Programming in R

STAT2005

Chapter 4

Introduction

- In previous chapters, we have introduced many useful built-in functions in R.
- In addition, we can write our own user defined function. The ability of writing user defined function is very important for researcher.
- Actually, R can be considered as a high level programming language and its syntax is very similar to C.
- The file `ch4.r` contains all the functions mentioned in this chapter and we can load these functions simply by the R command: `source("ch4.r")`.

Writing functions in R

A function takes inputs, do calculations (possibly printing intermediate results, drawing graphs, calling other functions, etc.), and produce outputs.

Syntax

```
myfunction <- function(arg1, arg2, ... ) {
     statements
     ...
     return(object)
}
```

If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically.

Let us write a simple R function with function name `se()` to calculate the standard error of sample mean.

Given a sample of size n, $x_1, \dots, x_n$ and let $\bar{x}$ and s be the sample mean and the sample standard deviation respectively.

The standard error of sample mean is defined as $se(x) = s/\sqrt{n}$.

This function can be implemented in a function `se()` as follow:

```
se<-function(x){
     n<-length(x)  # find the sample size
     return(sd(x)/sqrt(n))
}
```

Functions can be entered or edited using `fix(function_name)`.

First, let us enter the above function by `fix(se)`.

```
> se                  # display the function se
function(x){
    n<-length(x) # find the sample size
    return(sd(x)/sqrt(n))
}
```

Now type the function name `se` to make sure the function is correctly entered and test it using `se(1:20)`.

```
> se(1:20)       # compute se of 1:20
[1] 1.322876
```

Remarks

1. We can re-write `se()` by this one-line function without introducing the variable `n`:

```
se<-function(x){sd(x)/sqrt(length(x))}
```

2. It is important to write function using indentation. This will make the function easier to read and debug.

3. By writing functions, we will be able to minimize code redundancy and generalize codes to be reused later.

Example: Pooled sample standard deviation

Let us write another function `sp(x, y)` to compute the pooled sample standard deviation of two independent sample $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ of size m and n. The pooled sample standard deviation is defined as:

$$s_p = \sqrt{\frac{(m-1)s_x^2 + (n-1)s_y^2}{m+n-2}}$$

where s_x^2 and s_y^2 are the sample variances of x and y.

```r
sp<-function (x,y) {
     m<-length(x)
     n<-length(y)
     s2<-((m-1)*var(x)+(n-1)*var(y))/(m+n-2)
     return(sqrt(s2))
}
```

We can test this function using

```
> sp(1:10,3:7)
[1] 2.667468
```

We can save the output of our function to an object for later use. For example:

```
> result<-sp(1:10,3:7)
> result
[1] 2.667468
```

Scope of variables

- The scope of a variable tells us where the variable would be recognized.
- For example, variables defined within functions have local scope, so they are recognized only within the function.
- A variable with the same name could be created in a different function but there is no risk of a clash.

Example: scope of variables

In this example we create two functions `f` and `g`, both with local variables named `x`. `g` is called by `f` and modifies its instance of `x` without affecting the `x` in `f`:

```
f <- function() {
     x <- 1
     g() # g will have no effect on our local x
     return(x)
}
g <- function() {
     x <- 2
     # this changes g's local x, not the one in f
}
f()
[1] 1
```

Super assignment

For most cases, local scoping is a preferred behaviour because it prevented variable name clash. However, for some specific applications, global variables could be useful.

For example, suppose that we want to count how many times a function is being called.

```
f <- function() {
     if (!exists("f_count"))
          # check existence of f_count
          f_count <<- 1
     else
          f_count <<- f_count + 1
     return(f_count)
}
```

Loops and flow control

- As in many high level programming languages, loop and control is an essential part of the language.
- In R, there are `for` loop, `if` statement, `while` loop and `repeat` loop.
- Before going into details about these control statements, we give a review of how the logical operators work.

Logical expressions

- A logical expression is formed using the logical operators <, >, <=, >=, `==` (equal to), and `!=` (not equal to); and the logical operators `&` (and), `|` (or), and `!` (not).
- The order of operations can be controlled using parentheses `( )`.
- The value of a logical expression is either **`TRUE`** or **`FALSE`**. The integers `1` and `0` can also be used to represent **`TRUE`** and **`FALSE`**, respectively

Logical operators

Logical operators are essential for control flow, especially in the `if` statement and `while` loop. Here are some examples.

```
> a<-1;b<-2
> (a>0)
[1] TRUE
> (a==1)
[1] TRUE
> (a>=1)&(b<5)
[1] TRUE
> (a>=1)&(b!=2)
[1] FALSE
> (a>=1)|(b<1)
[1] TRUE
```

Difference between & and &&

Both & and && are logical operators representing "and". The difference is that & is a vectorised operator, meaning that it returns a vector; && evaluates from left to right examining only the first element of each vector. The same applies to | and ||.

```
> x<-1:6
> (x > 2) & (x < 5)
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE
> (x > 2) && (x < 5)
Error in (x > 2) && (x < 5) :
    'length = 6' in coercion to 'logical(1)'
> x[(x>2) & (x<5)]
[1] 3 4
> x[(x>2) && (x<5)]
Error in (x > 2) && (x < 5) :
    'length = 6' in coercion to 'logical(1)'
```

The for loop

The `for()` statement allows one to specify that a certain operation should be repeated a fixed number of times.

Syntax

```
for (x in v) { commands }
```

- This sets a variable called x equal to each of the elements of v, in sequence.
- v is usually a vector, but it could also be a list.
- For each value, whatever commands are listed within the curly braces `{}` will be performed.
- The curly braces serve to group the commands so that they are treated by R as a single command.
- If there is only one command to execute, the braces are not needed.

Example: Fibonacci sequence

- The Fibonacci sequence is a famous sequence in mathematics. The first two elements are defined as [1, 1].
- Subsequent elements are defined as the sum of the preceding two elements.
- For example, the third element is 2 (= 1 + 1), the fourth element is 3 (= 1 + 2), the fifth element is 5 (= 2 + 3), and so on.
- To obtain the first 12 Fibonacci numbers in R:

```r
Fibonacci <- numeric(12)
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:12) {
Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
}
```

Example: Modifying a vector

Given a vector `v<-1:5`, we want to increase each of the element by 1. Consider the following for loop.

```
for (x in v) x<-x+1
v
[1] 1 2 3 4 5
x
[1] 6
```

This does not work because in each iteration `x` is assigned a value of the element in v, the statement `x<-x+1` just changes the value of `x`, but not v.

The correct way to modify v is to write assignment statements in terms of v.

```
for (i in 1:length(v)) v[i]<-v[i]+1
```

Example: Infinity loop?

Consider the following for loop.

```
v <- c(1,1)
for (i in v) v <- c(v,1)
v
[1] 1 1 1 1
```

This example illustrates that v in the for loop is evaluated at the start of the loop, changing it subsequently does not affect the loop.

See `help("for")` for details.

The if statement

- The **`if()`** statement allows us to control which statements are executed.

Syntax

```
if (condition) {
     commands when TRUE # do if TRUE
     } else {
     commands when FALSE # do if FALSE
}
```

- This statement causes a set of commands to be invoked if condition evaluates to **TRUE**.
- The else part is optional, and provides an alternative set of commands which are to be invoked in case the logical variable is **FALSE**.

A simple example

```r
x <- 3
if (x > 2) y <- 2 * x else y <- 3 * x
```

Since `x > 2` is `TRUE`, `y` is assigned `2 * 3 = 6`.

If it hadn't been true, `y` would have been assigned the value of `3 * x`.

Example: Listing prime numbers

The function follows is intended to list all the prime numbers up to a given value n. The idea is as follows.

- Begin with a vector of numbers from 2 to n.
- Starting with 2, eliminate all multiples of 2 which are larger than 2.
- Then move to the next number remaining in the vector, in this case, 3. Now, remove all multiples of 3 which are larger than 3.
- Proceed through all remaining entries of the vector in this way. The entry for 4 would have been removed in the first round, leaving 5 as the next entry to work with after 3; all multiples of 5 would be removed at the next step, and so on.

```r
prime_list <- function(n) {
    if (n >= 2) {
        comp <- seq(2, n)
        primes <- c()
        for (i in seq(2, n)) {
            if (any(comp == i)) {
                primes <- c(primes, i)
                comp <- comp[(comp %% i) != 0]
            }
        }
        return(primes)
    } else {
        stop("Input value of n should be at least 2.")
    }
}
```

The `comp` object holds all the candidates for testing.

The `primes` object is set up initially empty, eventually to contain all of the primes that are less than or equal to `n`.

Each integer `i` from 2 through n is checked in sequence to see whether it is still in the vector.

The `any()` function returns a TRUE if at least one of the logical vector elements in its argument is TRUE. In the case that `i` is still in the `comp` vector, it must be a prime, since it is the smallest number that has not been eliminated yet.

All multiples of `i` are eliminated from `comp`, since they are necessarily composite, and `i` is appended to `primes`.

The expression `(comp %% i) == 0` would give TRUE for all elements of `comp` which are multiples of `i`.

Then we can eliminate all multiples of `i` from the `comp` vector using

```
comp <- c(comp[(comp %% i) != 0]
```

Note that this eliminates `i` as well, but we have already saved it in primes.

The while loop

- Sometimes we want to repeat statements, but the pattern of repetition isn't known in advance.
- We need to do some calculations and keep going as long as a condition holds. The `while()` statement accomplishes this.

Syntax

```
while (condition) {statements}
```

- The `condition` is evaluated, and if it evaluates to FALSE, nothing more is done.
- If it evaluates to TRUE the statements are executed, `condition` is evaluated again, and the process is repeated.

Example: Fibonacci numbers

Suppose we want to list all Fibonacci numbers less than 300.

We don't know beforehand how long this list is, so we wouldn't know how to stop the `for()` loop at the right time, but a `while()` loop is perfect:

```
Fib1 <- 1
Fib2 <- 1
Fibonacci <- c(Fib1)
while (Fib2 < 300) {
    Fibonacci <- c(Fibonacci, Fib2)
    oldFib2 <- Fib2
    Fib2 <- Fib1 + Fib2
    Fib1 <- oldFib2
}
```

Exercise

The variable `oldFib2` isn't strictly necessary.

Rewrite the Fibonacci while loop with the update of `Fib1` based just on the current values of `Fib1` and `Fib2`.

```r
Fib1 <- 1
Fib2 <- 1
Fibonacci <- c(Fib1)
while (Fib2 < 300) {
    Fibonacci <- c(Fibonacci, Fib2)
    Fib2 <- Fib1 + Fib2
    Fib1 <- max(Fibonacci)
}
print(Fibonacci)
```

Example: Compound interest

In this example we use a while loop to work out how long it will take to pay off a loan.

```
r <- 0.11                  # Annual interest rate
period <- 1/12             # Time between repayments
debt_initial <- 1000       # Amount borrowed
repayments <- 12           # Amount repaid each period
time <- 0
debt <- debt_initial
while (debt > 0) {
     time <- time + period
     debt <- debt*(1 + r*period) - repayments
}
cat('Loan will be repaid in', time, 'years\n')
```

The repeat loop

- Sometimes we don't want a fixed number of repetitions of a loop, and we don't want to put the test at the top of the loop the way it is in a while loop.
- In this situation we can use a repeat loop. This loop repeats until we execute a break statement.

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

- The **`break`** statement causes the loop to terminate immediately. This statements can also be used in `for` and `while` loops.
- The **`next`** statement causes control to return immediately to the top of the loop; it can also be used in any loop.

Example: Newton's method for root finding

- Newton's method is a popular numerical method to find a root of an algebraic equation $f(x) = 0$.
- If $f(x)$ has derivative $f'(x)$, then the following iteration should converge to a root of the above equation if started close enough to the root.

$$x_0 = \text{ initial guess,}$$
$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}.$$

- The idea is based on the Taylor approximation

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1})\, f'(x_{n-1}).$$

Suppose we want to find a root of

$$f(x) = x^3 + 2x^2 - 7 = 0$$

```
x <- 1
tolerance <- 0.000001
repeat {
    f <- x^3 + 2 * x^2 - 7
    if (abs(f) < tolerance) break
    f.prime <- 3 * x^2 + 4 * x
    x <- x - f / f.prime
}
x
```

The use of `next` inside a loop

Given a binary random sequence of 0 and 1 generated as follows.

```
> bseq<-sample(c(0,1),size=20,replace=T)
# generate random binary sequence
> bseq
 [1] 1 1 1 1 0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1
```

We are interested to find the maximum length of a consecutive series of equal number within the sequence. For example, the maximum length of `1` in `bseq` is 4; and the maximum length of `0` is 6.

- The functions `max1(v)` and `max0(v)`are designed to return the maximum length of `1` and `0` in the input binary sequence v respectively.
- First, we need to set a logical flag `is_prev1` to keep track the status of previous number if it is equal to `1` (`1` is TRUE, `0` is FALSE).
- We only need to write `max1(v)` since `max0(v)` can be obtained easily by applying `max1(1-v)`.

```r
max1<-function(v) {
      is_prev1<-FALSE        # initialize flag to False
      n1<-0; count<-0  # initialize counter
      for (i in v) {
            if ((i==1)&(is_prev1==TRUE)) {
                  count<-count+1
                  if (count>=n1) n1<-count
                  next       # skip to next element in v
            }
            if ((i==1)&(is_prev1==FALSE)) {
                  count<-1; is_prev1<-TRUE
                  if (count>=n1) n1<-count
                  next       # skip to next element in v
            }
            if ((i==0)&(is_prev1==TRUE)) {
                  count<-0   # reset counter
                  is_prev1<-FALSE
            }
      }
      return(n1)
}
```

```
max0<-function(v) { max1(1-v) }


maxlength<-function(v) {
     n1<-max1(v)
     n0<-max0(v)
     out<-list(n0,n1) # create list
     names(out)<-c("n0","n1") # apply label to out
     return(out) # output
}

> maxlength(bseq)
$n0
[1] 6

$n1
[1] 4
```

Example: Temperature conversion

We write a function to produce a conversion table of temperature from Fahrenheit (from 0 to 200 with increment 20) to their Celsius equivalent.

The general conversion formula is

$C = (F - 32) \times 5/9$,

where C is the temperature in degrees Celsius, F is the temperature in degrees Fahrenheit.

Version 1: Using for loop

```
ftoc.v1<-function(low,up,step) {
      # convert F to C from low to up with step
      f<-seq(low,up,step)
      # create vector of value f
      len<-length(f)
      # find the length of f
      c<-rep(0,len)
      # create a vector c
      for (i in 1:len) {
            # for loop
            c[i]<-(5/9)*(f[i]-32)
            # convert f to c element-wise
      }
      return(cbind(f,c))
      # output table
}
```

We test this function using

```
> ftoc.v1(0,200,20)
        f          c
 [1,]   0 -17.777778
 [2,]  20  -6.666667
 [3,]  40   4.444444
 [4,]  60  15.555556
 [5,]  80  26.666667
 [6,] 100  37.777778
 [7,] 120  48.888889
 [8,] 140  60.000000
 [9,] 160  71.111111
[10,] 180  82.222222
[11,] 200  93.333333
```

Version 2: Using while loop

```
ftoc.v2<-function(low,up,step) {
     f<-seq(low,up,step)
     # create a sequence in f
     len<-length(f)
     # find the length of f
     c<-rep(0,len)
     # create a vector of zeroes
     i<-1           # initialize the loop index i
     while (i<=len) {    # while loop
          c[i]<-(5/9)*(f[i]-32)
          i<-i+1
          # increase i by 1 for each iteration
     }
     return(cbind(f,c))
}
```

Version 3: Using repeat loop

```
ftoc.v3<-function (low,up,step) {
     f<-seq(low,up,step)
     len<-length(f)
     c<-rep(0,len)
     i<-1
     repeat {      # repeat loop
          if (i>len) break
          # break point inside the repeat loop
          c[i]<-(5/9)*(f[i]-32)
          i<-i+1
     }
     return(cbind(f,c))
}
```

Which version is better?

- The answer is they are all as good as (in fact, as bad as) each other.
- The choice is mainly depends on the programmer’s style.
- A better version is to use **vectorized** operation **without** using loop.
- Using vectorized operation is much faster that using loop especially in large scale computation.

Version 4: Using vector-based operation

```
ftoc<-function (low,up,step) {
   f<-seq(low,up,step) # create vector f
   c<-(5/9)*(f-32)     # compute vector c from f
   return(cbind(f,c))  # combine f and c
}
```

`ftoc.v1()`, `ftoc.v2()` and `ftoc.v3()` are written for illustrating the loop structure only.

In practice, we should write our function in R using vector-based operation whenever possible.

Commonly used programming structures and syntaxes

Syntax	Description
if (cond) expr	evaluates *cond*; if T, evaluates *expr*
if (cond) expr1 else expr2	evaluates *cond*; if T, *expr1*, if F, *expr2*
ifelse(cond, expr1, expr2)	a vectorized version of if-else
switch(expr, …)	evaluates *expr* and compare it to arguments
break	terminates current loop and jumps out
next	terminates current iteration and immediately starts next iteration of the loop
return(expr)	terminates current function and immediately returns the value of *expr*
stop(message)	terminates evaluation of the current function and display message
while (cond) expr	evaluates *cond*; if T evaluates *expr*, then goes back to the top of the loop, evaluates *cond* again
repeat expr	repeat *expr* indefinitely, some breaks should be include inside *expr*
for (name in expr1) expr2	evaluates *expr2* once for each name in *expr1*

Example: Random number generator

We want to write a function to generate n random numbers from a distribution specified by the user.

```
my.ran1 <- function (n, dist) {
     # n=sample size, dist="norm"or"uniform"
     # version 1: using if else
     if (dist == "norm") rnorm(n) else
     if (dist == "uniform") runif(n) else
     stop("Unknown distribution")
}
my.ran2 <- function(n, dist="norm"){
     # default value of dist is "norm"
     # version 2: using switch
     switch(dist, norm=rnorm(n), uniform=runif(n),
               stop("Unknown distribution"))
}
```

- Let us try these two functions with the following:

```
> my.ran1(5,"uniform")
# generate 5 random numbers from uniform(0,1)
[1] 0.2844390 0.3973675 0.3900139 0.8574467
0.2992833
> my.ran1(4,"t")             # error
Error in my.ran1(4, "t") : Unknown distribution
> my.ran2(6,"norm")
# generate 6 random numbers from normal(0,1)
[1] -0.6621585  1.2347028  1.2259735  0.4152614
0.1707441  1.3067714
> my.ran2(6)
# use default value for dist
[1]  1.8412086 -0.4728274  1.2872517 -0.2673689 -
0.9515996 -1.4746707
```

Note that in these functions, the parameter `dist` is a string.

In `my.ran1`, the if-else structure is nested.

In `my.ran2`, switch is used and the default value for `dist` is `norm`.

In both functions, `stop` will terminate the function and output an error message if `dist` is not `norm` or `uniform`.

Example: Signum function

Signum function is a very simple but useful function in mathematics or statistics. The signum function is defined as

$$\operatorname{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

```r
sgn1 <- function(x){
    # version 1: signum function using if else
    for(i in 1:length(x)) {
        if(x[i] > 0)
            x[i] <- 1
        else if (x[i] < 0)
            x[i] <- -1
    }
    return(x)
}
```

```
sgn2 <- function(x){
    # version 2: signum function using vectorized
    ifelse
    x<-ifelse(x>0, 1, ifelse(x<0, -1, 0))
    return(x)
}

sgn3 <- function(x){
    # version 3: signum function using selection
    x[x>0] <- 1
    x[x<0] <- -1
    return(x)
}
```

Not onlv version 2 and 3 is much faster than version 1, but also can handle **NA** (missing data) as illustrated in the following:

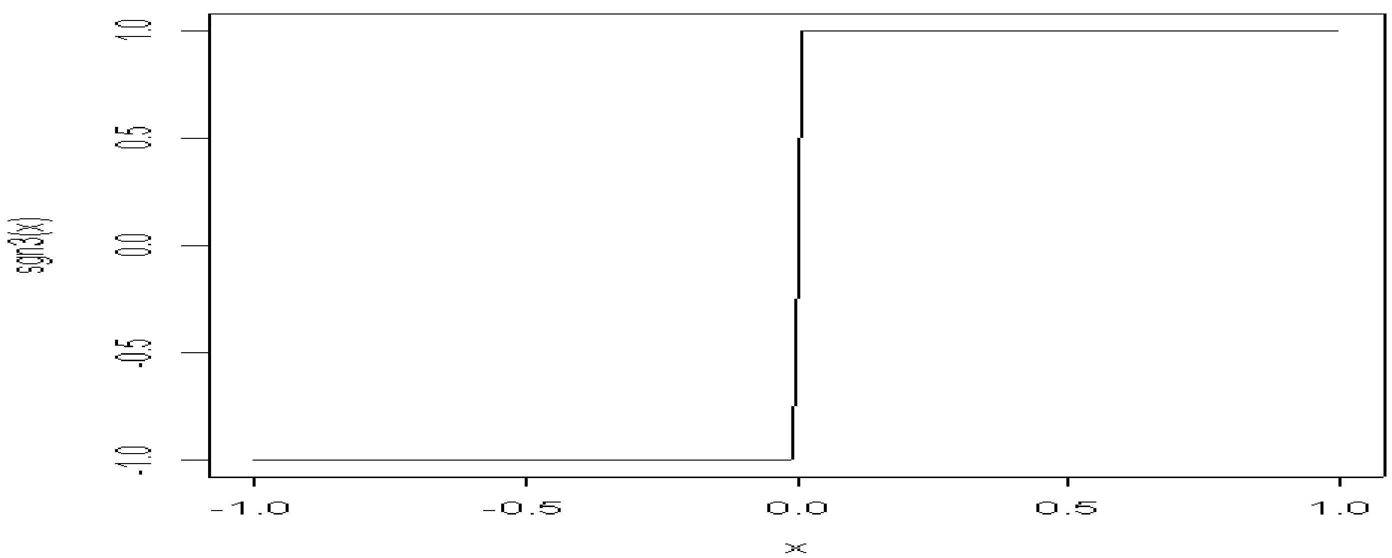
```
> x<- -3:3                  # create a vector
> y<-c(x,NA)                # create a vector with NA
> sgn1(x)
[1] -1 -1 -1  0  1  1  1
> sgn2(x)
[1] -1 -1 -1  0  1  1  1
> sgn3(x)
[1] -1 -1 -1  0  1  1  1
> sgn1(y)
Error in if (x[i] > 0) x[i] <- 1 else if (x[i] < 0) 
x[i] <- -1 :
        missing value where TRUE/FALSE needed
> sgn2(y)
[1] -1 -1 -1  0  1  1  1 NA
> sgn3(y)
[1] -1 -1 -1  0  1  1  1 NA
```

The `ifelse()` function can be used to avoid warning message as follow:

```
> x<-c(4:-2)
# create vector with negative numbers
> sqrt(x)
# warning: NaN = Not a Number
[1] 2.000000 1.732051 1.414214 1.000000 0.000000
NaN      NaN
Warning message:
In sqrt(x) : NaNs produced
> sqrt(ifelse(x>=0,x,NA))
# No warning message, NA = Not Applicable
[1] 2.000000 1.732051 1.414214 1.000000 0.000000
NA       NA
```

Once a function is defined, it can be used in the same way as the other built-in functions in R. For example, we can plot the `sgn3` function as follow:

```
> x<-seq(-1,1,0.01)
# create a vector from -1 to 1 with step 0.01
> plot(x,sgn3(x),type="l")
# plot sgn3 with type = line
```



Example: Roots of a quadratic equation

We know that the roots of the quadratic equation

$$a_2x^2 + a_1x + a_0 = 0$$

is given by

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_2}.$$

However, a program simply relies on the above formula would be problematic, since it cannot handle cases such as $a_2 = 0$ or $a_1^2 - 4a_2a_0 < 0$.

A proper design should be able to handle all possible input values of a_0, a_1 and a_2.

```r
quad <- 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
            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)
}
```