3

R programming

The R system is not only an interactive tool for exploring data sets and graphic representations, but it also serves as an excellent environment for programming. Comparably speaking, the programming syntax of R is easy to learn. Users even without previous programming experience can get started quickly in a couple of hours of learning some basic R control structures. This is exactly what we will do here. This chapter covers some basic programming skills, focused on the use of flow controls and on how to write functions for simple statistical problems.

3.1 Flow control and vectorization

The use of flow controls, either conditional or repetitive, are essentially programming skills.

3.1.1 Conditional execution

There are three functions that can be used for conditional execution: if, ifelse, and switch.

- The if-statement

  The syntax of the if-statement is
  \[
  \text{if} \ (\text{cond}) \ \text{expr} \\
  \text{if} \ (\text{cond}) \ \text{expr}_1 \ \text{else} \ \text{expr}_2
  \]

  where \text{cond} stands for condition, and \text{expr} stands for expression.

  A \text{cond} is a length-one logical value. The \text{cond} must evaluate to a single logical value and the result of the entire expression is then evident\(^1\). An \text{expr} is any valid R expression, and is often a compound expression, which is a series of expressions contained within curly braces.

  In the following example, we use the if-statement to decide the actual grade given the score that a student has. The initial value for the grade variable is \text{grade} = \text{’’Pass’’}, and an actual score is \text{score} = 50. Assume that the actual grade is based on a cut-off score of 60 (i.e., Fail if score < 60, or Pass otherwise). Then, the actual grade is obtained by:

\[\begin{align*}
\text{if} \ (\text{score} < 60) \ \
\text{grade} = \text{’’Fail’’} \\
\text{else} \ \
\text{grade} = \text{’’Pass’’}
\end{align*}\]

\(^1\)In recent versions of R, length-one numeric values also work, where zero corresponds to False and any non-zero value corresponds to True.
> grade = "PASS"
> score = 50
> if (score<60) grade = "FAIL"
> grade # actual grade
[1] "FAIL"
The above example can also be implemented using the *ifelse*-statement.

> score<-50
> if (score<60) grade = "FAIL" else grade = "PASS"
> grade
[1] "FAIL"

- **The *ifelse*-statement**

The *ifelse*-statement provides a more concise form, which takes three arguments, cond, a, and b.

```
ifelse(cond, a, b)
```

which returns a as the result if the condition *cond* is true, and b otherwise.

Retaking the previous example, we can use the *ifelse*-statement to do the same, yet in a more concise form.

```r
> score<-50
> grade <- ifelse(score>=60,'PASS','FAIL')
> grade
[1] "FAIL"
```

- **The *switch*-function**

The function *switch* is also a commonly used conditional execution form, the syntax of which is:

```
switch(expr, ...)
```

where the first argument *expr* is an expression to evaluate, and "..." stands for a list of alternatives, given explicitly.

Any number of additional arguments can be supplied, and they can be either named or unnamed. If the value of the expression is numeric, then the corresponding additional argument is evaluated and returned. If the expression returns a character value, then the additional argument with the matching name will be evaluated and returned. If no argument has a matching name, then the value of the first unnamed argument is returned.

In the following example, a character vector contains five elements representing either a single letter or a double letters. Then, each component is evaluated and a message is displayed showing a number if a single letter is found, or showing "Double letters" otherwise.

```r
> lett <- c('b','QQ','a','A','bb')
> for(ch in lett)
+ cat(ch,':',switch(EXPR = ch, a=1, A=1, b=2, B=2, 'No match!'),'
' )
b : 2
3.1.2 Repetitive execution

A repetitive execution is implemented by using a for-loop, or repeat-command, or while-command.

- The for-loop

The syntax of the for-loop is:

```
for (loop_variable in seq ) expr
```

where `seq` is actually any vector expression usually taking the form of a regular sequence, such as 1:5, and the statements of `expr` are executed for each value of the loop variable in the sequence.

A simple example using the for-loop follows:

```
> for(i in 1:5) print(1:i)
[1] 1
[1] 1 2
[1] 1 2 3
[1] 1 2 3 4
[1] 1 2 3 4 5
```

- The repeat command

The syntax of the repeat command is:

```
repeat expr
```

which repeatedly execute the expression `expr` until explicitly terminated.

- The while command

The syntax of the while command is:

```
while (cond) expr
```

where the while loop continues the execution of the expression `expr` while the condition `cond` holds true.

3.1.3 Vectorization

Loops are very inefficiently implemented in R. So, the use of loops should be avoided, whenever possible, and the technique of vectorization should be used instead. Consider exam scores of five persons, which are represented by `s <- c(80, 45, 55, 90, 75)`. Let `g` be a vector grade, the element of which is given the value 0 ("Fail") if the score is less than 60 and 1 ("Pass")
otherwise. By using the looping technique (e.g., the ifelse statement), the function is defined as:

```r
grade <- function (s) {
  n <- length(s)
  for (i in 1:n) {
    g <- ifelse(s < 60, 0, 1)
  }
  return(g)
}
```

```r
score <- c(80, 45, 55, 90, 75)
grade(score)
[1] 1 0 0 1 1
```

More efficiently, the above can be done via vectorization:

```r
score <- c(80, 45, 55, 90, 75)
grade <- function(s) {
  return(ifelse(s < 60, 0, 1))
}
grade[!score < 60] <- 0
grade
[1] 1 0 0 1 1
```

Obviously, vectorized expressions are computationally simpler, and more efficient, particularly with a large quantity of data. In R, many functions are vectorized (i.e., they can handle both scalars and vectors), such as `mean`, `sum`, and `apply`, just to list a few.

### 3.2 User-defined functions

Often than not, we need to define our own functions. A R function is defined by using the keyword `function`, followed by an opening parenthesis, a list of formal arguments (separated by commas), and a closing parenthesis, and then by the expression(s) for the body of the functions. The value returned by a R function is either the value that is explicitly returned by a call to `return()` or it is simply the value of the last expression.

In the following, three functions are defined. They do the same thing (i.e., calculate the square of a numeric number) though they look somewhat differently.

```r
sq1 <- function(x) x * x
sq1(5)
[1] 25
```

```r
sq2 <- function(x) return(x * x)
sq2(5)
[1] 25
```

```r
sq3 <- function(x) {
  y <- x * x
  return(y)
}```
3. R programming

Note that a single expression can be entered directly on the same line of the function keyword (as in sq1 and sq2). However, if there are several expressions or statements to execute, they must be entered at different lines, enclosed in braces (as in sq3). Also note that the above three functions are all vectorized. (Test them for yourself: if x<-1:10, what will be the outputs?)

3.2.1 A function for the normal likelihood

In statistics, the likelihood function (often simply the likelihood) is a function of the parameters of a statistical model. Informally, if we say that "probability" allows predicting unknown outcomes based on known parameters, then "likelihood" allows estimating unknown parameters based on known outcomes\(^2\). So, likelihoods play a key role in statistical inference.

Suppose we observe a sample of size \(n\), and the observations \(y = (y_1, \ldots, y_n)\) follow a normal distribution with mean \(\mu\) and variance \(\sigma^2\). The likelihood function is (or proportional to)

\[
L = \left(2\pi\sigma^2\right)^{-\frac{n}{2}} \exp\left(-\frac{\sum_{i=1}^{n} (y_i - \mu)^2}{2\sigma^2}\right) \tag{3.1}
\]

Computationally, it is preferable to compute the logarithmic likelihood (think why?):

\[
\log L = \left(-\frac{n}{2}\right) \log (2\pi\sigma^2) + \left(-\frac{1}{2\sigma^2}\right) \sum_{i=1}^{n} (y_i - \mu)^2 \tag{3.2}
\]

The R code for the logarithmic normal likelihood is:

```r
> loglike <- function(mu, sigma, yobs) {
+   n <- length(yobs)
+   var <- sigma * sigma
+   logL <- 0.5*n*log(2.0*pi*var) + sum((yobs-mu)^2)/(2.0*var)
+   return(-logL)
+ }
```

\(^2\)In a sense, likelihood works backwards from conditional probability. In a forward reasoning, given parameter \(B\), we use the conditional probability \(\Pr(A|B)\) to reason about outcome \(A\). In a backward reasoning, however, outcome \(A\) is given and the likelihood function \(L(B|A)\) is used to reason about parameter \(B\). Formally, a likelihood function is a conditional probability function considered as a function of its second argument, with its first argument held fixed, and also any other function proportional to such a function. Thus, the likelihood function for \(B\) is the equivalence class of functions \(L(b|A) = a \Pr(A|B = b)\) for any constant of proportionality \(a > 0\).
Now, let us randomly generate a sample of size from a normal distribution with mean 1.0 and standard deviation 1.2, and then calculate the logarithmic likelihood for parameters $\mu = 1.0$ and $\sigma^2 = 1.2$.

```r
> mu<-1.0
> sigma<-1.2
> seed<-123456
> y<-rnorm(n=100,mean=mu,sd=sigma)
> logL<-loglike(mu,sigma,y)
> logL # logarithm of likelihood
[1] -158.1327
> exp(logL) # likelihood
[1] 2.107775e-69
```

Here, `rnorm(n=,mean=,sd=)` is a function for generating n random samples from a normal distribution with the mean and standard deviation specified by `mean=` and `sd=`, respectively. If the two parameters `mean` and `sd` are not provided, then the random samples are generate from a standard normal distribution with mean 0 and an unit standard deviation.

### 3.2.2 Functions with default values

Using default values in a R function means that not every argument needs to be given specifically when calling the function. Presumably, some arguments can be given commonly appropriate default values, and these values may be omitted from a call to this function. In practice, the use of functions with default values brings a lot convenience in statistical computation using functions.

In the logarithmic normal likelihood function, for example, the variance may be assumedly known, say $\sigma^2 = 1.0$, and we would like to calculate likelihoods for a grid of x values. Then, the R function for calculating the logarithmic normal likelihood can be modified sightly, as shown below.

```r
> loglike <-function(mu,sigma=1,yobs) {
+ n <- length(yobs)
+ var <- sigma * sigma
+ logL <- 0.5*n*log(2.0*pi*var) + sum((yobs-mu)^2)/(2.0*var)
+ return(logL)
+ }
```

Further, by making use of the function `loglike`, a new function, `likemu`, can be defined, which calculates the likelihoods for a grid of values for mu with the variance fixed as $\sigma^2 = 1.0$.

```r
> likemu<-function(vmu,yobs) {
+ m<-length(vmu)
+ like<-numeric(m)
+ for (i in 1:m) {
+ like[i]<-exp(loglike(mu=vmu[i],yobs=yobs))
+ }
```
+ return(like)
+ }

Now, assume that there are 20 data points from a normal distribution with the variance approximately being 1.0. The likelihood values for varying values of the mean (i.e., from -2 to 2 with an increment of 0.1) is calculated as:

```r
> y=c(1.18,-0.84,-0.07,-2.00,-0.34,-1.84,-0.38,-2.39,-1.18,
+ 0.44,-0.21,0.43,-1.21,0.28,-1.19,0.19,-1.17,0.01)
> mean(y)
[1] -0.5716667
> mmu<-seq(-2,2,0.1)
> likmu<-likemu(vmu=mmu,yobs=y)
```

In Figure 3.1, the maximum likelihood value is observed at a location approximately correspond to the sample mean ($\mu \approx -0.57$). Think what this result implies.

```r
> plot(mmu,likmu,type="h")
```

FIGURE 3.1. Plot of likelihood values for a grid of mean values with the variance fixed at 1.0
3.2.3 Functions as arguments

In R, a function (or functions) can be passed as arguments in another function. In the following example, the general plotting function plots the values of a function \( f \) for a specified set of \( x \) values.

```r
genplot <- function(f, x=seq(-10,10,length=200),
                   ptype='', colour=2) {
  y <- f(x)
  plot(x, y, type=ptype, col=colour)
}

> genplot(sin, ptype='h')
> genplot(sin, ptype='h')
```

FIGURE 3.2. Plots of a generic sin function for a specified set of \( x \) values

In the above, we use a generic \( \sin \) function to generate values for \( y = \sin(x) \), and the values of \( y \) are plotted for a range of \( x \) values between -10 and 10 (Figure 3.2).

The function \( f \), which is passed as an argument, can also be user-defined (Figure 3.3).

```r
cubf <- function(x) x^3-6*x-6
> gen_plot(cubf, x=seq(-3,2,length=500))
```
3. R programming

3.2.4 Functions for Binary operators

As mentioned previously, a binary operations take two values, such as addition (+), subtraction (-), multiplication (*), and division (/). For example, adding 10 and 2 is mathematically denoted by 10 + 2. The R syntax follows this convention. In R, binary operators also include matrix multiplication %*% and outer product %o%.

Essentially, a binary operator is a function. Consider the addition operator (+), the code of which can be displayed by:

```r
> get('+')
function (e1, e2) .Primitive('+')
```

Clearly, the addition operator (+) is a function, which takes two parameters, `e1` and `e2`. When writing addition expression in R, however, we write, for example, `1 + 2` rather than `+(1; 2)`. Using such binary operators with the arguments on either side of the binary operator, instead of following the function convention, is much easier for us to understand.

R also allows us to define our own binary operators, e.g., in the form `%name%`. Suppose we want to define a binary `%m%` such that `a%m%b = ab-b`. The function is defined as:

```r
> '%m%' <- function(a, b) a*b-b
```

Then, we can use it in the same way as we use other binary operators (such as + or -).

```r
> 1 %m% 2
```
Next, a binary operation is defined for plotting $y$ over $x$. Practically, $y$ can be a numeric vector, or any function of $x$. In Figure 3.4, for example, we plot $0.3\cos(x) + 0.7\sin(2x)$ over a grid of $x$ values between 0.1 and 20.

```r
> "%p%" <- function(y,x) plot(x, y, type="l", col=2)
> x <- seq(0.1, 20, length=400)
> (0.3*cos(x) + 0.7*sin(2*x)) %p% x
```

### 3.2.5 Recursive functions

A recursive function is a function that calls itself. Recursive functions are convenience to use, but sometimes they may be inefficient means of solving problems in terms of run times.

Now, consider computing the factorial: $n! = n \ast (n - 1)!$ It is apparent that a recursive function can be used here, because, to compute $n!$, one can compute $(n - 1)!$, and then multiplied by $n$.

Numerically, we can see how this can be done using the recursive algorithm. As a starting point, we have:

$0! = 1$

Then, the factorial can be understood in the following recursive way:
Here, we enter this function in another way. Use a text editor to enter the following code.

```r
factorial <- function (n) {
  if (n==0) return(1)
  else return(n*factorial(n-1))
}
```

Save this function as “factorial.R”, and load this function by `source('factorial.R')`. Now, it is ready for use.

```r
> source('factorial.R')
> factorial(0)
[1] 1
> factorial(1)
[1] 1
> factorial(2)
[1] 2
> factorial(3)
[1] 6
> factorial(4)
[1] 24
> factorial(10)
[1] 3628800
```

### 3.3 Some issues related to R programming

#### 3.3.1 Lexical scope

Variables in the body of a R function can be grouped into three categories: formal parameters, local variables and free variables. The formal parameters of a function are those appearing in the argument list of the function, and their values are determined when call to the function (i.e., by the process of binding the actual function arguments to the formal parameters). Local variables are those whose values are determined by the evaluation of expressions in the body of the functions. Free variables are those belonging to neither of the two groups (i.e., not formal parameters nor local variables).

Consider the following function that calculates the area of a rectangle.

```r
area <- function(h, w) {
  s1 <- h * w
  print(h)
}
3. R programming

```
print(w)
print(s1)
print(s2)
```

In this function, h and w are formal parameters, s1 is a local variable and s2 is a free variable.

In R the value of a free variable is resolved by first looking in the environment in which the function was created. This is called lexical scope, which marks one of the major differences between S-Plus and R. Lexical scope can be confusing to R users, but, when probably use, it can provide a powerful mechanism for controlling evaluation and it ensures that intended sets of bindings between variables and values are used.

Define a function that calculates the volume of a cube.
```
cube <- function(w) {
  area <- function() w * w
  w * area()
}
```

The variable w is a formal parameter in the function cube, but it is a free variable in the function area, so its value is determined by the scoping rules. In S-Plus, the value of w is that associated with a global variable named w (i.e., static scope). In R, however, it is the parameter to the function cube because that is the active binding for the variable w at the time the function area was defined (i.e., lexical scope). So, the difference is that S-Plus looks for a global variable called w while R first looks for a variable called w in the environment created when cube was invoked.

In S, suppose that there is a globe variable w=3. A call to cube(2) will return 18 as the cube volume.
```
S> cube(2)
Error in sq(): Object 'w' not found
Dumped
S> w <- 3
S> cube(2)
[1] 18
```

In R, however, a call to the same function will return 8 as the answer.
```
> w<-3
> cube(2)
[1] 8
```

3.3.2 Exception handling

Exception handling is the process of dealing with the failure of a computation to complete successfully and in some sense to allow the user to interrupt computation. There are a number of tools in R that allow for general exception handling. The two most common sorts of exceptions are
errors (which can be raised by a call to `stop`) and warnings (which can be raised by a call to `warning`).

The typical behavior for an error is to halt the current evaluation and return control to the top-level R prompt\(^3\). The default behavior for warning is to wait until the current evaluation is finished and, then, to print the warning that occurred during the evaluation. Users can control the behavior by making use of various R options, which is not discussed in details here.

Next is a simple example demonstrating the use of `tryCatch` for conditionally evaluating expressions. In this example, two handlers are established, one for errors and the other for warnings.

```r
> foo <- function (x) {
+ if (x<3)
+ list() + x
+ else if (x<10)
+ warning('ouch')
+ else
+ 33
+ }
> foo(2)
Error in list() + x : non-numeric argument to binary operator
> foo(5)
Warning message:
In foo(5) : ouch
> foo(29)
[1] 33
>
> tryCatch(foo(2),error=function(e) ''This is an error'',
+ warning = function(e) ''This is an warning'')
[1] ''This is an error''
> tryCatch(foo(5),error=function(e) ''This is an error'',
+ warning = function(e) ''This is an warning'')
[1] ''This is an warning''
> tryCatch(foo(29),error=function(e) ''This is an error'',
+ warning = function(e) ''This is an warning'')
[1] 33
```

\(^3\)In some situation, however, this may not be desired. For example, a large simulation is being run, and one run may fail, which nevertheless should not halt the entire simulation.
3.3.3 Classes and generic functions

A class is a description of a thing, and an object is an instance of a class. For example,

```r
> y<-1:20
> y
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> class(y)
[1] "integer"
```

Here, we see that `y` is an object of the integer class. In R, the class of an object determines how it will be treated by what are known as generic functions. Put the other way round, a generic function performs a task or action on its arguments specific to the class of the argument itself. If the argument lacks any class attribute, or has a class not catered for specifically by the generic function in question, a default action is always provided.

The class mechanism offers the user the facility of designing and writing generic functions for special purposes. Examples of generic functions are `plot()` for displaying objects graphically, `summary()` for summarizing analyses of various types, and `anova()` for comparing statistical models.

The number of classes a generic function can handle can also be quite large. For example, the `summary()` function has a default method and variants for objects of classes. A complete list can be shown by a call to `methods(summary)`:

```r
> methods(summary)
[[1]] summary.aov summary.aovlist summary.connection
[[2]] summary.data.frame summary.Date summary.default
[[3]] summary.ecdf* summary.factor summary.glm
[[4]] summary.infl summary.lm summary.loess*
[[5]] summary.manova summary.matrix summary.mlm
[[6]] summary.nls* summary.packageStatus* summary.POSIXct
[[7]] summary.POSIXlt summary.ppr* summary.prcomp*
```

In this example there are 26 methods. Most of them can be seen by typing its name, such as `summary.data.frame`. However, five of them are asterisked, indicating that can not be viewed directly by typing their names. We can read these methods by, e.g.,

```r
> getAnywhere(summary.loess)
A single object matching 'summary.loess' was found
It was found in the following places
  registered S3 method for summary from namespace stats
  namespace:stats
with value
function (object, ...) {
  class(object) <- '"summary.loess"
  object
}
<environment: namespace:stats>
```
When called to the `summary` function, it performs a task or action on its arguments specific according to the class of the argument. In the following examples, the `summary` function gives descriptive statistics (e.g., minimum, quantiles, and maximum) for the numeric object `y`, a frequency table for the factor object `x`, and a list of regression results for the “lm” object.

```r
> y <- rnorm(20)
> class(y)
[1] "numeric"
> summary(y)

Min. 1st Qu.  Median   Mean 3rd Qu.   Max.   
-1.9370 -0.5221  0.2525  0.1297  1.1750  1.3420

> x <- sample(letters[1:4],20,replace=T)
> x <- as.factor(x)
> class(x)
[1] "factor"
> summary(x)

a  b  c  d
4  5  7  4

> lm<-lm(x~y)
> class(lm)
[1] "lm"
> summary(lm)

Call:
  lm(formula = x ~ y)

Residuals:
    Min     1Q   Median     3Q    Max
-1.8809 -0.5416  0.1368  0.6934  1.3954

Coefficients:  
      Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.0777    0.5302   0.147 0.8851
   yb   -0.1310    0.7113  -0.184 0.8562
   yc   -0.1334    0.6646  -0.201 0.8431
   yd    0.6571    0.7498   0.876 0.3942

Residual standard error: 1.06 on 16 degrees of freedom
Multiple R-squared: 0.09478, Adjusted R-squared: -0.07495
F-statistic: 0.5584 on 3 and 16 DF, p-value: 0.65

For advanced users of R, please refer to Appendix A for detailed descriptions of classes, generic functions, and object-oriented programming.
3.4 Exercises

1. Consider measurements of heights (in centimeters) of five persons at ages 8 and 15, respectively. Let \( x_1 = c(75.1, 108.9, 105.3, 83.9, 101.2) \) and \( x_2 = c(131.1, 175.8, 179.7, 154.6, 163.9) \). Now, we would like to know the change of height per year for each of them. Mathematically, this is to calculate:
\[
\Delta = \frac{1}{15} (x_2 - x_1)
\]
Then, (a) define a function that returns the change of height per year for each of them, and (b) define a function (i.e., binary operator \( \%\Delta\% \)) that takes \( x_2 \) and \( x_1 \) as the two parameters and returns the changes of height per year (\( \Delta \)) for the five persons.

2. Define a function, namely \texttt{center}, which is expected to return either the mean, or median, or mode, depending on the expression to be evaluated. The mean and median are given by the generic functions \texttt{mean} and \texttt{median}, and the mode is given by a user-defined function \texttt{mode}. (We’ll explain the mode function in Chapter 4).

\[
\text{mode} <- \text{function (x) \{ }
\text{ \ \ \ y <- as.integer(names(sort(-table(x)))[1]) }
\text{ \ \ \ print(y)}
\text{\}}
\]

Next, sample 20 numbers randomly with replacement from numbers 1, 2, 3, 4, and 5. Find the mean, median, and mode using the \texttt{center} function.

3. In mathematics, the Kronecker product, denoted by \( \otimes \), is an operation on two matrices of arbitrary size resulting in a block matrix. If \( A \) is an \( m \times n \) matrix and \( B \) is a \( p \times q \) matrix, then the Kronecker product \( A \otimes B \) is the \( mp \times nq \) block matrix

\[
A \otimes B = \begin{bmatrix}
a_{11}B & \ldots & a_{1n}B \\
\vdots & \ddots & \vdots \\
a_{m1}B & \ldots & a_{mn}B
\end{bmatrix}
\]

where \( A = \begin{bmatrix}
a_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \ldots & a_{mn}
\end{bmatrix} \)

Then, (a) Define binary operator (denoted by \( \%\otimes\% \)) for the Kronecker product; (b) calculate \[
\begin{bmatrix}
7 & 0 \\
0 & 5
\end{bmatrix} \otimes \begin{bmatrix}
1.0 & 0.2 \\
0.2 & 1.0
\end{bmatrix}
\].

4. In mathematics, the Fibonacci numbers\(^5\) are the following sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots. Note that the first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two:
\[
0 + 1 = 1 \\
1 + 1 = 2
\]

\(^5\)The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of filius Bonaccio, “son of Bonaccio”).
3. R programming

1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13

In mathematical terms, the sequence \( F_n \) of Fibonacci numbers is defined by the recurrence relation
\[
F_n = F_{n-1} + F_{n-2}
\]
with seed values
\[
F_0 = 0 \quad \text{and} \quad F_1 = 1
\]
Write a recursive R function that gives the Fibonacci numbers.

5. The Bayesian information criterion (BIC) or Schwarz Criterion is a criterion for model selection among a class of parametric models with different numbers of parameters.

The BIC is an asymptotic result derived under the assumptions that the data distribution is in the exponential family. Let \( x \) = the observed data; \( n \) = the number of data points in \( x \) (i.e., the number of observations); \( k \) = the number of free parameters to be estimated (If the estimated model is a linear regression, \( k \) is the number of regressors, including the constant); \( p(x|k) \) = the likelihood of the observed data given the number of parameters; \( L \) = the maximized value of the likelihood function for the estimated model.

The formula for the BIC is:
\[
BIC = -2 \cdot \ln L + k \ln(n)
\]
Under the assumption that the model errors or disturbances are normally distributed, this becomes (up to an additive constant, which depends only on \( n \) and not on the model):
\[
BIC = n \ln(\frac{RSS}{n}) + k \ln(n)
\]
where RSS is the residual sum of squares from the estimated model.
Write a function that gives BIC values of normal data with overall mean \( \mu \) and variance \( \sigma^2 \).

6*. Lexical scope and exception handling: The following R codes are used to mimic a bank account. A functioning bank account needs to have a balance or total, a function for making withdrawals, a function for making deposits and a function for stating the current balance. This is achieved by creating the three functions within account and then returning a list containing them. When account is invoked it takes a numerical argument total and returns a list containing the three functions. Because these functions are defined in an environment which contains total, they will have access to its value.

The special assignment operator, \(<<<\rangle\rangle\), is used to change the value associated with total. This operator looks back in enclosing environments for an environment that contains the variable total. When such an environment is found, it replaces the value, in that environment, with the value of right
3. R programming

hand side. If the global or top-level environment is reached without finding the variable total then that variable is created and assigned to there.

```r
open.account <- function(total) {
  list(deposit = function(amount) {
    if(amount <= 0)
      stop("Deposits must be positive! \n")
    total <- total + amount
    cat(amount, "deposited. Your balance is", total, "\n")
  },
  withdraw = function(amount) {
    if(amount > total)
      stop("You don't have that much money! \n")
    total <- total - amount
    cat(amount, "withdrawn. Your balance is", total, "\n")
  },
  balance = function() {
    cat("Your balance is", total, "\n")
  }
}
```

1) Predict what will be the outputs of the following. Then, run the codes and see what you actually get as the outputs.

```r
ross <- open.account(100)
robert <- open.account(200)
ross$withdraw(30)
ross$balance()
robert$balance()
```

2) Modify the codes by using the tryCatch function for conditionally evaluating the over-drawing problem.

3.5 Project: Additive genetic relationship matrix

The probability of identical genes by descent occurring in two individuals is termed as the coancestry or the coefficient of kinship (Falconer, 1989) and the additive genetic relationship between two individuals is twice their coancestry. The matrix which indicates the additive genetic relationship among individuals is called the numerator relationship matrix (A). It is a symmetric matrix with its diagonal element for animal \( i \) \( (\alpha_{ii}) \) being equal to \( 1 + F_i \), where \( F_i \) is the inbreeding coefficient of animal \( i \) (Wright, 1922). The diagonal element represents twice the probability that two gametes taken at random from animal \( i \) will carry identical alleles by descent. The off-diagonal element, \( a_{ij} \), equals the numerator of the coefficient of relationship (Wright, 1922) between animal \( i \) and \( j \).
The matrix $A$ can be computed using path coefficient, but a recursive method has described by Henderson (1976), which is computationally more convenient to be taken. The algorithm of the recursive method is as follows. Let there be $n$ animals in the pedigree. First, code the animals from 1 to $n$ and order them such that parents precede their progeny. Then, the $A$ matrix can be computed recursively.

If both parents (say $s$ and $d$) of animal $i$ are known

$a_{ji} = a_{ij} = 0.5(a_{js} + a_{jd}); \text{ for } j = 1 \text{ to } (i - 1)$

$a_{ii} = 1 + 0.5(a_{sd})$

If only one parent ($s$, or $d$) is known and assumed unrelated to the mate

$a_{ji} = a_{ij} = 0.5(a_{js}); \text{ for } j = 1 \text{ to } (i - 1)$

$a_{ii} = 1$

If both parents are unknown and are assumed unrelated

$a_{ji} = a_{ij} = 0; \text{ for } j = 1 \text{ to } (i - 1)$

$a_{ii} = 1$

1. Define a function for the numerator relationship matrix $A$ for an arbitrarily pedigree with $n$ individuals.

2. Calculate the numerator relationship matrix $A$ for the pedigree given below.

<table>
<thead>
<tr>
<th>Calf</th>
<th>Sire</th>
<th>Dam</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>unknown</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

3. Multiplying the matrix $A$ by the additive genetic variance ($\sigma_u^2$) leads to the covariance among breeding values of the individuals (denoted as $A\sigma_u^2$).

Let $u_i$ be the breeding value for animal $i$, then $\text{var}(u_i) = (1 + F_i)\sigma_u^2$. Define a function which takes two animal ids (say $i$ and $j$) as the input parameters and returns the covariance of breeding values between the two individuals. Specifically, that the function is expected to return the variance of the breeding value of individual $i$, if two animal ids are the same (i.e., $i = j$).
3. R programming