Question 1

Wenjing Xu, Yanfei Zhou, Yijia Zhao

Question 1 Metropolis-Hastings

Part 1

Implement Metropolis-Hastings algorithm to sample from a Beta distribution with parameters (6,4), we need the following steps.

1. Choose a starting value θ_0. This will be generated from a uniform distribution (as required by question setting).
2. At each iteration, draw a candidate θ^* from a jumpting functin with the form $\phi_{prop}|\phi_{old} \sim Beta(c\phi_{old}, c(1-\phi_{old}))$. This will help us to define the proposal function.
3. Define an acceptence ratio, so that if we accept θ^* and add it into the chain with the probability given by min{1, acceptance ratio}.
4. If θ^* is not accepted, then stay at that point.
5. Repeat the process until θ does not change.

First we define the target function whose domain only takes values x from [0,1], and return the density function of beta(x,6,4), and return 0 when x>1 or x<0.

```
target = function(x){
  if (any(x>1) | any(x<0)){
    return(0)}
  else{
    return (dbeta(x,6,4))
  }
}
```

Then we define the proposal function which generate one sample from the distribution $\phi_{prop}|\phi_{old} \sim Beta(c\phi_{old}, c(1-\phi_{old}))$, where ϕ_{old} is the last sample we had and c is a constant.

```
proposalfunction<-function(x){
  return(rbeta(1,shape1=c*x,shape2=c*(1-x)))
}
```

The code for Metropolis-Hastings algorithm is shown as follows. Set the constant c to be 1, the number of iteration to be 10000, and randomly generate a first starting value from a uniform distribution $[0, 1]$

```
set.seed(1234)
num_iteration=10000
c=1
# initialize an arbitrary point to be the staring values, and set the number of iterations
start=runif(1,0,1)

run_metropolis_MCMC <- function(startvalue, iterations){
  # create a chain to store all the samples generated in each iterations
  chain = rep(0,iterations)
  # assign the first element in the chain to be the staring value
  chain[1] = startvalue
  for (i in 1:iterations){
    # set the last sample we generated in chain as currentx
    currentx = chain[i]
    # generate a candidate x* (here named proposal) using the jumping distribution given i
n the question, with parameter phi equals current x
    proposal = proposalfunction(chain[i])
    # calculate the acceptance ratio (here named probab), which will be used to decide whe
ther to accept or reject the candidate
    probab = (target(proposal)/target(chain[i]))*(dbeta(chain[i],shape1=c*proposal,shape2=
c*(1-proposal)/dbeta(proposal,shape1=c*chain[i],shape2=c*(1-chain[i]))))
    # generate a uniform random number u on [0,1], and compare u with probab to determine
 either to keep the  candidate in chain or drop it
    if (runif(1) < probab){
      # if u<= acceptance ratio, accept the candidate by setiing next element in the chain
 to proposal
      chain[i+1] = proposal
    }else{
      # if u<= acceptance ratio, reject the candidate proposal and set next element same a
s last element(currentx)
      chain[i+1] = chain[i]
    }
  }
  #return the chain which stores all generated samples from the target distribution
  return(chain)
}
```

Calculate how many different samples we have drawn from running the algorithm once, (equivalent to saying how many newly generated samples have been accepted) store the ratio as accpetance_rate

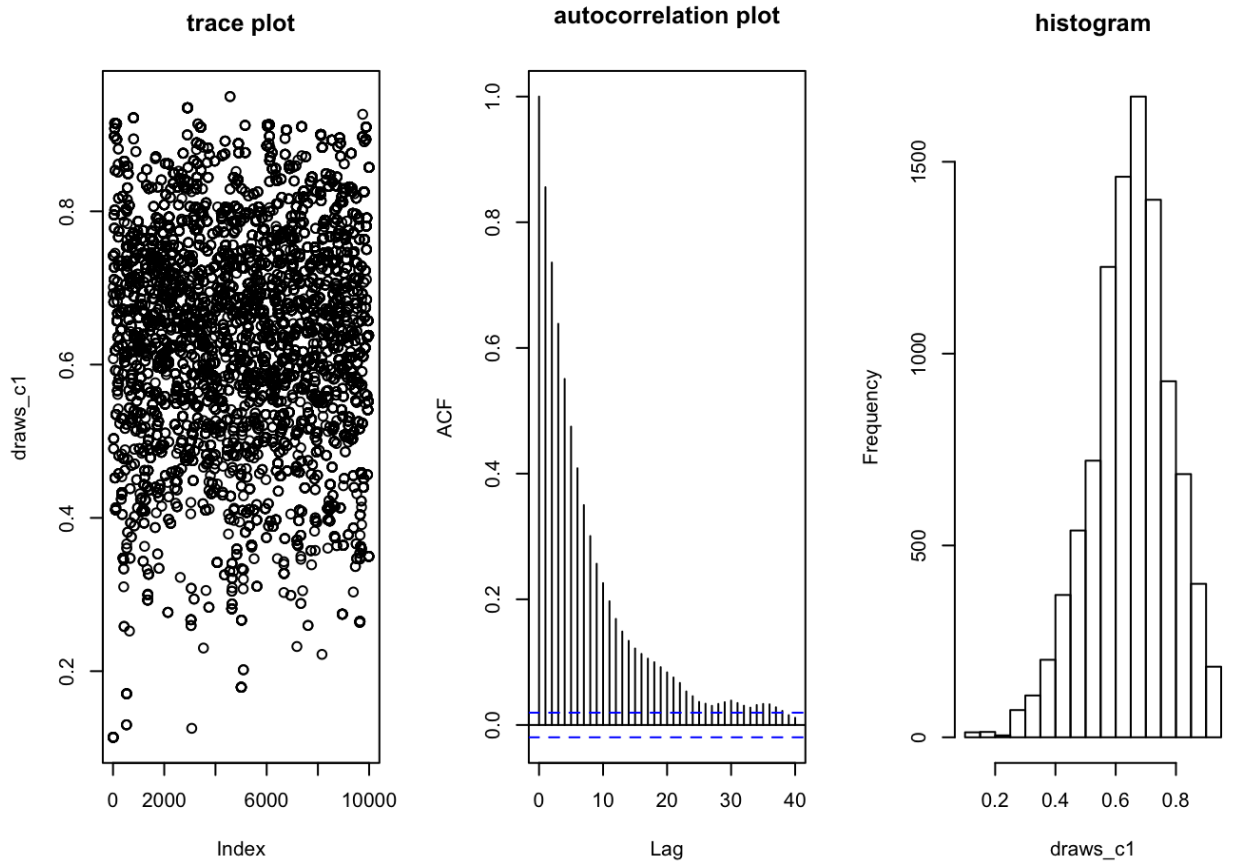
```
set.seed(3)
draws_c1 = run_metropolis_MCMC(startvalue=start, num_iteration)
acceptance_rate_c1 = 1-mean(duplicated(draws_c1))
acceptance_rate_c1
```

```
## [1] 0.1870813
```

Part 2

We provide a trace plot of this sampler and an autocorrelation plot, as well as a histogram of the draw to evaluate the performance of the sampler.

```
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c1, main="trace plot"); acf(draws_c1,main="autocorrelation plot"); hist(draws_c
1,main="histogram")
```

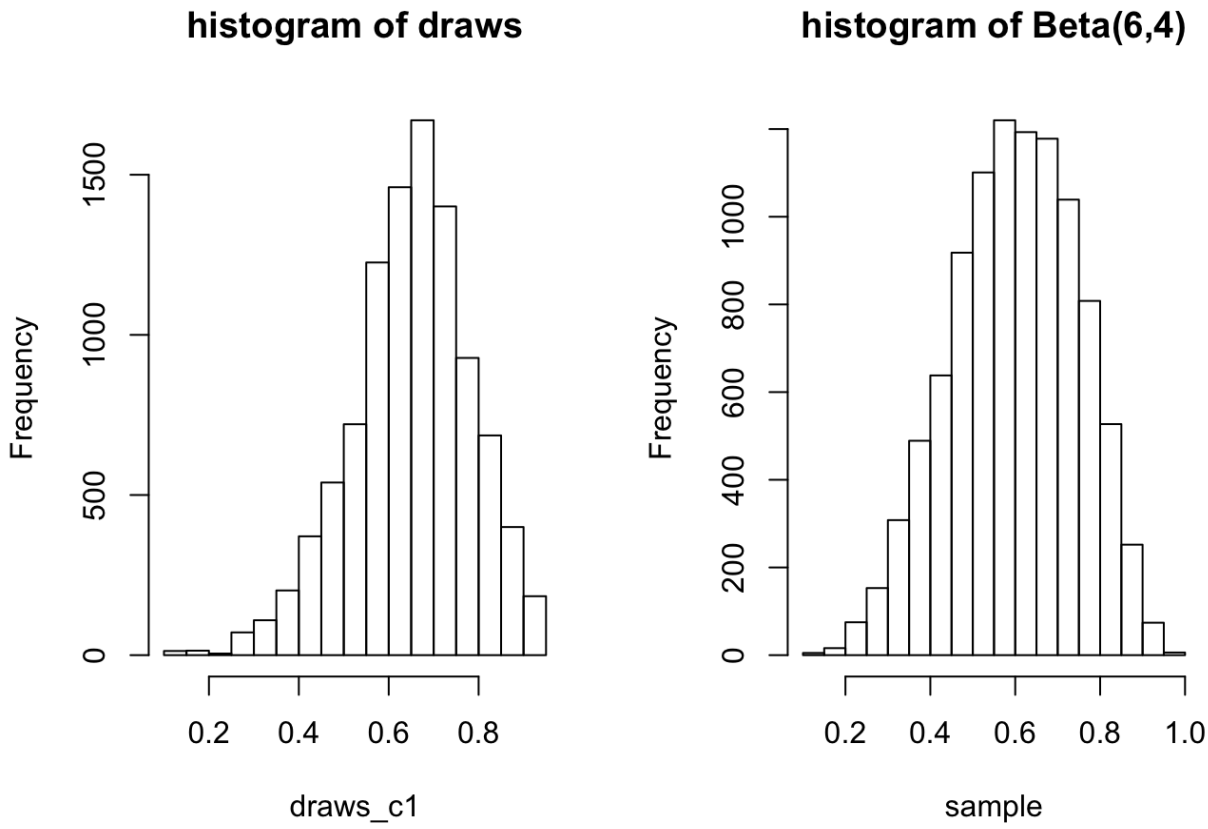


From the trace plot we can see, the sampler we generated are spread pattenlessly across index and the values are always between [0,1], however, the values are concentrated between [0.4,0.9]; from autocorrelation plot we see the autocorelation between consecutive samplers are high but decreases as the lag increases. This make sense as the next sample point we decided to add into the resulting vector is highly dependent on the previous sample, but as the simulation proceeds forward, the correlation between next samplers and further previous samplers are decreasing.

Compare the histogram of draws with target distribution of Beta(6,4).

- First, graphically compare these two.

```
par(mfrow=c(1,2))
hist(draws_c1, main = "histogram of draws",nclass = 15)
sample <- rbeta(num_iteration,6,4)
hist(sample, main = "histogram of Beta(6,4)",nclass = 15)
```



We found that our algorithm generate a sample whose histogram is very similar to the beta distribution histogram with the required shape parameters.

- Second, use Kolmogorov-Smirnov statistic to compare these two.

```
ks.test(draws_c1,"pbeta",6,4)
```

```
## Warning in ks.test(draws_c1, "pbeta", 6, 4): ties should not be present for
## the Kolmogorov-Smirnov test
```

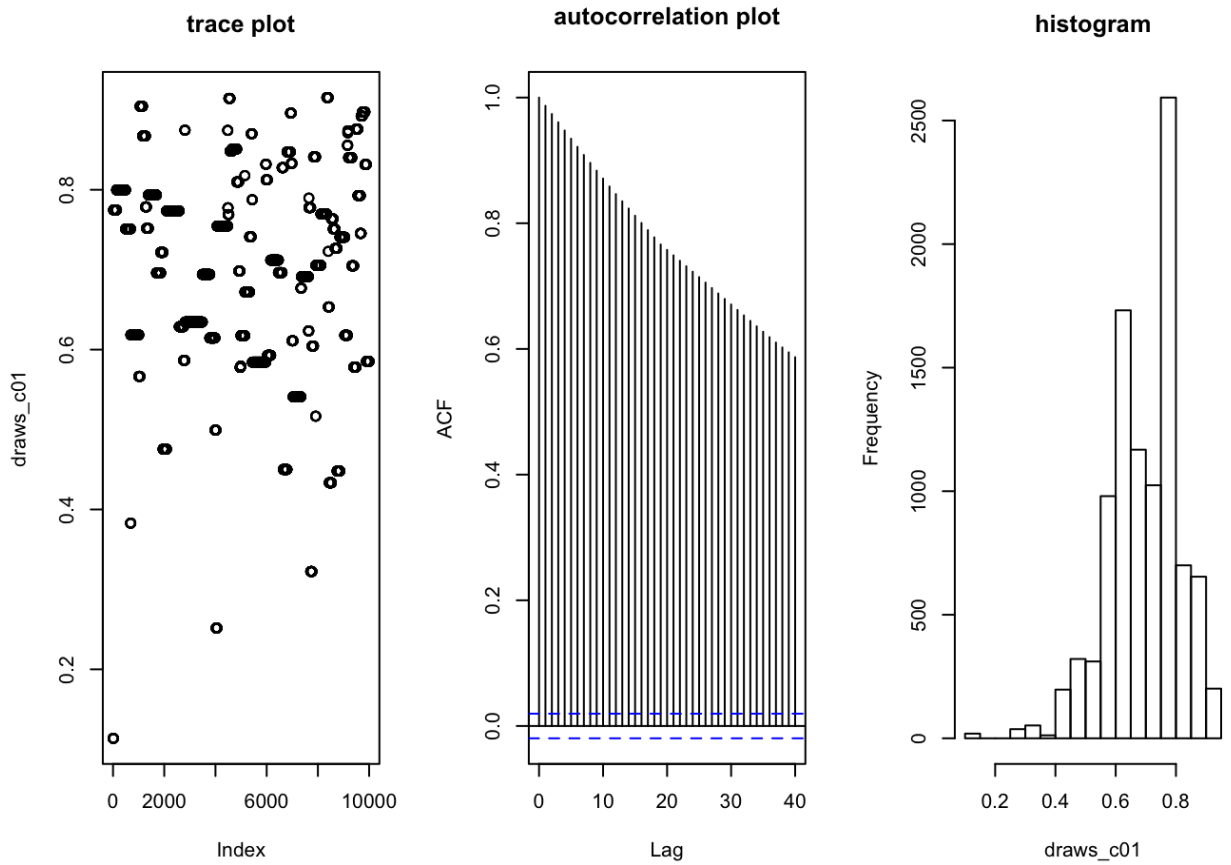
```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws_c1
## D = 0.16419, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

The Kolmogorov–Smirnov statistic measures a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. Because the p-value is extremely small, we reject the null hypothesis that the sample is drawn from the reference (Beta(6,4)) distribution. Notice that we got the error message because we have replicated values in the sample we drawn.

Part 3

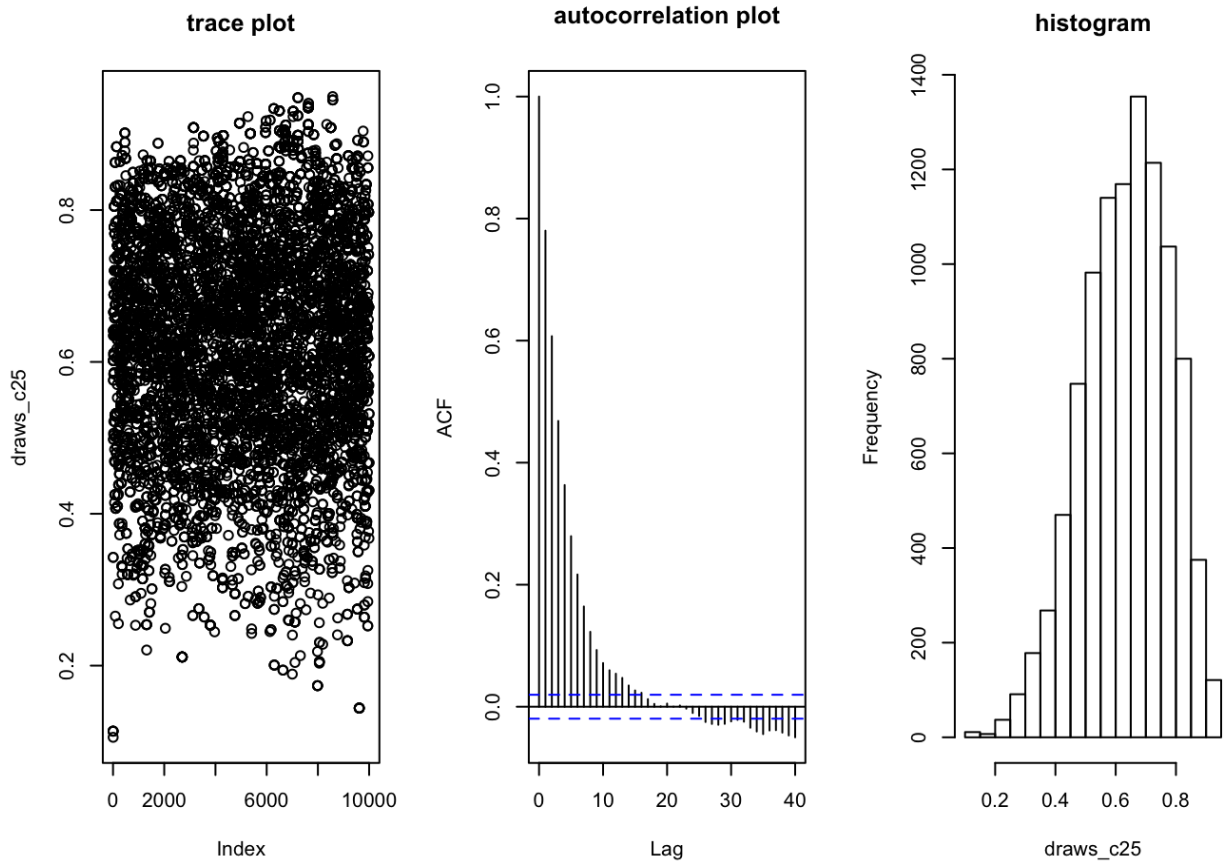
Re-run this sampler with c = 0.1, c = 2.5 and c = 10, and compare their results.

```
c=0.1
draws_c01 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))
plot(draws_c01,main="trace plot"); acf(draws_c01,main="autocorrelation plot"); hist(draws_
c01,main="histogram")
```



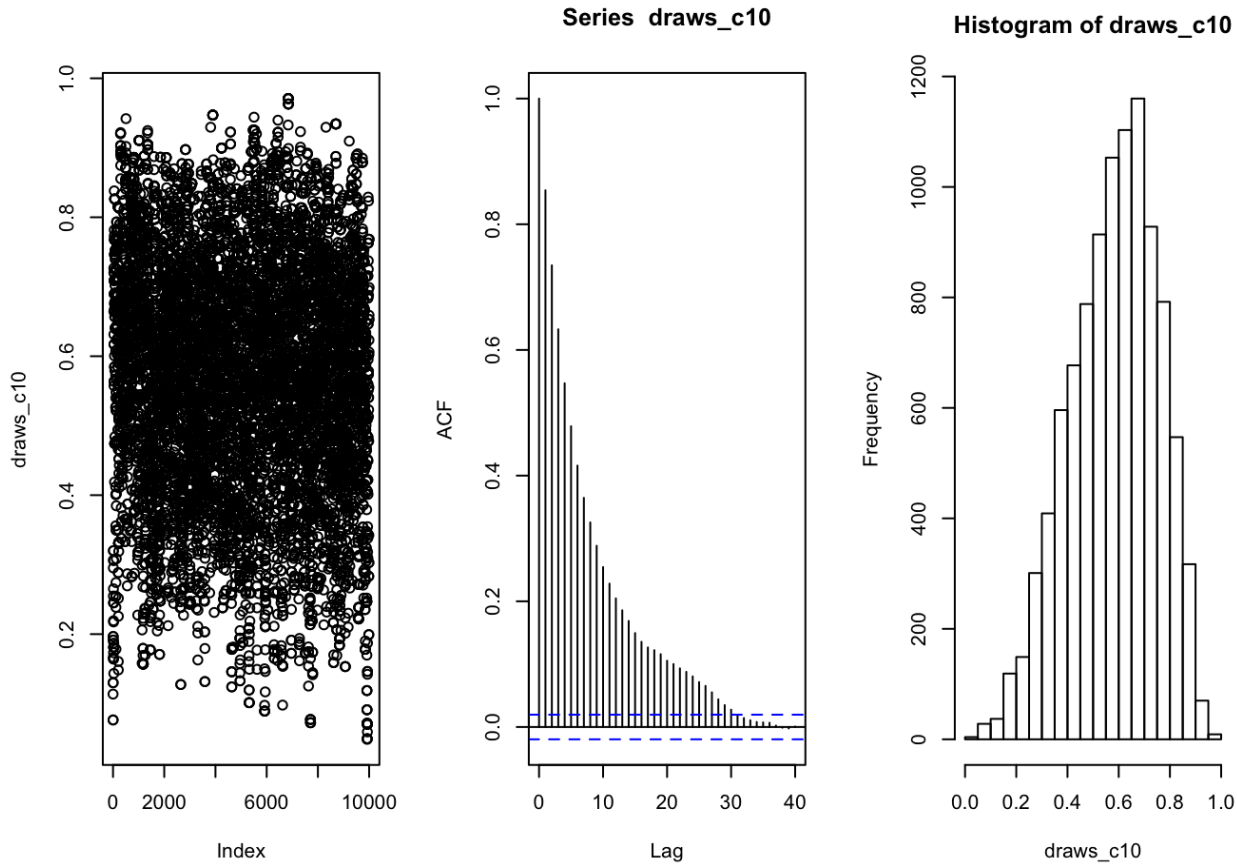

```
acceptance_rate_c01 = 1-mean(duplicated(draws_c01))
```

```
c=2.5
draws_c25 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c25,main='trace plot'); acf(draws_c25,main='autocorrelation plot'); hist(draws_
c25,main='histogram')
```




```
acceptance_rate_c25 = 1-mean(duplicated(draws_c25))
```

```
c=10
draws_c10 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c10); acf(draws_c10); hist(draws_c10)
```




```
acceptance_rate_c10 = 1-mean(duplicated(draws_c10))
```

Compare the acceptance rate for different c values,

```
paste0("acceptance rate when c=0.1: ",acceptance_rate_c01)
```

```
## [1] "acceptance rate when c=0.1: 0.00879912008799122"
```

```
paste0("acceptance rate when c=1: ",acceptance_rate_c1)
```

```
## [1] "acceptance rate when c=1: 0.187081291870813"
```

```
paste0("acceptance rate when c=2.5: ",acceptance_rate_c25)
```

```
## [1] "acceptance rate when c=2.5: 0.399260073992601"
```

```
paste0("acceptance rate when c=10: ",acceptance_rate_c10)
```

```
## [1] "acceptance rate when c=10: 0.650734926507349"
```

Comparing the acceptance rate of simulation for different values of c, we see the acceptance rate increases as c increases, meaning the algorithm are more efficient when c has larger value. From the trace plots we can also see, when $c = 10$, we generate most data points for the target distribution, with same number of iterations.

Compare the histogram plots, the histograms tend to be more consistent with the target beta(6,4) distribution as we increase the value of c.