tot` on line 4 is held in common by all the threads. Each thread makes its contribution to that grand total on line 30.

But even the variable `nval` on line 18 is held in common with all the threads (during the execution of `mutlinks()`), as it is declared outside the block beginning on line 21. So, even though it is a local variable in terms of C scope, it is global to all the threads. Indeed, we could have declared `tot` on that line, too. It needs to be shared by all the threads, but since it's not used outside `mutlinks()`, it could have been declared on line 18.

Line 29 contains another pragma, `atomic`. This one applies only to the single line following it—line 30, in this case—rather than to a whole block. The purpose of the `atomic` pragma is to avoid what is called a *race condition* in parallel-processing circles. This term describes a situation in which two threads are updating a variable at the same time, which may produce incorrect results. The `atomic` pragma ensures that line 30 will be executed by only one thread at a time. Note that this implies that in this section of the code, our parallel program becomes temporarily serial, which is a potential source of slowdown.

Where is the manager's role in all of this? Actually, the manager is the original thread, and it executes lines 18 and 19, as well as `.C()`, the R function that makes the call to `mutlinks()`. When the worker threads are activated in line 21, the manager goes dormant. The worker threads become dormant once they finish line 31. At that point, the manager resumes execution. Due to the dormancy of the manager while the workers are executing, we do want to have as many workers as our machine has cores.

The function `procpairs()` is straightforward, but note the manner in which the matrix `m` is being accessed. Recall from the discussion in Chapter 15 on interfacing R to C that the two languages store matrices differently: column by column in R and row-wise in C. We need to be aware of that difference here. In addition, we have treated the matrix `m` as a one-dimensional array, as is common in parallel C code. In other words, if `n` is, say, 4, then we treat `m` as a vector of 16 elements. Due to the column-major nature of R matrix storage, the vector will consist first of the four elements of column 1, then the four of column 2, and so on. To further complicate matters, we must keep in mind that array indices in C start at 0, instead of starting at 1 as in R.

Putting all of this together yields the multiplication in line 12. The factors here are the (k,i) and (k,j) elements of the version of `m` in the C code, which are the (i+1,k+1) and (j+1,k+1) elements back in the R code.

16.3.5 Other OpenMP Pragmas

OpenMP includes a wide variety of possible operations—far too many to list here. This section provides an overview of some OpenMP pragmas that I consider especially useful.

16.3.5.1 The omp barrier Pragma

The parallel-processing term *barrier* refers to a line of code at which the threads rendezvous. The syntax for the `omp barrier` pragma is simple:

```
#pragma omp barrier
```

When a thread reaches a barrier, its execution is suspended until all other threads have reached that line. This is very useful for iterative algorithms; threads wait at a barrier at the end of every iteration.

Note that in addition to this explicit barrier invocation, some other pragmas place an implicit barrier following their blocks. These include `single` and `parallel`. There is an implied barrier immediately following line 31 in the previous listing, for example, which is why the manager stays dormant until all worker threads finish.

16.3.5.2 The omp critical Pragma

The block that follows this pragma is a *critical section*, meaning one in which only one thread is allowed to execute at a time. The `omp critical` pragma essentially serves the same purpose as the `atomic` pragma discussed earlier, except that the latter is limited to a single statement.

NOTE *The OpenMP designers defined a special pragma for this single-statement situation in the hope that the compiler can translate this to an especially fast machine instruction.*

Here is the `omp critical` syntax:

```
1 #pragma omp critical
2 {
3     // place one or more statements here
4 }
```

16.3.5.3 The omp single Pragma

The block that follows this pragma is to be executed by only one of the threads. Here is the syntax for the `omp single` pragma:

```
1 #pragma omp single
2 {
3     // place one or more statements here
4 }
```

This is useful for initializing sum variables that are shared by the threads, for instance. As noted earlier, an automatic barrier is placed after the block. This should make sense to you. If one thread is initializing a sum, you wouldn't want other threads that make use of this variable to continue execution until the sum has been properly set.

You can learn more about OpenMP in my open source textbook on parallel processing at *http://heather.cs.ucdavis.edu/parprocbook*.

16.3.6 GPU Programming

Another type of shared-memory parallel hardware consists of graphics processing units (GPUs). If you have a sophisticated graphics card in your machine, say for playing games, you may not realize that it is also a very powerful computational device—so powerful that the slogan "A supercomputer on your desk!" is often used to refer to PCs equipped with high-end GPUs.

As with OpenMP, the idea here is that instead of writing parallel R, you write R code interfaced to parallel C. (Similar to the OpenMP case, *C* here means a slightly augmented version of the C language.) The technical details become rather complex, so I won't show any code examples, but an overview of the platform is worthwhile.

As mentioned, GPUs do follow the shared-memory/threads model, but on a much larger scale. They have dozens, or even hundreds, of cores (depending on how you define *core*). One major difference is that several threads can be run together in a block, which can produce certain efficiencies.

Programs that access GPUs begin their run on your machine's CPU, referred to as the *host*. They then start code running on the GPU, or *device*. This means that your data must be transferred from the host to the device, and after the device finishes its computation, the results must be transferred back to the host.

As of this writing, GPU has not yet become common among R users. The most common usage is probably through the CRAN package `gputools`, which consists of some matrix algebra and statistical routines callable from R. For instance, consider matrix inversion. R provides the function `solve()` for this, but a parallel alternative is available in `gputools` with the name `gpuSolve()`.

For more about GPU programming, again see my book on parallel processing at *http://heather.cs.ucdavis.edu/parprocbook*.

16.4 General Performance Considerations

This section discusses some issues that you may find generally useful in parallelizing R applications. I'll present some material on the main sources of overhead and then discuss a couple of algorithmic issues.

16.4.1 Sources of Overhead

Having at least a rough idea of the physical causes of overhead is essential to successful parallel programming. Let's take a look at these in the contexts of the two main platforms, shared-memory and networked computers.

16.4.1.1 Shared-Memory Machines

As noted earlier, the memory sharing in multicore machines makes for easier programming. However, the sharing also produces overhead, since the two cores will bump into each other if they both try to access memory at the same time. This means that one of them will need to wait, causing overhead. That overhead is typically in the range of hundreds of nanoseconds (billionths of seconds). This sounds really small, but keep in mind that the CPU is working at a subnanosecond speed, so memory access often becomes a bottleneck.

Each core may also have a *cache*, in which it keeps a local copy of some of the shared memory. It's intended to reduce contention for memory among the cores, but it produces its own overhead, involving time spent in keeping the caches consistent with each other.

Recall that GPUs are special types of multicore machines. As such, they suffer from the problems I've described, and more. First, the *latency*, which is the time delay before the first bit arrives at the GPU from its memory after a memory read request, is quite long in GPUs.

There is also the overhead incurred in transferring data between the host and the device. The latency here is on the order of microseconds (millionths of seconds), an eternity compared to the nanosecond scale of the CPU and GPU.

GPUs have great performance potential for certain classes of applications, but overhead can be a major issue. The authors of `gputools` note that their matrix operations start achieving a speedup only at matrix sizes of 1000 by 1000. I wrote a GPU version of our mutual outlinks application, which turned out to have a runtime of 3.0 seconds—about half of the `snow` version but still far slower than the OpenMP implementation.

Again, there are ways of ameliorating these problems, but they require very careful, creative programming and a sophisticated knowledge of the physical GPU structure.

16.4.1.2 Networked Systems of Computers

As you saw earlier, another way to achieve parallel computation is through networked systems of computers. You still have multiple CPUs, but in this case, they are in entirely separate computers, each with its own memory.

As pointed out earlier, network data transfer causes overhead. Its latency is again on the order of microseconds. Thus, even accessing a small amount of data across the network incurs a major delay.

Also note that `snow` has additional overhead, as it changes numeric objects such as vectors and matrices to character form before sending them, say from the manager to the workers. Not only does this entail time for the conversion (both in changing from numeric to character form and

in charging back to numeric at the receiver), but the character form tends to make for much longer messages, thus longer network transfer time.

Shared-memory systems can be networked together, which, in fact, we did in the previous example. We had a hybrid situation in which we formed `snow` clusters from several networked dual-core computers.

16.4.2 Embarrassingly Parallel Applications and Those That Aren't

> It's no shame to be poor, but it's no great honor either.
>
> —Tevye, *Fiddler on the Roof*

> Man is the only animal that blushes, or needs to.
>
> —Mark Twain

The term *embarrassingly parallel* is heard often in talk about parallel R (and in the parallel processing field in general). The word *embarrassing* alludes to the fact that the problems are so easy to parallelize that there is no intellectual challenge involved; they are embarrassingly easy.

Both of the example applications we've looked at here would be considered embarrassingly parallel. Parallelizing the `for i` loop for the mutual outlinks problem in Section 16.1 was pretty obvious. Partitioning the work in the KMC example in Section 16.2.4 was also natural and easy.

By contrast, most parallel sorting algorithms require a great deal of interaction. For instance, consider merge sort, a common method of sorting numbers. It breaks the vector to be sorted into two (or more) independent parts, say the left half and right half, which are then sorted in parallel by two processes. So far, this is embarrassingly parallel, at least after the vector is divided in half. But then the two sorted halves must be merged to produce the sorted version of the original vector, and that process is *not* embarrassingly parallel. It can be parallelized but in a more complex manner.

Of course, to paraphrase Tevye, it's no shame to have an embarrassingly parallel problem! It may not exactly be an honor, but it is a cause for celebration, as it is easy to program. More important, embarrassingly parallel problems tend to have low communication overhead, which is crucial to performance, as discussed earlier. In fact, when most people refer to embarrassingly parallel applications, they have this low overhead in mind.

But what about nonembarrassingly parallel applications? Unfortunately, parallel R code is simply not suitable for many of them for a very basic reason: the functional programming nature of R. As discussed in Section 14.3, a statement like this:

```
x[3] <- 8
```

is deceptively simple, because it can cause the entire vector `x` to be rewritten. This really compounds communication traffic problems. Accordingly, if your application is not embarrassingly parallel, your best strategy is probably to write the computationally intensive parts of the code in C, say using OpenMP or GPU programming.

Also, note carefully that even being embarrassingly parallel does not make an algorithm efficient. Some such algorithms can still have significant communication traffic, thus compromising performance.

Consider the KMC problem, run under `snow`. Suppose we were to set up a large enough number of workers so that each worker had relatively little work to do. In that case, the communication with the manager after each iteration would become a signficant portion of run time. In this situation, we would say that the *granularity* is too fine, and then probably switch to using fewer workers. We would then have larger tasks for each worker, thus a *coarser* granularity.

16.4.3 Static Versus Dynamic Task Assignment

Look again at the loop beginning on line 26 of our OpenMP example, reproduced here for convenience:

```
for (i = me; i < nval; i += nth) {
   mysum += procpairs(i,m,nval);
}
```

The variable `me` here was the thread number, so the effect of this code was that the various threads would work on nonoverlapping sets of values of `i`. We do want the values to be nonoverlapping, to avoid duplicate work and an incorrect count of total number of links, so the code was fine. But the point now is that we were, in effect, preassigning the tasks that each thread would handle. This is called *static* assignment.

An alternative approach is to revise the `for` loop to look something like this:

```
int nexti = 0;  // global variable
...
for ( ; myi < n; ) {  // revised "for" loop
   #pragma omp critical
   {
      nexti += 1;
      myi = nexti;
   }
   if (myi < n)  {
      mysum += procpairs(myi,m,nval);
      ...
   }
}
...
```

This is *dynamic* task assignment, in which it is not determined ahead of time which threads handle which values of `i`. Task assignment is done during execution. At first glance, dynamic assignment seems to have the potential for better performance. Suppose, for instance, that in a static assignment

setting, one thread finishes its last value of `i` early, while another thread still has two values of `i` left to do. This would mean our program would finish somewhat later than it could. In parallel-processing parlance, we would have a *load balance* problem. With dynamic assignment, the thread that finished when there were two values of `i` left to handle could have taken up one of those values itself. We would have better balance and theoretically less overall runtime.

But don't jump to conclusions. As always, we have the overhead issue to reckon with. Recall that a `critical` pragma, used in the dynamic version of the code above, has the effect of temporarily rendering the program serial rather than parallel, thus causing a slowdown. In addition, for reasons too technical to discuss here, these pragmas may cause considerable cache activity overhead. So in the end, the dynamic code could actually be substantially slower than the static version.

Various solutions to this problem have been developed, such as an OpenMP construct named `guided`. But rather than present these, the point I wish to make is that they are unnecessary. In most situations, static assignment is just fine. Why is this the case?

You may recall that the standard deviation of the sum of independent, identically distributed random variables, divided by the mean of that sum, goes to zero as the number of terms goes to infinity. In other words, sums are approximately constant. This has a direct implication for our load-balancing concerns: Since the total work time for a thread in static assignment is the sum of its individual task times, that total work time will be approximately constant; there will be very little variation from thread to thread. Thus, they will all finish at pretty close to the same time, and we do not need to worry about load imbalance. Dynamic scheduling will not be necessary.

This reasoning does depend on a statistical assumption, but in practice, the assumption will typically be met sufficiently well for the outcome: Static scheduling does as well as dynamic in terms of uniformity of total work times across threads. And since static scheduling doesn't have the overhead problems of the dynamic kind, in most cases the static approach will give better performance.

There is one more aspect of this to discuss. To illustrate the issue, consider again the mutual outlinks example. Let's review the outline of the algorithm:

```
1  sum = 0
2  for i = 0...n-1
3     for j = i+1...n-1
4        for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5  mean = sum / (n*(n-1)/2)
```

Say `n` is 10000 and we have four threads, and consider ways to partition the `for i` loop. Naively, we might at first decide to have thread 0 handle the `i` values 0 through 2499, thread 1 handle 2500 through 4999, and so on. However, this would produce a severe load imbalance, since the thread that

handles a given value of i does an amount of work proportional to n-i. That, in fact, is why we staggered the values of i in our actual code: Thread 0 handled the i values 0, 4, 8 ..., thread 1 worked on 1, 5, 9, ..., and so on, yielding good load balance.

The point then is that static assignment might require a bit more planning. One general approach to this is to randomly assign tasks (i values, in our case here) to threads (still doing so at the outset, before work begins). With a bit of forethought such as this, static assignment should work well in most applications.

16.4.4 Software Alchemy: Turning General Problems into Embarrassingly Parallel Ones

As discussed earlier, it's difficult to attain good performance from nonembarrassingly parallel algorithms. Fortunately, for statistical applications, there is a way to turn nonembarrassingly parallel problems into embarrassingly parallel ones. The key is to exploit some statistical properties.

To demonstrate the method, let's once again turn to our mutual outlinks problem. The method, applied with w workers on a links matrix m, consists of the following:

1. Break the rows of m into w chunks.
2. Have each worker find the mean number of mutual outlinks for pairs of vertices in its chunk.
3. Average the results returned by the workers.

It can be shown mathematically that for large problems (the only ones you would need parallel computing for anyway), this chunked approach gives the estimators of the same statistical accuracy as in the nonchunked method. But meanwhile, we've turned a nonparallel problem into not just a parallel one but an embarrassingly parallel one! The workers in the preceding outline compute entirely independently of each other.

This method should not be confused with the usual chunk-based approaches in parallel processing. In those, such as the merge-sort example discussed on page 347, the chunking is embarrassingly parallel, but the combining of results is not. By contrast, here the combining of results consists of simple averaging, thanks to the mathematical theory.

I tried this approach on the mutual outlinks problem in a 4-worker snow cluster. This reduced the runtime to 1.5 seconds. This is far better than the serial time of about 16 seconds, double the speedup obtained by the GPU and approaching comparability to the OpenMP time. And the theory showing that the two methods give the same statistical accuracy was confirmed as well. The chunked method found the mean number of mutual outlinks to be 249.2881, compared to 249.2993 for the original estimator.

16.5 Debugging Parallel R Code

Parallel R packages such as `Rmpi`, `snow`, `foreach`, and so on do not set up a terminal window for each process, thus making it impossible to use R's debugger on the workers. (My `Rdsm` package, which adds a threads capability to R, is an exception to this.)

What then can you do to debug apps for those packages? Let's consider `snow` for a concrete example.

First, you should debug the underlying single-worker function, such as `mtl()` in Section 16.2. Here, we would set up some artificial values of the arguments and then use R's ordinary debugging facilities.

Debugging the underlying function may be sufficient. However, the bug may be in the arguments themselves or in the way we set them up. Then things get more difficult.

It's even hard to print out trace information, such as values of variables, since `print()` won't work in the worker processes. The `message()` function may work for some of these packages; if not, you may need to resort to using `cat()` to write to a file.

Regression

Linear Regression

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments. The other variable is called response variable whose value is derived from the predictor variable.

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is −

```
y = ax + b
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

StepstoEstablishaRegression

A simple example of regression is predicting weight of a person when his height is known. Todo this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height andcorresponding weight.

- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation usingthese
- Get a summary of the relationship model to know the average error in prediction. Alsocalled **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

Input Data

Below is the sample data representing the observations −

```
# Values of height
151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.
63, 81, 56, 91, 47, 57, 76, 72, 62, 48
```

lm()Function

This function creates the relationship model between the predictor and the

response variable.Syntax

The basic syntax for **lm()** function in linear regression is −

```
lm(formula,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the Coefficients

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function

.relation <- lm(y~x) print(relation)

When we execute the above code, it produces the following result −

```
Call:
lm(formula = y

Coefficien
ts:              x
(Intercept    0.67
```

Get the Summary of the Relationship

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function.

relation <- lm(y~x)

print(summary(relation))

When we execute the above code, it produces the following result −

Output:

```
Call:
lm(formula = y ~ x)

Residuals:
  Min    1Q   Median   3Q   Max
-6.3002-1.6629  0.0412     1.8944  3.9775

Coefficients:
       Estimate Std. Error t value
Pr(>|t|) (Intercept) -38.45509
                    8.04901  -4.778
0.00139 **
```

```
Residual standard error: 3.253 on 8 degrees of
freedom Multiple R-squared:  0.9548,
                              Adjusted R-squared:
```

predict() FunctionSyntax

The basic syntax for predict() in linear regression is –

predict(object, newdata)

Following is the description of the parameters used −

- **object** is the formula which is already created using the lm() function.
- **newdata** is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

# The resposne vector.

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

# Find weight of a person with height 170

a <- data.frame(x = 170)

result <-  predict(relation,a)

print(result)
```

When we execute the above code, it produces the following result −

```
      1
76.22869
```

Visualize the Regression Graphically

Create the predictor and response variable.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

relation <- lm(y~x)

Give the chart file a name.

png(file = "linearregression.png")

Plot the chart.

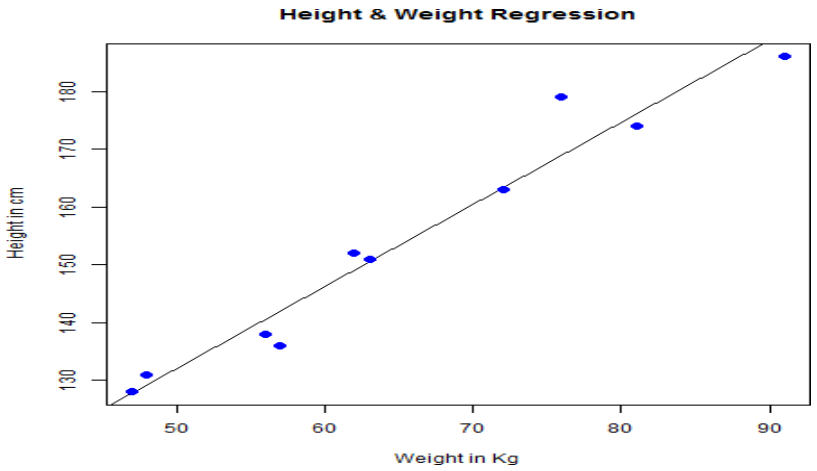
plot(y,x,col = "blue",main = "Height & Weight Regression",

abline(lm(x~y)), cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")

Save the file.

dev.off()

When we execute the above code, it produces the following result −



R-MultipleRegression

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

```
y = a + b1x1 + b2x2 +...bnxn
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value ofthe coefficients using the input data. Next we can predict the value of the response variable for agiven set of predictor variables using these coefficients.

lm()Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in multiple regression is −

```
lm(y ~ x1+x2+x3...,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictorvariables.
- **data** is the vector on which the formula will be applied.

Example:

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

```
input <- cars[,c("mpg","disp","hp","wt")]
```

Create Relationship Model & get the Coefficients

```
input <- mtcars[,c("mpg","disp","hp","wt")]

# Create the relationship model.
model <- lm(mpg~disp+hp+wt, data = input)

# Show the model.
print(model)

# Get the Intercept and coefficients as vector elements.
```

When we execute the above code, it produces the following result −

```
Call:
lm(formula = mpg ~ disp + hp + wt,

Coefficien
ts:          dis       h       w
(Intercept  -0.000937  -0.031157  -
```

Create Equation for Regression Model

Based on the above intercept and coefficient values, we create the mathematical equation.

```
Y = a+Xdisp.x1+Xhp.x2+Xwt.x3or
Y = 37.15+(-0.000937)*x1+(-0.0311)*x2+(-3.8008)*x3
```

Apply Equation for predicting New Values

We can use the regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.

For a car with disp = 221, hp = 102 and wt = 2.91 the predicted mileage is

```
Y = 37.15+(-0.000937)*221+(-0.0311)*102+(-3.8008)*2.91 = 22.7104
```

R-LogisticRegression

The Logistic Regression is a regression model in which the response variable (dependentvariable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

```
y = 1/(1+e^-(a+b1x1+b2x2+b3x3+...))
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

The basic syntax for **glm()** function in logistic regression is −

Generalized Linear Models Using R

GLMs (Generalized linear models) are a type of statistical model that is extensively used in the analysis of non-normal data, such as count data or binary data. They enable us to describe the connection between one or more predictor variables and a response variable in a flexible manner.

Major components of GLMs

- A probability distribution for the response variable
- A linear predictor function of the predictor variables
- A link function that connects the linear predictor to the response variable's mean.

The probability distribution and link function used is determined by the type of response variable and the research topic at hand. R includes methods for fitting GLMs, such as the glm() function. The user can specify the formula for the model, which contains the response variable and one or more predictor variables, as well as the probability distribution and link function to be used, using this function.

Mathematical Formulation of GLM

In Generalized Linear Models (GLMs), the response variable Y is assumed to follow a distribution from the exponential family. The model relates the expected value of Y, denoted μ, to the predictors X via a link function:

$$g(\mu)=X\beta$$

Here, β is the vector of model coefficients and $g(\cdot)$ is a specified link function. The variance of Y is given by:

$$Var(Y)=\phi V(\mu)$$

where $V(\mu)$ is the variance function and ϕ is a dispersion parameter.

Classical Linear Regression as a Special Case

In linear regression, $Y=X\beta+\epsilon$, with $\epsilon\sim N(0,\sigma 2)$, is a special case where:

- $g(\mu)=\mu$ (identity link)
- $V(\mu)=1$
- $\phi=\sigma^2$

Estimation

Model parametersβ are estimated via maximum likelihood. For observations (xi,yi), the likelihood is:

$$L(\beta)=\prod^{n}_{i=1} f(y_i|\mu_i)$$

where $f(\cdot)$ is the density function of the assumed distribution and μ_i is the expected value of Y_i given x_i.

GLM model families

There are several GLM model families depending on the make-up of the response variable. These includes three well-known GLM model families:

- **Binomial:** The binomial family is used for binary response variables (i.e., two categories) and assumes a binomial distribution.

```
model <- glm(binary_response_variable ~ predictor_variable1 + predictor_variable2,family =
binomial(link = "logit"), data = data)
```

- **Gaussian:** This family is used for continuous response variables and assumes a normal distribution. The link function for this family is typically the identity function.

```
model <- glm(response_variable ~ predictor_variable1 + predictor_variable2,
        family = gaussian(link = "identity"), data = data)
```

- **Gamma:** The gamma family is used for continuous response variables that are strictly positive and have a skewed distribution.

```
model <- glm(positive_response_variable ~ predictor_variable1 + predictor_variable2,family =
gamma(link = "inverse"), data = data)
```

- **Quasibinomial:** When a response variable is binary but has a higher variance than would be predicted by a binomial distribution, the quasibinomial model is utilized. This could happen if the response variable has excessive dispersion or additional variation that the model is not taking into account.

```
model <- glm(response_variable ~ predictor_variable1 + predictor_variable2,
        family = quasibinomial(), data = data)
```

Building a Generalized Linear Model

1. Loading the Dataset

We will use the "mtcars" dataset in R to illustrate the use of generalized linear models. This dataset includes data on different car models, including mpg, horsepower (hp) and weight. (wt). The response variable will be "mpg," and the predictor factors will be "hp" and "wt."

```
data(mtcars)
head(mtcars)
```

To create a generalized linear model in R, we must first select a suitable probability distribution for the answer variable.

- If the answer variable is binary (e.g., 0 or 1), we could use the Bernoulli distribution.
- If the response variable is a count (for example, the number of vehicles sold), the Poisson distribution may be used.

2. Building the model

To create a generalized linear model in R, use the glm() tool. We must describe the model formula (the response variable and the predictor variables) as well as the probability distribution family.

- data(mtcars)

- model <- glm(mpg ~ hp + wt, data = mtcars, family = gaussian)

The Gaussian family is used in this example, which implies that the response variable has a normal distribution.

3. Calculate summary of the model

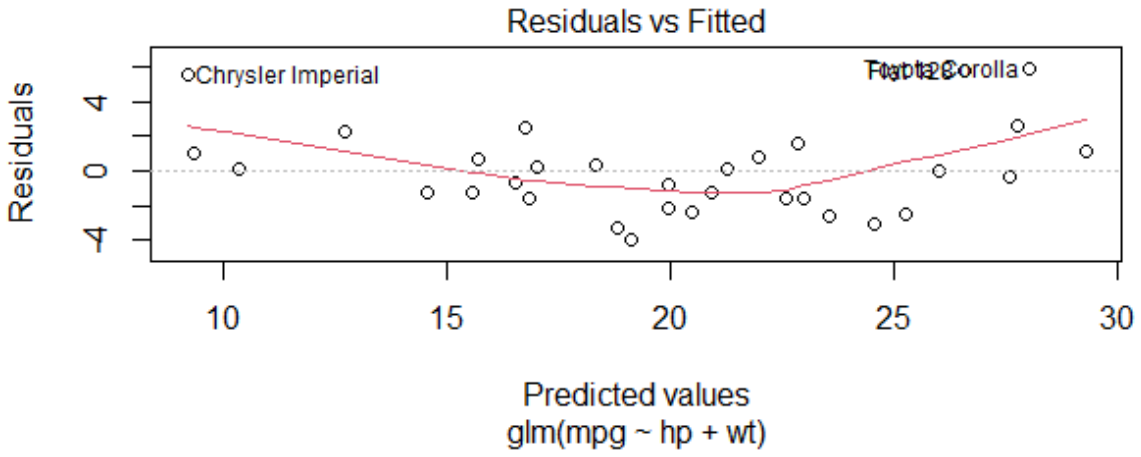
```
summary(model)
```

A one unit hp increase predicts a 0.03177 mpg decrease, while one unit wt increase predicts a 3.87783 mpg decrease.

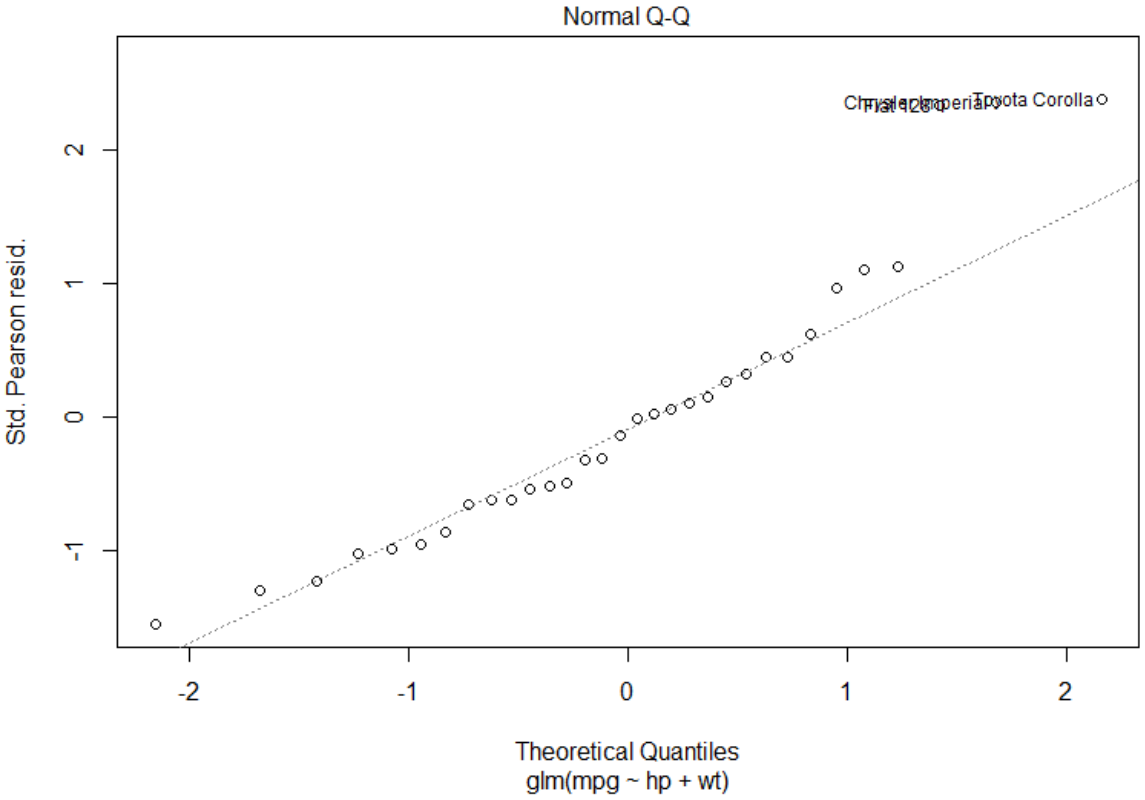
4. Visualize the model

After creating an extended linear model, we must evaluate its fit to the data. This can be accomplished with the help of diagnostic graphs such as the residual plot and the Q-Q plot.

```
plot(model, which = 1)

plot(model, which = 2)
```



The residual plot displays the residuals (differences between measured and predicted values) plotted against the fitted values. (i.e. the predicted values). We want to see a random scatter of residuals around zero, which indicates that the model is capturing the data trends.



The residuals Q-Q plot displays the residuals plotted against the anticipated values if they were normally distributed. The points should follow a straight line, showing that the residuals are normally distributed.

Time series Analysis

Time series is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day. Another example is the amount of rainfall in a region at different months of the year. R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored in an R object called time-series object. It is also a R data object like a vector or data frame.The time series object is created by using the ts() function.

Syntax
The basic syntax for ts() function in time series analysis is –

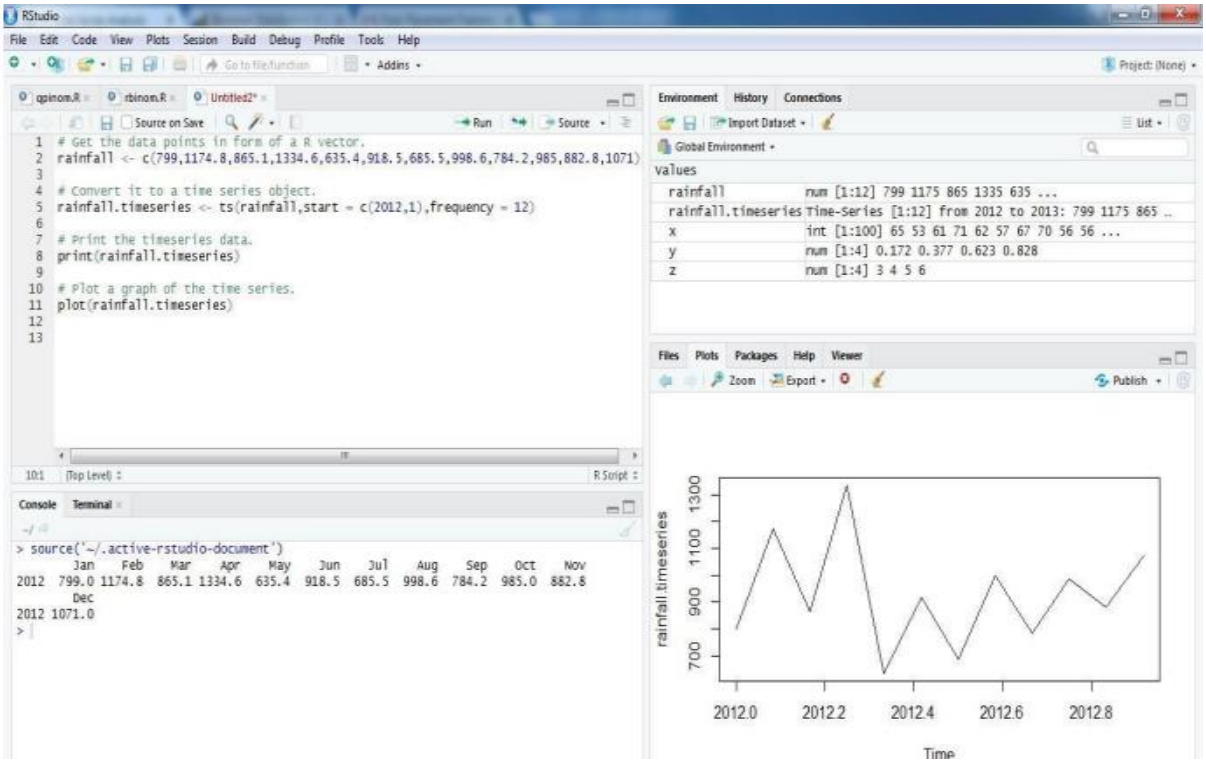
timeseries.object.name <- ts(data, start, end, frequency)

Following is the description of the parameters used –
- data is a vector or matrix containing the values used in the time series.
- start specifies the start time for the first observation in time series.
- end specifies the end time for the last observation in time series.
- frequency specifies the number of observations per unit time.

Except the parameter "data" all other parameters are optional.

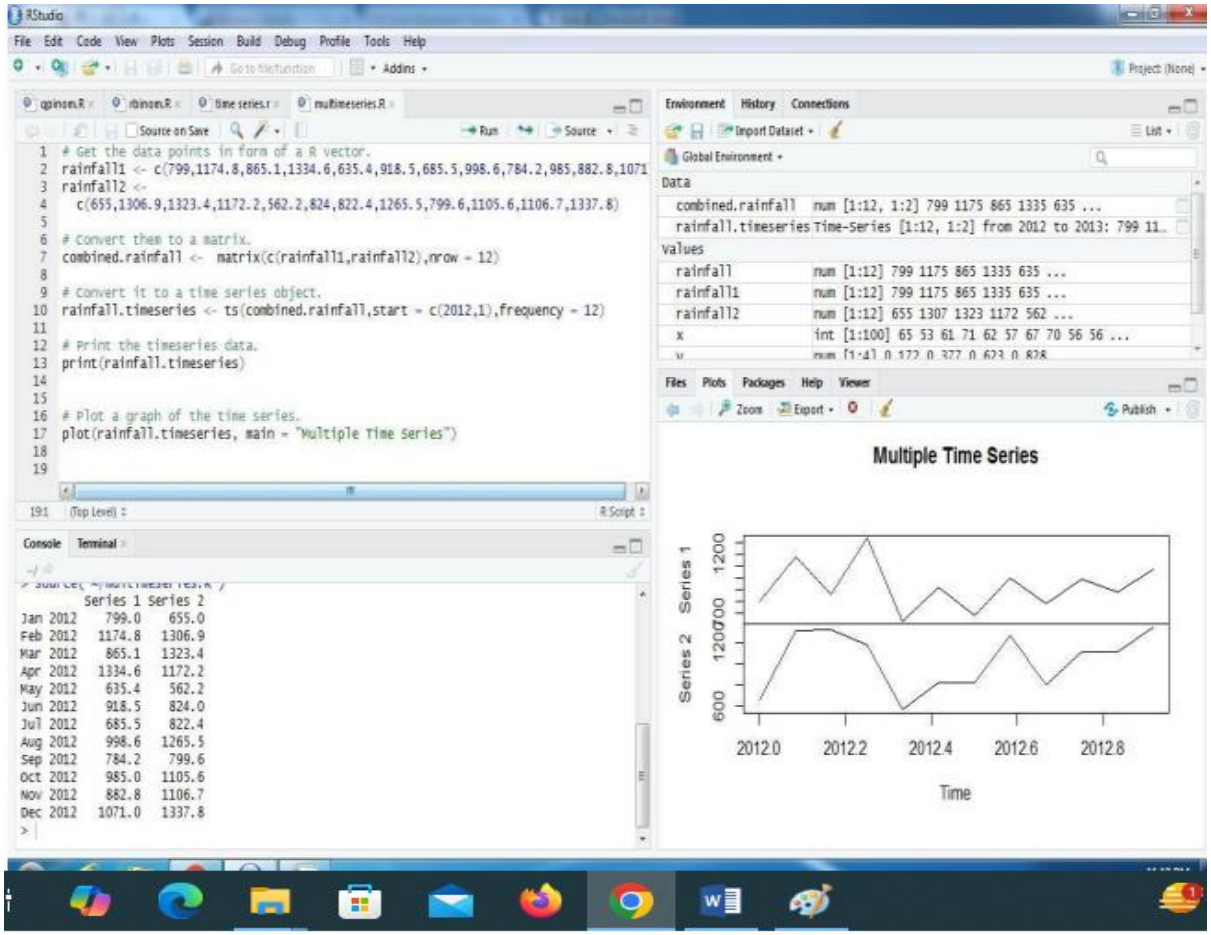
Example:

Consider the annual rainfall details at a place starting from January 2012. We create an R time series object for a period of 12 months and plot it.



MultipleTimeSeries

We can plot multiple time series in one chart by combining both the series into a matrix.



Autocorrelation

Autocorrelation is used to measure the degree of similarity between a time series and a lagged version of itself over the given range of time intervals. We can also call autocorrelation as "serial correlation" or "lagged correlation". It is mainly used to measure the relationship between the actual values and the previous values.

In R, we can calculate the autocorrelation in a vector by using the module tseries. Within this module, we have to use acf() method to calculate autocorrelation.

Syntax:
acf(vector, lag, pl)

Parameter:

- *vector is the input vector*
- *lag represents the number of lags*
- *pl is to plot the auto correlation*

Example: R program to calculate auto correlation in a vector with different lags

```
# load tseries module
library(tseries)

```

```
# create vector1 with 8 time periods
vector1=c(34,56,23,45,21,64,78,90)

# calculate auto correlation with no lag
print(acf(vector1,pl=FALSE))

# calculate auto correlation with lag 0
print(acf(vector1,lag=0,pl=FALSE))

# calculate auto correlation with lag 2
print(acf(vector1,lag=2,pl=FALSE))

# calculate auto correlation with lag 6
print(acf(vector1,lag=6,pl=FALSE))
```

Output:

```
Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6      7
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064 -0.151

Autocorrelations of series 'vector1', by lag

0
1

Autocorrelations of series 'vector1', by lag

    0     1     2
1.000 0.257 0.208

Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064
```

The same function can be used to visualize the output produced for that we simply have to set pl to TRUE

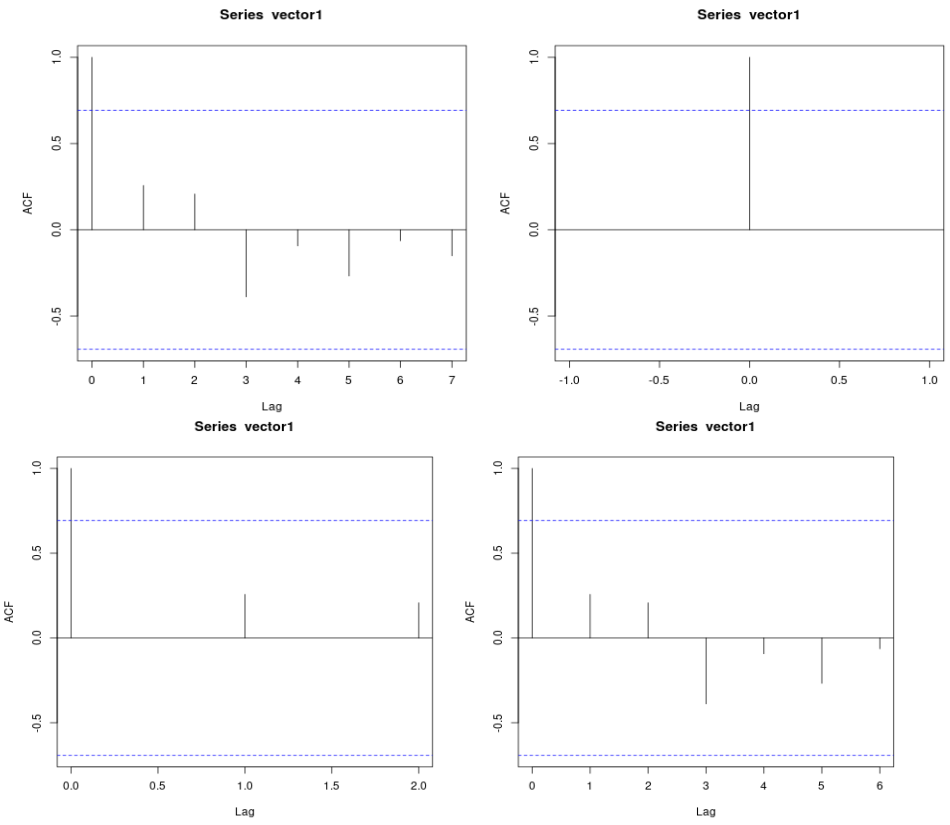
Example: Data visualization

```
# load tseries module
library(tseries)

# create vector1 with 8 time periods
vector1=c(34,56,23,45,21,64,78,90)

# calculate auto correlation with no lag
print(acf(vector1,pl=TRUE))

# calculate auto correlation with lag 0
print(acf(vector1,lag=0,pl=TRUE))
```

-
- *# calculate auto correlation with lag 2*
- print(acf(vector1,lag=2,pl=**TRUE**))
-
- *# calculate auto correlation with lag 6*
- print(acf(vector1,lag=6,pl=**TRUE**))

Output:

```
Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6      7
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064 -0.151

Autocorrelations of series 'vector1', by lag

0
1

Autocorrelations of series 'vector1', by lag

    0     1     2
1.000 0.257 0.208

Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064
```

- 

Clustering in R Programming

Clustering is an unsupervised learning technique where a dataset is divided into groups, or clusters, based on similarities among data points. It helps identify natural groupings within the data without prior labeling. Each cluster has data points that are closer to one another than to other clusters. This approach is commonly employed in data mining and pattern recognition to reveal latent structures, group behavior, or trends in the data. Clustering is particularly helpful when we need to explore data, reduce dimensionality, or pre-process data for additional supervised learning activities.

Types of Clustering in R Programming
In **R**, there are several clustering techniques, each suited for different data types and clustering challenges. Each method has its own advantages and is designed to handle specific data characteristics such as the number of clusters, their shapes, and whether or not noise is present in the data.

1. K-means clustering
The most common method is K-means, where the number of clusters (k) is set beforehand. It is computationally efficient for large datasets but can struggle with irregularly shaped clusters or varying densities. it is a data-partitioning technique that seeks to assign each observation to the cluster with the closest mean after dividing the data into k clusters.

2. Hierarchical Clustering
Hierarchical Clustering, on the other hand, creates a hierarchy of clusters by either merging smaller clusters (agglomerative) or splitting larger ones (divisive). This approach builds a tree like structure known as a dendrogram, which allows you to see the relationships between clusters at various levels of similarity. Hierarchical clustering is great for small to medium datasets where understanding the relationships between clusters is important, but it can be computationally expensive for larger datasets.

3. Spectral Clustering
Spectral Clustering transforms the clustering problem into a graph partitioning problem. By constructing a similarity graph from the data and performing clustering based on the eigenvalues of the graph's Laplacian matrix, it is able to capture complex, non convex clusters. This method works particularly well for datasets where the clusters are not linearly separable, but it can be computationally intensive and requires careful tuning of the similarity matrix.

4. Fuzzy Clustering
Fuzzy Clustering (or Fuzzy C-Means) is a soft clustering technique where data points are assigned membership scores for each cluster, rather than being definitively assigned to one cluster. This means a data point can belong to multiple clusters with varying degrees of membership. Fuzzy clustering is useful when the boundaries between clusters are not well defined, and it allows for more nuanced grouping, but interpreting the membership scores can be more complex.

5. Density Based Clustering
Density Based Clustering is a broader category that includes methods like DBSCAN. These methods focus on finding clusters based on regions of high data density, rather than relying on a distance metric. Density based methods are robust to noise and can find clusters of

arbitrary shapes. However, they can be sensitive to the parameters used, such as the minimum number of points needed to form a cluster.

6. Ensemble Clustering

Ensemble Clustering takes a different approach by combining the results of multiple clustering algorithms or multiple runs of the same algorithm to create a more reliable clustering solution. By aggregating the results of different methods, ensemble clustering aims to improve performance and reduce the risk of overfitting. This method is particularly useful when there is uncertainty about which clustering technique is the most appropriate, and it can provide more robust and stable results.

Each of these clustering techniques has its strengths and weaknesses, making it important to choose the right one based on the specific characteristics of your data and the goals of your analysis. Whether you're working with large datasets, noisy data, or data that requires soft assignments, there's a clustering method in R that can be tailored to your needs.

Implementation of K-Means Clustering in R Programming

We will implement K-Means Clustering algorithm here since it is simple and easy to understand. K-Means is an iterative hard clustering technique that uses an unsupervised learning algorithm. In this, total numbers of clusters are pre defined by the user and based on the similarity of each data point, the data points are clustered. This algorithm also finds out the centroid of the cluster.

Algorithm

1. **Specify number of clusters (K):** Let us take an example of k = 2 and 5 data points.
2. **Randomly assign each data point to a cluster:** In the below example, the red and green color shows 2 clusters with their respective random data points assigned to them.
3. **Calculate cluster centroids:** The cross mark represents the centroid of the corresponding cluster.
4. **Reallocate each data point to their nearest cluster centroid:** Green data point is assigned to the red cluster as it is near to the centroid of red cluster.
5. **Reconfigure cluster centroid.**

Syntax:

kmeans(x, centers, nstart)

where,

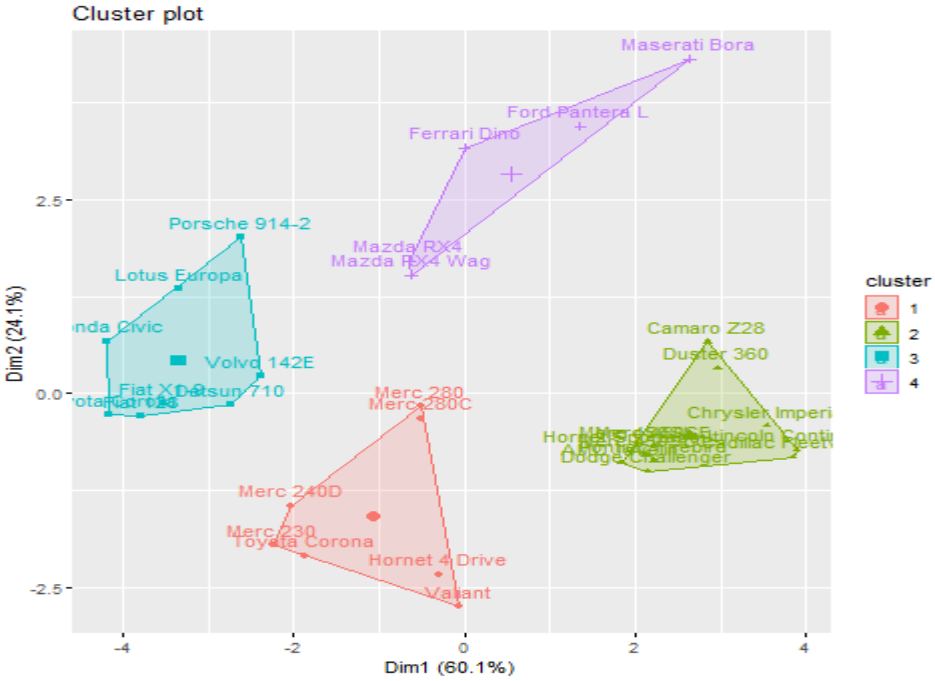
- **x :** represents numeric matrix or data frame object.
- **centers** : represents the **K** value or distinct cluster centers.
- **nstart** : represents number of random sets to be chosen.

Example

```
install.packages("factoextra")
library(factoextra)

df <- mtcars
df <- na.omit(df)
df <- scale(df)

km <- kmeans(df, centers = 4, nstart = 25)
fviz_cluster(km, data = df)

km <- kmeans(df, centers = 5, nstart = 25)
fviz_cluster(km, data = df)
```

Output:

When k = 4



When k = 5