STAT37810_HW1

Seung ah Ha

2018 10 10

4.1.1, exercise 2

(a)

```
fib <- numeric(30)
fib[1]=1 ; fib[2] = 1
for(i in 3:30){
  fib[i] <- fib[i-2]+fib[i-1]
}

ratio <- c()
ratio[1] <- 1
for(i in 2:30){
  ratio[i] <- fib[i]/fib[i-1]
}
```

Yes, they seem to converge to the value 1.618034.

(b)

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

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

The value is computed as 1.618034, and this is the same value with the value in (a), the value that the ratio converges to. This can be proved as following:

$\frac{F_{n+1}}{F_n} = \frac{F_n+F_{n-1}}{F_n} = 1 + \frac{F_{n-1}}{F_n}$ Let $x_n = \frac{F_{n+1}}{F_n}$. Then, $x_n = 1 + \frac{1}{x_{n-1}}$. We can also notice that $1 \leq x_n \leq 2$. As $n \to \infty$, $x^* = 1 + \frac{1}{x^*}$. This equation can be written as $(x^*)^2 - x^* - 1 = 0$ and the solution for this equation is the golden ratio, $\frac{1+\sqrt{5}}{2}$, and only this can be the solution for the above equation since the solution needs to be positive.

4.1.1, exercise 3

(a)

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

The answer is 15, which is computed by $\sum_{i=1}^{5} i$.

(b)

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

This gives the vector: 1 2 3 4 5

(c)

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

This yields the vector: 0 1 2 3 4 5 The difference between (b) is that in (b), 'answer' was started as NULL, so nothing is there. However, in (c), 'answer' was started from the real value 0, so it includes this value.

(d)

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

This gives the value 120, and this answer is computed as $120 = 1 * 2 * 3 * 4 * 5$.

(e)

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

This gives the vector of remainders after divided by 31; 3 21 23 6 11 15 12 22 30 24 13 29 17 26 27 3

4.1.2, exercise 4

```
gic <- function(n, P){
  if(n<=3){i=0.04} else {i=0.05}
  I <- P*((1+i)^n-1)
  return(I)
}
```

Using this gic function, if we let P=1000 and n=5, then, the I will be

```
gic(5, 1000)
```

```
## [1] 276.2816
```

4.1.2, exercises 5

```
mortgage <- function(n, P, open){
  if(open==TRUE) {i=0.005} else {i=0.004}
```

```
  R=(P*i)/(1-(1+i)^(-n))
  return(R)
}

mortgage(2,100, open = T)
```

```
## [1] 50.37531
```

4.1.3, exercise 2

```
Fibonacci <- c(1,1)
while(max(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
```

4.1.3, exercise 4

```
i <- 0
i0 <- 0.006
while(abs(i-i0)>=0.000001){
  i <- i0
  i0 <- (1-(1+i)^(-20))/19
}
print(i)
```

```
## [1] 0.004955135
```

4.1.3, exercise 5

```
i <- 0
i0 <- 0.006
iter <- 0
while(abs(i-i0)>=0.000001){
  i <- i0
  i0 <- (1-(1+i)^(-20))/19
  iter <- iter+1
}
print(iter)
```

```
## [1] 74
```

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)

The Eratosthenes function gives all the prime numbers less than n. It starts with eliminating all the even numbers except 2, which is the multiples of 2.

(b)

Let's say that $n = a * b$, where both $a, b \neq n$. If this n has a prime factor greater than $\sqrt{n}$, then it should have a prime factor less than $\sqrt{n}$. This is because if both $a \geq \sqrt{n}$ and $b \geq \sqrt{n}$, then $a * b > n$. Thus, if we cross out the multiples of all the numbers less or equal to $\sqrt{n}$, all the multiples of those crossed out numbers will be eliminated also. Therefore, once $p \geq \sqrt{n}$, all remaining entries in sieve are prime.

(c) ***

```
Eratosthenes_break <- function(n) {
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    repeat{
    if (length(sieve) == 0) break
      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.")
  }
}
```

4.2.1, exercise 2

(a)

```
compound.interest <- function(P, i.r, n){
  return(P*(1+i.r)^n)
}
```

(b)

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

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

4.2.1, exercise 3

```
bisect0 <- function(f, val1, val2, tol=0.00001) {
  repeat {
    f1 <- f(val1)
    f2 <- f(val2)
    val3 <- (val1+val2)/2
    f3 <- f(val3)
    if(val3==0){
      print(val3)
      break
    } else {
      if(sign(f1)==sign(f2)) {
      val1 <- val3
    } else { val2 <- val3 }
    }
    if(abs(val1-val2)<tol){
      print((val1+val2)/2)
      break
    }
  }
}
```

Thus, if we want to look for the function f, we can make a function for it, such as

```
f<- function(x){x^2-2*x+1}
bisect0(f, 0,3,0.00001)
```

```
## [1] 2.999997
```

4.4.1, exercise 1

```
mergesort <- function(x, dec=FALSE){
  if(dec==FALSE){
  len <- length(x)
```

```
  if(len<2) result<-x
  else{
    y <- x[1:(len/2)]
    z <- x[(len/2+1):len]
    y <- mergesort(y)
    z <- mergesort(z)
    result <- c()
    while(min(length(y), length(z))>0){
      if(y[1]<z[1]){
        result <- c(result, y[1])
        y <- y[-1]
      } else {
        result <- c(result, z[1])
        z <- z[-1]
      }
    }
    if(length(y)>0) {result <- c(result, y)}
    else {result <- c(result, z)}
  }
  return(result)
  }
  else {
  len <- length(x)
  if(len<2) result<-x
  else{
    y <- x[1:(len/2)]
    z <- x[(len/2+1):len]
    y <- mergesort(y, dec=TRUE)
    z <- mergesort(z, dec=TRUE)
    result <- c()
    while(min(length(y), length(z))>0){
      if(y[1]>z[1]){
        result <- c(result, y[1])
        y <- y[-1]
      } else {
        result <- c(result, z[1])
        z <- z[-1]
      }
    }
    if(length(y)>0) {result <- c(result, y)}
    else {result <- c(result, z)}
  }
  return(result)
  }
}
```

4.4.1, exercise 2

(a)

```
newton <- function(f, g, x0, y0, tol=0.00001, n=100){
  install.packages("numDeriv", repos='http://cran.us.r-project.org')
  library(numDeriv)

  x <- c(); y <- c()
  x[1] <- x0; y[1] <- y0
  for(i in 2:n){
  u <- f(c(x[i-1], y[i-1]))
  v <- g(c(x[i-1], y[i-1]))
  q <- grad(f, c(x[i-1], y[i-1]))[1]
  r <- grad(f, c(x[i-1], y[i-1]))[2]
  s <- grad(g, c(x[i-1], y[i-1]))[1]
  t <- grad(g, c(x[i-1], y[i-1]))[2]
  d <- q*t-r*s
  x[i] <- x[i-1]-(u*t-v*r)/d
  y[i] <- y[i-1]-(q*v-s*u)/d

  if(abs(x[i]-x[i-1]) < tol & abs(y[i]-y[i-1]) < tol){
    return(c(x[i], y[i]))
  }
  }
}
```

(b)

```
func <- function(a){a[1]+a[2]}
gfun <- function(b){b[1]^2+2*b[2]^2-2}
x0 <- 0.5; y0 <- -0.5
newton(func, gfun, x0, y0, tol = 0.00001, n=100)
```

```
##
## The downloaded binary packages are in
##  /var/folders/nv/29kjp8095_n_2zxvx871q1mc0000gn/T//RtmpaXXcZ5/downloaded_packages
```

```
## [1]  0.8164966 -0.8164966
```

Chapter 4 exercise 1

```
directpoly <- function(x, c){
  n <- length(c)
  result <- 0
  for(i in 1:n){
    result <- result+c[i]*x^(i-1)
  }
  return(result)
}
```

Chapter 4 exercise 2

```
hornerpoly <- function(x, c){
  n <- length(c)
  a <- matrix(0, length(x), n)
  a[,n] <- c[n]
  for(i in (n-1):1){
    a[,i] <- a[,i+1]*x+c[i]
  }
  return(a[,1])
}
```

Chapter 4 exercise 3

(a)

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

```
##    user  system elapsed
##   1.324   0.146   1.518
```

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

```
##    user  system elapsed
##   0.722   0.442   1.420
```

User time is a bit shorter from the hornerpoly function, but system time and total elapsed time are longer in the hornerpoly function.

(b)

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

```
##    user  system elapsed
##   0.243   0.035   0.280
```

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

```
##    user  system elapsed
##   0.210   0.056   0.271
```

If the coefficients get smaller, the hornerpoly function seems more efficient than the directpoly function, because for all user, system and elapsed time, the hornerpoly function takes shorter time than the another one.