CHAPTER 5

Programming with functions

In this chapter we cover the creation of functions, the rules that they must follow, and how they relate to and communicate with the environments from which they are called. We also present some tips on the construction of efficient functions, with especial reference to how functions are treated in R.

Functions are one of the main building blocks for large programs: they are an essential tool for structuring complex algorithms. In some other programming languages *procedures* and *subroutines* play the same role as functions in R.

5.1 Functions

A function has the form

```
name <- function(argument_1, argument_2, ...) {
    expression_1
    expression_2
    ...
    return(output)
}
```

Here `argument_1`, `argument_2`, etc., are the names of variables and `expression_1`, `expression_2`, and `output` are all regular R expressions. `name` is the name of the function. Note that some functions have no arguments, and that the braces are only necessary if the function comprises more than one expression.

To call or run the function we type

```
name(x1, x2, ...)
```

The value of this expression is the value of the expression `output`. To calculate the value of `output` the function first copies the value of `x1` to `argument_1`, `x2` to `argument_2`, and so on.[1] The arguments then act as variables within the function. We say that the arguments have been *passed* to the function. Next

[1] If fact, to save time R only makes a new copy of an argument if its value is changed within the function. However, to understand how a function works it suffices to think that all the arguments are copied when the function is called.

the function evaluates the grouped expressions contained in the braces `{ }`; the value of the expression `output` is returned as the value of the function.

A function may have more than one `return` statement, in which case it stops after executing the first one it reaches. If there is no statement `return(output)` then the value returned by the function is the value of the last expression in the braces (as long as it is not assigned to a variable).

A function *always* returns a value. For some functions the value returned is unimportant, for example if the function has written its output to a file then there may be no need to return a value as well. In such cases one usually omits the `return` statement, or returns `NULL`.

If, when called, the value returned by a function (or any expression) is not assigned to a variable, then it is printed. The expression `invisible(x)` has the same value as `x`, but its value is not printed.

5.1.1 Example: roots of a quadratic 3 `quad3.r`

As an example we write our program for finding the roots of a quadratic as a function. The command `rm(list=ls())` has no effect on the main workspace if executed inside a function, so we have moved it outside. (The reason behind this should become clear in Section 5.2.)

Note that the name of the function does not have to match the name of the program file, but when a program consists of a single function this is conventional.

```
# program spuRs/resources/scripts/quad3.r

quad3 <- function(a0, a1, a2) {
  # find the zeros of a2*x^2 + a1*x + a0 = 0
  if (a2 == 0 && a1 == 0 && a0 == 0) {
    roots <- NA
  } else if (a2 == 0 && a1 == 0) {
    roots <- NULL
  } else if (a2 == 0) {
    roots <- -a0/a1
  } else {
    # calculate the discriminant
    discrim <- a1^2 - 4*a2*a0
    # calculate the roots depending on the value of the discriminant
    if (discrim > 0) {
        roots <- (-a1 + c(1,-1) * sqrt(a1^2 - 4*a2*a0))/(2*a2)
    } else if (discrim == 0) {
        roots <- -a1/(2*a2)
    } else {
        roots <- NULL
    }
  }
  return(roots)
}
```

To use the function we first load it (using `source` or by copying and pasting into R), then call it, supplying suitable arguments.

```
> rm(list = ls())
> source("../scripts/quad3.r")
> quad3(1, 0, -1)

[1] -1  1

> quad3(1, -2, 1)

[1] 1

> quad3(1, 1, 1)

NULL
```

The most important advantage of using a function is that once it is loaded, it can be used again and again without having to reload it. User-defined functions can be used in the same way as predefined functions are used in R. In particular they can be used within other functions.

The second most important use of functions is to break down a programming task into smaller logical units. Large programs are typically made up of a number of smaller functions, each of which does a simple well-defined task.

5.1.2 Example: n choose r `n_choose_r.r`

The number of ways that you can choose r things from a set of n, ignoring the order in which you choose them, is n choose r, which we write as $\binom{n}{r}$.

As is well known, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. One way to write a function for calculating $\binom{n}{r}$, is to first write a function to calculate $n!$, and then use it within our function for $\binom{n}{r}$.

```
# program spuRs/resources/scripts/n_choose_r.r

n_factorial <- function(n) {
    # Calculate n factorial
    n_fact <- prod(1:n)
    return(n_fact)
}

n_choose_r <- function(n, r) {
    # Calculate n choose r
    n_ch_r <- n_factorial(n)/n_factorial(r)/n_factorial(n-r)
    return(n_ch_r)
}
```

Here it is in action.

```
> rm(list = ls())
> source("../scripts/n_choose_r.r")
> n_choose_r(4, 2)

[1] 6

> n_choose_r(6, 4)

[1] 15
```

As an aside we note that $\binom{n}{r}$ can be defined for any real value of n and non-negative integer value of r, using the definition $\binom{n}{r} = n(n-1)\cdots(n-r+1)/r!$. This generalisation is useful for defining certain probability distributions (amongst other things).

Finally, note that more efficient functions that achieve the same goal are available in R; specifically, `choose` and `factorial`.

5.1.3 Example: Winsorised mean `wmean.r`

Let $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ be a sample of real numbers and let $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$ be the ordered sample. The k-th trimmed mean of $\mathbf{x}$ is defined as

$$\bar{x}_k = \frac{x_{(k+1)} + \cdots + x_{(n-k)}}{n-2k}.$$

That is, we discard the k smallest and k largest values then take the average. The trimmed mean is less susceptible to outliers than the untrimmed mean.

The k-th Winsorised mean is defined as

$$w_k = \frac{(k+1)x_{(k+1)} + x_{(k+2)} + \cdots + x_{(n-k-1)} + (k+1)x_{(n-k)}}{n}.$$

That is, instead of discarding the k-th largest and k-th smallest values, we replace them by $x_{(n-k)}$ and $x_{(k+1)}$, respectively. The Winsorised mean can be used when you think that your sample may contain occasional extraordinary values, either because of errors or because you are not measuring what you think you are measuring (this would be a conceptual rather than a measurement error).

Here is a function for calculating the k-th Winsorised mean.

```
# program spuRs/resources/scripts/wmean.r

wmean <- function(x, k) {
    # calculate the k-th Windsorised mean of the vector x
    x <- sort(x)
    n <- length(x)
    x[1:k] <- x[k+1]
    x[(n-k+1):n] <- x[n-k]
    return(mean(x))
}
```

Here it is in practice.

```
> source("../scripts/wmean.r")
> x <- c( 8.244, 51.421, 39.020, 90.574, 44.697,
+         83.600, 73.760, 81.106, 38.811, 68.517)
> mean(x)

[1] 57.975

> wmean(x, 2)

[1] 59.8773

> x.err <- x
> x.err[1] <- 1000
> mean(x.err)

[1] 157.1506

> wmean(x.err, 2)

[1] 65.9695
```

5.1.4 Program flow using functions

When a function is executed the computer sets aside space for the function variables, makes a copy of the function code, then transfers control to the function. When the function is finished the output of the function is passed back to the main program, then the copy of the function and all its variables are deleted. We illustrate this using the program below. Note that we have numbered the lines of the function `swap` separately from the main program. The flow of the program is charted in Table 5.1.

```
    # swap.r

f1  swap <- function(x) {
      # swap values of x[1] and x[2]
f2    y <- x[2]
f3    x[2] <- x[1]
```

Table 5.1 *Control flow for the program* `swap.r`

	main program	function swap (1st)		function swap (2nd)		
line	x	x	y	x	y	comments
p1	(7, 8, 9)					
f1	(7, 8, 9)	(7, 8)				control transferred to swap from p2
f2	(7, 8, 9)	(7, 8)	8			
f3	(7, 8, 9)	(7, 7)	8			
f4	(7, 8, 9)	(8, 7)	8			
f5	(7, 8, 9)	(8, 7)	8			swap returns (8, 7); control returned to line p2, function variables deleted
p2	(8, 7, 9)					
f1	(8, 7, 9)			(7, 9)		control transferred to swap from p3
f2	(8, 7, 9)			(7, 9)	9	
f3	(8, 7, 9)			(7, 7)	9	
f4	(8, 7, 9)			(9, 7)	9	
f5	(8, 7, 9)			(9, 7)	8	swap returns (9, 7); control returned to line p3, function variables deleted
p3	(8, 9, 7)					

```
f4    x[1] <- y
f5    return(x)
f6  }

p1  x <- c(7, 8, 9)
p2  x[1:2] <- swap(x[1:2])
p3  x[2:3] <- swap(x[2:3])
```

5.2 Scope and its consequences

Arguments and variables defined within a function exist *only* within that function. That is, if you define and use a variable `x` inside a function, it does not exist outside the function. If variables with the same name exist inside and outside a function, then they are separate and do not interact at all. You can think of a function as a separate environment that communicates with the outside world only through the values of its arguments and its output expression.[2] For example if you execute the command `rm(list=ls())` inside a function (which is only rarely a good idea), you only delete those objects that are defined inside the function.

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

[1] 2

> x

Error: Object "x" not found

> y

Error: Object "y" not found

> y <- 10
> test(1)

[1] 2

> y

[1] 10
```

That part of a program in which a variable is defined is called its *scope*. Restricting the scope of variables within a function provides an assurance that calling the function will not modify variables outside the function, except by assigning the returned value.

Beware however, the scope of a variable is not symmetric. That is, variables defined inside a function cannot be seen outside, but variables defined outside the function *can* be seen inside the function (provided there is not a variable with the same name defined inside). This arrangement allows for elegant programming in certain situations (in particular when programming recursively, see Section 5.5), but it also makes it possible to write a function whose behaviour depends on the context within which it is run. Consider the following example:

```
> test2 <- function(x) {
+     y <- x + z
+     return(y)
+ }
> z <- 1
> test2(1)
```

[2] This statement is not entirely accurate, but provides a useful model.

```
[1] 2

> z <- 2
> test2(1)

[1] 3
```

The moral of this example is that it is advisable to ensure that the variables you use in a function either are arguments, or have been defined in the function. Exercise 4 gives a subtle example of what can go wrong. Conversely, an example where we deliberately make use of this aspect of scoping, to simplify our coding, is given in Section 12.4.1.

5.3 Optional arguments and default values

To give the argument `argument_1` the default value `x1` we use `argument_1 = x1` within the function definition. If an argument has a default value then it may be omitted when calling the function, in which case the default is used.

If you omit an argument then there is possible ambiguity regarding which arguments are assigned to which variables. To avoid this R assigns arguments to variables from the left, unless an argument is named.

```
> test3 <- function(x = 1, y = 1, z = 1) {
+     return(x * 100 + y * 10 + z)
+ }
> test3(2, 2)

[1] 221

> test3(y = 2, z = 2)

[1] 122
```

5.4 Vector-based programming using functions

We have mentioned that many R functions are vectorised, meaning that given vector input the function acts on each element separately, and a vector output is returned. This is a very powerful aspect of R that allows for compact, efficient, and readable code.

To further facilitate vector-based programming, R provides a family of powerful and flexible functions that enable the vectorisation of user-defined functions: `apply`, `sapply`, `lapply`, `tapply`, and `mapply`.

The effect of `sapply(X, FUN)` is to apply function `FUN` to every element of vector X. That is, sapply(X, FUN) returns a vector whose i-th element is the value of the expression FUN(X[i]). If FUN has arguments other than X[i], then they can be included using sapply(X, FUN, ...), which returns FUN(X[i], ...) as the i-th element. That is, the arguments ... are passed directly from sapply to FUN, thus allowing you to use a function with more than one argument, though note that the values of the arguments ... are the same each time. To vectorise over more than one argument, use mapply.

If you wish to apply a function that takes a vector argument to each of the rows (or columns) of a matrix, then use the function apply, which is a more flexible but more complex version of sapply.

We cover tapply in Section 6.4.1, and provide more detail about sapply and lapply in Section 6.4.2. See also help(apply).

5.4.1 Example: density of primes primedensity.r

Here we give an example of sapply in action. The idea is to write a function prime that tests if a given integer is prime. We then use sapply to apply prime to the vector 2:n, so that we know all the primes less than or equal to n.

Let $\rho(n)$ be the number of primes less than or equal to n. Both Legendre and Gauss famously asserted that

$$\lim_{n\to\infty} \frac{\rho(n)\log(n)}{n} \to 1.$$

The result was eventually proved some time later by Hadamard and de la Vallée Poussin in 1896. The proof is hard, but we can easily check the result numerically. Our program uses the function cumsum(x), which returns the cumulative sums of x as a vector. We apply it to a logical vector of TRUE/FALSE values, which R coerces into a 1/0 vector before computing the cumulative sum.

```
# spuRs/resources/scripts/primedensity.r
# estimate the density of primes (using a very inefficient algorithm)

# clear the workspace
rm(list=ls())

prime <- function(n) {
    # returns TRUE if n is prime
    # assumes n is a positive integer
    if (n == 1) {
        is.prime <- FALSE
    } else if (n == 2) {
        is.prime <- TRUE
    } else {
        is.prime <- TRUE
        for (m in 2:(n/2)) {
            if (n %% m == 0) is.prime <- FALSE
        }
    }
    return(is.prime)
}

# input
# we consider primes <= n
n <- 1000

# calculate the number of primes <= m for m in 2:n
# num.primes[i] == number of primes <= i+1
m.vec <- 2:n
primes <- sapply(m.vec, prime)
num.primes <- cumsum(primes)

# output
# plot the actual prime density against the theoretical limit
par(mfrow = c(1, 2))
plot(m.vec, num.primes/m.vec, type = "l",
    main = "prime density", xlab = "n", ylab = "")
lines(m.vec, 1/log(m.vec), col = "red")

plot(m.vec, num.primes/m.vec*log(m.vec), type = "l",
    main = "prime density * log(n)", xlab = "n", ylab = "")
par(mfrow = c(1, 1))
```

Executing the command `source("primedensity.r")` gives the output of Figure 5.1.

We see that at the point $n = 1000$ the prime density $\rho(n)/n$ is not particularly close to $1/\log(n)$, though the rate of decay looks correct. To see better convergence you will need to take much larger n, however this will take a long time as it takes longer and longer to check each number to see if it is prime.

It does not help that the algorithm we used is inefficient. The function `prime` can be made more efficient in two ways. First, we need only check for factors up to $\sqrt{n}$, since if $n = ab$ then at least one of a and b is less than or equal to $\sqrt{n}$. Second, once we find one factor we don't need to keep checking. Incorporating these two refinements we get the following:

```
# program spuRs/resources/scripts/prime.r

prime <- function(n) {
    # returns TRUE if n is prime
    # assumes n is a positive integer
    if (n == 1) {
        is.prime <- FALSE
```

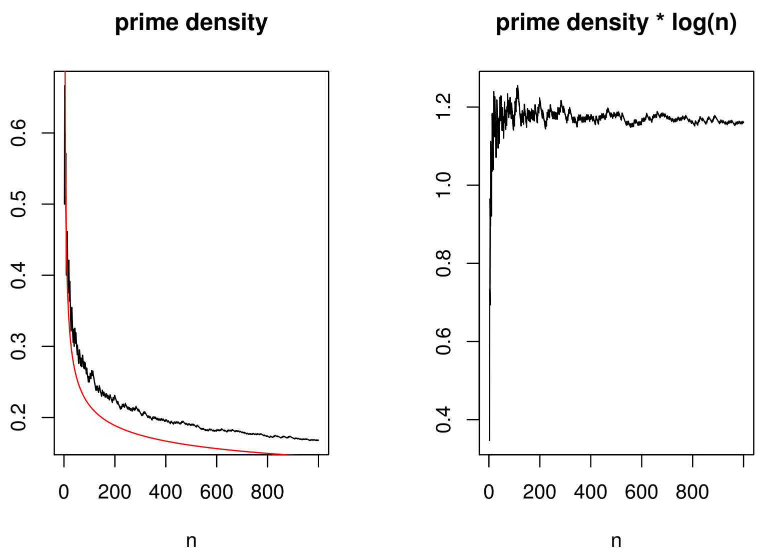

Figure 5.1 *The density of primes. Output from Example 5.4.1.*

```
    } else if (n == 2) {
        is.prime <- TRUE
    } else {
        is.prime <- TRUE
        m <- 2
        m.max <- sqrt(n)  # only want to calculate this once
        while (is.prime && m <= m.max) {
            if (n %% m == 0) is.prime <- FALSE
            m <- m + 1
        }
    }
    return(is.prime)
}
```

However, if what you really want to do is not just check that n is prime, but rather find all the primes less than or equal to n, then a much more efficient algorithm is the 'Sieve of Eratosthenes' (ca. 240 BC). An implementation of Eratosthenes' algorithm is given in Section 5.5.

5.5 Recursive programming

Recursive programming is a powerful programming technique, made possible by functions. A recursive program is simply one that calls itself. This is useful because many algorithms are recursive in nature.

5.5.1 Example: n factorial 2 `nfact2.r`

We can write $n!$ as $n*((n-1)!)$. We implement this recursive definition below. Note that the program uses `cat` statements to provide some feedback, and we have numbered the lines for the purpose of charting the program flow.

```
    # function nfact2.r

 1  nfact2 <- function(n) {
        # calculate n factorial
 2      if (n == 1) {
 3          cat("called nfact2(1)\n")
 4          return(1)
 5      } else {
 6          cat("called nfact2(", n, ")\n", sep = "")
 7          return(n*nfact2(n-1))
 8      }
 9  }
```

```
> source("../scripts/nfact2.r")
> nfact2(6)

called nfact2(6)
called nfact2(5)
called nfact2(4)
called nfact2(3)
called nfact2(2)
called nfact2(1)
[1] 720
```

When you chart the flow through a recursive function, it is important to remember that when a function is called, a *new* copy of the function is created with a *new* set of function variables. For example, calling `nfact2(3)` gives the program flow shown in Table 5.2. We write $i.j$ to indicate line j within the i-th nested function call.

5.5.2 Example: Sieve of Eratosthenes `primesieve.r`

The Sieve of Eratosthenes is an algorithm for finding all of the primes less than or equal to a given number n. It works as follows:

Table 5.2 *Control flow through the function* `nfact2`

	nfactorial (1st call)	nfactorial (2nd call)	nfactorial (3rd call)	
line	n	n	n	comments
1.1	3			
1.2	3			$n \neq 1$ so go to line 6
1.6	3			print 'called nfactorial(3)'
2.1	3	2		nfactorial(2) called on line 1.7
2.2	3	2		$n \neq 1$ so go to line 6
2.6	3	2		print 'called nfactorial(2)'
3.1	3	2	1	nfactorial(1) called on line 2.7
3.2	3	2	1	$n = 1$ so go to line 3
3.3	3	2	1	print 'called nfactorial(1)'
3.4	3	2	1	return 1, delete variables, return control to line 2.7
2.7	3	2		return 2, delete variables, return control to line 1.7
1.7	3			return 6, delete variables, return control to calling line

1. Start with the list $2, 3, \ldots, n$ and largest known prime $p = 2$.
2. Remove from the list all elements that are multiples of p (but keep p itself).
3. Increase p to the smallest element of the remaining list that is larger than the current p.
4. If p is larger than $\sqrt{n}$ then stop, otherwise go back to step 2.

Here is a recursive implementation of the algorithm. You may find that it takes you some time to understand how it works.

```
# program spuRs/resources/scripts/primesieve.r
# loadable spuRs function

primesieve <- function(sieved, unsieved) {
  # finds primes using the Sieve of Eratosthenes
  # sieved: sorted vector of sieved numbers
  # unsieved: sorted vector of unsieved numbers

  # cat("sieved", sieved, "\n")
  # cat("unsieved", unsieved, "\n")
  p <- unsieved[1]
  n <- unsieved[length(unsieved)]
  if (p^2 > n) {
      return(c(sieved, unsieved))
  } else {
      unsieved <- unsieved[unsieved %% p != 0]
      sieved <- c(sieved, p)
      return(primesieve(sieved, unsieved))
  }
}
```

Here it is in action:

```
> rm(list = ls())
> source("../scripts/primesieve.r")
> primesieve(c(), 2:200)

 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53
[17]  59  61  67  71  73  79  83  89  97 101 103 107 109 113 127 131
[33] 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

It can be shown that the Sieve of Eratosthenes uses $O(n(\log n)(\log\log n))$ operations to find all the primes less than or equal to n. (The notation $g(x) = O(f(x))$ means there exists a constant c such that $\lim_{x\to\infty} g(x)/f(x) \leq c$. In other words $g(x)$ grows no faster than a constant times $f(x)$.) You should try to calculate how many operations are used by the algorithm given in Example 5.4.1. You will see that it is much less efficient.

5.6 Debugging functions

Often code will be used in circumstances under which you cannot control the type of input (numeric, character, logical, etc.). Unexpected input can lead to undesirable consequences, for example, the function could fail to work and the user may not know why. Worse still, the function could seem to work but return plausible nonsense, and the user may be none the wiser. It can be worth performing simple checks on the input to be sure that it conforms to your expectations. (Useful considerations here are: what will your function do if the input is the wrong type, or the right type but incomplete?) The `stop` function is useful in these circumstances: `stop("Your message here.")` will cease processing and print the message to the user.

The `browser` function is very useful to invoke inside your own functions. The command `browser()` will temporarily stop the program, and allow you to inspect its objects. You can also step through the code, executing one expression at a time.

When in the browser environment, R commands can be entered and evaluated as normally, but some commands have specific new interpretations. The important ones are:

n enters the step-through debugger. In step-through mode,

- n evaluates the current step and prints the next step to be evaluated. The return key has the same effect.
- c continues evaluation from the next expression to the end of the current set of expressions, whether that be the end of the current loop or the end of the function (cont has the same effect).
- Q stops evaluation and exits the browser, returning the user to the top-level prompt.

c stops the browser and continues evaluation, starting at the next statement (the return key and cont both have the same effect).

A commented example of its application follows. my_fun attempts to multiply its input by the (undefined) variable z

```
> my_fun <- function(x) {
+     browser()
+     y <- x * z
+     return(y)
+ }

> my_fun(c(1,2,3))

Called from: my_fun(c(1,2,3))
Browse[1]>
```

browser catches the execution and presents us with a prompt. Using n, we will step through the function one line at a time. At each point, R shows us the next line to be evaluated. We signify our input using curly braces; thus: { Enter }.

```
Browse[1]> n

debug: y <- x * z
Browse[1]>

Browse[1]> { Enter }

Error in my_fun(c(1, 2, 3)) : object "z" not found
```

The result makes it clear to us that the problem in our function is in the line y <- x * z. Here, the problem is obvious: the code calls for an object z, which does not exist. In any case, we can run the function again, return to that point in the proceedings, and take a look around.

```
> my_fun(c(1,2,3))

Called from: my_fun(c(1,2,3))
Browse[1]>
```

```
Browse[1]> n

debug: y <- x * z
Browse[1]>
```

We know that there is a problem here. We identify and examine the objects to locate the problem.

```
Browse[1]> ls()

[1] "x"
Browse[1]>

Browse[1]> Q

>
```

It is clear that something is missing in the environment.

See `?browser` and `?debug` for more information, and note that we provide more advice on debugging in Section 8.3.

5.7 Exercises

1. The (Euclidean) length of a vector $v = (a_0, \ldots, a_k)$ is the square root of the sum of squares of its coordinates, that is $\sqrt{a_0^2 + \cdots + a_k^2}$. Write a function that returns the length of a vector.
2. In Exercise 3.9.2 you wrote a program to calculate $h(x, n)$, the sum of a finite geometric series. Turn this program into a *function* that takes two arguments, x and n, and returns $h(x, n)$.

   Make sure you deal with the case $x = 1$.
3. In this question we simulate the rolling of a die. To do this we use the function `runif(1)`, which returns a 'random' number in the range (0,1). To get a random integer in the range $\{1, 2, 3, 4, 5, 6\}$, we use `ceiling(6*runif(1))`, or if you prefer, `sample(1:6,size=1)` will do the same job.

   (a). Suppose that you are playing the gambling game of the Chevalier de Méré. That is, you are betting that you get at least one six in 4 throws of a die. Write a program that simulates one round of this game and prints out whether you win or lose.

   Check that your program can produce a different result each time you run it.

   (b). Turn the program that you wrote in part (a) into a function `sixes`, which returns `TRUE` if you obtain at least one six in n rolls of a fair die, and returns `FALSE` otherwise. That is, the argument is the number of rolls n, and the value returned is `TRUE` if you get at least one six and `FALSE` otherwise.

   How would you give n the default value of 4?

(c). Now write a program that uses your function `sixes` from part (b), to simulate N plays of the game (each time you bet that you get at least 1 six in n rolls of a fair die). Your program should then determine the proportion of times you win the bet. This proportion is an estimate of the *probability* of getting at least one 6 in n rolls of a fair die.

Run the program for $n = 4$ and $N = 100$, 1000, and 10000, conducting several runs for each N value. How does the *variability* of your results depend on N?

The probability of getting no 6's in n rolls of a fair die is $(5/6)^n$, so the probability of getting at least one is $1 - (5/6)^n$. Modify your program so that it calculates the theoretical probability as well as the simulation estimate and prints the difference between them. How does the *accuracy* of your results depend on N?

You may find the `replicate` function useful here.

(d). In part (c), instead of processing the simulated runs as we go, suppose we first store the results of every game in a file, then later postprocess the results.

Write a program to write the result of all N runs to a textfile `sixes_sim.txt`, with the result of each run on a separate line. For example, the first few lines of the textfile could look like

```
TRUE
FALSE
FALSE
TRUE
FALSE
.
.
```

Now write another program to read the textfile `sixes_sim.txt` and again determine the proportion of bets won.

This method of saving simulation results to a file is particularly important when each simulation takes a very long time (hours or days), in which case it is good to have a record of your results in case of a system crash.

4. Consider the following program and its output

```
# Program spuRs/resources/scripts/err.r

# clear the workspace
rm(list=ls())

random.sum <- function(n) {
    # sum of n random numbers
    x[1:n] <- ceiling(10*runif(n))
    cat("x:", x[1:n], "\n")
    return(sum(x))
}

x <- rep(100, 10)
show(random.sum(10))
show(random.sum(5))

> source("../scripts/err.r")
x: 8 5 4 2 10 6 8 9 3 2
[1] 57
x: 2 2 3 5 9
[1] 521
```

Explain what is going wrong and how you would fix it.

5. For $r \in [0,4]$, the *logistic map* of $[0,1]$ into $[0,1]$ is defined as $f(x) = rx(1-x)$.

   Given a point $x_1 \in [0,1]$ the sequence $\{x_n\}_{n=1}^{\infty}$ given by $x_{n+1} = f(x_n)$ is called the *discrete dynamical system* defined by f.

   Write a function that takes as parameters x_1, r, and n, generates the first n terms of the discrete dynamical system above, and then plots them.

   The logistic map is a simple model for population growth subject to resource constraints: if x_n is the population size at year n, then x_{n+1} is the size at year $n+1$. Type up your code, then see how the system evolves for different starting values x_1 and different values of r.

   Figure 5.2 gives some typical output.

6. The Game of Life is a cellular automaton and was devised by the mathematician J.H. Conway in 1970. It is played on a grid of cells, each of which is either alive or dead. The grid of cells evolves in time and each cell interacts with its eight neighbours, which are the cells directly adjacent horizontally, vertically, and diagonally.

   At each time step cells change as follows:

   - A live cell with fewer than two neighbours dies of loneliness.
   - A live cell with more than three neighbours dies of overcrowding.
   - A live cell with two or three neighbours lives on to the next generation.
   - A dead cell with exactly three neighbours comes to life.

   The initial pattern constitutes the first generation of the system. The second generation is created by applying the above rules simultaneously to every cell in the first generation: births and deaths all happen simultaneously. The rules continue to be applied repeatedly to create further generations.

   Theoretically the Game of Life is played on an infinite grid, but in practice we use a finite grid arranged as a torus. That is, if you are in the left-most column of the grid then your left-hand neighbours are in the right-most column, and if you are in the top row then your neighbours above are in the bottom row.

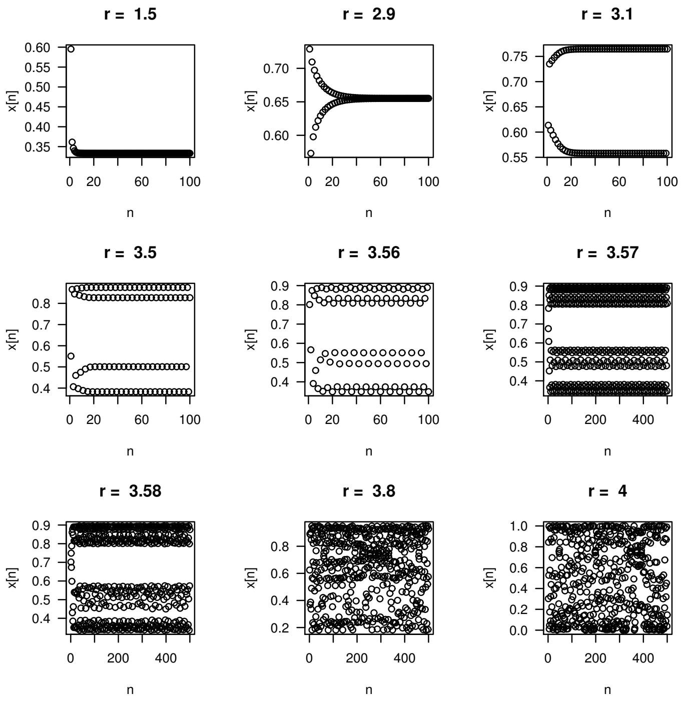

Figure 5.2 *The logistic map described in Exercise 5.*

Here is an implementation of the Game of Life in R. The grid of cells is stored in a matrix A, where A[i,j] is 1 if cell (i, j) is alive and 0 otherwise.

```
# program spuRs/resources/scripts/life.r

neighbours <- function(A, i, j, n) {
    # A is an n*n 0-1 matrix
    # calculate number of neighbours of A[i,j]
    .
    .
    .
}

# grid size
n <- 50

# initialise lattice
A <- matrix(round(runif(n^2)), n, n)

finished <- FALSE
while (!finished) {
    # plot
    plot(c(1,n), c(1,n), type = "n", xlab = "", ylab = "")
    for (i in 1:n) {
        for (j in 1:n) {
            if (A[i,j] == 1) {
                points(i, j)
            }
        }
    }

    # update
    B <- A
    for (i in 1:n) {
        for (j in 1:n) {
            nbrs <- neighbours(A, i, j, n)
            if (A[i,j] == 1) {
                if ((nbrs == 2) | (nbrs == 3)) {
                    B[i,j] <- 1
                } else {
                    B[i,j] <- 0
                }
            } else {
                if (nbrs == 3) {
                    B[i,j] <- 1
                } else {
                    B[i,j] <- 0
                }
            }
        }
    }
    A <- B

    ## continue?
    #input <- readline("stop? ")
    #if (input == "y") finished <- TRUE
}
```

Note that this program contains an infinite loop! To stop it you will need to use the escape or stop button (Windows or Mac) or control-C (Unix). Alternatively, uncomment the last two lines. To get the program to run you will need to complete the function `neighbours(A, i, j, n)`, which calculates the number of neighbours of cell (i, j). (The program `forest_fire.r` in Section 21.2.3 uses a similar function of the same name, which you may find helpful.)

Once you get the program running, you might like to initialise it using the glider gun, shown in Figure 5.3 (see `glidergun.r` in the `spuRs` package). Many other interesting patterns have been discovered in the Game of Life.[3]

7. The number of ways you can choose r things from a set of n, ignoring the

[3] M. Gardner, *Wheels, Life, and Other Mathematical Amusements*. Freeman, 1985.

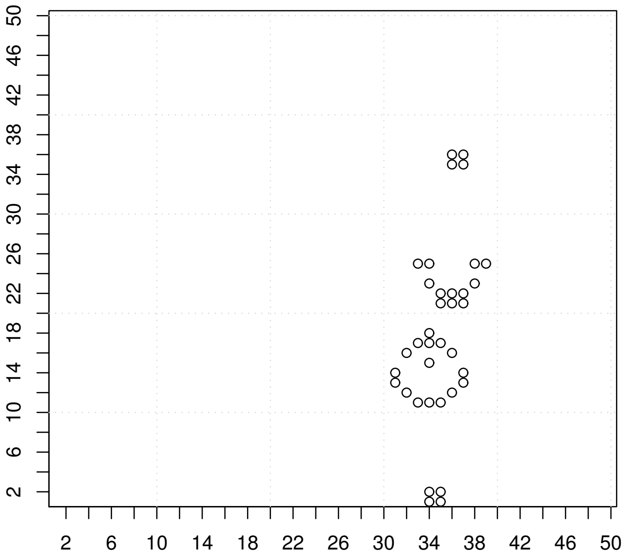

Figure 5.3 *The glider gun, from Exercise 6.*

order in which they are chosen, is $\binom{n}{r} = n!/(r!(n-r)!)$. Let x be the first element of the set of n things. We can partition the collection of possible size r subsets into those that contain x and those that don't: there must be $\binom{n-1}{r-1}$ subsets of the first type and $\binom{n-1}{r}$ subsets of the second type. Thus

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}.$$

Using this and the fact that $\binom{n}{n} = \binom{n}{0} = 1$, write a recursive function to calculate $\binom{n}{r}$.

8. A classic puzzle called the *Towers of Hanoi* uses a stack of rings of different sizes, stacked on one of 3 poles, from the largest on the bottom to the smallest on top (so that no larger ring is on top of a smaller ring). The object is to move the stack of rings from one pole to another by moving one ring at a time so that larger rings are never on top of smaller rings.

   Here is a recursive algorithm to accomplish this task. If there is only one ring, simply move it. To move n rings from the pole `frompole` to the pole `topole`, first move the top $n-1$ rings from `frompole` to the remaining `sparepole`, then move the last and largest from `frompole` to the empty `topole`, then move the $n-1$ rings on `sparepole` to `topole` (on top of the largest).

The following program implements this algorithm. For example, if there are initially 8 rings, we then move them from pole 1 to pole 3 by calling `moverings(8,1,3)`.

```
# Program spuRs/resources/scripts/moverings.r

# Tower of Hanoi

moverings <- function(numrings, frompole, topole) {
  if (numrings == 1) {
    cat("move ring 1 from pole", frompole,
        "to pole", topole, "\n")
  } else {
    sparepole <- 6 - frompole - topole # clever
    moverings(numrings - 1, frompole, sparepole)
    cat("move ring", numrings, "from pole", frompole,
        "to pole", topole, "\n")
    moverings(numrings - 1, sparepole, topole)
  }
  return(invisible(NULL))
}
```

Check that the algorithm works for the cases `moverings(3, 1, 3)` and `moverings(4, 1, 3)`, then satisfy yourself that you understand why it works.

Use mathematical induction to show that, using this algorithm, moving a stack of n rings will require exactly $2^n - 1$ individual movements.