Final Project

Alice Mee Seon Chung, Seulmin Yang

11/3/2017

Metropolis-Hastings

1.

Describing algorithm

1. Set a randomly selected start value from a uniform distribution (0,1)
2. Calculate ϕ_{new} from $Beta(c\phi_{old}, c(1-\phi_{old})$ using proporsal function.
3. Compute the acceptance probability. The Beta distribution is not symmetric, so we need to modify like below

$$Posterior = \frac{f(\phi_{new})}{f(\phi_i)}$$

$$Proposal = \frac{g(\phi_{new}|\phi_i)}{g(\phi_i|\phi_{new})}$$

$$acceptanceprobability = Posterior * Proposal$$

Final accpeptance probability will be $min(1, appceptanceprobability)$.

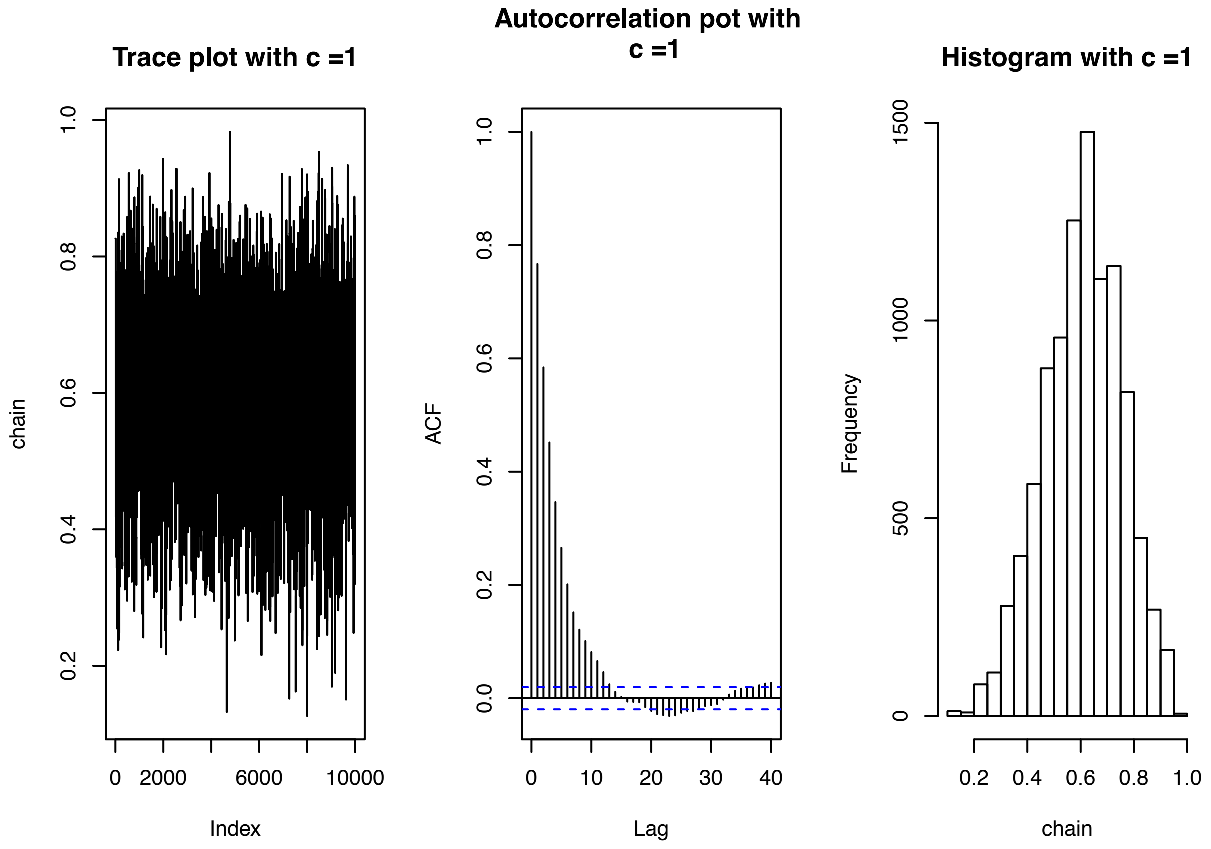
4. If the accpeptance probability greater than a number from uniform distribution(0,1), then accept ϕ_{new} and assign $\phi_{i+1} = \phi_{new}$, otherwise, keep the current value and assign $\phi_{i+1} = \phi_i$.
5. Repeat 2~4 to finish all i times and get final chain.

R codes

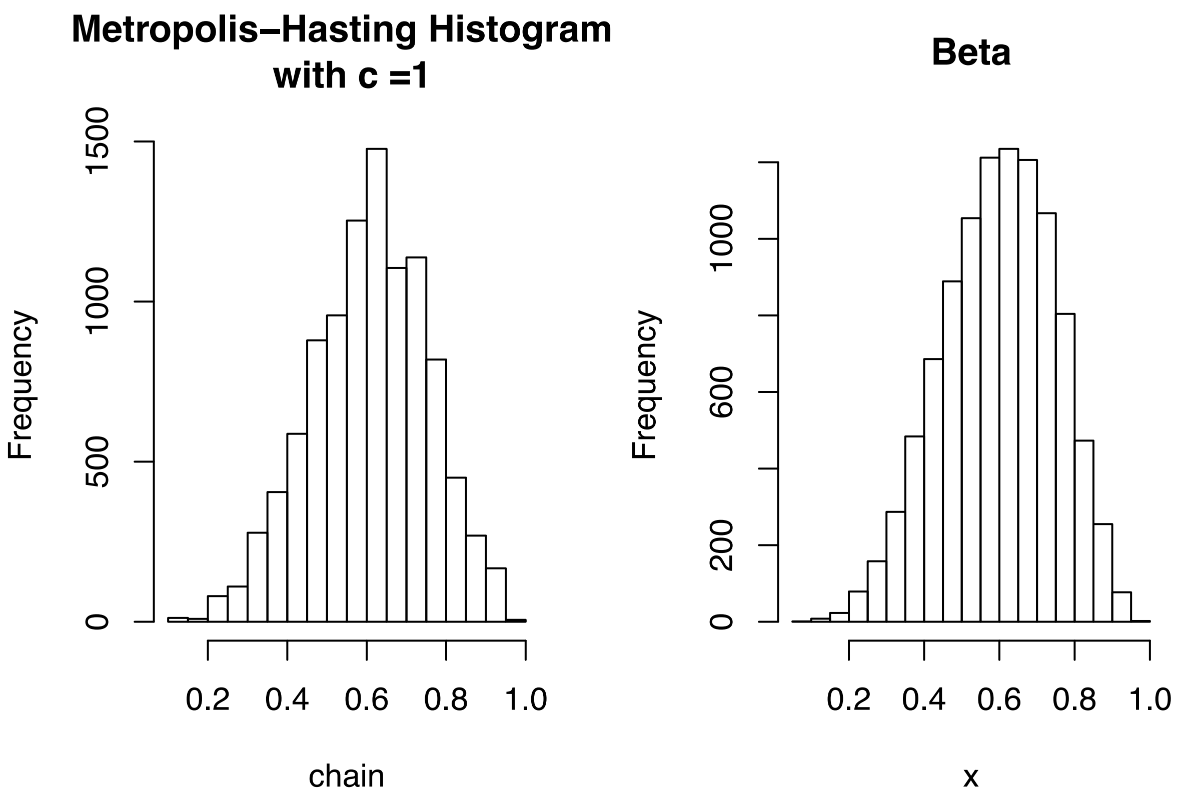
```r
# Set alpha and beta and startvalue
alpha <- 6
beta <- 4
startvalue <-runif(1)

# Proposal function in the problem to get new phi
proposalfunction <- function(c, phi_old){
  return(rbeta(1, c* phi_old, c*(1-phi_old)))
}

# Posterior function to compute the posetiror
posterior_beta<-function(x, alpha, beta){
  return (dbeta(x, alpha, beta))
}

# Proposal function to compute the proposal
proposal<-function(x, c, y){
  return (dbeta(x,c*y, c*(1-y)))
}

# Modify the code from Assignment2
run_betaMH <- function(startvalue, c, iterations){

  chain = rep(0, iterations)
  chain[1] = startvalue
  for (i in 1:iterations){
    new_beta = proposalfunction(c, startvalue)
    posterior = posterior_beta(new_beta, alpha, beta) / posterior_beta(chain[i], alpha, beta)
    proposal = proposal(chain[i], c, new_beta) / proposal(new_beta, c, chain[i])
    probab = posterior * proposal
    # if acceptance probability  is greater than the criteria, then accept new_beta
    # otherwise keep the current beta value
    if (runif(1) < min(1, probab)){
      chain[i+1] = new_beta
      startvalue = new_beta
    }else{
      chain[i+1] = chain[i]
    }
  }
  return(chain)
}
```

2.

```r
alpha <- 6
beta <- 4
startvalue <-runif(1)
chain<-run_betaMH(startvalue, c=1, iterations = 10000)

par(mfrow=c(1,3))
plot(chain,  type = 'l', main ='Trace plot with c =1')
acf(chain, main ='Autocorrelation pot with \n c =1')
hist(chain, main ='Histogram with c =1')
```



```r
x = rbeta(10000, 6, 4)
par(mfrow=c(1,2))
hist(chain, main ='Metropolis-Hasting Histogram \n with c =1')
hist(x, main ='Beta')
```



From above graphs, I think the performace of the sampler with c as 1 does not look fine. The traceplot showed the values of the parameter taken during the runtime of the chain. From two histograms plots, we can observe that the distibution of two histrograms seems different, especially in the center part. It seems that the sampler result may not follow the $Beta(6,4)$ distribution.

```
d<-ks.test(chain,x);d
```

```
## Warning in ks.test(chain, x): p-value will be approximate in the presence
## of ties
```

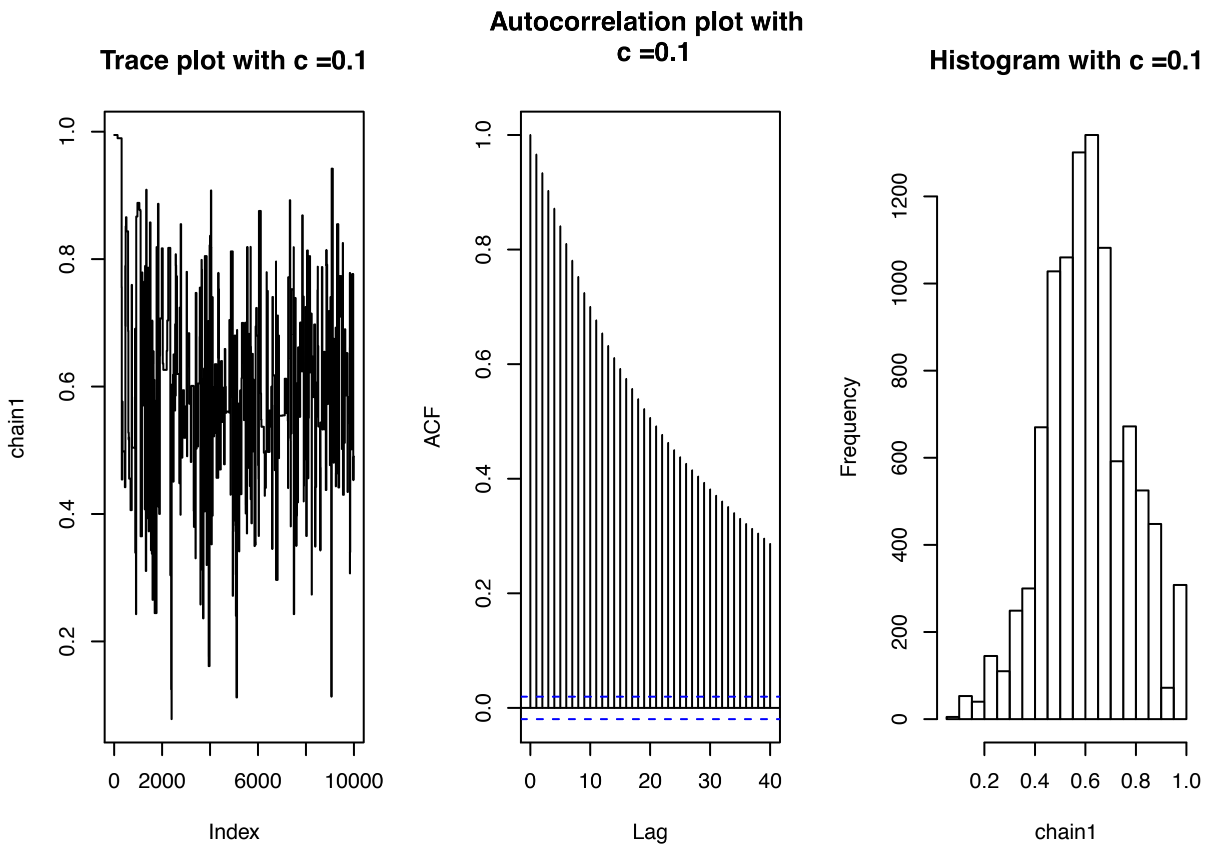
```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  chain and x
## D = 0.037547, p-value = 1.508e-06
## alternative hypothesis: two-sided
```

In addition to graphical tools, we also can check the result from Kolmogorov-Smirnov statistic. From the result, since p-value is less than 1.508e-06, which is close to 0, we can reject the null hypothesis. Thus we can conlucde that two distribution are not same.

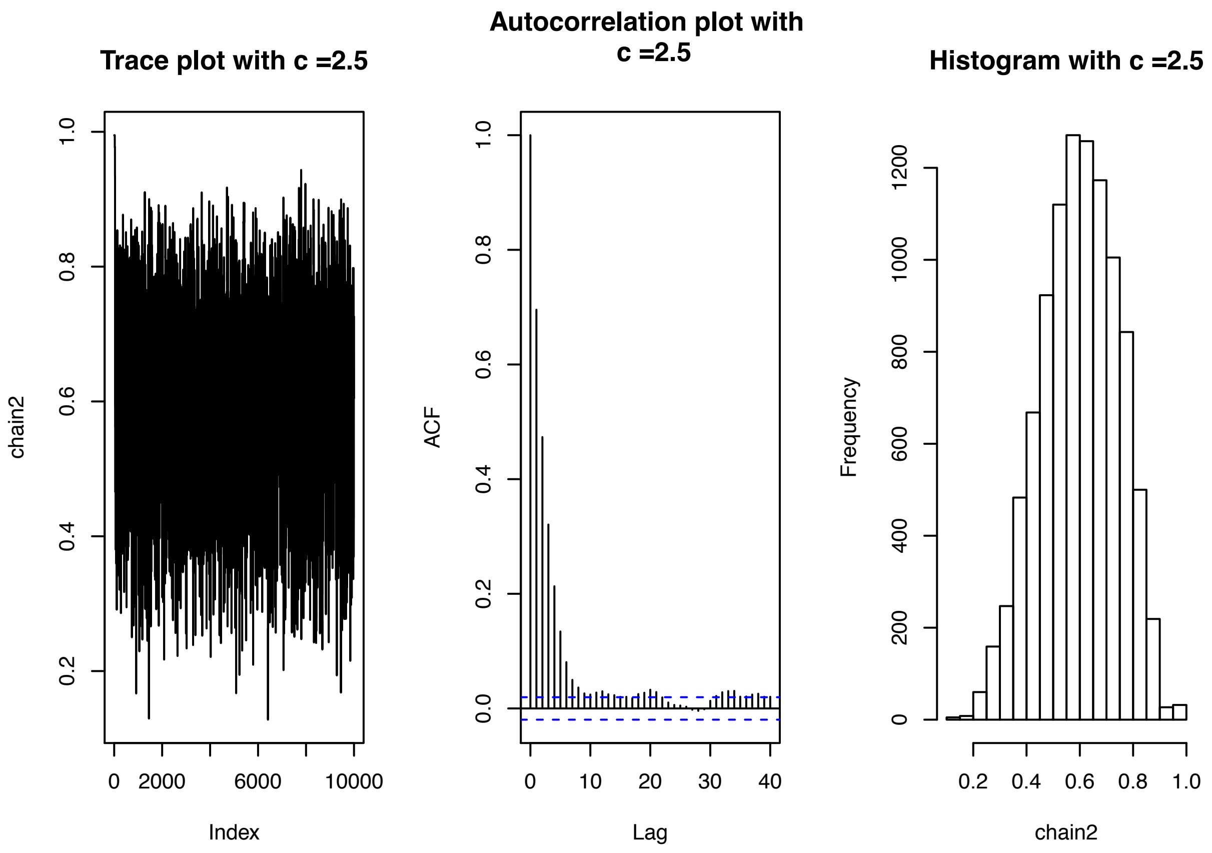
Therefore the performance of the sampler with c as 1 is not good, and we can suspect that it may associated with the value of c.

3.

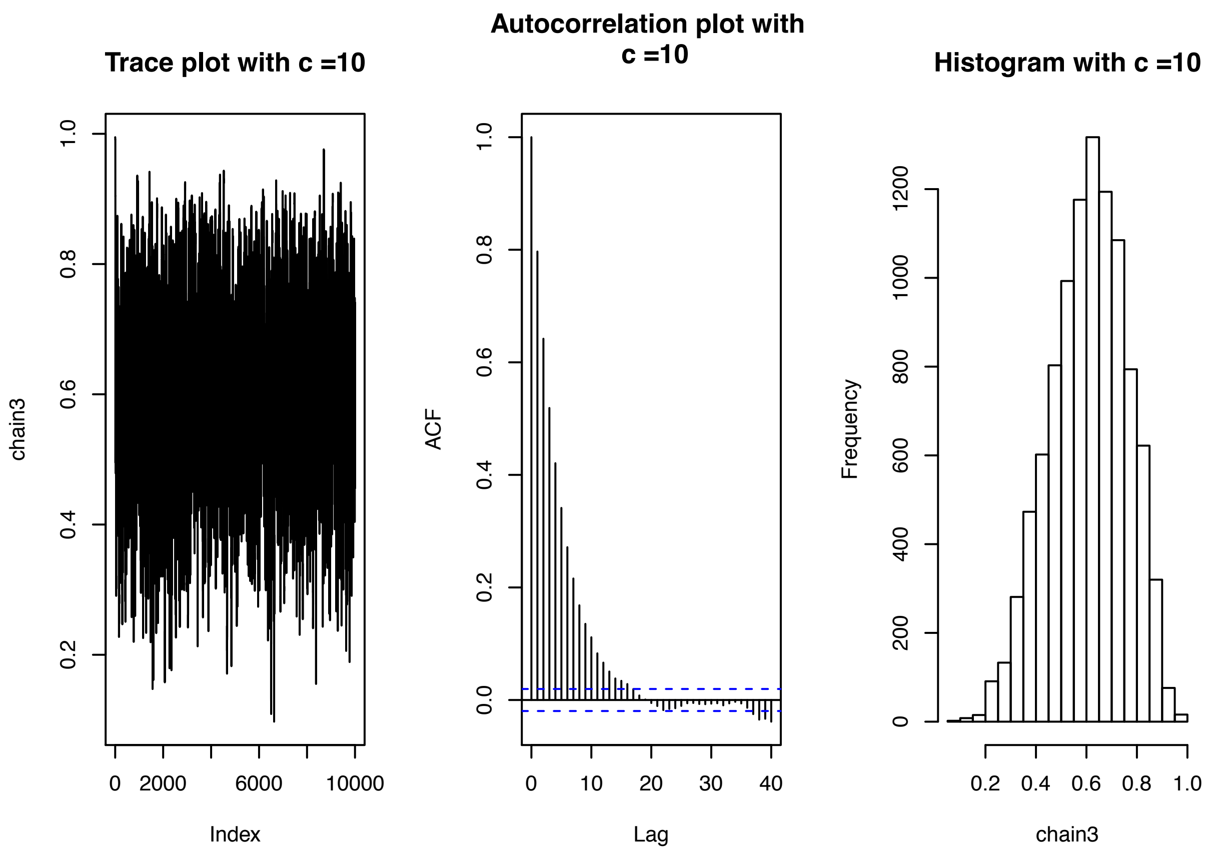
```r
alpha <- 6
beta <- 4
startvalue <-runif(1)
par(mfrow=c(1,3))

# c= 0.1
chain1<-run_betaMH(startvalue, c=0.1, iterations = 10000)
plot(chain1,  type = 'l', main ='Trace plot with c =0.1')
acf(chain1, main ='Autocorrelation plot with \n c =0.1')
hist(chain1, main ='Histogram with c =0.1')
```



```r
# c = 2.5
chain2<-run_betaMH(startvalue, c= 2.5, iterations = 10000)
plot(chain2,  type = 'l', main ='Trace plot with c =2.5')
acf(chain2, main ='Autocorrelation plot with \n c =2.5')
hist(chain2, main ='Histogram with c =2.5')
```



```r
# c = 10
chain3<-run_betaMH(startvalue, c=10, iterations = 10000)
plot(chain3,  type = 'l', main ='Trace plot with c =10')
acf(chain3, main ='Autocorrelation plot with \n c =10')
hist(chain3, main ='Histogram with c =10')
```



We rerun the sampler with c = 0.1, c = 2.5, and c = 10 and plot s, autocorrelation plots and histrograms for these samplers. If we see these graphs all together, the one thing that we can see obviously is the differences in autocorrelation plots. When c = 0.1, the autocorrelation plot shows the most ACF compared with c = 2.5 and c = 10. From the histrgrams, we can see that the histrogram with c = 0.1 looks different from the histrograms with the histograms of c = 2.5 and c = 10 and the histrograms with c = 2.5 and c = 10 seems similar to the target distribution. From autocorrelation plots, the the graph with c = 2.5 is less lagging and has less ACF than c = 10, and the histrograms seems very similar to the target distribution. Thus I think the sampler with c = 2.5 is most effective at drawing from the target distribution.

In addition to graphical tools, we also can check the result from Kolmogorov-Smirnov statistic for these three samplers.

```
# c = 0.1
ks.test(chain1,x)
```

```
## Warning in ks.test(chain1, x): p-value will be approximate in the presence
## of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  chain1 and x
## D = 0.059886, p-value = 5.551e-16
## alternative hypothesis: two-sided
```

```
# c = 2.5
ks.test(chain2,x)
```

```
## Warning in ks.test(chain2, x): p-value will be approximate in the presence
## of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  chain2 and x
## D = 0.015034, p-value = 0.2084
## alternative hypothesis: two-sided
```

```r
# c = 10
ks.test(chain3,x)
```

```
## Warning in ks.test(chain3, x): p-value will be approximate in the presence
## of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  chain3 and x
## D = 0.032842, p-value = 4.136e-05
## alternative hypothesis: two-sided
```

From the results, since only the p-value of sampler with c = 2.5 is greater than 0.05, we can not reject the null hypothesis. So we can conlucde that the sampler with c = 2.5 follow the target distribution, $Beta(6, 4)$.

As c increases, we can get the distribution that follows closer to the target distribution efficiently. However, it is not always true that large c is optimal option because it can increases laggs. To increase efficiency and to optimize the process, we could include the appropriate numbers of draws needed for thining and burn-in instead of simply increasing c.

Gibbs sampling

1.

Algorithms

From the marginal distribution, we can find the density function, cumulative density function and inverse cumululative density function. Here, B is a known positve constant.

Marginal ditribution

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

Density function

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

Cumulative density function

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

Inverse of Cumulative density function

$$F^{-1}(p_1|y) = (\frac{-1}{y}) * log(1 - p_1(1 - e^{-By}), 0 \leq p_1 \leq 1$$
$$F^{-1}(p_1|x) = (\frac{-1}{x}) * log(1 - p_1(1 - e^{-Bx}), 0 \leq p_1 \leq 1$$

We use inverse transform sampling to generate the sample from the conditional $p(x|y)$, so we use random number from uniform(0,1) to get x using `Inverse of Cumulative density function` as we define above. Same method will apply to the case of y.

1. Set first x with a random number from uniform(0,1) and first y with a random number from uniform(0,1).
2. Generate new x using inverse function with given y and new y using inverse function with given x.
3. In the next step, new x and new y in the previous step will be the given x and the given y to generate new x and new y.
4. Repeat 2-3 to finish all T times and get final matrix with the sample.

Here, we set thin value as 10 (default).

NOTE

Python codes and the solutions for later problems are written in jupyter notebook with pdf form. Those pages will attach from next page.

Gibbs Sampling

November 3, 2017

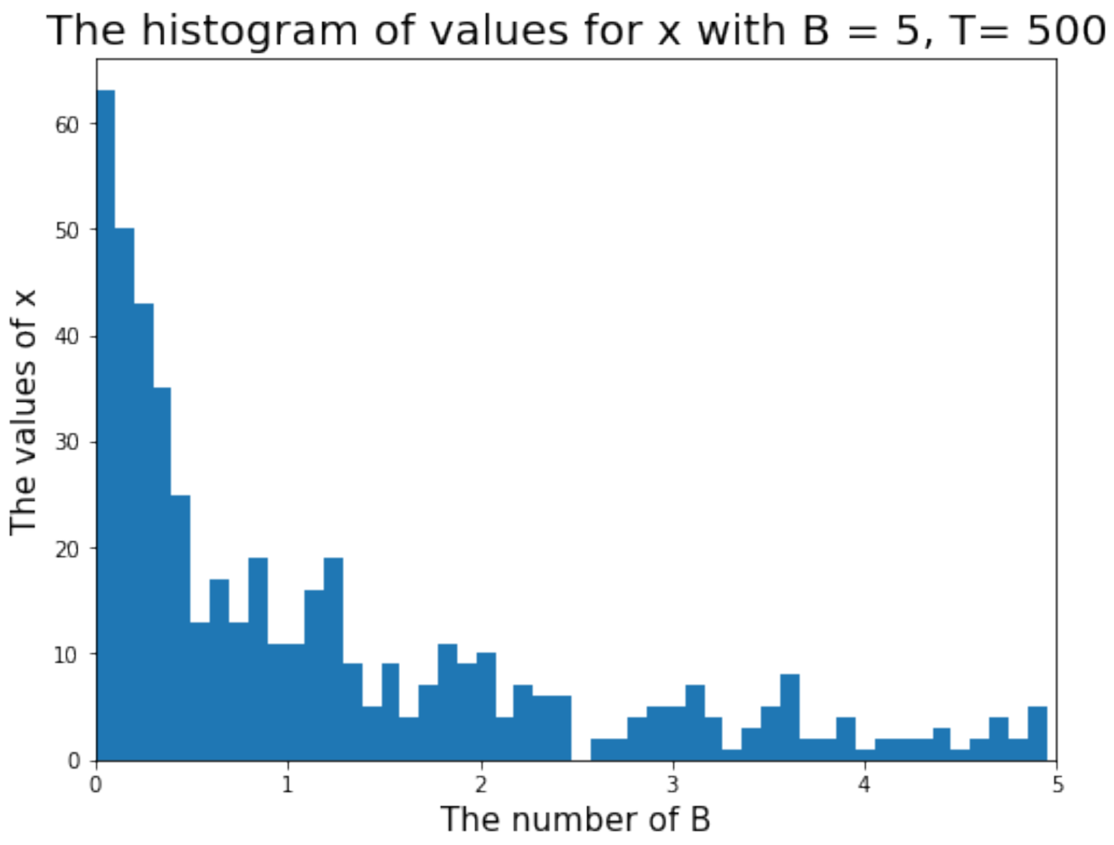
1 Python codes

```
In [1]: import numpy as np
        import random
        import matplotlib.pyplot as plt
        %matplotlib inline

        def pxy(a, b):
            '''
            Marginal distribution function.
            This function can be used into both p(x|y) and p(y|x)
            '''
            return a * np.exp((-a)*b)

        def densityfxy(a, b, B):
            '''
            Density function in an interval(0,B).
            This function can be used into both f(x|y) and f(y|x).
            '''
            return (b*np.exp((-a)*b))/(1-np.exp((-B)*b))

        def cdffxy(a, b, B):
            '''
            Cumulative density function in an interval(0,B).
            This function can be used into both F(x|y) and F(y|x).
            '''
            return (1-np.exp((-a)*b))/(1-np.exp((-B)*b))

        def inversefxy(a ,B):
            '''
            Inverse function in an interval(0,B).
            This function can be used into both F^-1(p1|y) and F^-1(p1|x),
            where 0 < p1 < 1.
            Let p1 be the r.v from uniform distribution.
            '''
            p1 = random.uniform(0,1)
            return (-1/a)*(np.log(1-p1*(1-np.exp((-B)*a))))
```

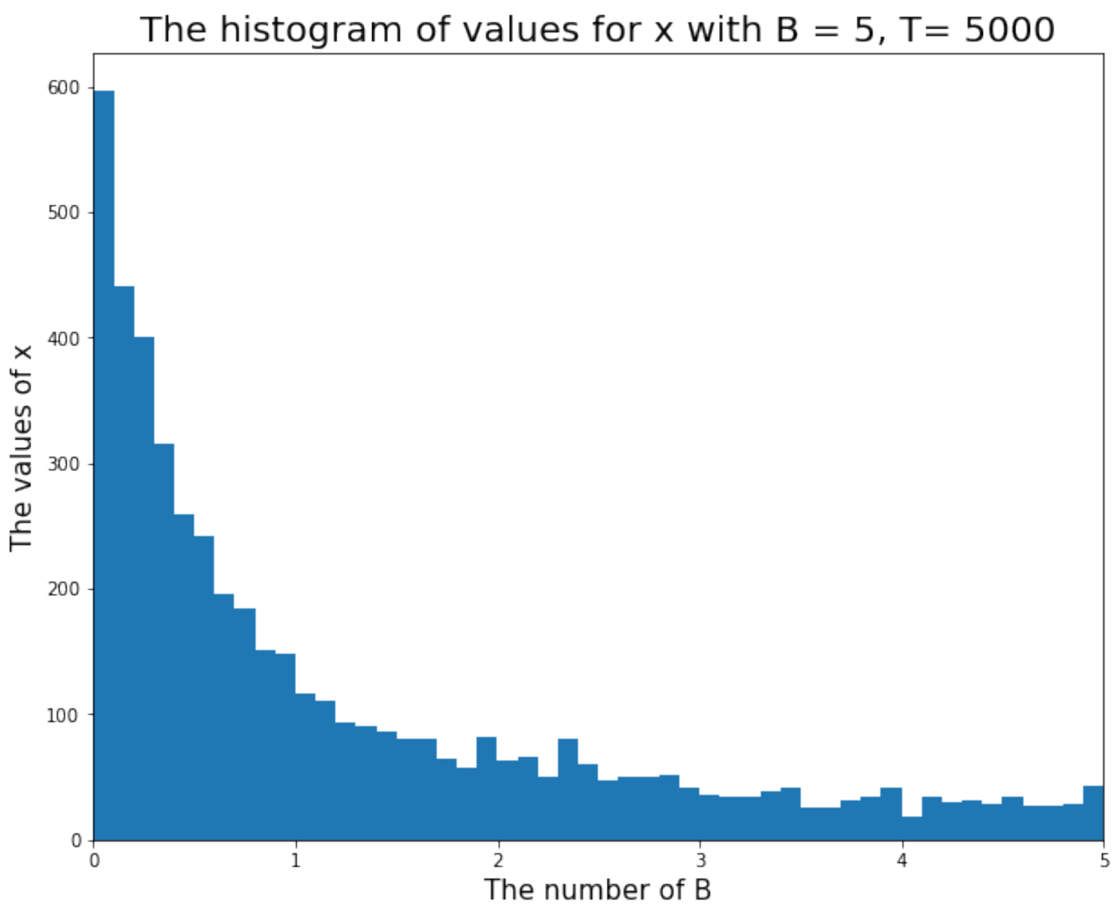
```
In [2]: # Gibbs sampling method
        def gibbs(N, B, thin):
            matrix = np.zeros((N, 3))
            matrix[:,0] = np.arange(1,N+1)
            # select random numbers from uniform distribution
            x = random.uniform(0,1)
            y = random.uniform(0,1)
            for i in range(0, N):
                for j in range(0, thin):
                    # generate updated x value
                    x = inversefxy(y,B)
                    # generate updated y value
                    y = inversefxy(x,B)
                    # assign updated x,y values to column 2,3
                    matrix[i,1:3] = [x,y]
            return matrix
```

2 2

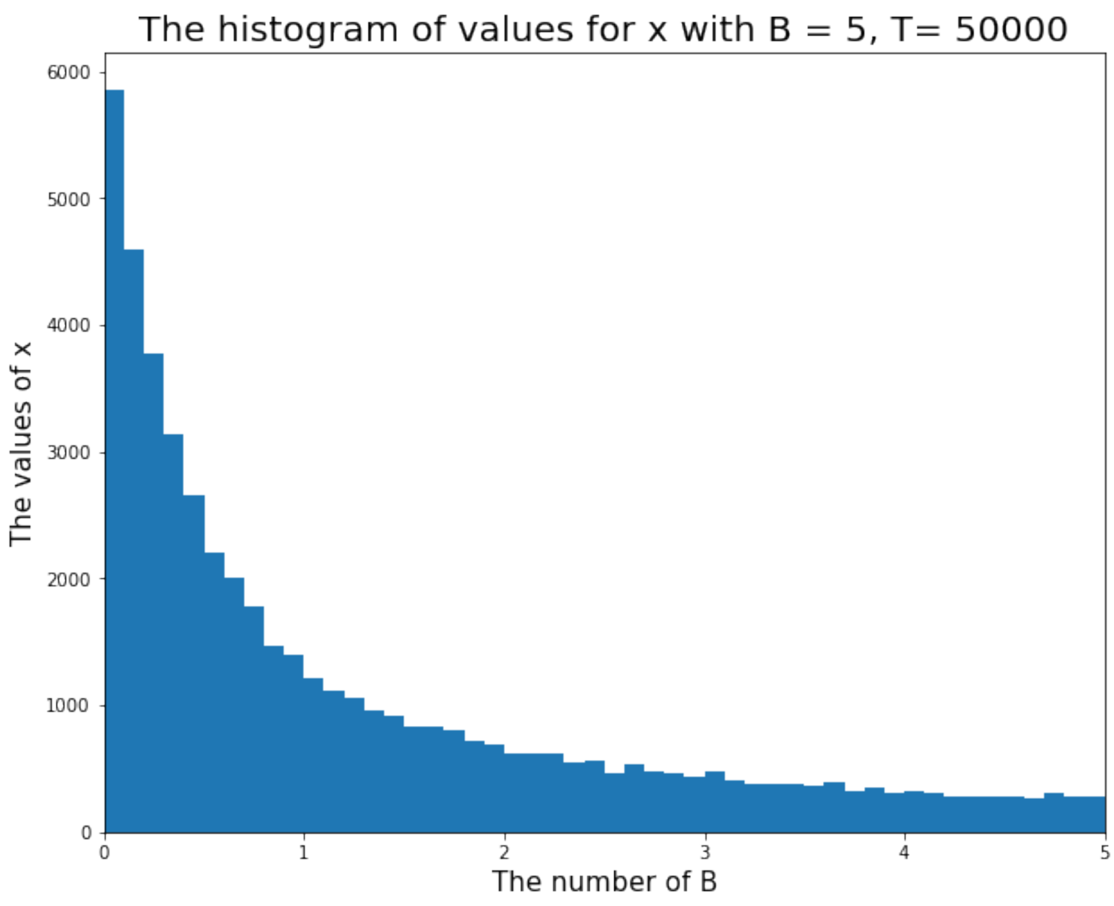
```
In [3]: # Plot histogram with T = 500
        N, B, thin = 500, 5, 10
        result1 = gibbs(N, B, thin)
        plt.subplots(figsize=(8, 6))
        plt.hist(result1[:,1], bins = 50)
        plt.title ('The histogram of values for x with B = 5, T= 500',
                   fontsize= 20)
        plt.xlabel('The number of B', fontsize= 15)
        plt.ylabel('The values of x', fontsize= 15)
        plt.xlim(0,5)
        plt.show()
```



```
In [4]: # Plot histogram with T = 5000
        N, B, thin = 5000, 5, 10
        result2 = gibbs(N, B, thin)
        plt.subplots(figsize=(10, 8))
        plt.hist(result2[:,1],bins = 50)
        plt.title ('The histogram of values for x with B = 5, T= 5000',
                   fontsize= 20)
        plt.xlabel('The number of B',fontsize= 15)
        plt.ylabel('The values of x',fontsize= 15)
        plt.xlim(0,5)
        plt.show()
```



```
In [5]: # Plot histogram with T = 50000
        N, B, thin = 50000, 5, 10
        result3 = gibbs(N, B, thin)
        plt.subplots(figsize=(10, 8))
        plt.hist(result3[:,1],bins = 50)
        plt.title ('The histogram of values for x with B = 5, T= 50000',
                   fontsize= 20)
        plt.xlabel('The number of B',fontsize= 15)
        plt.ylabel('The values of x',fontsize= 15)
        plt.xlim(0,5)
        plt.show()
```



From these three histograms, we can see that as T inceases, the histograms become smoother.

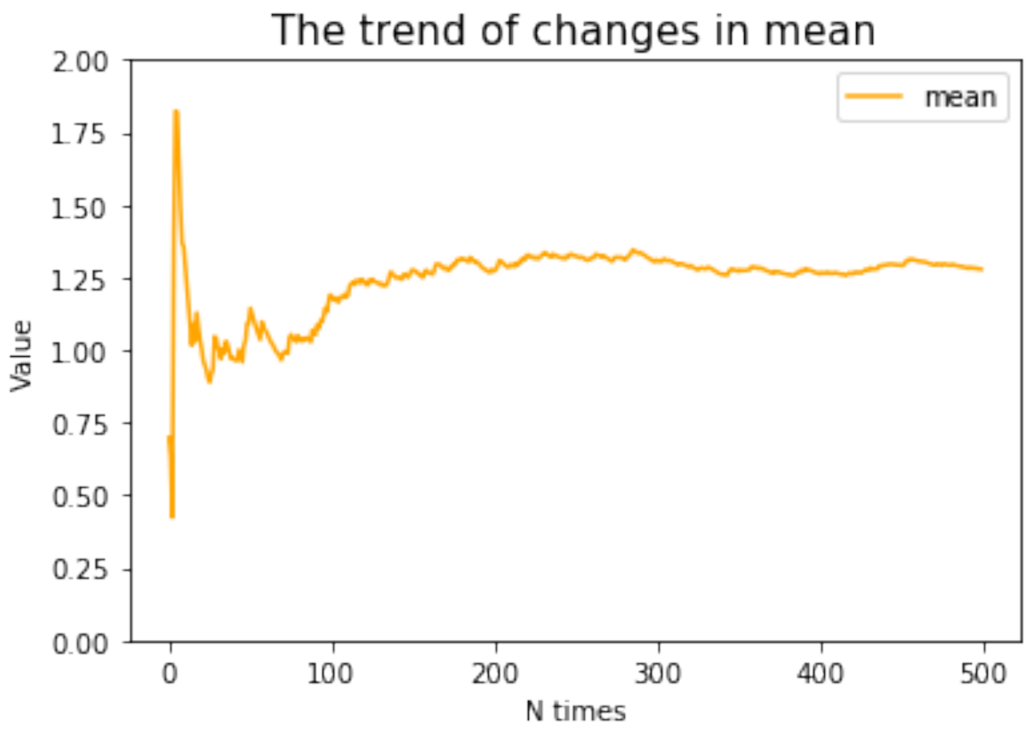
3 3.

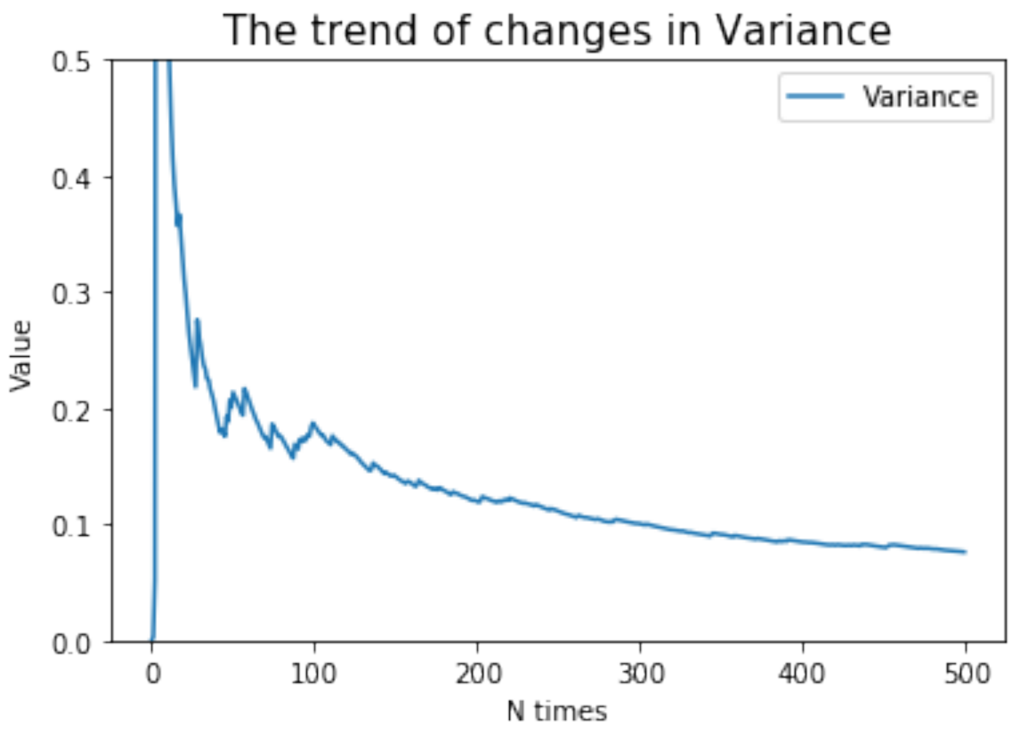
```
In [18]: random.seed(200)
         result500 = gibbs(500,5,10)
         result5000 = gibbs(5000,5,10)
         result50000 = gibbs(50000,5,10)

         # calulate the means from each iterations.
         estimated_expectation_500 = np.mean(result500[:,1])
         estimated_expectation_5000 = np.mean(result5000[:,1])
         estimated_expectation_50000 = np.mean(result50000[:,1])

In [19]: estimated_expectation_500

Out[19]: 1.2776584241390458

In [20]: estimated_expectation_5000
```

```
Out[20]: 1.2662195687732802

In [21]: estimated_expectation_50000

Out[21]: 1.2584644918562236
```

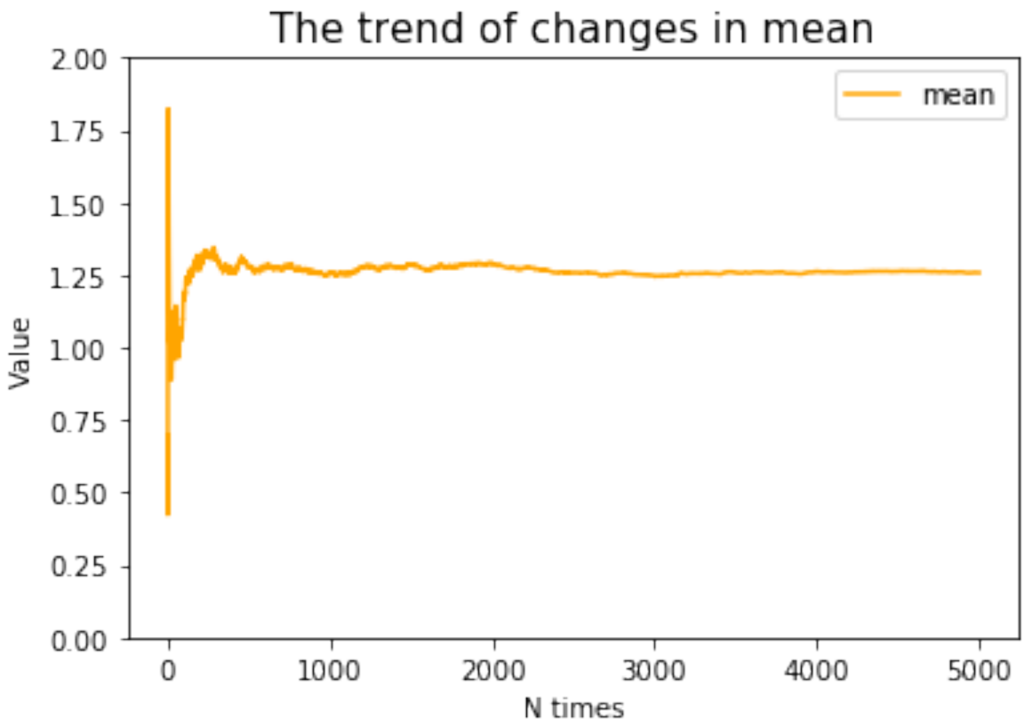
As we can see from the results, the expection changes 1.277658 to 1.266219 to 1.258464 by using 500, 5000 and 5000 samples respectively. The expectation become stable as T increases like 500, 5000 and 50000 samples from the sampler. We can see this trend by using graphs.

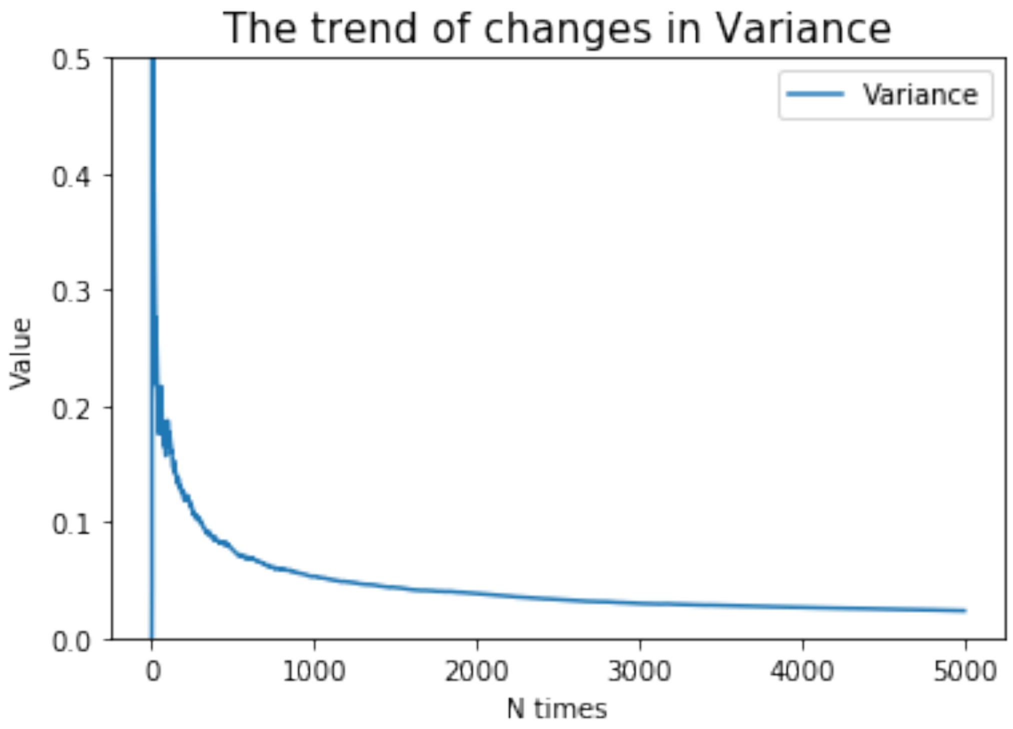
```
In [25]: # define function do simulation and make graph at the same time
         # the function takes T times as argument
         def simulation_and_graph(N):
             simulation_matrix = np.zeros((N,3))
             simulation_matrix[:,0] = np.arange(N)
             simulation = gibbs(N, 5, 10)
             for i in range(N):
                 mean = np.mean(simulation[0:i+1,1])
                 var =  np.var(simulation[0:i+1,1])/np.sqrt(i+1)
                 simulation_matrix[i,1:3] = [mean, var]

             plt.subplots(figsize=(6, 4))
             plt.plot(simulation_matrix[:,1], label = 'mean', color = 'orange')
             plt.title ('The trend of changes in mean',fontsize= 15)
             plt.xlabel('N times',fontsize= 10)
             plt.ylabel('Value',fontsize= 10)
             plt.legend(loc='upper right')
             plt.ylim(0,2)
             plt.legend

             plt.subplots(figsize=(6, 4))
             plt.plot(simulation_matrix[:,2], label = 'Variance')
             plt.title ('The trend of changes in Variance',fontsize= 15)
             plt.xlabel('N times',fontsize= 10)
             plt.ylabel('Value',fontsize= 10)
             plt.legend(loc='upper right')
             plt.ylim(0,0.5)
             plt.legend

In [29]: # T = 500
         random.seed(200)
         simulation_and_graph(500)
```


The trend of changes in mean
mean
Value
N times
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
100
200
300
400
500

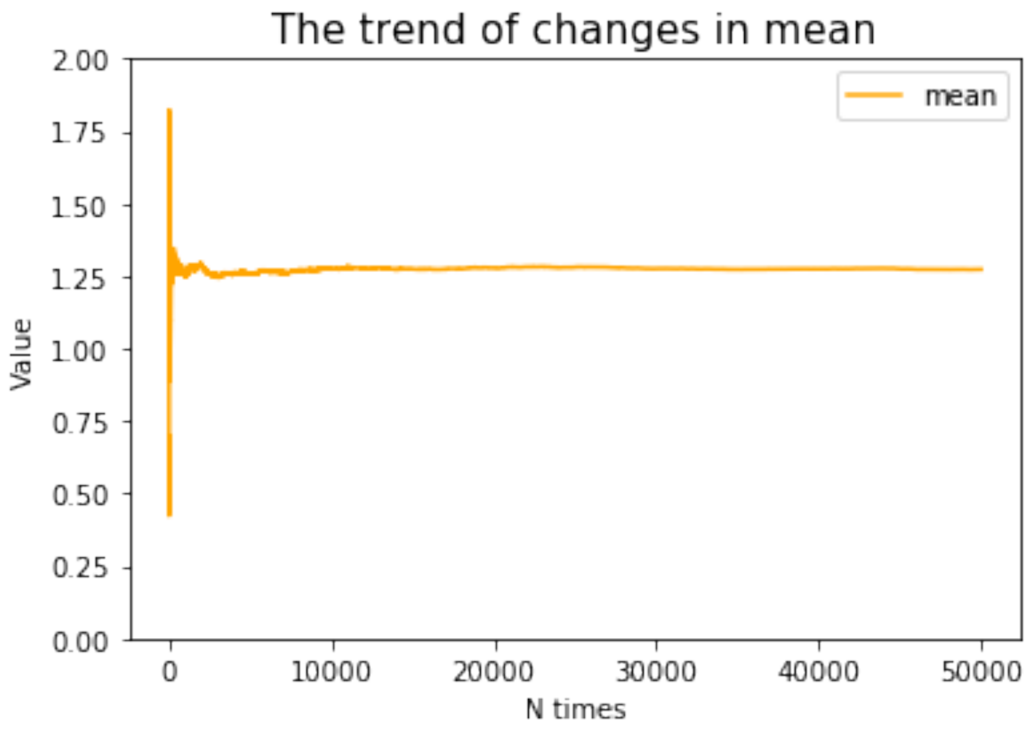

The trend of changes in Variance
Variance
Value
N times
0.5
0.4
0.3
0.2
0.1
0.0
0
100
200
300
400
500

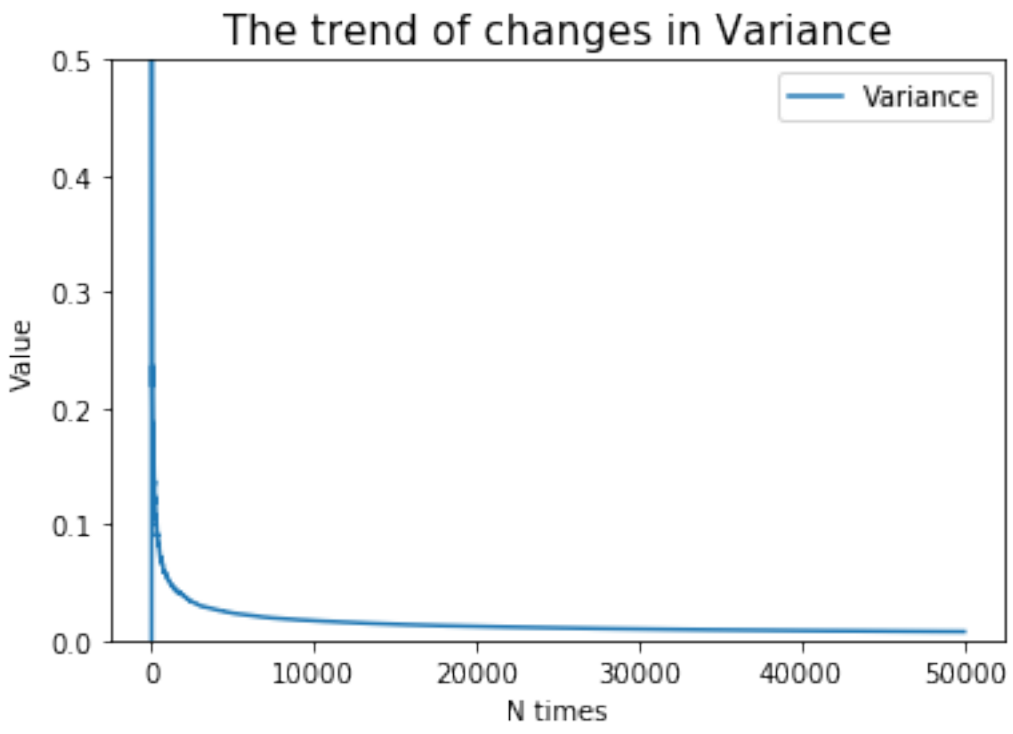
```
In [30]: # T = 5000
         random.seed(200)
         simulation_and_graph(5000)
```





```
In [31]: # T = 500
         random.seed(200)
         simulation_and_graph(50000)
```





As you see in the above graphs that shows how means and variance change as T increases, the trand of mean and varianve varies a lot at first and becomes stable and converges. Especially variance becomes small as T increases.

```
In [ ]:
```

K-means

1.

Algorithms

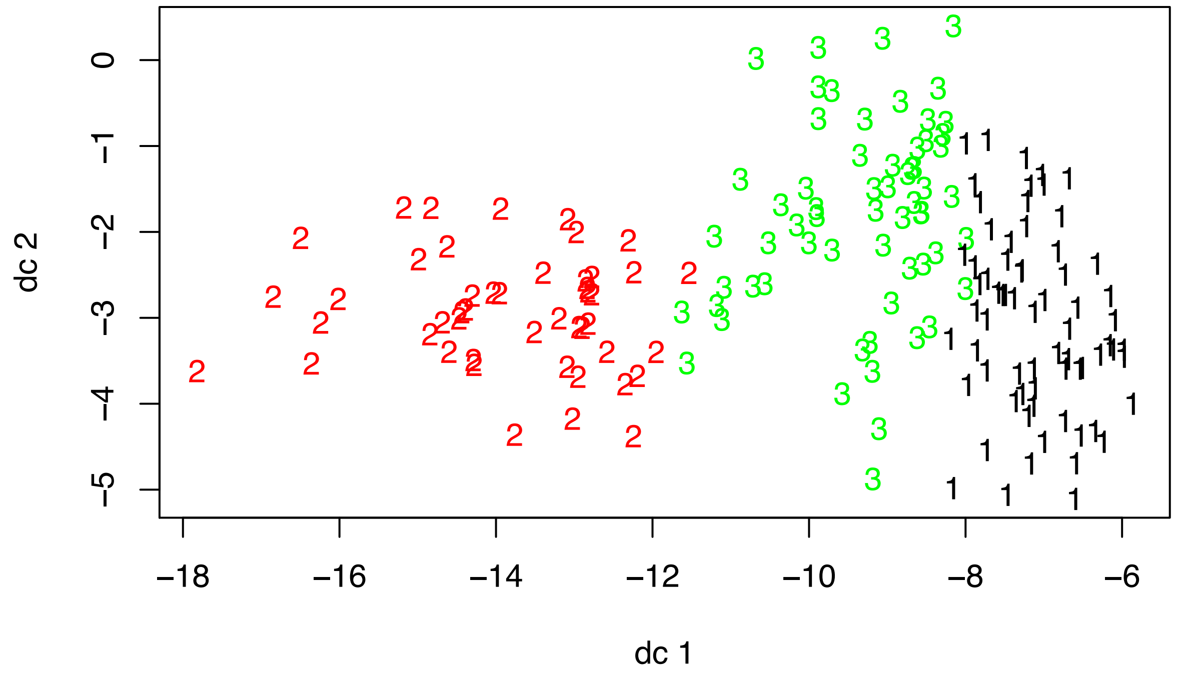
1. Select the number of clusters and randomly select the first random centers from the data.
2. For every points in the data, calculate the euclidean distance with the cluster centers, choose the minimum distance and assign into the according group.
3. Recalculate the means of cluster centers within each groups and set as new cluster centers.
4. Compare the previous cluster centers and new cluster centers, if the centers changed then repeat until the centers do not change.
5. Get and return the final cluster result.

R codes

```r
data(wine, package = 'rattle.data')
data.wine <- wine[-1]
data.iris <- iris[-5]

# the number of cluster groups
cluster_num = 3

mykmeans<-function(data, cluster_num){
  data <- as.matrix(data)
  rows <- nrow(data)
  cols <- ncol(data)
  center <- matrix(0, cluster_num, cols)
  resultmatrix <- matrix (0, rows, 2)
  eucliddist <- matrix(0, 1, cluster_num)
  # randomly select the first sample
  random <- sample(rows, cluster_num, replace = FALSE)
  ## STEP 1
  # update center matrix
  for (i in 1:cluster_num){
    center[i,] <- data[random[i],] # use random to get first center points
  }
  previouscenter <- matrix(0, cluster_num, cols)
  # within this whle loop, compare current centers and previous centers.
  # if those points are changed, we do STEP2 and STEP3 again.
  # if those points are not changes, then we return the final results.
    while(sum(abs(previouscenter)) != sum(abs(center))){
  ## SETP 2
  # update euclidian distance matrix for every rows
  for (i in 1:rows){
    for (j in 1:cluster_num){
      # calculate euclidean distance measure
      eucliddist[,j] <- sqrt(sum((data[i,]-center[j,])^2))
      }
      # assign every rows to one of cluster groups and record every calculated distance
      resultmatrix[i,] = c(min <- which.min(eucliddist),
                        eucliddist[min <- which.min(eucliddist)])
  }
  ## STEP 3
    previouscenter <- center
    # calculate a new center for every cluster group
    for (i in 1:cluster_num){
      index = which(resultmatrix[,1]==i, arr.in=TRUE)
      center[i,] = colMeans(data[index,])
    }
  }
  return(resultmatrix)
}
```

Wine data

```
kmeans_wine <- mykmeans(data.wine, 3)
plotcluster(data.wine, as.list(kmeans_wine[,1]))
```



As we can see in the above cluster plot, it is seperated into three groups but the boundary of each cluster is quite blurry. Some data points overlaps each other. So the clusters here doesn't seem to be well-seperated.

Let's develope a method to quantify how well my algorithm's clusters correspond to the three wine types. I simply want to compare with the original dataset. In the dataset, there is `Type` variable and it is also clustered into three groups. So we can compare the cluster result with original data, find and count the matched pairs and divide by the number of whole data points. It is simple ratio and not a complex method to compute.

```
correct_ratio = sum(kmeans_wine[,1] == wine[1])/length(kmeans_wine[,1])
```

So if we canculate with above cluster results, then this correct ratio will be 0.1685393. It is less than 20%, so we can not say that it is good clustering and it explains why we have overlaps in cluster plot.

Second method is using built-in R methid `randIndex`. This method computes the Rand index, a measure of the similarity between two clusterings.

```
table <- table(wine$Type, kmeans_wine[,1])
ari<- randIndex(table); ari
```

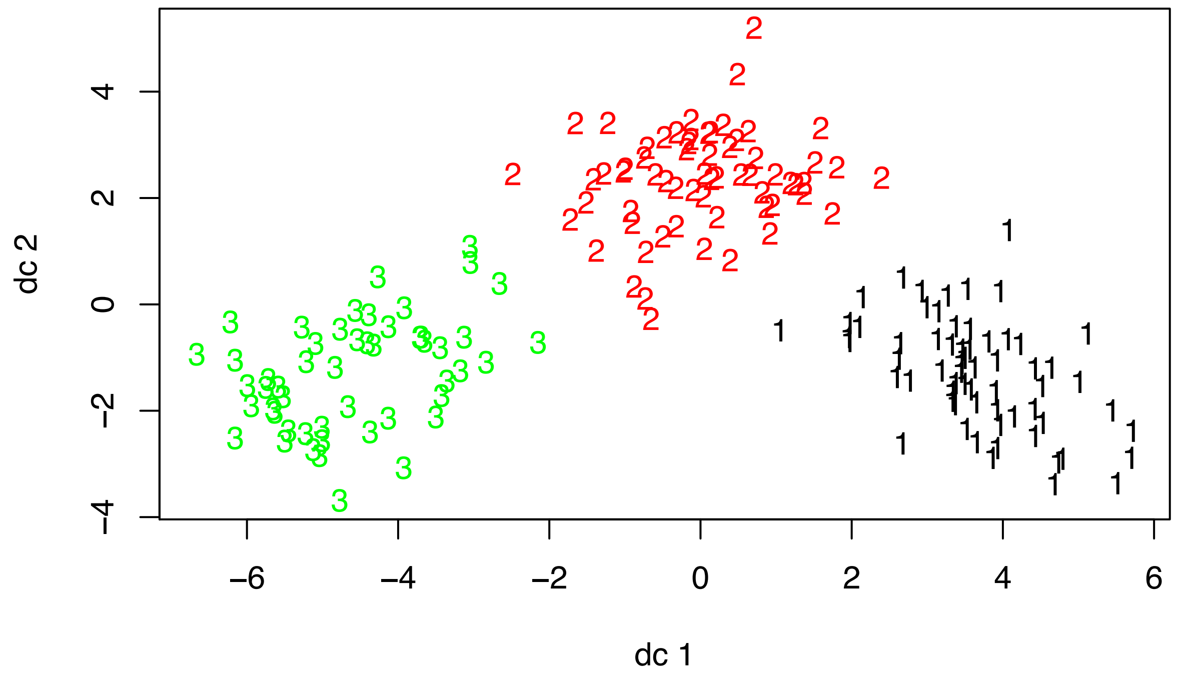
```
##       ARI
## 0.3711137
```

The results is 0.3711137 and it is not high value to say two clustertings are similar. Thus, my kmeans algorigtm does not seperate clusters well.

```
#scaled
scaled.wine = scale(wine[-1])
```

Now we will use scale function to scale data. This function will calculate the mean and standard deviation of the entire vector, then scale each element by those values by subtracting the mean and dividing by the sd. So it minimize the effect of the outliers in the dataset. If we do not scale this data observations, then outliers or unusual data observations will affect and distort the results from whatever algorithms we use to analyze the data. Thus if we use scale function here, our clustering results will be more precise and have less errors.

```r
kmeans_wine.scaled <- mykmeans(scaled.wine, 3)
plotcluster(scaled.wine, as.list(kmeans_wine.scaled[,1]))
```



This time we used scaled `wine` data and cluster plot seems work well. There is no overlapping at the boundaries of cluster groups. So the clusters here seem to be well-seperated into three groups.

We can calulate correct ratio.

```r
correct_ratio1 = sum(kmeans_wine.scaled[,1] == wine[1])/length(kmeans_wine.scaled[,1])
```

So if we canculate with above cluster results, then this correct ratio will be 0.9550562. It is very close to 1, so we can say that it is good clustering. Also compare to previous result with unscaled data, it increases almost two times.

We also can use second method `randIndex`.

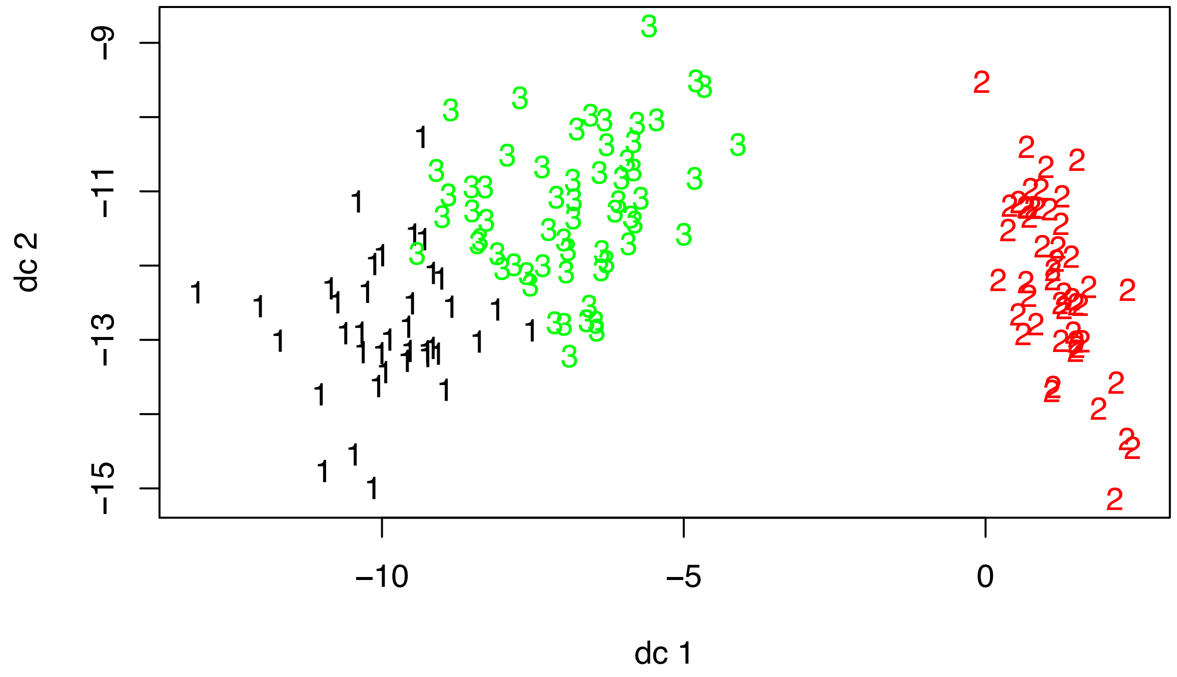
```r
table1 <- table(wine$Type, kmeans_wine.scaled[,1])
ari1<- randIndex(table1); ari1
```

```
##       ARI
## 0.8635988
```

The results is 0.8635988 and it is high value to say two clustertings are similar. Thus, my kmeans algorigtm seperates clusters well with scaled `wine` data.

Iris data

```
kmeans_iris <- mykmeans(data.iris, 3)
plotcluster(data.iris, as.list(kmeans_iris[,1]))
```



As we can see in above cluster plot with unscaled iris data, it is seperated into three groups but the boundary of cluster 3 and 2 are vague. Here also, Some data points overlaps each other. So the clusters here doesn't seem to be well-seperated.

In the iris dataset, there is `Species` variable and it is also clustered in three groups. So we can use `simple ratio` that we define earlier.

```
iris$Species<-as.numeric(iris$Species, levels = sort(iris$Species))
correct_ratio3 = sum(kmeans_iris[,1] == iris[5])/length(kmeans_iris[,1])
```

So if we canculate with above cluster results, then this correct ratio will be 0.0933333. It is less than 10%, so we can not say that it is good clustering and it explains why we have overlaps in cluster plot.

We can use second method, built-in R method `randIndex`.

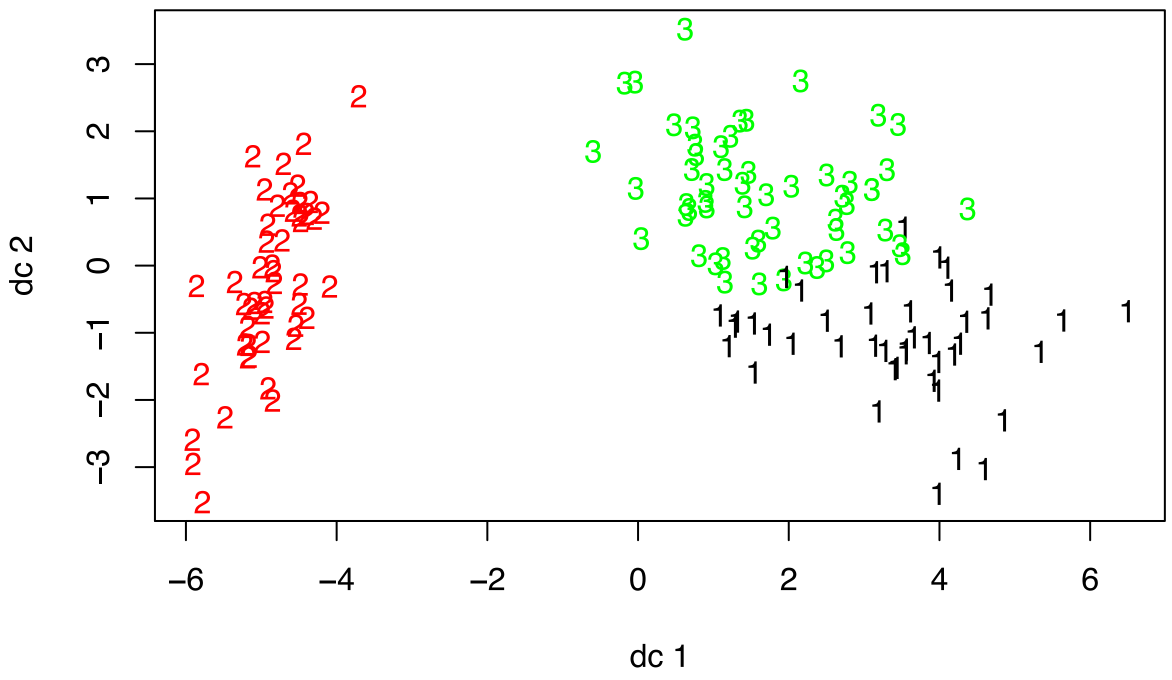
```
table3 <- table(iris$Species, kmeans_iris[,1])
ari3<- randIndex(table3); ari3
```

```
##       ARI
## 0.7302383
```

The results is 0.7302383 and it is greater than 0.5 but not a high value to say two clustertings are similar. Thus, my kmeans algorigtm does not seperate clusters well with unsacled iris data.

```
#scaled
scaled.iris = scale(iris[-5])
```

```
kmeans_iris.scaled <- mykmeans(scaled.iris, 3)
plotcluster(scaled.iris, as.list(kmeans_iris.scaled[,1]))
```



This time we used scaled `iris` data and cluster plot stil does not seem work well. There is overlapping at the boundaries of cluster group 1 and 3 and the boundary of 1 and 2 are very close. So the clusters here does not seem to be well-seperated into three groups.

We can calulate correct ratio.

```
correct_ratio4 = sum(kmeans_iris.scaled[,1] == iris[5])/length(kmeans_iris.scaled[,1])
```

So if we canculate with above cluster results, then the correct ratio will be 0.1133333. It is less than 15%, so we can not say that it is good clustering. Also compare to previous result with unscaled data, it increases almost two times.

We also can use second method `randIndex`.

```
table4<- table(iris$Species, kmeans_iris.scaled[,1])
ari4<- randIndex(table4); ari4
```

```
##       ARI
## 0.5923326
```

Like previous results, ARI is 0.5923326 and it is greater than 0.5 but not a high value to say two clustertings are similar. Thus, my kmeans algorigtm does not seperate clusters well with scaled iris data. So the scale method does not helpful to cluster `iris` data.