assignment-1-cardinalbraxiatel

Patrick Walker

October 13, 2018

all my problems are from the 2007 edition.

4.1.1.2.a

```
Fibonacci <- numeric(1000000)
ratio <- numeric(1000000)
ratio[1] <- NaN
ratio[2] <- 1
Fibonacci[1] <- 1
Fibonacci[2] <- 1
for(i in 3:1000){
  Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i-1]
  ratio[i] <- Fibonacci[i]/Fibonacci[i-1]
}
return(head(Fibonacci))
```

```
## [1] 1 1 2 3 5 8
```

The sequence appears to converge quite quickly #4.1.1.2.b

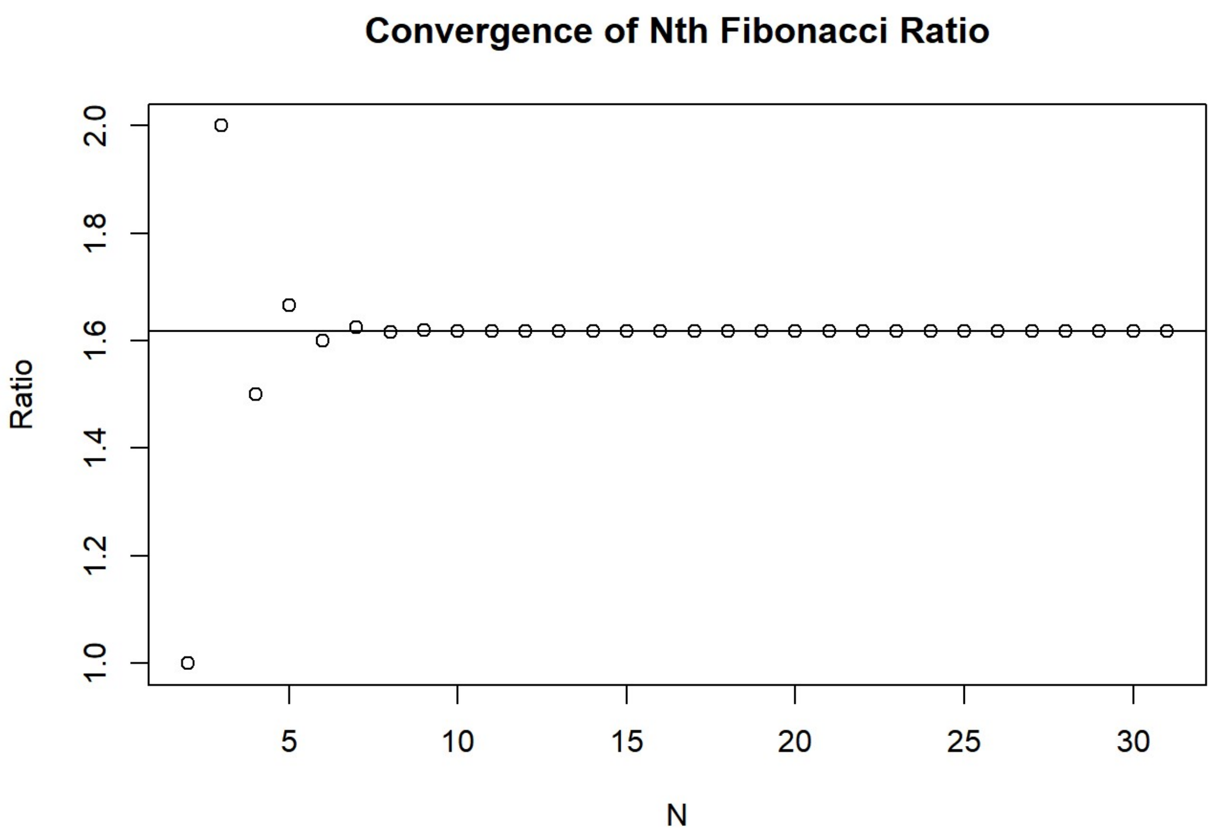
```
GR <- (1 + sqrt(5))/2



for(j in 3:length(Fibonacci)){
  Fibonacci[j] <- Fibonacci[j - 2] + Fibonacci[j-1]
  ratio[j] <- Fibonacci[j]/Fibonacci[j-1]
  if(ratio[j]  == GR){print(j)
    break}}
```

```
## [1] 41
```

```
  plot(2:31, ratio[2:31], main = "Convergence of Nth Fibonacci Ratio", xlab = "N", yla
b = "Ratio")
  abline(h = GR)
```



Although not quite a proof, by Johannes Kepler's proof that the limit of this ratio at the nth observation (as n goes to infinity) the ratio will converge to the golden ratio. According to R, the floating point differences between the ratio and the GR vanishes and the series essentially converges. The line below indicates the Golden Ratio.

4.1.1.3.a

This code is the equivalent of the sum function, starting from 0. So 0 + 1 + 2 + 3 + 4 + 5 = 15. This is borne out by the code.

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

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

4.1.1.3.b

This code first assigns an empty vector to answer. Each time through the loop the c() argument combines the vector answer from the last iteration with whatever the current value of j is. This simply leads to a vector from 1:5, albeit in a roundabout manner.

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

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

4.1.1.3.c

This code is similar to the last one save that it sets the answer variable to zero rather than to the empty set. This means that the output vector is 0 to 5.

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

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

4.1.1.3.d

This code will output a single value which is the equivalent of 5! = 120

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

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

4.1.1.4.e

This code will output a vector that combines the previous answer vector whose first value is 3 and whose 2:15 elements represent the 7*answer[j] mod 31. I do not believe that I would be able to predict the sequence of numbers in a finite amount of time. Perhaps if I had the use of a very clever computer.

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

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

4.1.2.4

```
calc_interest <- function(n = 3, P = 1000){

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

calc_interest(n = 10, P = 100)
```

```
## [1] 62.88946
```

4.1.2.5

```
cal_mort <- function(n = 36, P = 250000, open = TRUE){

if(open == TRUE){i <- 0.005} else {i <- 0.004}

R <- (P*i)/(1-(1+i)^(-n))

return(R)

}

cal_mort()
```

```
## [1] 7605.484
```

4.1.3.2

```
while_loop <- function(){

  Fibonacci <- c(1,1)

  while(max(Fibonacci) < 300){

    Fibonacci <- c(Fibonacci, Fibonacci[length(Fibonacci)] + Fibonacci[length(Fibonacc
i)-1])
  }
  print((Fibonacci))

  return(length(Fibonacci))

}

while_loop()
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

```
## [1] 14
```

4.1.3.4

```
while_loop_2 <- function(){

  oldi <- 0
  i <- 0.006

  while(abs(i - oldi) > 0.000001){

    oldi <- i

    i <- (1-(1+oldi)^(-20))/19

  }
  return(i)

}
while_loop_2()
```

```
## [1] 0.004954139
```

4.1.3.5

```r
while_loop_3 <- function(){

  oldi <- 0
  i <- 0.006
  numb <- 0
  while(abs(i - oldi) > 0.000001){

    oldi <- i

    i <- (1-(1+oldi)^(-20))/19


    numb <- numb + 1
  }
  print(i)

  return(numb)
}
while_loop_3()
```

```
## [1] 0.004954139
```

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

4.1.5.2

The if statment, along with its accompanying else statment, prevents trivial values of n being used. Sieve is a vector containing all the numbers from 2 through n. Primes starts as an empty vector. The while loop is based on whether or not the length of the sieve vector is still positive. The first entry of sieve is used as p, a test value, which for the first iteration is 2. p is then combined into the previous primes vector. The next line updates sieve by indexing sieve by those entries of sieve that don’t have p as a factor. If this process is repeated ad infinitum, all those numbers which are prime will be transfered from sieve into primes, resulting in a zero length sieve, which ends the loop. The function ends by returning the primes vector.

```r
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.")
       }

}

Eratosthenes(100)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
## [24] 89 97
```

4.1.5.2.b

The below function is esssentially the same but p takes an initial value outside the loop, the while loop is based on whether or not p < sqrt(n), and that to return all the prime numbers the code combines those prime numbers less than sqrt(n) along with those remaining values of sieve. This results in the same output as the previous code.

```r
Eratosthenes2 <- function(n) {
  # Print prime numbers up to n (based on the sieve of Eratosthenes)
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    p <- 2

    while (p < sqrt(n)) {


      primes <- c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]
      p <- sieve[1]

    }
    combined <- c(primes,sieve)
    return(combined)
  } else {
    stop("Input value of n should be at least 2.")
  }

}
Eratosthenes2(100)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
## [24] 89 97
```

4.1.5.2.c

This function functions very similar to the code from part a, the primary modification is that it check for p >= sqrt(n) and if its true it breaks the while loop. The primes less than sqrt(n) and the remaining entries of sieve are combined together and returned, again equivalent to the last two sets of code.

```
Eratosthenes3 <- 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]

      if(p >= sqrt(n)){break}

      primes <- c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]


    }
    combined <- c(primes,sieve)
    return(combined)
  } else {
    stop("Input value of n should be at least 2.")
  }

}

Eratosthenes3(100)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
## [24] 89 97
```

4.2.1.2.a_and_b

```
compound.interest <- function(P = 1000, i.r = 1, n = 30){

  i.r <- i.r/100

  acc_cap <- P*(1 + i.r)^n

  return(paste("$", acc_cap))
}
compound.interest()
```

```
## [1] "$ 1347.84891533291"
```

4.2.1.3

```r
func <- function(x){x-1}
bi <- function(l = -1,h = 3 ,x = seq(from = l, to = h)){

 if(func(h) > 0){(hbound <- h) && (lbound <- l)} else {(lbound <- h) && (hbound <- l)}

  if(func(l)*func(h) >= 0){return("F(x)s must have differing signs")}else{
    mfunc <- 1
        while(abs(mfunc) > 0.0000001){

      mid <- (hbound-lbound)/2



      mfunc <- func(mid)



      if(mfunc > 0){hbound <- mid} else {lbound <- mid}


    }
    return(mid)
  }

}

bi()
```

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

4.4.1.1

```
# 1. Use a merge sort to sort a vector
  mergesort <- function (x, descending = 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 en
d
                    # 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(descending == TRUE){
        for(i in 1:length(result)){
          y[i] <- result[length(result) - i + 1]
        }
       result <- y
      }
      return(result)
      }
```

4.4.1.2

x + y = 0 x^2 +2y^2 - 2 = 0 subbing 1 into 2 gives 3y^2 = 2 y=sqrt(2/3) = -x sqrt(2/3) = 0.8164966

```r
f <- function(x,y){x+y}            #function 1 requires use input of function
fe <- expression(x+y)              #fe uses the same function as above, but requires c
onversion
g <- function(x,y){x^2 + 2*y^2 -2} #similar to above but for g
ge <- expression(x^2 + 2*y^2 -2)

Newton <- function(){
  xv <- c(1,1)                    #creates vectors for x and y
  yv <- c(1,1)


  fdx <- D(fe,"x")               # takes derivatives of above according to Newton's me
thod
  fdy <- D(fe, "y")
  gdx <- D(ge, "x")
  gdy <- D(ge, "y")

iff <- abs(f(xv[1],yv[1])) + abs(g(xv[1],yv[1])) # variable that will be used in whil
e loop

  i <- 2                                      #counter for vectors since this is a whi
le loop

  while((iff) > 0.00001){

    x <- xv[i-1]                #creates useful values for x and y, i-1th entry

    y <- yv[i-1]


    fxn1 <- eval(fdx)          #evaluates above derivatives using x and y from above

    fyn1 <- eval(fdy)

    gxn1 <- eval(gdx)

    gyn1 <- eval(gdy)


    fn_1 <- f(xv[i-1],yv[i-1])    # various functions evaluated at i-1
    gn_1 <- g(xv[i-1],yv[i-1])
    dn_1 = fxn1*gyn1 - fyn1*gxn1

    xv[i] <- xv[i-1]-(gyn1*fn_1 - fyn1*gn_1)/dn_1 #updates xv and yv
    yv[i] <- yv[i-1]-(fxn1*gn_1 - gxn1*fn_1)/dn_1

    iff <- abs(f(xv[i],yv[i])) + abs(g(xv[i],yv[i]))  #updates iff
    i <- i + 1                                      #updates i

    if(i == 3000){break}                           #Emergency bailout to prevent infinite l
oop
  }
  print(xv[length(xv)])
  return(yv[length(yv)])

}
Newton()
```

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

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

CH4.1

```
directpoly <- function(x = 2, coe = c(1,2,3)){
  n <- length(coe)
  powers <- c(0:(n-1))
  temp <- c()
  for(i in 1:n){
  temp[i] <- coe[i]*x^(powers[i])
  }

  return(sum(temp))
}
directpoly()
```

```
## [1] 17
```

CH4.2

```r
hornerpoly <- function(x = c(2,3), coe = c(1,2,3)){
  n <- length(coe)
  a <- matrix(nrow = n,ncol = length(x))
  a[n,] <- coe[n]
  print(a[n,])
  for(z in 1:length(x)){
  for(j in 1:(n-1)){
    a[n-j,z] <- x[z]*a[n-j+1,z] + coe[n-j]
  }

  }


  a1s <- (a[1,])
  return(a1s)
}
```

4.4.1.3.a

The system elapsed time is 0.1 for the directpoly function while the horner function used 0.06. The latter is faster.

4.4.1.3.b

The notation to run this has coe = c(-3,17,2) The system elapsed time is 0.03 for the directpoly function while the horner function is 0.05. This time the former is faster, suggesting that we should determine the length of our polynomials before running a function.