Flow control and functions in R

Yijia Zhao

1.Section 4.1.1, exercises 2

```
n <- 30
f <- numeric(n)
f[1:2] <- c(1,1)
for (i in 3:n){
  f[i] <-f[i-2] + f[i-1]
}
ratio <- numeric(n)
ratio[1] <- 1
ratio[2:n] <- f[2:n]/f[1:(n-1)]
ratio
```

```
##  [1] 1.000000 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000
##  [8] 1.615385 1.619048 1.617647 1.618182 1.617978 1.618056 1.618026
## [15] 1.618037 1.618033 1.618034 1.618034 1.618034 1.618034 1.618034
## [22] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
## [29] 1.618034 1.618034
```

```
(1+sqrt(5))/2
```

```
## [1] 1.618034
```

```
ratio - (1+sqrt(5))/2
```

```
##  [1] -6.180340e-01 -6.180340e-01  3.819660e-01 -1.180340e-01  4.863268e-02
##  [6] -1.803399e-02  6.966011e-03 -2.649373e-03  1.013630e-03 -3.869299e-04
## [11]  1.478294e-04 -5.646066e-05  2.156681e-05 -8.237677e-06  3.146529e-06
## [16] -1.201865e-06  4.590718e-07 -1.753498e-07  6.697766e-08 -2.558319e-08
## [21]  9.771908e-09 -3.732537e-09  1.425702e-09 -5.445699e-10  2.080072e-10
## [26] -7.945178e-11  3.034772e-11 -1.159184e-11  4.427569e-12 -1.691314e-12
```

(a)According to the result, the sequence appears to converge.

(b)According to the result, the sequence converges to golden ratio. Now let us prove it theoretically.

Since $f_n = f_{n-2} + f_{n-1}$, we can derive that $f_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$. Therefore, $f_n/f_{n-1} = [(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]/[(\frac{1+\sqrt{5}}{2})^{n-1} - (\frac{1-\sqrt{5}}{2})^{n-1}] = \frac{1+\sqrt{5}}{2} + \sqrt{5}/[(\frac{1+\sqrt{5}}{2})^{n-1} - (\frac{1-\sqrt{5}}{2})^{n-1}] \to \frac{1+\sqrt{5}}{2}$.

2.Section 4.1.1, exercises 3

(a)15;(b)1,2,3,4,5;(c)0,1,2,3,4,5;(d)120;(e)3,21,23,6,11,15,$\cdots$

```
answer <- 0
for (j in 1:5) answer <- answer + j
answer
```

```
## [1] 15
```

```
answer <- NULL
for (j in 1:5) answer <- c(answer, j)
answer
```

```
## [1] 1 2 3 4 5
```

```
answer <- 0
for (j in 1:5) answer <- c(answer, j)
answer
```

```
## [1] 0 1 2 3 4 5
```

```
answer <- 1
for (j in 1:5) answer <- answer * j
answer
```

```
## [1] 120
```

```
answer <- 3
for (j in 1:15) answer <- c(answer, (7 * answer[j]) %% 31)
answer
```

```
##  [1]  3 21 23  6 11 15 12 22 30 24 13 29 17 26 27  3
```

If I do not now the rule of the last sequence, I can still predict successive elements because answer[16]=answer[1], which means the following elements are replication of former ones.

3.Section 4.1.2, exercises 4

```
GIC_interest <- function(initial_amount, years){
 if (years <= 3){
   return(initial_amount*((1+0.04)^years-1))
 }else{
   return(initial_amount*((1+0.05)^years-1))
 }
}
```

4.Section 4.1.2, exercises 5

```
mortgage_rate <- function(n, p, open){
  if (open == TRUE){
    i <- 0.005
  } else{
    i <- 0.004
  }
  R <- p*i/(1-(1+i)^(-n))
  return(R)
}
```

5.Section 4.1.3, exercises 2

```
Fibonacci <- c(1,1)
while(Fibonacci[length(Fibonacci)]<300){
  Fibonacci <- c(Fibonacci, Fibonacci[length(Fibonacci)] + Fibonacci[length(Fibonacci)-1])
}
print(Fibonacci[-length(Fibonacci)])
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

6.Section 4.1.3, exercises 4

```
fixed_point <- function(i){
  diff <- 1.000000
  times <- 0
  while (diff >=  0.000001){
    temp <- i
    i <- (1-(1+i)^(-20))/19
    diff <- abs(i-temp)
    times <- times + 1
  }
  return(paste(times,i))
}
fixed_point(0.006)
```

```
## [1] "74 0.00495413936538335"
```

```
fixed_point(0.002)
```

```
## [1] "115 0.00491729014790449"
```

```
fixed_point(0.03)
```

```
## [1] "104 0.00495342231600276"
```

```
fixed_point(0.6)
```

```
## [1] "106 0.00495377931234107"
```

When I try other starting guess, it still converges to the fixed point which is around 0.0049 in about 100 times of iterations.

7.Section 4.1.3, exercises 5

It has been designed to calculate the number of iterations in the previous answer.

8.Section 4.1.5, exercise 2

```
Eratosthenes <- function(n) {
  # Print prime numbers up to n (based on the sieve of Eratosthenes)
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    while (length(sieve) > 0) {
      p <- sieve[1]
      primes <- c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]
      }
    return(primes)
  }else{
      stop("Input value of n should be at least 2.")
  }
}
```

(a)We can see that if the input number n is less than 2, the function will output the error information “Input value of n should be at least 2.”. If the input number is no less than 2, the vector primes will record every

number which can not be exactly divided by any number from 2 to itself-1. That is to say, the function will output all prime numbers no more than the input number n.

To verify this result, we can do some calculations with this function.

```
Eratosthenes(49)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47
```

We can see that the function outputs all prime numbers no more than 41.

(b)If a number k is not a prime number, it must can be exactly divided by a number which is no more than its square root $\sqrt{k} \leq \sqrt{n}$. So when $p >= \sqrt{n}$, all remaining numbers in sieve can not be exactly divided by numbers no more than its square root. So they must be prime numbers.

(c)In view of the result of (b), we can modify our function as follows.

```
Eratosthenes <- function(n) {
  # Print prime numbers up to n (based on the sieve of Eratosthenes)
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    while (length(sieve) > 0) {
      p <- sieve[1]
      primes <- c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]
      if (p >= sqrt(n)){
        break()
      }
      }
    return(c(primes,sieve))
  }else{
      stop("Input value of n should be at least 2.")
  }
}
Eratosthenes(49)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47
```

9.Section 4.2.1, exercises 2

(a)

```
compound.interst <- function(primary, interestrate, periods){
  return(primary*((1+interestrate)^periods))
}
```

(b)

```
compound.interst(1000,0.01,30)
```

```
## [1] 1347.849
```

He will have $1347.849 in the bank at the end of 30 months.

10.Section 4.2.1, exercises 3

Here we ask users to input the function f and the interval $[a, b]$ to find the zero(requiring only one zero existing in the interval) and output the point whose absolute function value is no more than 0.000 001.

```
find.zero <- function(f, a, b){
  x <- c(a,b)
  y <- c(f(a),f(b))
  diff <- min(abs(y))
  if (y[1]*y[2] <= 0){
     while (diff > 0.000001){
       xnew <- mean(x)
       id <- (y*f(xnew)>0)
       x[id] <- xnew
       y[id] <- f(xnew)
       diff <- min(abs(y))
  }
    zero <- x[abs(y)<= 0.000001]
    return(zero)
  }else{
    return("There is no single zero in this interval.")
  }
}
```

Give an example of calculation.

```
f1 <- function(x){
  return(sin(x))
}
find.zero(f1,3,4)
```

```
## [1] 3.141592
```

```
find.zero(f1,0.5,1)
```

```
## [1] "There is no single zero in this interval."
```

11.Section 4.4.1, exercises 1

```
mergesort <- function (x, decreasing = FALSE) {
  # Check for a vector that doesn't need sorting
  len <-length(x)
  if (len < 2) result <- x
  else {
    # 2: sort x into result
    # 2.1: split x in half
    y <- x[1:(len %/% 2)]
    z <- x[(len %/% 2 + 1):len]
    # 2.2: sort y and z
    y <- mergesort(y)
    z <- mergesort(z)
    # 2.3: merge y and z into a sorted result
    result <- c()
    # 2.3.1: while (some are left in both piles)
    while (min(length(y), length(z)) > 0) {
      # 2.3.2: put the smallest first element on the end
      # 2.3.3: remove it from y or z
      if (y[1] < z[1]) {
        result <- c(result, y[1])
        y <- y[-1]
```

```
        } else {
          result <- c(result, z[1])
          z <- z[-1]
        }
      }
    # 2.3.4: put the leftovers onto the end of result
    if (length(y) > 0)
      result <- c(result, y)
    else
      result <- c(result, z)
  }
  if (decreasing == TRUE){
    result <- result[length(result):1]
  }
  return(result)
}
a <- c(5,3,7,0,1,6,4)
mergesort(a, TRUE)
```

```
## [1] 7 6 5 4 3 1 0
```

```
mergesort(a, FALSE)
```

```
## [1] 0 1 3 4 5 6 7
```

12.Section 4.4.1, exercises 2

(a)

```
Newton.method <- function(f,g,x0,y0,dist){
  partial.fx <- D(f,'x')
  partial.fy <- D(f,'y')
  partial.gx <- D(g,'x')
  partial.gy <- D(g,'y')
  x <- x0
  y <- y0
  while (abs(eval(f))>dist||abs(eval(g))>dist){
    d <- eval(partial.fx)*eval(partial.gy)-eval(partial.fy)*eval(partial.gx)
    xnew <- x - (eval(partial.gy)*eval(f)-eval(partial.fy)*eval(g))/d
    ynew <- y - (eval(partial.fx)*eval(g)-eval(partial.gx)*eval(f))/d
    x <- xnew
    y <- ynew
  }
  return(c(x,y))
}
```

(b) Because Newton method can be fixed around one solution, we can find two solutions with different initial guesses.

```
ans1 <- Newton.method(expression(x+y),expression(x^2+2*y^2-2),-1,1,0.000001)
ans2 <- Newton.method(expression(x+y),expression(x^2+2*y^2-2),1,-1,0.000001)
```

So the two solutions of this system are (x, y) = (-0.8164966, 0.8164966) and (x, y) = (0.8164966, -0.8164966).

13.Chapter 4 exercises 1

```
directpoly <- function (x, poly.coef){
  ans <- 0
  for (i in 1:length(poly.coef) ){
    ans <- ans + poly.coef[i]*(x^(i-1))
  }
  return(ans)
}
```

14.Chapter 4 exercises 2

```
hornerpoly <- function (x, poly.coef){
  a <- poly.coef[length(poly.coef)]
  for (i in (length(poly.coef)-1):1 ){
    a <- a*x + poly.coef[i]
  }
  return(a)
}
```

Test the function with examples.

```
hornerpoly(2,c(3,2,1))
```

```
## [1] 11
```

```
hornerpoly(c(1,2,3),c(3,2,1))
```

```
## [1]  6 11 18
```

The answers are correct.

15.Chapter 4 exercises 3

(a)

```
system.time(directpoly(x=seq(-10, 10, length=5000000), c(1, -2, 2, 3, 4, 6, 7)))
```

```
##    user  system elapsed
##    1.23    0.05    1.28
```

```
system.time(hornerpoly(x=seq(-10, 10, length=5000000), c(1, -2, 2, 3, 4, 6, 7)))
```

```
##    user  system elapsed
##    0.11    0.05    0.16
```

According to the results, hornerpoly is far more efficient than directpoly in this case.

(b)

```
system.time(directpoly(x=seq(-10, 10, length=5000000), c(-3,17,2)))
```

```
##    user  system elapsed
##    0.27    0.02    0.28
```

```
system.time(hornerpoly(x=seq(-10, 10, length=5000000), c(-3,17,2)))
```

```
##    user  system elapsed
##    0.09    0.00    0.09
```

When the number of polynomial coefficients is smaller, hornerpoly is still more efficient than directpoly. But the difference is not so distinct.