Lab Note 2. Rounding Functions and Control Statements

Cheng Peng

MAT325 Numerical Analysis

Contents

1 Introduction

In this lab note, we introduce some R functions commonly used in error analysis. Various control statements are the building blocks in developing numerical algorithms. We will focus on developing pseudo-code and implementing it through examples.

2 Rounding and Related Functions

There are several R functions we can use to handle decimals and manage significant digits in numerical analysis. A list of a few such functions in the base R.

2.1 round()

`round(number,digits)` rounds the number to the number of digits provided. Here is an example of the round function in action.

```
round(1234.56789,3)    # keep three decimal places
```

```
## [1] 1234.568
```

```
round(1234.56789,5)    # R only displays 7 digits by default
```

```
## [1] 1234.568
```

```
round(1234.56789,0)    # keep no decimal place
```

```
## [1] 1235
```

```
round(1234.56789,-3)   # set three digits before decimal points to zeros
```

```
## [1] 1000
```

```
round(1234.56789,-5)
```

```
## [1] 0
```

2.2 Controlling Number of Displayed Digits

There are different ways to change the number of displayed digits. We can change the default `option(digits = 7)` to `option(digits = 10)` to display 10 digits (Caution: this is a global option!).

```
options(digits = 10)
round(1234.56789,3)    # keep three decimal places
```

```
## [1] 1234.568
```

```
round(1234.56789,5)    # R only displays 7 digits by default
```

```
## [1] 1234.56789
```

```
round(1234.56789,6)    # R only displays 7 digits by default
```

```
## [1] 1234.56789
```

The c-style formatting function can also be used to display the desired number of digits.

```
sqrt(2)                    # this will display 10 digits. why?
```

```
## [1] 1.414213562
```

```
sprintf("%.20f", sqrt(2))   # Note this displays a string value
```

```
## [1] "1.41421356237309514547"
```

```
options(digits = 20)        # change options to display 20 digits
sqrt(2)
```

```
## [1] 1.4142135623730951455
```

```
options(digits = 7)         # change the options back to the default
```

2.3 Signif()

`Signif()` is an R **rounding function** with the format of `signif(number,digits)` and it **rounds** the number to the number of digits provided.

```
signif(1234.566789,4)
```

```
## [1] 1235
```

```
signif(1234.566789,7)
```

```
## [1] 1234.567
```

```
signif(1234.566789,1)
```

```
## [1] 1000
```

The `signif function` round to the given number of digits. Both the round and `signif functions` use standard rounding conventions.

2.4 `floor()` and `ceiling()`

`floor()` is a rounding function with the format of `floor(number)` and it rounds the number to **the nearest integer that is less than its value**.

```
floor(3.14159)
```

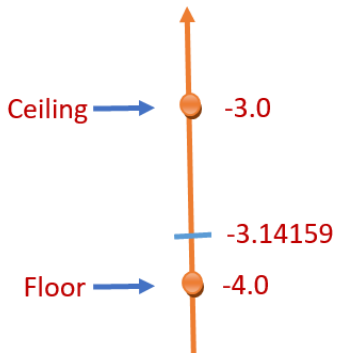
```
## [1] 3
```

```
floor(-3.14159)
```

```
## [1] -4
```

`floor()` just drops the decimal places of a decimal number. In the above example, $-3.14159 = -4 + 0.85841$, `floor()` throws 0.85841 away and only keeps the integral part -4.

`ceiling()` is a rounding function with the format of `ceiling(number)` that rounds the number to **the nearest integer that is greater than its value**.




```
ceiling(3.14159)
```

```
## [1] 4
```

```
ceiling(-3.14159)
```

```
## [1] -3
```

2.5 `trunc()`

`trunc()` is a rounding function with the format of `trunc(number)` that **drops all digits after the decimal point**.

```
trunc(3.14159)
```

```
## [1] 3
```

```
trunc(-3.14159)
```

```
## [1] -3
```

3 Control Statements

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 decide after assessing the variable.

There are 8 types of control statements in R:

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

We will introduce `if/if-else` statements and `for/while` loops in this note and use them to implement the algorithm with the given pseudo-code given in the lecture note.

3.1 Conditional Statements

3.1.1 IF Statement

This control structure checks whether the expression provided in parenthesis is true or not. If true, the execution of the statements in braces `{}` continues. The syntax is given by

Syntax:

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

Example 1:

```
x <- 100                                # numerical scalar
##
if(x > 10){                             # conditional statement
print(paste(x, "is greater than 10")) # paste() is a string function. one
                                        # can pass a numerical argument to
                                        # this string function.
}
```

```
## [1] "100 is greater than 10"
```

Example 2:

```r
x <- pi                        # numerical scalar
##
if(x == 3.14159){              # conditional statement
print(paste(x, "is pi!"))
}
```

3.1.2 IF-ELSE Statement

If the **if condition** is determined as true, it will execute the statements written *inside the if block*. Otherwise, if the **if condition** is not satisfied, it will return the statements written inside the *else block*.

Syntax

```
if (condition) {# The statement will execute
                # if the condition is satisfied.
     statement_1
     ..........
   }
else { # The statement will execute if the condition
       # turns out to be not satisfied.
   statement_2
   ..........
 }
```

Example 3

```r
x <- pi                        # numerical scalar
##
if(x == 3.14159){              # conditional statement
   print(paste(x, "is pi!"))
   } else {
   print(paste(x, "is NOT pi!"))  # Caution: x will be printed as a string!
  }
```

```
## [1] "3.14159265358979 is NOT pi!"
```

3.1.3 ELSE-IF Statement

It continuously checks certain conditions. If any of the if the condition is satisfied, it will return statements written inside the block. If none of the if the condition is true, it will execute the statements written inside the else block.

Syntax

```
if (condition){
     expression
  } else if (condition){
     expression
  } else if (condition){
     expression
  } else { # The statement will execute if none of the if
           # conditions turn out to be true.
    statement
}
```

Example 4

```r
y = 10

if (x > y) {
    print(x)
} else if (y == x) {
    print("x and y are equal")
} else {
    print(y)
}
```

```
## [1] 10
```

3.2 Loops in R

A loop is used to perform repetitive tasks. There are three types of loops in R: for-loop, while-loop, and repeat-while-loop. These loops are used with `if-else` like conditional statements to execute statements conditionally.

3.2.1 FOR Loops

A **for-loop** will run statements a pre-set number of times n.

Syntax

```r
 for (i in 1:n){
  executable statements
 }
```

Example 5

Find 10 factorial (10!). Note that $10! = 0 \times 1 \times 2 \times \cdots \times 9 \times 10$. We only need 10 iterations of the cumulative product to find the value.

```r
result = 1  # initial value to start the process of iteration
for(i in 1:10){
  result = result * i
}
cat("\n 10! =", result,".\n")
```

```
##
##  10! = 3628800 .
```

`for-loop` is usually used jointly with other conditional statements defined by `if-else`, `break`, `next`, etc.

Example 6. Let X =c(10 18 42 46 11 29 15 22 20 8 49 47 1 16 23 13 32 27 19 37 31 43 39 24 44 12 3 2 9 7 50 4 30 6 45 38 17 14 26 36 21 41 40 5 33 34 25 28 35 48). We want to select 10 smallest odd numbers and store them in a vector.

```r
# define the vector
 X=c(10, 18, 42, 46, 11, 29, 15, 22, 20, 8, 49, 47, 1, 16, 23, 13, 32, 27,
     19, 37, 31, 43, 39, 24, 44, 12, 3, 2, 9, 7, 50, 4, 30, 6, 45, 38, 17,
     14, 26, 36, 21, 41, 40, 5, 33, 34, 25, 28, 35, 48)
# Note that the largest of the first 10 smallest odd number is 19.
oddVec = NULL      # store the first 10 smallest odd numbers
k = 1              # initial index of oddVec
for (i in 1:100){
```

```
    xi = X[i]
    I = X[i]%%2          # %% gives the remainder of the division
    if(I == 0){
        next             # skip the current iteration and jump to the next
    } else{
       if(xi > 19) {
          next           # skip the current iteration and jump to the next
      } else{
          oddVec[k] = xi
          if (k == 10) break     # beak the iteration once all
                                 # desired odd numbers are selected!
          k = k + 1              # update the index for the next valid odd number
      }
    }
  }
oddVec
```

```
##  [1] 11 15  1 13 19  3  9  7 17  5
```

3.2.2 WHILE Loops

The while loop repeats statements as long as a certain condition is true. Stated another way, the while loop will stop when the condition is false (for example, the user types 0 to exit). Each time the loop starts running, it checks the condition.

See the example in the next section.

3.2.3 REPEAT-IF-WHILE Loops

This loop is equivalent to the **DO-WHILE** loop in other languages such as SAS and C++.

The `repeat loop` does not have any condition to **terminate** the loop. We need to put an **exit condition** implicitly with a **break statement** inside the loop.

Syntax:

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

Example 6

```
i = 1
repeat {
  print(i)
  i = i + 1
  if(i >= 5 )
    break
}
```

```
## [1] 1
## [1] 2
```

```
## [1] 3
## [1] 4
```

4 Numerical Example

Example 4 (Numerical Implementation): The nth Taylor polynomial for $f(x) = e^x$ expanded about $x_0 = 0$ is

$$P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

and the value of e to six decimal places is 2.718282. **Construct an algorithm** to determine the **minimal value of** n required for

$$|e - P_n(1)| < 10^{-5},$$

without using the Taylor polynomial remainder term.

Solution: The objective is to determine the degrees of the Taylor polynomial evaluated at $x = 1$ to approximate e. The input values are (1) tolerance TOL and the initial degree of the Taylor polynomial. The output is the smallest degree of the Taylor polynomial that meets $|e - P_n(1)| < TOL$.

Caution: In general, one should consider including two stopping rules: error tolerance and maximum iterations. In this particular example, the maximum number of iterations is simply the solution to the problem. Therefore, there is one stopping rule: TOL

4.1 VERSION 1

This is modified from the pseudo-code from the example in the lecture note.

```
INPUT  initial degree: n,
            tolerance: TOL,

OUTPUT the desired degree N of the polynomial

Step 1. SET   n = 0;
              Pn = 1;
             ERR = exp(1) - 1;
Step 2. WHILE ERR >= TOL DO:
        1.    n = n + 1
              Pn = Pn + 1/n!          # n! = n factorial
             ERR = exp(1) - Pn        # exp(1) = 2.718282
        2. IF |ERR| < TOL DO:
               OUTPUT (N)
               WRITE (tolerance achieved!)
               STOP
           ELSE DO:                 # optional: print out something
                                    # to monitor the iterative process
               WRITE(iteration number)
          ENDIF
        ENDWHILE
```

The following is the R code for implementing the above pseudo-code.

```
 n = 0
 TOL = 10^(-5)
 Pn = 1
 ERR = exp(1) - Pn
# Loop Starts
while(ERR >= TOL){
  n = n + 1
  Pn = Pn + 1/factorial(n)
  ERR = exp(1) - Pn
  if(ERR < TOL){
  cat("\n\n The desired degrees of the Taylor polynomial is ", n+1,". \n")
  cat("Tolerance achieved!\n\n\n")
  } else{
  cat("\n Iteration number: ", n, ". ")
  }
 }
```

```
##
##  Iteration number:  1 .
##  Iteration number:  2 .
##  Iteration number:  3 .
##  Iteration number:  4 .
##  Iteration number:  5 .
##  Iteration number:  6 .
##  Iteration number:  7 .
##
##  The desired degrees of the Taylor polynomial is  9 .
## Tolerance achieved!
```

4.2 Version 2

Change the order for updating the degrees of the polynomial and add additional outputs

```
INPUT  initial degree: n,
            tolerance: TOL,

OUTPUT the desired degree N of the polynomial

Step 1. SET   n = 0;
              Pn = 0;
             ERR = exp(1);
Step 2. WHILE ERR >= TOL DO:
        1.  Pn = Pn + 1/n!          # n! = n factorial
           ERR = exp(1) - Pn        # exp(1) = 2.718282
        2. IF |ERR| < TOL DO:
               OUTPUT (N, Pn, absolute error)
               WRITE (tolerance achieved!)
               STOP
           ELSE DO:                 # optional: print out something
                                    # to monitor the iterative process
               WRITE (iteration number, absolute error)
```

```
            n = n + 1
        ENDIF
    ENDWHILE
```

The following is the R code for implementing the above pseudo-code.

```r
n = 0
TOL = 10^(-5)
Pn = 0
ERR = exp(1)
# Loop Starts
while(ERR >= TOL){
  Pn = Pn + 1/factorial(n)
  ERR = exp(1) - Pn
  if(ERR < TOL){
     cat("\n\nTolerance achieved!\n\n")
     cat("\n degree N = ", n+1, ".")
     cat("\n e = ", exp(1), ".")
     cat("\n Pn =", Pn, ".")
     cat("\n Absolute Error =", ERR, ".")
  } else{
        n = n + 1
     cat("\n Iteration numer: ", n, ". Absolute Error = ", ERR, ".")
  }
 }
```

```
##
##  Iteration numer:  1 . Absolute Error =  1.718282 .
##  Iteration numer:  2 . Absolute Error =  0.7182818 .
##  Iteration numer:  3 . Absolute Error =  0.2182818 .
##  Iteration numer:  4 . Absolute Error =  0.05161516 .
##  Iteration numer:  5 . Absolute Error =  0.009948495 .
##  Iteration numer:  6 . Absolute Error =  0.001615162 .
##  Iteration numer:  7 . Absolute Error =  0.0002262729 .
##  Iteration numer:  8 . Absolute Error =  2.786021e-05 .
##
## Tolerance achieved!
##
##
##  degree N =  9 .
##  e =  2.718282 .
##  Pn = 2.718279 .
##  Absolute Error = 3.058618e-06 .
```