Final 37810

Shiting Zhu, Lijing Wang, Jingting Li

November 1, 2015

Question 1

Metropolis Hastings

1. The algorithm is Metropolis-Hastings. We first choose the new phi based on the proposal function.

- Step 0: Pick a start point ϕ_0, which is a random number from uniform distribution (0,1)
- Step 1: Pick a new candidate(ϕ_{new}) from the jumping distribution, which is Beta(cϕ_{old},c(1-ϕ_{old})). Notice the ϕ_{new} cannot be either 0 or 1, otherwise it will loop in 0 or 1 forever. For this step we used a function called phi in the r code.
- Step 2: Calculate the accaptance probability. Since the jumping distribution is not symmetric in this question, so

$$\frac{P(\phi_{new})}{P(\phi_n)} = \frac{dbeta(\phi_{new}, 6, 4)}{dbeta(\phi_n, 6, 4)}$$

$$\frac{Q(\phi_{new} * |\phi_n)}{Q(\phi_n|\phi_{new})} = \frac{dbeta(\phi_{new}, c\phi_n, c(1-\phi_n))}{dbeta(\phi_n, c\phi_{new}, c\phi_{new})}$$

$$A(\phi_n -> \phi_{new}) = min(1, \frac{\frac{P(\phi_{new})}{P(\phi_n)}}{\frac{Q(\phi_{new}*|\phi_n)}{Q(\phi_n|\phi_{new})}})$$

.

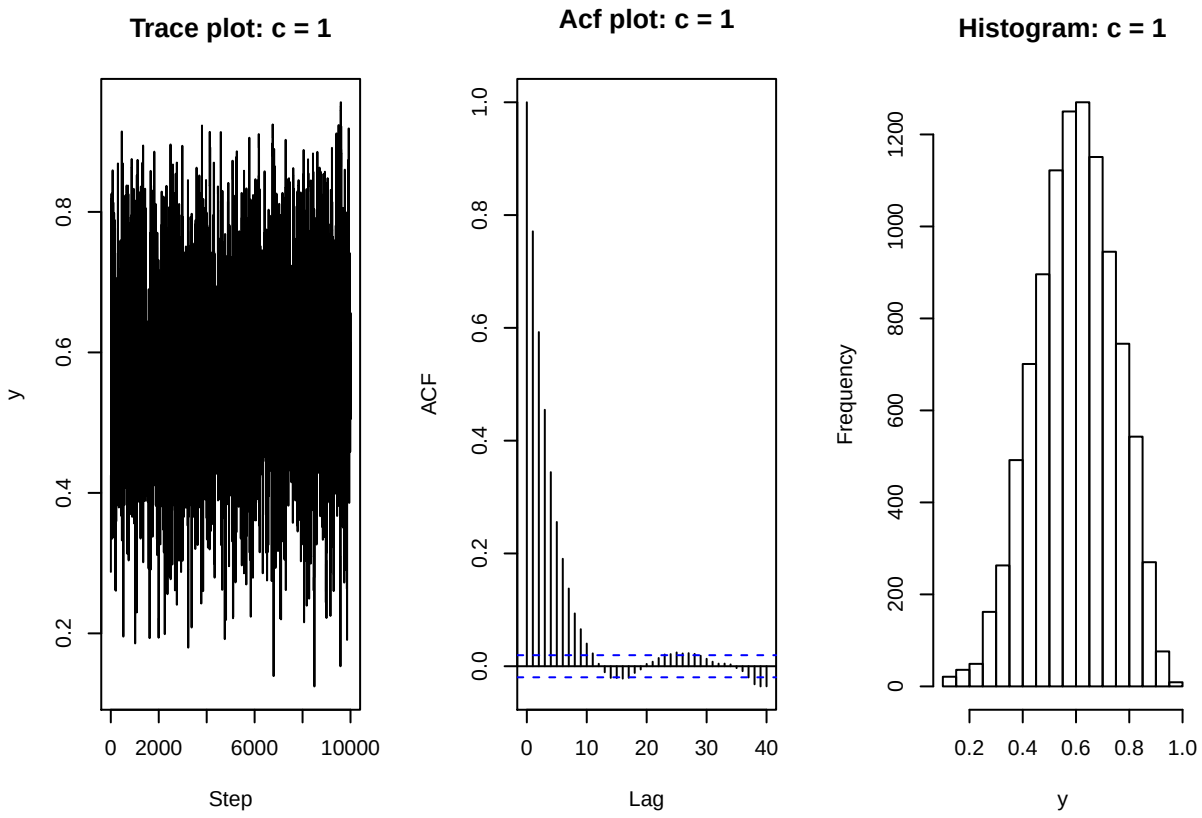
- Step 3: Sample a number from uniform(0,1).If it is more than A(ϕ_n->ϕ_{new}), we accept ϕ_{new} and set $\phi_{n+1} = \phi_{new}$. Otherwise, $\phi_{n+1} = \phi_n$.
- Step 4 : Repeat Step 1 to Step 3 until we get length n chain.

```
library(coda)
set.seed(123)

Alpha <- 6
Beta <- 4

# Use the proposal function to get new phi, noticeing phi cannot be 0 or 1
phi <- function(c,oldphi){
  newphi = 0
  # Making sure new phi is neither 0 nor 1
  while (newphi == 0 || newphi == 1){
  newphi = rbeta(1,c*oldphi,c*(1-oldphi))
  }
  return(newphi)
}
```

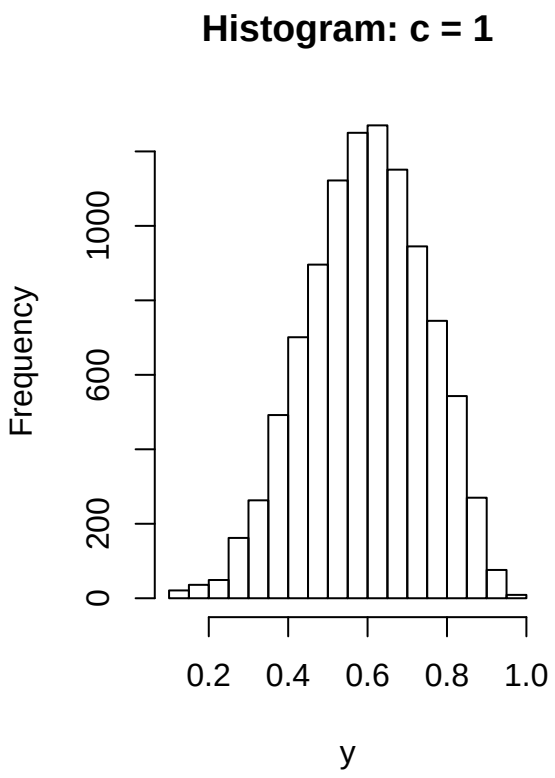
```r
# Using similar strcuture of assignment 3 to build MCMC chain
run_metropolis_MCMC <- function(startvalue,c,iterations){
  # set chain
  chain = rep(0, iterations+1)
  chain[1] = startvalue
  for (i in 1:iterations){
    phi = phi(c,chain[i])
    posterior = dbeta(phi,Alpha,Beta)/dbeta(chain[i],Alpha,Beta)
    proposal = dbeta(phi,c*chain[i],c*(1-chain[i]))/dbeta(chain[i],c*phi,c*(1-phi))
    probab = min(1,posterior/proposal)
    # accept new value if random number uniform (0,1) is less than
    # acceptance probability
    if (runif(1) < probab){
      chain[i+1] = phi
    # reject new value if random number uniform (0,1) is greater
    # or equal than acceptance probability
    }else{
      chain[i+1] = chain[i]
    }
  }
  return(chain)
}

startvalue = runif(1)
chain = run_metropolis_MCMC(startvalue,1, 10000)
acceptance = 1-mean(duplicated(chain))
```

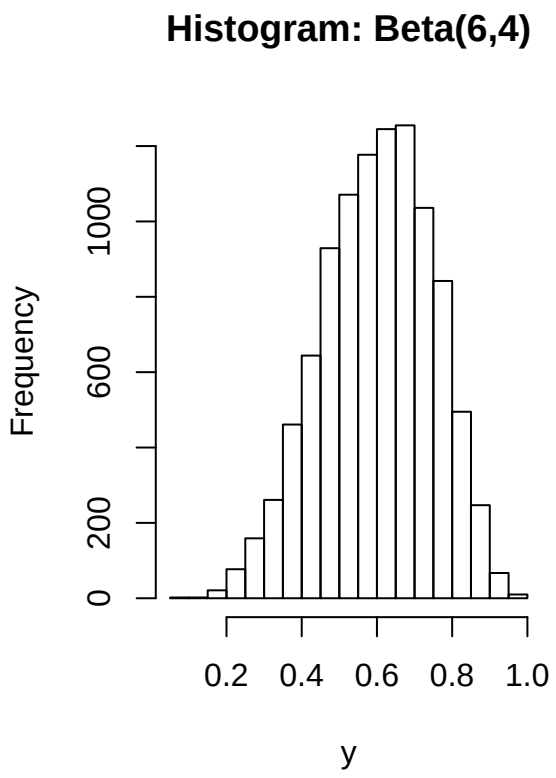
2. Based on the plots, we think the performance of the sampler when c = 1 is ok. The traceplot shows the movement of the MCMC chain values. The autcorrection plot is good and not much thinning is needed. The histograms of the MCMC chain and target distribution Beta(6,4) are pretty close. The Kolmogorov-Smirnov Statistic is showing good result and supporting our conclusions.

```r
par(mfrow=c(1,3))  #1 row, 3 columns

  traceplot(as.mcmc(chain), type="l", main = "Trace plot: c = 1", xlab="Step", ylab="y")
  acf(chain, main = "Acf plot: c = 1")
  hist(chain, main = "Histogram: c = 1", xlab="y")
```




```
 target = rbeta(10001,6,4)
par(mfrow=c(1,2))
 hist(chain, main = "Histogram: c = 1", xlab="y")
 hist(target, main = "Histogram: Beta(6,4)", xlab="y")
```






```
  ks.test(chain,"pbeta",6,4)
```

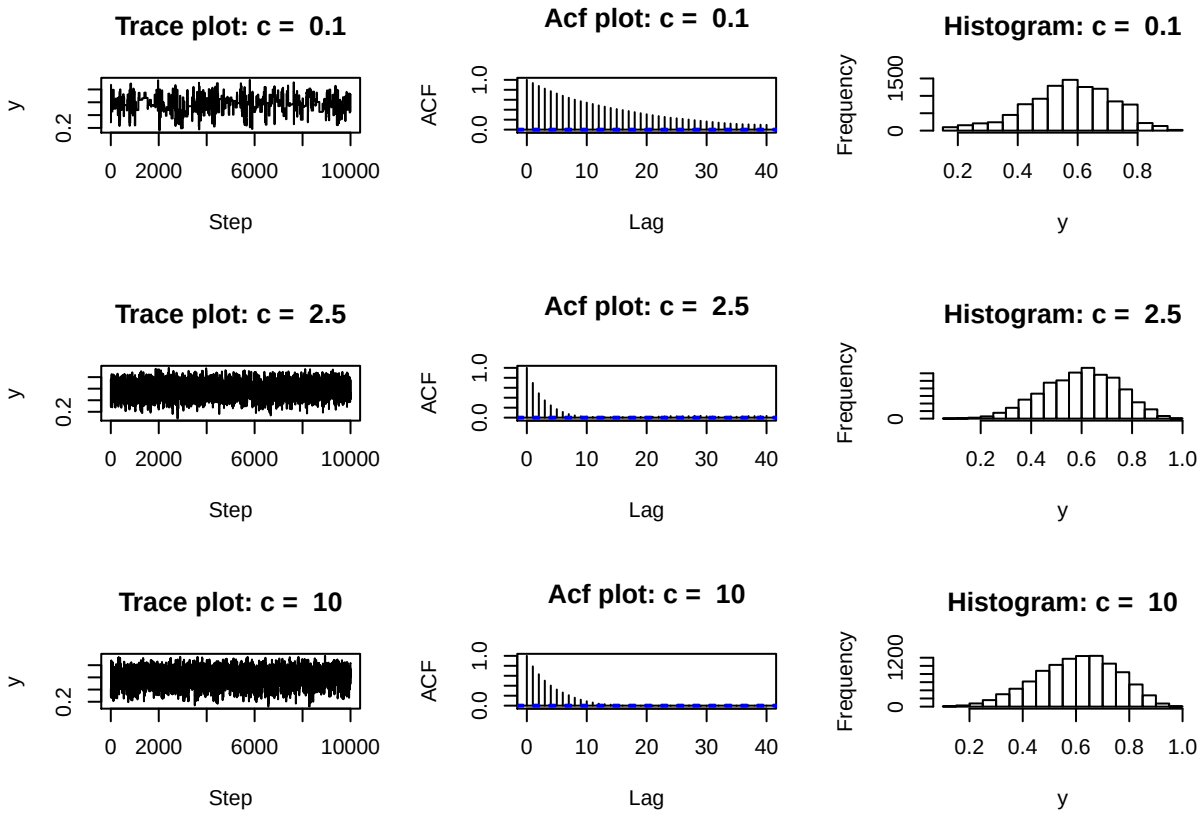
```
## Warning in ks.test(chain, "pbeta", 6, 4): ties should not be present for
## the Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  chain
## D = 0.022219, p-value = 0.0001029
## alternative hypothesis: two-sided
```

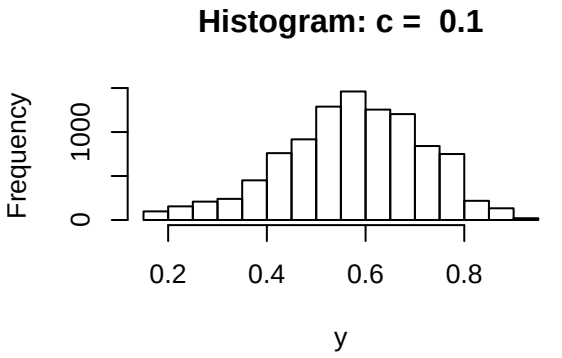
3. From the result we can see when c = 2.5 gives the best result. Based on the autocorrection plots, c = 0.1 is pretty bad since it requires a lot of burn-in. c = 10 and c = 2.5 have very close autocorrection plots but when c = 2.5 it is a little bit better. c = 0.1 histogram is very different than our target distribution while c = 2.5 and c = 10 histograms are very close to our target distribution histogram. The Kolmogorov-Smirnov Statistic result is consistent with our conclusion.

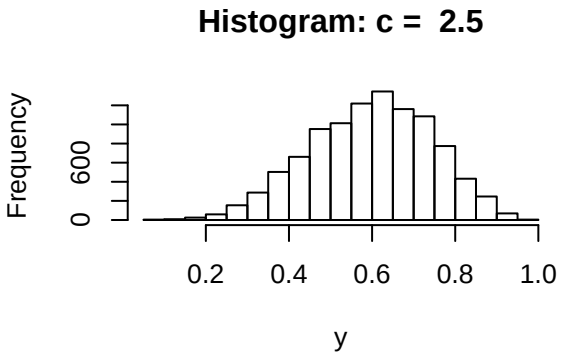
```
Cs <- c(0.1, 2.5, 10)
  # calculate MCMC chian for all the c
result <- sapply(Cs, run_metropolis_MCMC, startvalue = startvalue,iterations = 10000)

# set a graph function which will automatically have the trace plot,
# cf plot and histograms of all the c in vector Cs
graph <- function(Cs){
  n = length(Cs)
  par(mfrow = c(n, 3))
  for( i in 1:n ) {
```

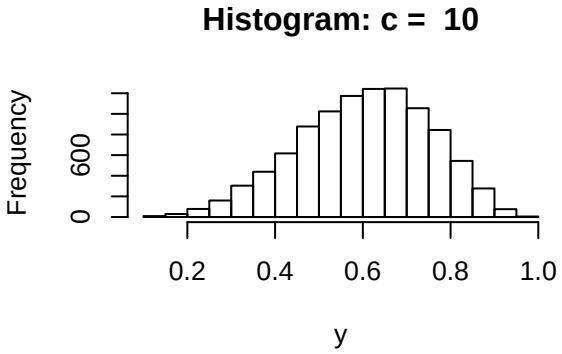
```
  traceplot(as.mcmc(result[,i]), type="l", main = paste("Trace plot: c = ", Cs[i]), xlab="Step", ylab="y")
  acf(result[,i], main = paste("Acf plot: c = ", Cs[i]))
  hist(result[,i], main = paste("Histogram: c = ", Cs[i]), xlab="y")
  }
}

graph(Cs)
```

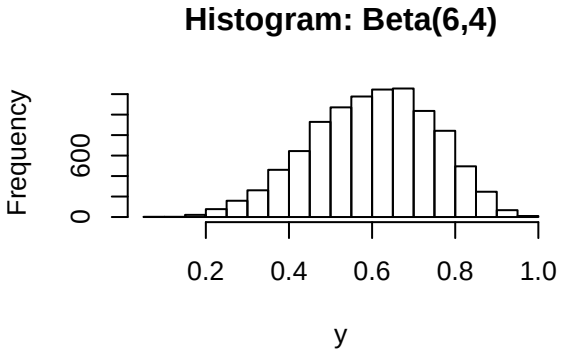



```
par(mfrow=c(2,2))
for( i in 1:length(Cs) ) {
hist(result[,i], main = paste("Histogram: c = ", Cs[i]), xlab="y")
}
hist(target, main = "Histogram: Beta(6,4)", xlab="y")
```










```
ks.test(result[,1],"pbeta",6,4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  result[, 1]
## D = 0.092739, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(result[,2],"pbeta",6,4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  result[, 2]
## D = 0.017795, p-value = 0.003549
## alternative hypothesis: two-sided
```

```
ks.test(result[,3],"pbeta",6,4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  result[, 3]
## D = 0.015745, p-value = 0.01405
## alternative hypothesis: two-sided
```

Question 2

Gibbs Sampling

```
Gibbs_<-function(x0,y0,iterations,B=5,burnIn=0)
## 5 parameters are needed in this function, with B=5 and burnIn=0 by default
{
  if(x0>0&&x0<B&&y0>0&&y0<B)
## to check whether the starting values are in the domain.
  {
    x<-c(x0,rep(NA,iterations-1))
## Initialize the Markov chain
    y<-c(y0,rep(NA,iterations-1))
## Initialize the Markov chain
    for(i in 1:(iterations-1))
    {
      x[i+1]<-(-log(1-runif(1)*(1-exp(-y[i]*B)))/y[i])
## use inverse transform sampling to draw sample from conditional distribution
## p(x^{i+1}|y^{i})
      y[i+1]<-(-log(1-runif(1)*(1-exp(-x[i+1]*B)))/x[i+1])
## use inverse transform sampling to draw sample from conditional distribution
## p(y^{i+1}|x^{i+1})
    }
    if(burnIn>0)
    {
        x<-x[-(1:burnIn)]
        y<-y[-(1:burnIn)]
        print(length(x))
## discard the first bunch of draws for the burn-in process
    }
    return(data.frame(x,y))
  }
 else
   stop("Initial values incorrect")
## print the information for incorrect starting values
}
```

To estimate the marginal distribution generated from the conditional distributions using Gibbs sampling, we first pick up the starting values for X and Y. Denote them as `x0` and `y0`. `iterations` stands for the number of draws we want to get from Gibbs sampling. `B` is the number given in the exercise, which is 5 here. `burnIn` is the number of draws we want to discard for the burn-in process, the default for `burnIn` is 0.

In this case, both the conditional distribution of $X|Y$ and $Y|X$ are truncated exponential distribution. The domains of both conditional pdf's are $[0, B]$. So the starting values should satisfy $0 <$`x0`$< B$ and $0 <$`y0`$< B$. Therefore I put a restriction `if(x0>0&&x0<B&&y0>0&&y0<B)` on the input of starting values to check if they satisfy the requirement.

Since

$$p(x|y) \propto ye^{-yx}, 0 < x < B$$

$$p(y|x) \propto xe^{-yx}, 0 < y < B$$

Consider $p(x|y)$ only. To get samples using inverse transform sampling, first we need to figure out the normalizing constant for the conditional pdf and then the inverse function of cdf.

The normalizing constant can be calculated using the formula $c = \frac{1}{\int f(x|y)\,\mathrm{d}x}$, if $p(x|y) \propto f(x|y)$. In this case, $f(x|y) = ye^{-yx}, 0 < x < B$.
Therefore

$$c = \frac{1}{\int_0^B ye^{-yx}\,\mathrm{d}x} = \frac{1}{1-e^{-By}}$$

.

Then we start to calculate the cdf $H(x|y)$ for $X|Y$.
We have

$$p(x|y) = \frac{ye^{-yx}}{1-e^{-By}}, 0 < x < B$$

Using the formula $F(x) = \int_{-\infty}^{x} p(z)\,\mathrm{d}z$, we get the cdf of $X|Y$:

$$F(x|y) = \frac{1-e^{-xy}}{1-e^{-By}}, 0 < x < B$$

Then we can write down the inverse function of $F(x|y)$, Denote as $F^{-1}(u|y), where 0 \leq u \leq 1$.
We have:

$$F^{-1}(u|y) = -\frac{log(1-u(1-e^{-By}))}{y}, 0 \leq u \leq 1$$

According to inverse transform sampling, to draw a sample from $p(x|y)$, we first generate u from $Unif[0,1]$, then $x = F^{-1}(u|y)$ is from the conditional distribution of X|Y.

We draw samples from $p(y|x)$ using the same method.

Now we start generating samples from p(x,y).

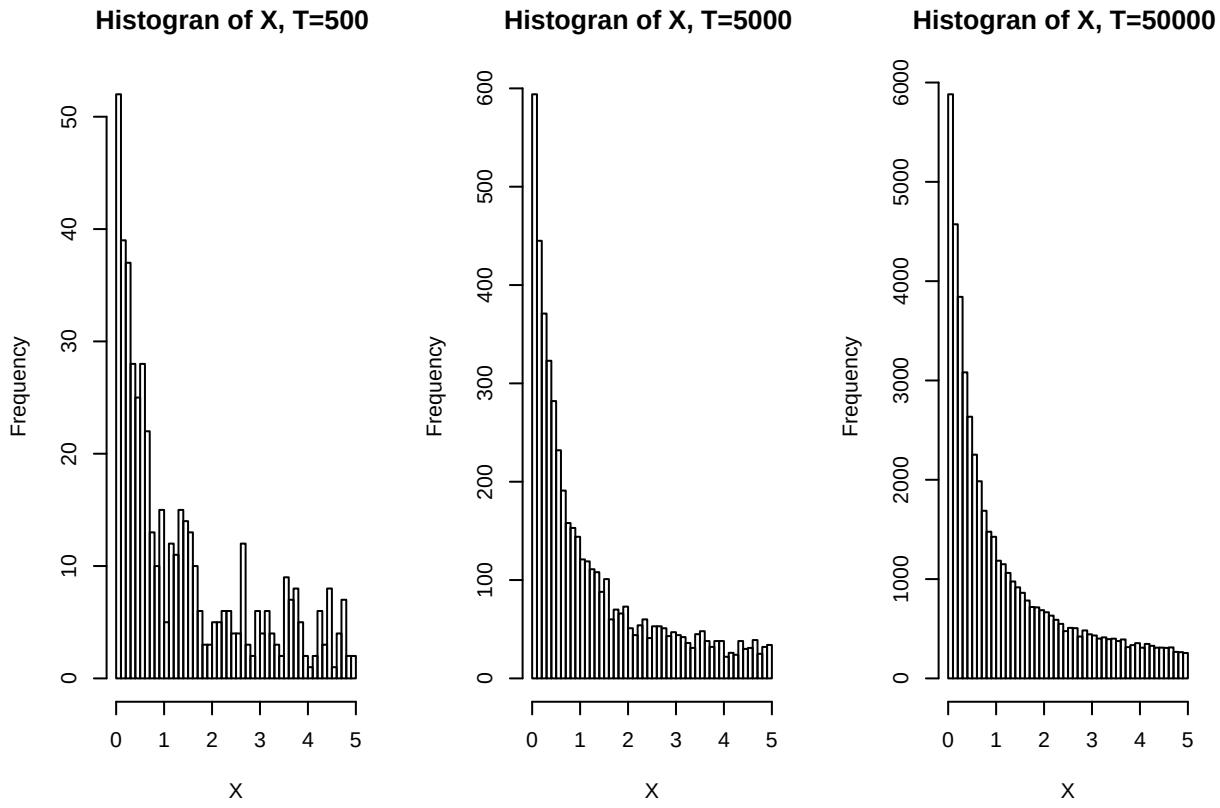
Here we start Gibbs sampling with `x0` and `y0`, draw a value $x^{(1)}$ from the full conditional $p(x|$ `y0` $)$ using inverse transform sampling. Then use the updated $x^{(1)}$ to draw a sample from $p(y|x^{(1)})$.

To get more samples using Gibbs sampling, continually use the most updated values of x and y when generating samples from conditional distribution. To be more specific, generate $x^{(i+1)}$ from $p(x|y^{(i)})$ and $y^{(i+1)}$ from $p(y|x^{(i+1)})$. Repeat it for $n =$ `iterations` times.

For the burn-in process, discard the first $m =$ `burnIn` samples from the Markov chain to reduce the influence of starting values on the Markov chain.

For $B = 5$ and sample size $T = 500, 5000, 50000$,to see the histogram of of values for x generated from Gibbs sampling, We have:

```
par(mfrow=c(1,3))
x<-Gibbs_(1,2,50000,B=5,burnIn=0)
hist(x[1:500,1],breaks=60, main="Histogran of X, T=500 ",xlab="X")
hist(x[1:5000,1],breaks=60, main="Histogran of X, T=5000 ",xlab="X")
hist(x[1:50000,1],breaks=60, main="Histogran of X, T=50000 ",xlab="X")
```



Here we picked up the starting values arbitrarily.

The plots above shows that as T increases, the histogram tend to be more smooth.

Use the mean of the generated samples to estimate the expectation of X.

```
set.seed(123)
mean_500<-c()
mean_5000<-c()
mean_50000<-c()
for(i in 1:10)
{
  x<-Gibbs_(1,2,50000,B=5,burnIn=0)
  mean_500<-c(mean_500,mean(x[1:500,1]))
  mean_5000<-c(mean_5000,mean(x[1:5000,1]))
  mean_50000<-c(mean_50000,mean(x[1:50000,1]))
}
mean_500
```

```
##  [1] 1.323771 1.392163 1.425209 1.300402 1.319226 1.200004 1.389191
##  [8] 1.305194 1.360014 1.357299
```

```
mean_5000
```

```
##  [1] 1.266212 1.263367 1.264459 1.280466 1.269729 1.280995 1.252972
##  [8] 1.279271 1.254663 1.296133
```

```
mean_50000
```

```
##  [1] 1.259416 1.272679 1.266894 1.275371 1.269562 1.270537 1.267437
##  [8] 1.267778 1.261333 1.262746
```

```
c(var(mean_500),var(mean_5000),var(mean_50000))
```

```
## [1] 0.0040109521 0.0001785322 0.0000253901
```

As we can see, the mean is around 1.26, as sample size T increases, the variance of the estimator also decreases.

Question 3

K-Means

```
set.seed(1234)
k_means<-function(data,k){
  row<-nrow(data)
  col<-ncol(data)
  # index matrix stores the classifying result and the Euclidean distance
  index<-matrix(0,nrow=row,ncol=2)
  # stores the center of each cluster
  center<-matrix(0,nrow=k,ncol=col)
  rand<-as.vector(sample(1:row,size=k))
  for(i in 1:k){
    index[rand[i],1]<-i
    center[i,]<-data[rand[i],]
    center<-matrix(center,nrow=k,ncol=col)
  }
  changed=TRUE
  while(changed){
    previous<-c()
    for(i in 1:row){
      # set a large initial distance for updating
      initialDistance<-1e4
      previous[i]<-index[i,1]
      # find the most suitable cluster for each data and update the index matrix
      for(j in 1:k){
        currentDistance<-sqrt(sum((data[i,]-center[j,])^2))
        if(currentDistance<initialDistance){
           initialDistance<-currentDistance
           index[i,1]<-j
           index[i,2]<-currentDistance
        }
      }
    }
    # check if there is change in index matrix
    for(i in 1:row){
      if(previous[i]!=index[i,1]){
```

```
        changed<-TRUE
      break
      }
      else
        changed<-FALSE
    }
    # compute new center matrix
    for(i in 1:k){
    cluster<-as.matrix(data[index[,1]==i,])
    center[i,]<-colMeans(cluster)
    }
}
result<-index[,1]
return(result)
}
```

K-Means algorithm:
At first, the parameter in k_means function are data that you need to classify and the number of clusters k. Define an index matrix "index", the first column of it stores the cluster of every data, and the second column is the Euclidean distance between the data and its cluster's center.
Define "center" as a matrix to save the center of each cluster, compute all measurements' centers and save them seperately in 13 columns.
Then we obtain initial centers by finding k random data from the dataset.
Define "changed" as an index to imply if there is change in classifying, or the first column of "index". If any data has changed its cluster, then mark "changed" as TRUE; if not, mark it as FALSE.
Next, we start to classify the dataset.
Steps:

- 1.For each data, compute the Euclidean distance between it and the center of each cluster. Compare these k distances, find the minimum one and put the data into that cluster. Update the matrix "index" at the same time.
- 2.If "changed" is marked as TRUE, continue. If "changed" is marked as FALSE, then the while loop stops.
- 3.Recompute the k centers. For each cluster, find all data that are in it and compute the mean of them as the new center.
  Finally, return the result of classifying.