# Final_Project

Pin-wei, Yu

A082021

6/22/2020

## Problem 1. Goodness-of-Fit and Bootstrap

Given the following data:

$$6,7,3,4,7,3,7,2,6,3,7,8,2,1,3,5,8,7.$$

We want to know whether this data is coming from a binomial distribution with parameters $(8, p)$, where $p \in [0,1]$ is unknown.

```
data <- sort(c(6,7,3,4,7,3,7,2,6,3,7,8,2,1,3,5,8,7))
```

### (a) (5%): Compute the corresponding Kolmogorov-Smirnov statistics.

```
Table_data <- table(data)
Names <- names(Table_data)
L <- length(Names)
p <-mean(data)/8
D <-c()
for(i in c(1:L)){
  e <- length(which(data <=Names[i]))/length(data)
  t <- pbinom(i,8,p)
  d <- abs(e-t)
  D<-c(D,d)

}
Kolmogorov_Smirnov_statistics <- max(D)
cat("The corresponding Kolmogorov-Smirnov statistics is :",Kolmogorov_Smirnov_statistics)

## The corresponding Kolmogorov-Smirnov statistics is : 0.262332
```

### (b) (10%): Write down the algorithm of approximating the p-value of this Kolmogorov-Smirnov statistics based on bootstrap with 10^4 sample.

#### Algorithm

Step1: Set $B = 10000$ (Times we bootstrap) ; calculate $\hat{p}$ for the given data.

Step2: Set $Y_i$ follow Binomial(n, $\hat{p}$ ) i.i.d. , $i = 1,2, \ldots, n$, where n is the length of the given data.

Step3: Let $\hat{p}_B = \frac{\sum_{i=1}^{n} Y_i}{n}$, Set $Y_i \sim$ Binomial(n, $\hat{p}_B$) , $CDF$:F(x)

Step4: Set $D_B = Maximum_x |F_e(x) - F(x)|$, where$F_e(x) = \frac{\#i{:}Y_i \leq x}{n}$

Step5: Repeat Step2~Step4 B times

Step6: Calculate the frequency of $D_B$ bigger than D in 1(a)

### (c) (5%): Write a program based on your algorithm in (b). Determine the result of the hypothesis testing.

```
B = 10000
Kolmogorov_Smirnov_statistics_bootstrap <- c()
for(i in c(1:B)){
  # data_bootstrap <- sample(data,length(data),replace = TRUE)
  data_bootstrap <- rbinom(length(data),8,mean(data)/8)
  p_bootstrap <- mean(data_bootstrap)/8
  Table_data_bootstrap <- table(data_bootstrap)
  Names_bootstrap <- as.numeric(names(Table_data_bootstrap))
  L_bootstrap <- length(Names_bootstrap)
  D <-c()
  for(i in c(1:L_bootstrap)){
    e <- length(which(data_bootstrap <= Names_bootstrap[i]))/length(data_bootstrap)
    t <- pbinom(Names_bootstrap[i],8,p_bootstrap)
    d <- abs(e-t)
    D<-c(D,d)
  }
  K_S_s <- max(D)
  Kolmogorov_Smirnov_statistics_bootstrap <- c(Kolmogorov_Smirnov_statistics_bootstrap,K_
S_s)
}
sum(Kolmogorov_Smirnov_statistics_bootstrap > Kolmogorov_Smirnov_statistics) / B

## [1] 1e-04
```

In the hypothesis testing, we set $\alpha = 0.05$.

$H_0$:The given data follows $Binomial(8, p)$.

```
> sum(Kolmogorov_Smirnov_statistics_bootstrap > Kolmogorov_Smirnov_statistics) / B
[1] 1e-04
```

Thus, we reject $H_0$.

## Problem 2. Hastings-Metropolis Algorithm

### (a) (5%): Write down the MCMC algorithm to sample μ from (1).

Step1: Initial $\mu_1$, set $n = 1$

Step2: Generate v $\sim$ q( u , v ) $with$= U_n ; Set $q$: $Unif$(0,2), where $q$ is the proposal with Independence Metropolis-Hastings Algorithm.

Step3: Let$\alpha_{(\mu,v)} = min(\frac{h(\mu)}{h(v)}, 1)$

Step4: Generate $k \sim Unif(0,1)$

Step5: Set $U_{n+1} = v$, if $k \leq \alpha_{(\mu,v)}$ ; Set $U_{n+1} = U_n$, if $k \geq \alpha_{(\mu,v)}$

Step6: Set $n = n + 1$

Step7: Repeat Step2~Step6 N times to get one sample point $U_N$ approximately.

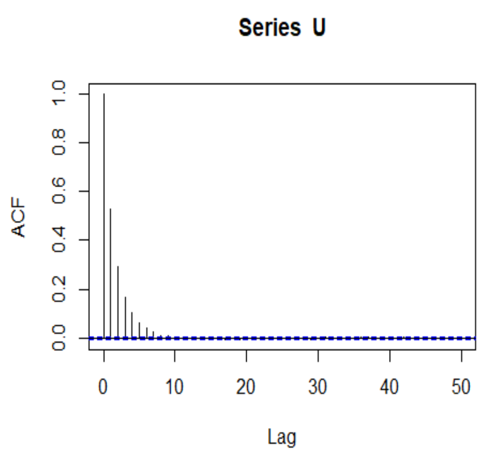
### (b) (10%): Based on your algorithm in a), sample 10^4 µ from (1). Report the histogram, the mean and the standard error of your sampled µ.

#### Make the likelihood function

```
data<-c(0.8605 ,1.2175, -0.9772 ,-0.0378 ,2.9478,-0.2710 ,0.0380, 1.1110 ,2.4136, 0.2516,
 0.3485, 0.6765 ,2.7070, 0.5617,1.0066, 2.3637 ,1.4502, 1.6041 ,1.2023 ,1.6049)
f_x <- function(x){
  return((2*pi)^(-0.5)*exp(-(x)^2/2))
}
h_u <- function(u,X_list){
  h <- 1
  for(i in c(1:length(data))){
    k <- f_x(X_list[i]-u)
    h<- h*k
  }
  h = h*f_x(u)
  return(h)
}
```

#### Main programming

```
n <- 1
N <- 100000
U<-c(rep(0,N+1))
U[1]<- mean(data)
while(n<N+1){
  v <- runif(1,0,2)
  alpha <- min(h_u(v,data)/h_u(U[n],data),1)
  K <- runif(1,0,1)
  if(K <= alpha){
    U[n+1]=v
  }else{
    U[n+1]=U[n]
  }
  n <-n+1
}

acf(U)
```

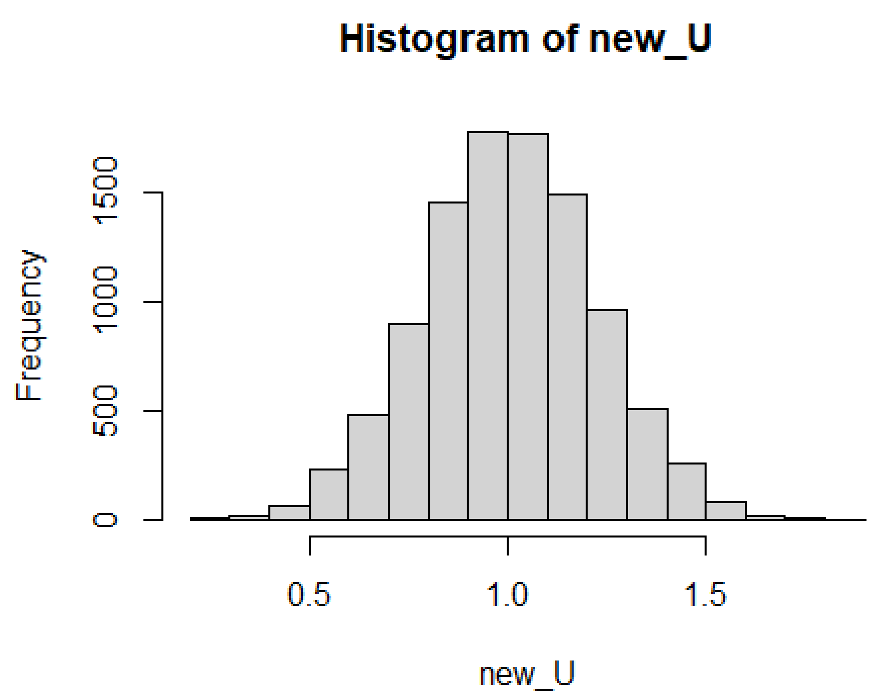



Thus, we take only 1 point when the 10 points generated.

```r
new_U <- rep(0,10000)
for(n in c(1:10000)){
  new_U[n] = U[10*n+1]
}
```

### Histogram, mean and the standard error of samplings

```r
hist(new_U)
```




```r
se<-sqrt(var(new_U)/length(new_U))

cat("The mean of sampled mu is :",mean(new_U))

## The mean of sampled mu is : 1.006089

cat("The standard error of sampled mu is :", se)

## The standard error of sampled mu is : 0.002173192

length(new_U)

## [1] 10000
```

## (c) (5%): Check the dependency of your sampled μ to make sure that they are i.i.d. samples.

```r
Box.test(new_U)

## 
##  Box-Pierce test
## 
## data:  new_U
## X-squared = 1.8143, df = 1, p-value = 0.178
```

In Box.test, the null hypothesis is that the series are independence. Set $\alpha = 0.05$.

The p-value in our data is such big. Thus, we accept $H_0$. The series are independent.

## Problem 3. Gibbs Sampler

Suppose that for random variables $X, Y, N$,

$$P\{X=i, y \leq Y \leq y+dy, N=n\} \propto C_i^n y^{i+\alpha-1}(1-y)^{n-i+\beta-1} e^{-\lambda} \frac{\lambda^n}{n!} dy$$

where $n \in \mathbb{N}, i \in \{0, \cdots, n\}, y \geq 0$, and $\alpha, \beta, \gamma$ are constants.

### （a) (10%): Derive the conditional distribution of X given (Y, N ), Y given (X, N ), and N given (X, Y )

$$P(X=i|y \leq Y \leq dy, N=n) \propto C_i^n y^i (1-y)^{n-i} \propto binom(n, y)$$

$$P(y \leq Y \leq dy|X=i, N=n) \propto y^{i+\alpha-1}(1-y)^{n-i+\beta-1}dy \propto Beta(i+\alpha, n-i+\beta)$$

$$P(N=n|X=i, y \leq Y \leq dy) \propto C_i^n \frac{[\lambda(1-y)]^n}{n!} = \frac{[\lambda(1-y)]^n}{(n-i)!}$$

### (b) (5%):Based on a),write down the Gibbs sampler algorithm to sample(X,Y,N))

Step1: Initial $X_0 = (x_0, y_0, n_0)$

Step2: Generate

$$X_t \sim P(X|Y, N)$$

$$Y_t \sim P(Y|X, N)$$

$$N_t \sim P(N|X, Y)$$

$$t = 1,2, \dots.$$

#### Function Setting

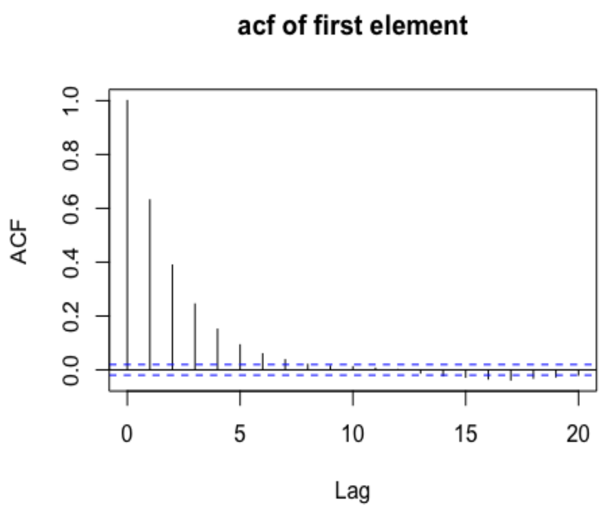
In the simulation, we set $\alpha = \beta = \lambda = 5$

```
Gibbs = function(r.trans,nsimu,x0){
  x = matrix(0,nrow = length(x0), ncol = nsimu)
  xc = x0
  for(i in 1:nsimu){
    xp = r.trans(xc)
    xc = xp
    x[,i] = xc
  }
  return(x)
}
#Set alpha=beta=lambda=5
trans2 = function(xsamp){ #xsamp = (i,y,n)
  i = xsamp[1]
  y = xsamp[2]
  n = xsamp[3]
  alpha = 5
  beta = 5
  lambda = 5
  i = rbinom(1, n, y) # n>=i
```

```
  y = rbeta(1, i+alpha, n-i+beta)
  n = rpois(1,(1-y)*lambda)+i
  return(c(i, y, n))
}
```

### Main Programming

```
set.seed(35)
nsimu = 10^4
x0 = c(1,0.5,2)
result = Gibbs(trans2, nsimu, x0)
#plot(1:10000, result[2,1:10000])

acf(result[1,],lag.max = 20,main = "acf of first element")
```




```
X = result[1,seq(from=1000,to=nsimu,by=20)]
Y = result[2,seq(from=1000,to=nsimu,by=20)]
N = result[3,seq(from=1000,to=nsimu,by=20)]
```

### (c) (5%): Use the algorithm in b) to simulated 104 pairs of (X, Y, N). Report EX, EY and EN

```
mean(X)

## [1] 2.43459

mean(Y)

## [1] 0.4888099

mean(N)

## [1] 5.077605
```

## Problem 4. Simulated Annealing

Consider a traveling salesman problem in which the salesman starts at city 0 and must travel in turn to each of the 10 cities 1, · · · , 10 according to some permutation of 1, · · · , 10. Let the reward earned by the salesman when he goes directly from city i to city j be Ui,j

### (a) (5%):Write down the simulated annealing algorithm for finding the maximum of the salesman's reward

Step1: Initial $X_0$,and $x_{0t}$ is the t-th city visited, $t = 1,2,...,10$

Step2: Let $\lambda_n = log(1+n)$, set $n = 0$

Step3: Generate $y$ as random permutation from 1～10

Step4: Set $\alpha_n = min(\frac{(1+n)^{V(y)}}{(1+n)^{V(X_n)}}, 1)$, where $V(X_n)$ is the reward from the n-th cities-visted order.

Step5: Set $X_{n+1} = y$ with probability $\alpha_n$, and Set $X_{n+1} = X_n$ with probability $1 - \alpha_n$

Step6: Repeat Step3~Step5 until convergence.

### (b) (5%): Generate 100 random numbers U0,k,k = 1,··· ,10,Ui,j,i not equal to j, i,j = 1,··· ,10.

```
U_0_k <- runif(10,0,1)
Reward_Matirx <- matrix(0,nrow = 10, ncol = 10) #row index is the start city
for(i in c(1:10)){
  for(j in c(1:10)){
    if(i!=j){
      Reward_Matirx[i,j]=runif(1,0,1)
    }
  }
}
print(Reward_Matirx)

##            [,1]        [,2]      [,3]      [,4]      [,5]       [,6]      [,7]
##  [1,] 0.0000000 0.240105873 0.3897836 0.8923836 0.7147038 0.20823068 0.5202180
##  [2,] 0.6352739 0.000000000 0.4419182 0.8918153 0.7956574 0.41394950 0.9584327
##  [3,] 0.7623370 0.584046909 0.0000000 0.3816479 0.1237732 0.85111591 0.5215559
##  [4,] 0.6881411 0.420394934 0.7598632 0.0000000 0.6983119 0.23371109 0.8962232
##  [5,] 0.1227167 0.083698788 0.9673856 0.5781007 0.0000000 0.06096208 0.1697567
##  [6,] 0.6264305 0.448143804 0.2301667 0.9878745 0.6290218 0.00000000 0.5149164
##  [7,] 0.2920446 0.001369312 0.3255674 0.1063150 0.2689678 0.68282124 0.0000000
##  [8,] 0.1378250 0.351119312 0.9537526 0.8713562 0.8080511 0.87857758 0.9160856
##  [9,] 0.9348439 0.551942663 0.4541713 0.7348359 0.5867169 0.74730962 0.4997243
## [10,] 0.8531985 0.622953833 0.3444602 0.6993650 0.5179730 0.78976385 0.3715065
##            [,8]       [,9]      [,10]
##  [1,] 0.9800454 0.86902400 0.77778751
##  [2,] 0.6046278 0.97946566 0.03283878
##  [3,] 0.4680437 0.06426993 0.71478779
##  [4,] 0.3918316 0.51860239 0.74017851
##  [5,] 0.8392268 0.49885639 0.15842853
##  [6,] 0.5650888 0.56258735 0.75176007
##  [7,] 0.3978356 0.29517195 0.17645552
##  [8,] 0.0000000 0.86292897 0.22589290
##  [9,] 0.8584717 0.00000000 0.19455651
## [10,] 0.9668830 0.06292634 0.00000000
```

**(c) (5%): Based on your algorithm in a) and the sampled Ui,j in b), do a simulated annealing. Report the maximum reward and the corresponding order of cities that the salesman should travel.**

### Function Setting

```
library(doParallel)

## Loading required package: foreach

## Warning: package 'foreach' was built under R version 3.6.2

## Loading required package: iterators

## Loading required package: parallel

library(foreach)
library(doSNOW)

## Loading required package: snow

##
## Attaching package: 'snow'

## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, clusterSplit, makeCluster, parApply,
##     parCapply, parLapply, parRapply, parSapply, splitIndices,
##     stopCluster

library(pracma)
#cpu.cores <- detectCores()
#cl <- makeCluster(4)
#registerDoParallel(cl)


set.seed(35)
Calculate_Reward <- function(initial_reward , reward_matrix, city_sequence){
  reward <- initial_reward[city_sequence[1]]
  for(i in c(1:9)){
    r <- reward_matrix[city_sequence[i],city_sequence[i+1]]
    reward <- reward + r
  }
  return(reward)
}


#B is the Markov-Chain length
Simulated_Annealing <- function(B,U_0_k,Reward_Matirx){
  X_0 <- randperm(10, 10)
  n <- 0
  anealing_reward <- c()
  cities_order <- matrix(numeric(10*B),nrow = 10 ,ncol = B)
  while(n<B){
    X_n <- X_0
    y <-randperm(10, 10)
```

```r
    V_x_n <- Calculate_Reward(U_0_k,Reward_Matirx,X_n)
    V_y <- Calculate_Reward(U_0_k,Reward_Matirx,y)
    alpha<-min((1+n)^(V_y)/(1+n)^(V_x_n),1)
    if(runif(1,0,1)<alpha){
      X_n <- y
    }
    anealing_reward <- c(anealing_reward,V_y)
    cities_order[,n] <- y
    n <- n+1
  }
  Maximum_reward <- max(anealing_reward)
  Corresponding_cities_order <- cities_order[,which(anealing_reward == Maximum_reward)[1]]
  List <- c(Maximum_reward,Corresponding_cities_order)
  return(List)
}
```

### Main Programming

```r
Reward_order <- Simulated_Annealing(10000,U_0_k,Reward_Matirx)
cat("The Maximum Reward is:",Reward_order[1])

## The Maximum Reward is: 8.247418

cat("The Optimal Cities-Visited-Order is:",Reward_order[2:11])

## The Optimal Cities-Visited-Order is: 5 10 4 8 1 9 2 7 6 3
```

## (d) (5%): Repeat b) and c) for 106 times, and report the mean and variance for the maximum reward.

Programming:

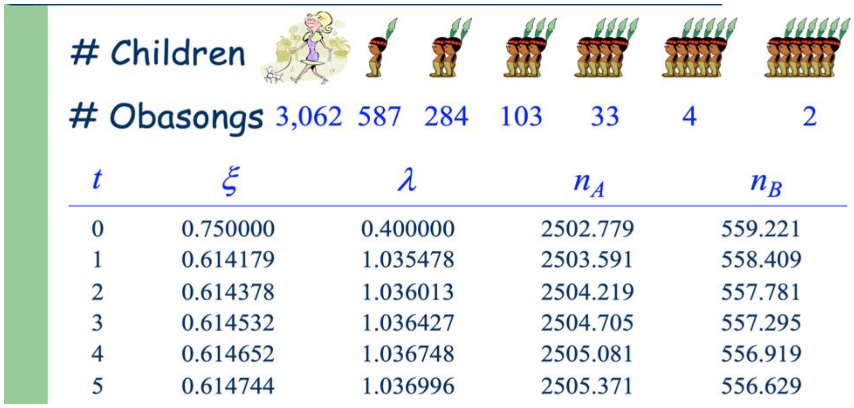
```r
i <- 1
R<- c(numeric(1000000))
while(i<1000001){
  R[i]<-Simulated_Annealing(1000,U_0_k,Reward_Matirx)[1]
  print(i)
  i<-i+1

}
```

Output:

```
 [ reached getOption("max.print") -- omitted 999000 entries ]
> length(R)
[1] 1000000
> Mean_Maximum_Reward <- mean(R)
> Variance_Maximum_Reward <-var(R)
> cat("The Mean of Maximum Reward is:",Mean_Maximum_Reward)
The Mean of Maximum Reward is: 7.973674
> cat("The Variance of Maximum Reward is:",Variance_Maximum_Reward)
The Variance of Maximum Reward is: 0.04418592
>
```

# Problem 5. EM Algorithm

Consider the binomial/Poison mixture problem in the slide of Week 12-1, page 26-38. The data is given in page 38.



| # Children | | | | | | | |
|---|---|---|---|---|---|---|---|
| # Obasongs | 3,062 | 587 | 284 | 103 | 33 | 4 | 2 |

| $t$ | $\xi$ | $\lambda$ | $n_A$ | $n_B$ |
|---|---|---|---|---|
| 0 | 0.750000 | 0.400000 | 2502.779 | 559.221 |
| 1 | 0.614179 | 1.035478 | 2503.591 | 558.409 |
| 2 | 0.614378 | 1.036013 | 2504.219 | 557.781 |
| 3 | 0.614532 | 1.036427 | 2504.705 | 557.295 |
| 4 | 0.614652 | 1.036748 | 2505.081 | 556.919 |
| 5 | 0.614744 | 1.036996 | 2505.371 | 556.629 |

## (a) (10%): Write down the EM algorithm for this problem.

Step1: Initial $\xi^{(0)}$& $\lambda^{(0)}$

Step2: Set $n_A^{(t)}$=$E_{\xi(t),\lambda_{(t)}}$ $[n_A|n_{obs}]$= $\frac{n_0\xi^{(t)}}{N\xi^{(t)}+(1-\xi^{(t)})\exp(-\lambda_{(t)})}$ , where $n_0 = n_A + n_B$

Step3: Set

$$\xi^{(t+1)} = \frac{n_A^{(t)}}{N}$$

$$\lambda^{(t+1)} = \frac{\sum_{x=1}^{6} x\, n_x^{(t)}}{N - n_A^{(t)}}$$

, where

$$n_x^{(t)} = N\frac{\lambda_{(t)}^{x}\exp(-\lambda_{(t)})}{x!}(1-\xi_{(t)}),\; x = 1 \sim 6$$

### Function Setting

```
Updating_n_A <- function(xi,lambda,n0){
  n_a<- (n0*xi)/(xi+(1-xi)*exp(-lambda))
  return(n_a)
}

Updating_Xi <- function(n_A,N){
  X<- n_A/N
  return(X)
}

Updating_Lambda <- function(N,na,total_children){
  L <- total_children/(N-na)
  return(L)
}
Obasongs <- c(3062,587,284,103,33,4,2)
n_0<- Obasongs[1]
N <- sum(Obasongs)
sum_of_children <- sum(c(0,1,2,3,4,5,6)*Obasongs)
```

### (b) (5%): Reconstruct the table in page 38 by setting the initial as ξ0 = 0.75 and λ0 = 0.4.

```
B <- 6

Xi_List <- c(numeric(B))
Xi_List[1]<- 0.75
Lambda_List <-c(numeric(B))
Lambda_List[1] <- 0.4
n_A_List <- c(numeric(B))

t <- 1
while(t<B+1){
  n_A_List[t]<- Updating_n_A(Xi_List[t],Lambda_List[t],n_0)
  Xi_List[t+1] <- Updating_Xi(n_A_List[t],N)
  Lambda_List[t+1] <- Updating_Lambda(N,n_A_List[t],sum_of_children)
  t<-t+1
}
n_B_List <- n_0 - n_A_List

Table <- matrix(numeric(30),nrow = 6,ncol = 5)
Table[,1] <- c(0:5)
Table[,2] <- Xi_List[1:6]
Table[,3] <- Lambda_List[1:6]
Table[,4] <- n_A_List
Table[,5] <- n_B_List
print(Table)

##      [,1]      [,2]     [,3]     [,4]     [,5]
## [1,]    0 0.7500000 0.400000 2502.779 559.2210
## [2,]    1 0.6141789 1.035478 2503.591 558.4087
## [3,]    2 0.6143782 1.036013 2504.219 557.7805
## [4,]    3 0.6145324 1.036427 2504.705 557.2948
## [5,]    4 0.6146516 1.036748 2505.081 556.9194
## [6,]    5 0.6147437 1.036996 2505.371 556.6293
```

### (c) (5%): Repeat b), but with ξ0 = 0.5 and λ0 = 0.6.

```
B <- 6

Xi_List <- c(numeric(B))
Xi_List[1]<- 0.5
Lambda_List <-c(numeric(B))
Lambda_List[1] <- 0.6
n_A_List <- c(numeric(B))

t <- 1
while(t<B+1){
  n_A_List[t]<- Updating_n_A(Xi_List[t],Lambda_List[t],n_0)
  Xi_List[t+1] <- Updating_Xi(n_A_List[t],N)
  Lambda_List[t+1] <- Updating_Lambda(N,n_A_List[t],sum_of_children)
  t<-t+1
}
n_B_List <- n_0 - n_A_List

Table <- matrix(numeric(30),nrow = 6,ncol = 5)
Table[,1] <- c(0:5)
Table[,2] <- Xi_List[1:6]
```

```r
Table[,3] <- Lambda_List[1:6]
Table[,4] <- n_A_List
Table[,5] <- n_B_List
print(Table)
```

```
##      [,1]      [,2]      [,3]     [,4]      [,5]
## [1,]    0 0.5000000 0.6000000 1977.000 1085.0004
## [2,]    1 0.4851533 0.7759770 2057.210 1004.7897
## [3,]    2 0.5048369 0.8068234 2129.758  932.2417
## [4,]    3 0.5226401 0.8369140 2194.153  867.8468
## [5,]    4 0.5384425 0.8655676 2250.272  811.7277
## [6,]    5 0.5522141 0.8921879 2298.334  763.6665
```