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:

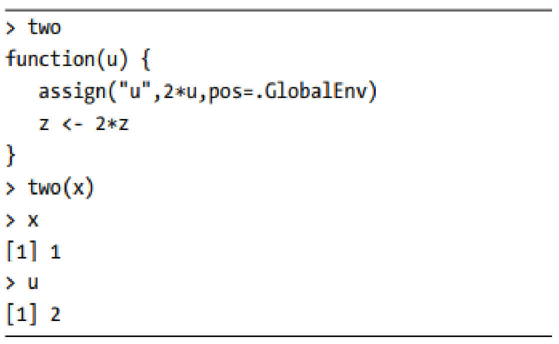
• x: Even though x was the actual argument to two() in the example, it retained the value 1 after the call. This is because its value 1 was copied to the formal argument u, which is treated as a local variable within the function. Thus, when u changed, x did not change with it.

• 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

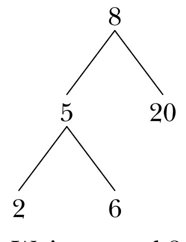
Treelike data structures are common in both computer science and statistics. In R, for example, the rpart library for a recursive partioning approach to regression and classification is very popular. Trees obviously have applications in genealogy, and more generally, graphs form the basis of analysis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

```
# class "bookvec" of vectors that count writes of their elements

# each instance of the class consists of a list whose components are the
# vector values and a vector of counts

# 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.

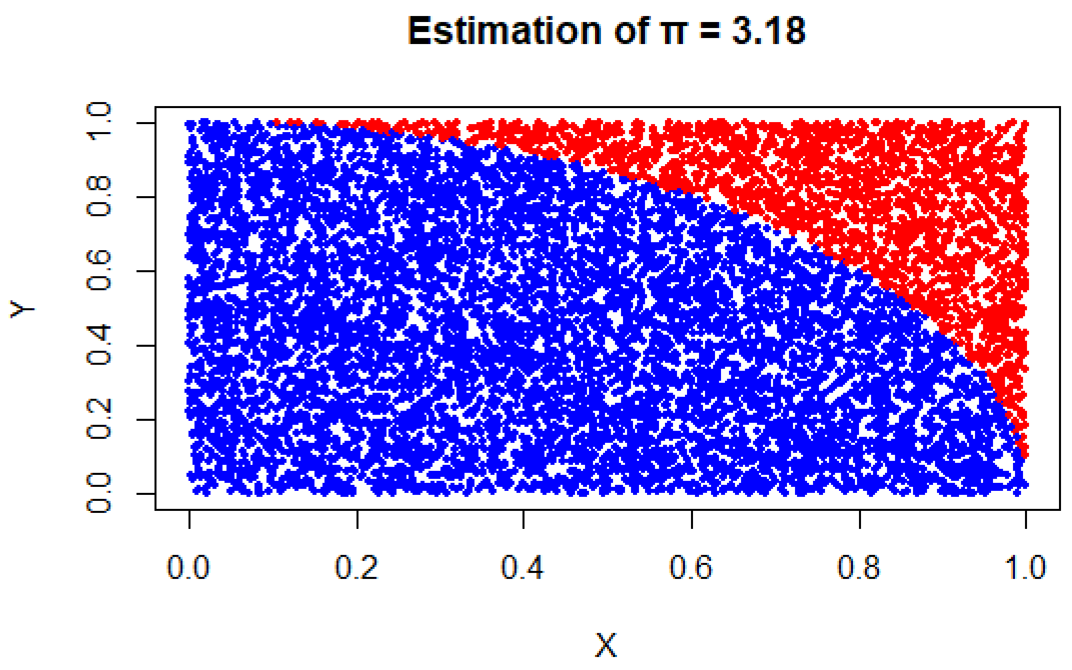
Let's start with a simple Monte Carlo simulation to estimate the value of π.

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.

```
# 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:

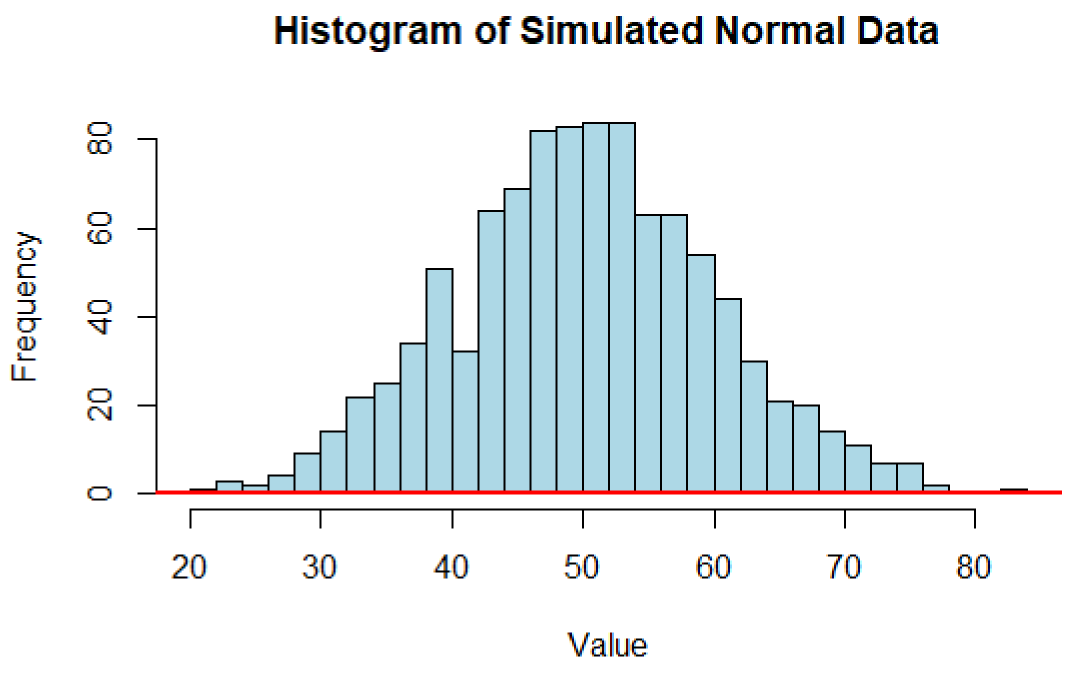


- **runif(n):** Generates n random numbers uniformly distributed between 0 and 1.
- **inside:** Logical vector indicating whether each point lies inside the unit circle.
- **sum(inside) / n * 4:** Estimates π using the ratio of points inside the circle to total points.

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

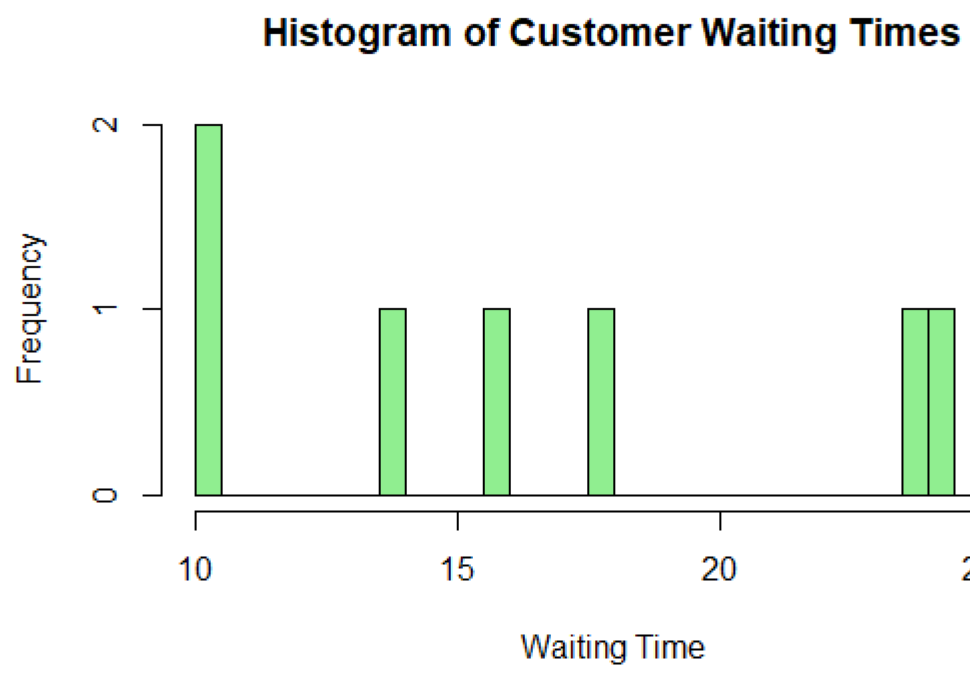
Let's simulate the operation of a simple queue system using the simmer package.

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

```r
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.

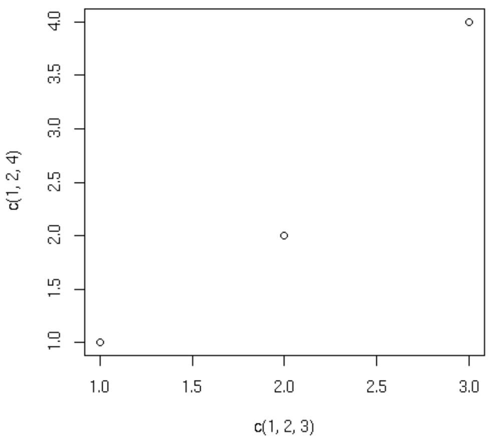


Figure 12-1: Simple point plot

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.

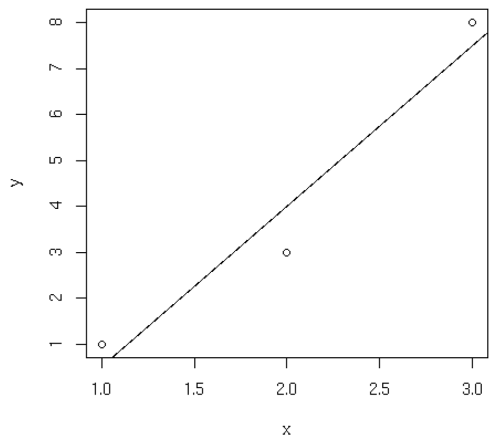


Figure 12-2: Using `abline()`

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.

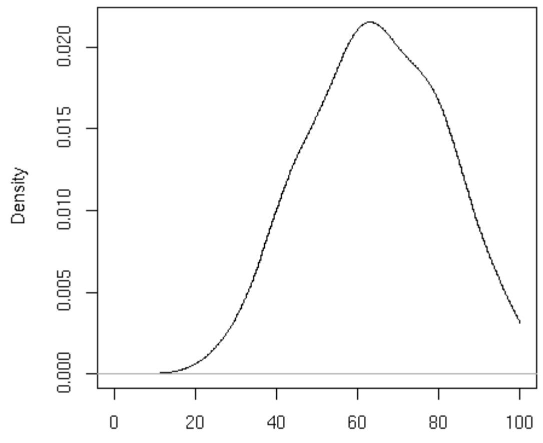


Figure 12-3: Plot of first density

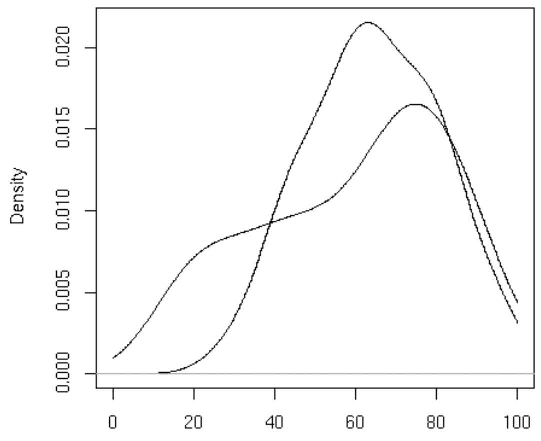


Figure 12-4: Addition of second density

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.

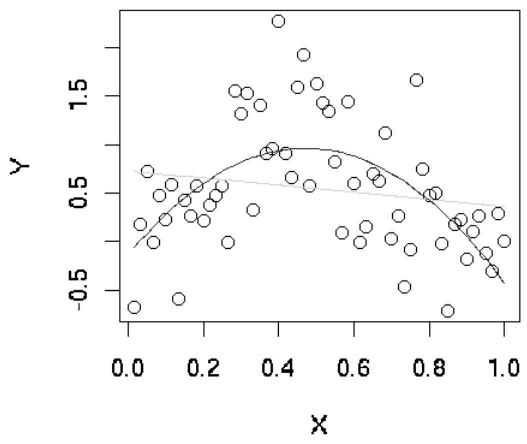


Figure 12-5: Plotting a polynomial fit

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 multicurve 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.

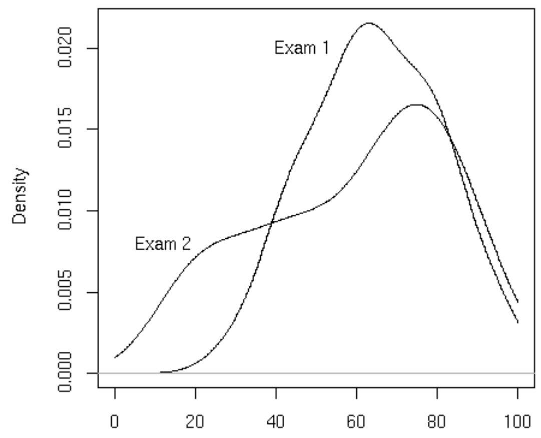


Figure 12-6: Placing text

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.)

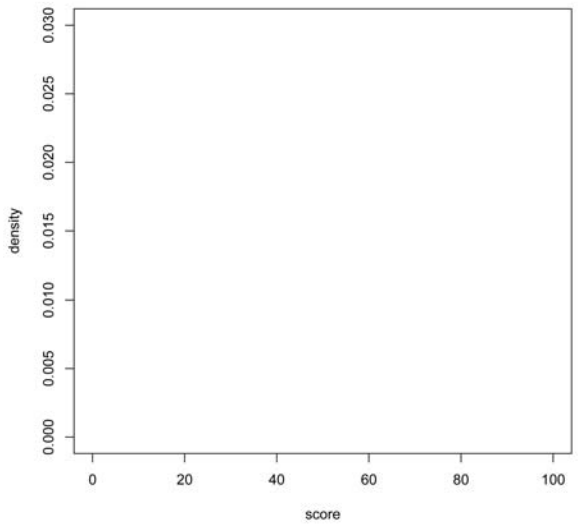


Figure 12-7: Axes only

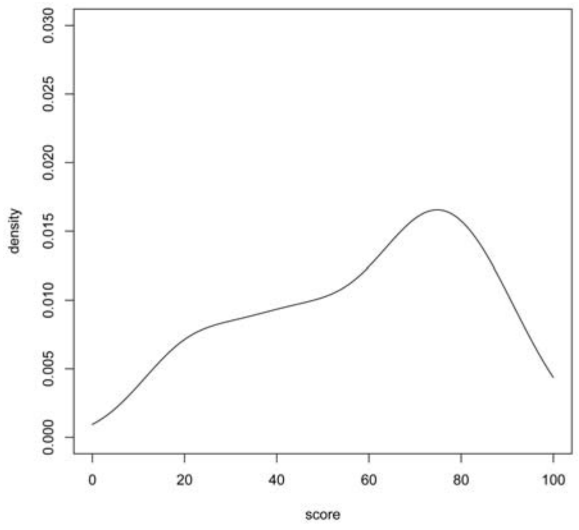


Figure 12-8: Addition of `d2`

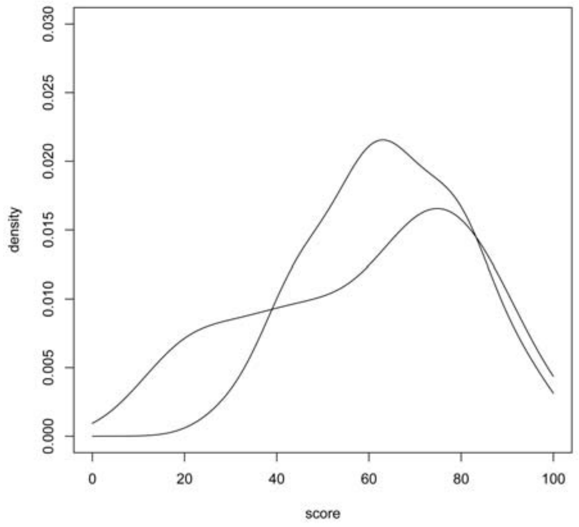


Figure 12-9: Addition of `d1`

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)
```

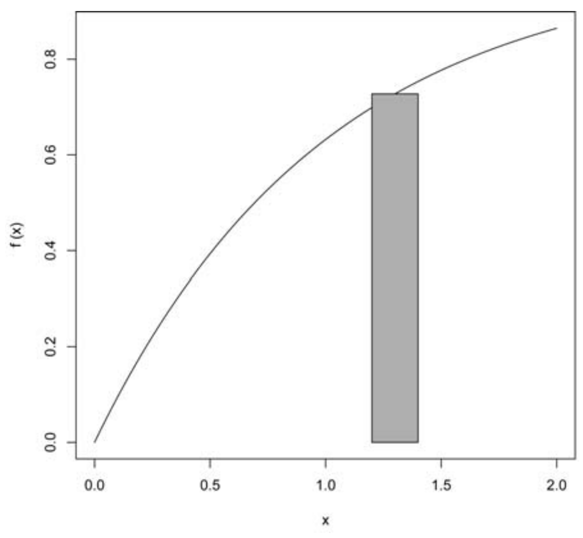


Figure 12-10: Rectangular area strip

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")
```

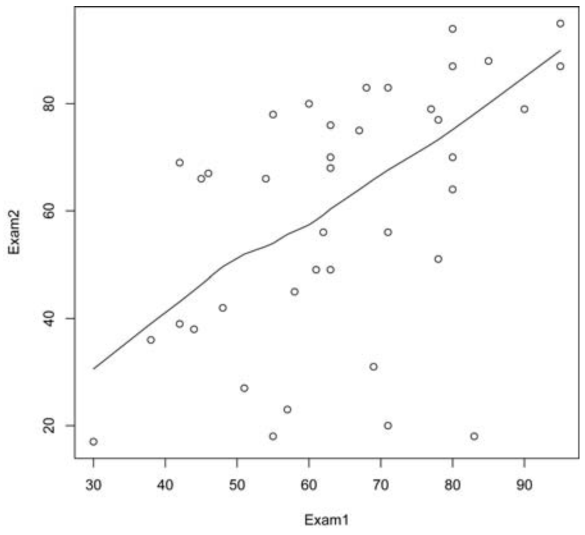


Figure 12-11: Smoothing the exam score relation

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.

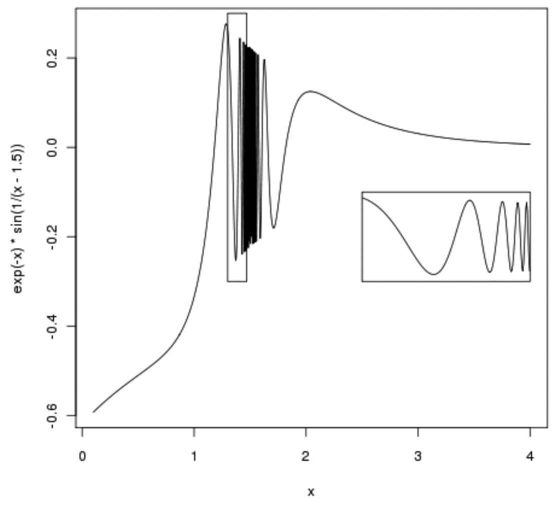


Figure 12-12: Adding an inset graph

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.

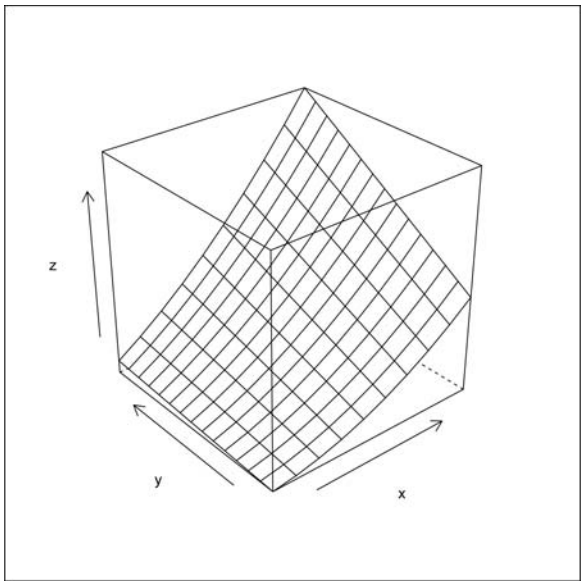


Figure 12-13: Example of using `wireframe()`

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.