STAT 37810 Group Final Project Report

Boxin Zhao, Yilun Dai, Zhen Dai, Zhengyang Fang

11/04/2018

This is the group final report. Question 1 & 3 are completed with R Markdown, question 2 is completed with Python Jupyter notebook. The first part is based on the code of Yilun Dai, the second part is based on the code of Zhen Dai, and the third part is based on the code of Zhengyang Fang. This final report is written by Boxin Zhao, who checked all codes and corrected mistakes before merging them together.

Note that all four of us wrote our own final reports on three questions. Individual works can be found in files named by individual names. After compeleting individual works, we split the job to make a group final report. Specifically, Yilun Dai is responsible for part one, Zhen Dai for Part two, Zhengyang Fang for part three, and Boxin Zhao for proofreading all codes and combining them together. Everyone fully participated in every question and made their own solutions, and we only splited the job when making group report.

Part I Metropolis-Hastings

This part is originally written by Yilun Dai, and reivised by Boxin Zhao

1. algorithm and implementation

- Functions

Target function is a beta distribution with shape1 = 6 and shape2 = 4 Proposal function generates a ϕ_{new} from a beta distribution with shape1 = cϕ_{old} and shape2 = c(1 - ϕ_{old})

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

proposalfunction <- function(c, shape){

  return(rbeta(1, c * shape, c * (1 - shape)))
}
```

- Algorithm

let t(x) stand for the target function

let p(x) stand for the proposal function

let q(x) stand for the acceptance rate funtion

Given x_i, generate Y_i ~ p(y|x_i)

Then the acceptance probability for $X_{i+1} = Y_i$ is q(x).

The probability for $X_{i+1} = X_i$ is 1 - q(x).

Since beta distribution is asymmetric, we need to implement a correction factor when calculating the acceptance probability.

acceptance probability $= \frac{t(p(x_i))}{t(x_i)} \times correction$

where correction $= \frac{\phi(x_i, c \times p(x_i), c \times (1-p(x_i))}{\phi(p(x_i), c \times p(x_i), c \times (1-p(x_i))}$

when p(x) >= 1, then jump to the new Y_i as well.

- Code

```
run_metropolis_MCMC <- function(c, startvalue, iterations){
    chain <- rep(0, iterations)
    chain[1] <- startvalue
    for (i in 1:iterations){
        # generate candidate Y_{i} using proposal function
        proposal <- proposalfunction(c, chain[i])
        # calculate the correction factor
        correction <- dbeta(chain[i], c * proposal, c * (1- proposal))/
          dbeta(proposal, c * proposal, c * (1 - proposal))
        # calculate the acceptance rate
        probab <- (target(proposal) / target(chain[i]) )* correction
        # generate a uniform random value
        # and compare it with the acceptance rate.
        # If acceptance rate >= this value,
        # then we have X_{i+1} = Y_{i}.
        if (runif(1) <= probab){
            chain[i+1] = proposal
        }else{
          # Otherwise, we keep x_{i} as the new X_{i+1} = x_{}
            chain[i+1] = chain[i]
        }
    }
    return(chain)
}
```

2. initial run: c = 1

c = 1, initial runs = 10000, generate a start value from a uniform distribution on [0, 1], with no burn-in time.

From the histogram we can see that randomly generated values from the target is less centered than the sample.

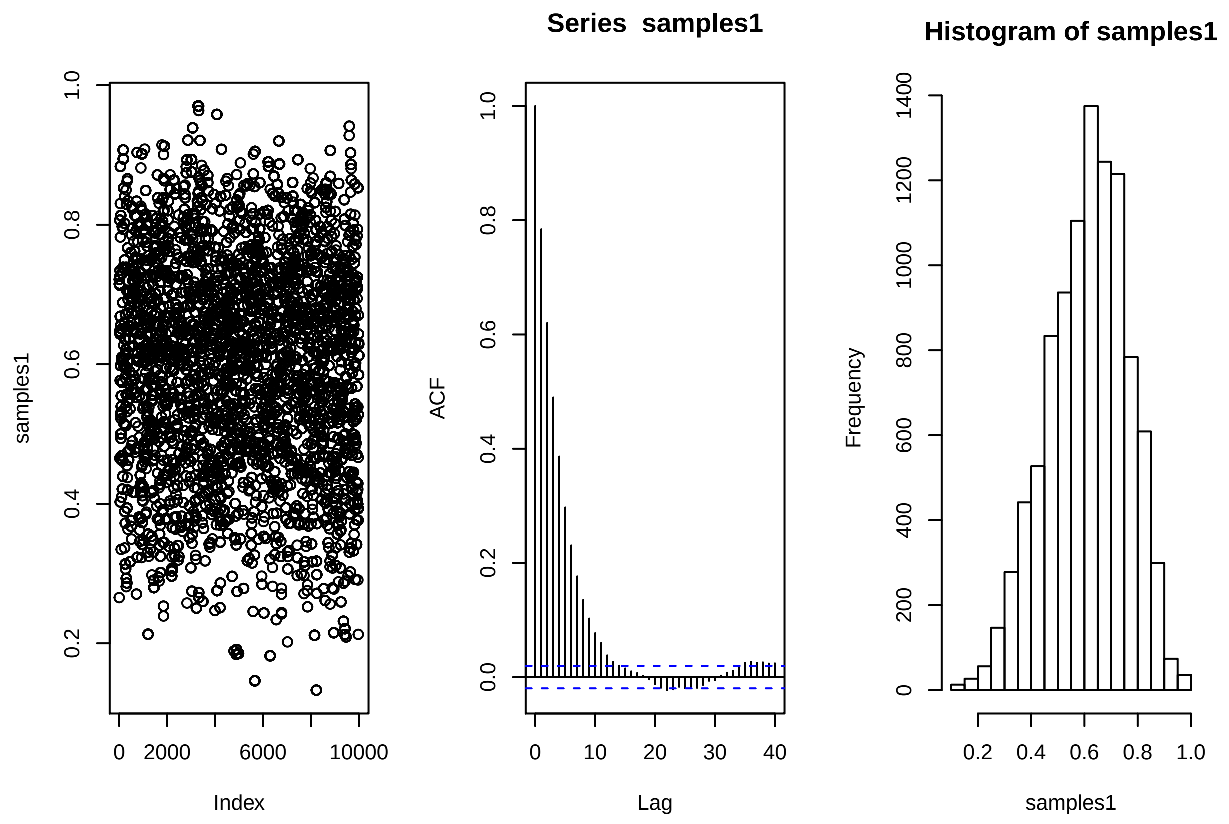
In ks-test, D-value is greater than critical value at a confidence level of 0.95.

```
set.seed(1)
start <- runif(1, 0, 1)
samples1 <- run_metropolis_MCMC(1, start, 10000)
acceptance_rate1 <- 1 - mean(duplicated(samples1))
print(c("acceptance rate is: ", acceptance_rate1))
```

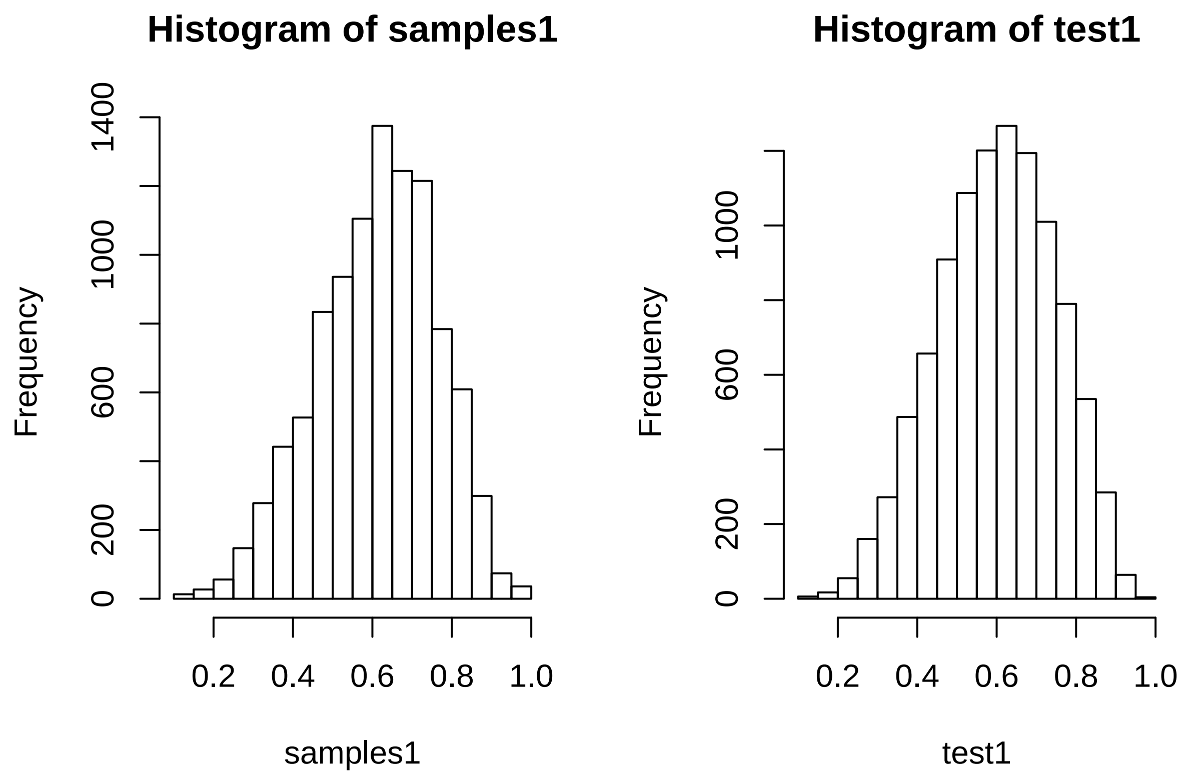
```
## [1] "acceptance rate is: " "0.257374262573743"
```

Provide a trace plot of this sampler and an autocorrelation plot, as well as a histogram of the draws.

```
 par(mfrow=c(1,3))  #1 row, 3 columns
 plot(samples1); acf(samples1); hist(samples1)  #plot commands
```



```r
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test1 = rbeta(10000,6,4)
par(mfrow=c(1,2))
hist(samples1)
hist(test1)
```



- the Kolmogorov Smirnov statistic

```
ks.test.critical.value(10000, 0.95)
```

```
## [1] 0.013581
```

```
ks.test(samples1, pbeta, 6, 4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  samples1
## D = 0.05242, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

3. runs

(a) without adding burn-in time

(1) c = 0.1

When c = 0.1, acceptance rate ~ 0.04.

From the histogram we can see that the sample does not have the same shape as the randomly generated values from the target.

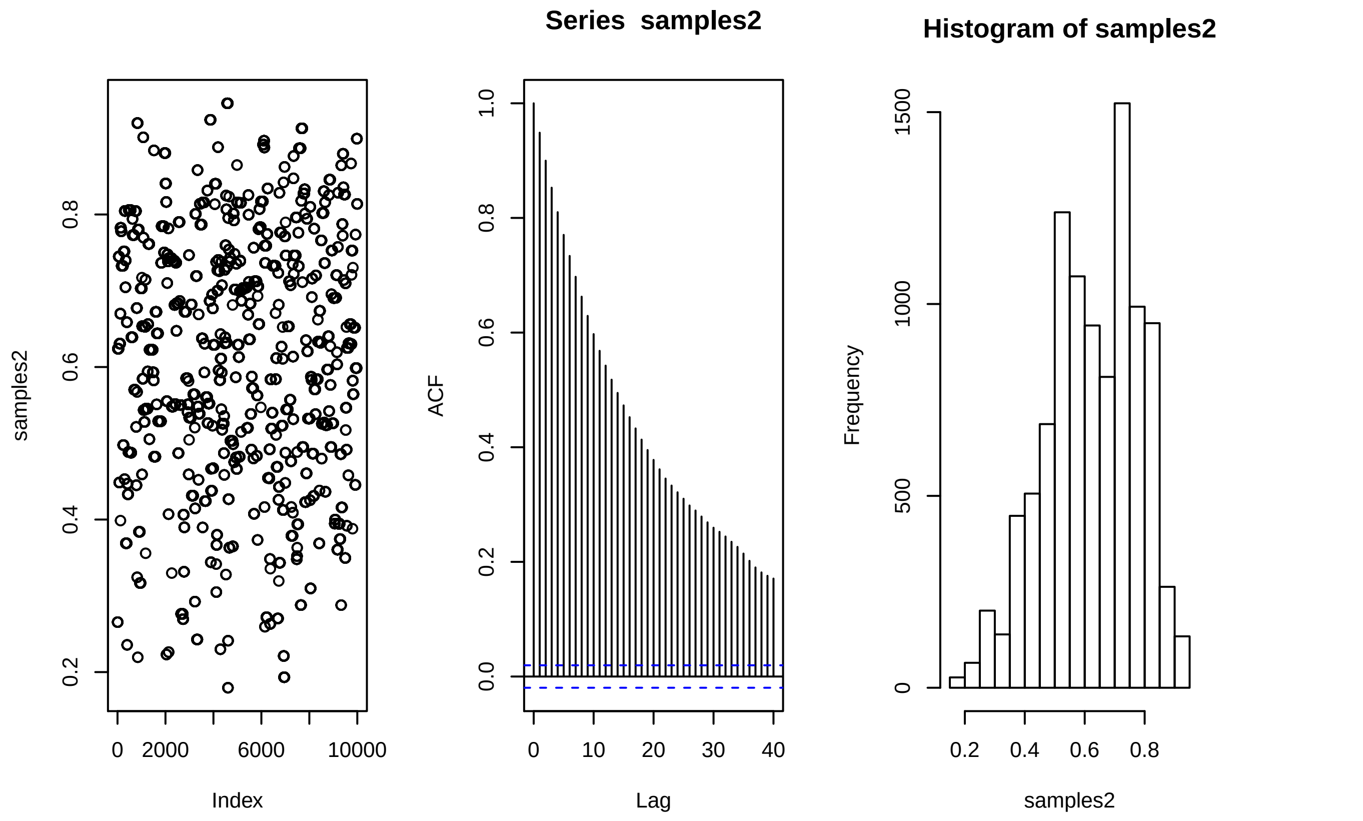
In ks-test, D-value is smaller than critical value at a confidence level of 0.95.

```
samples2 <- run_metropolis_MCMC(0.1, start, 10000)
acceptance_rate2 <- 1 - mean(duplicated(samples2))
```

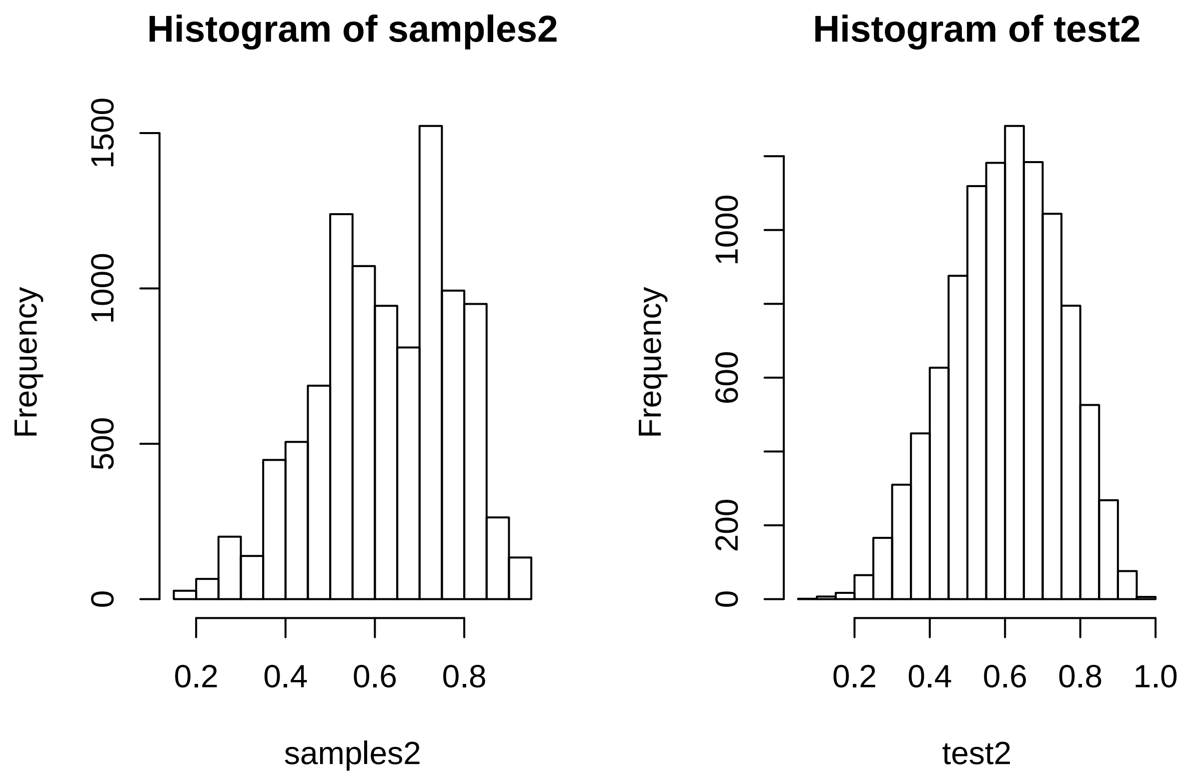
```r
print(acceptance_rate2)
```

```
## [1] 0.04159584
```

```r
 par(mfrow=c(1,3))  #1 row, 3 columns
 plot(samples2); acf(samples2); hist(samples2)  #plot commands
```



```r
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test2 = rbeta(10000,6,4)
par(mfrow=c(1,2))
hist(samples2)
hist(test2)
```



- the Kolmogorov Smirnov statistic

```
ks.test(samples2, pbeta, 6, 4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  samples2
## D = 0.11709, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

(2) c = 2.5

When c = 2.5, acceptance rate ~ 0.4.

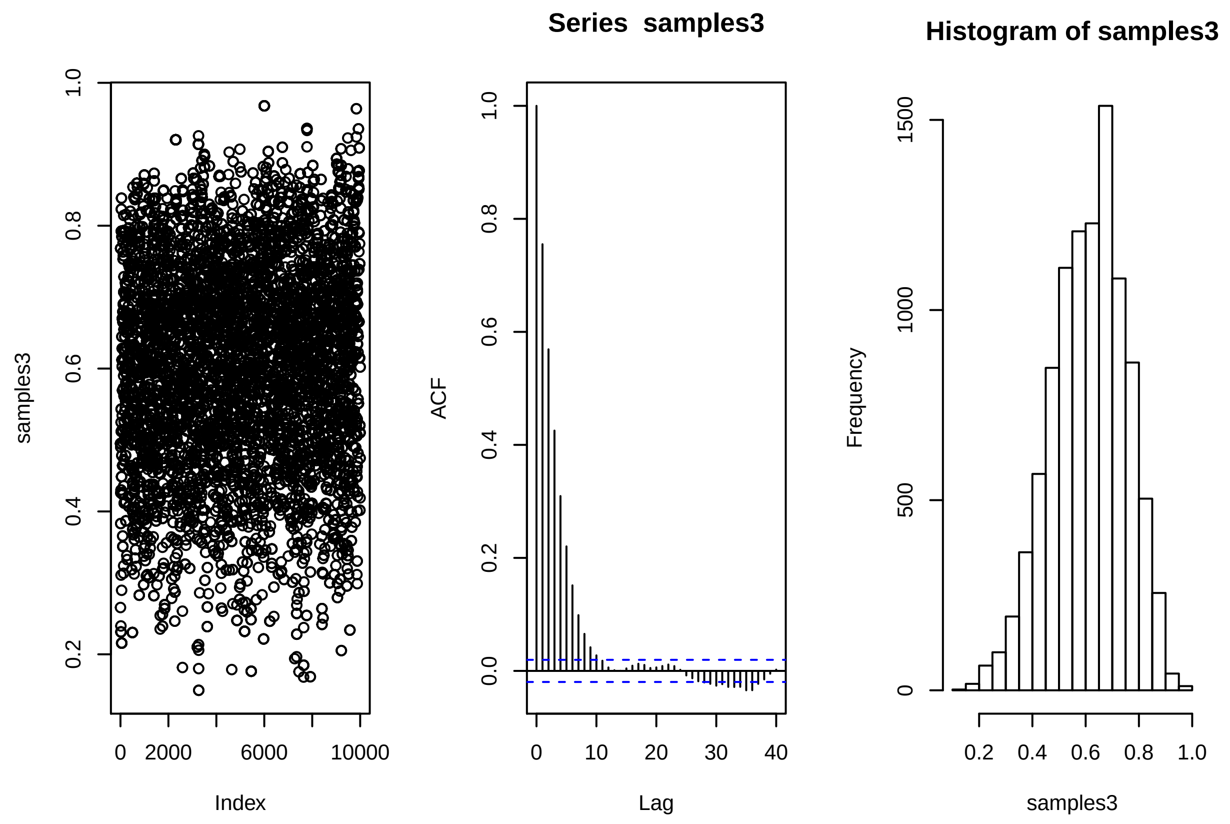
From the histogram we can see that the sample has a closer shape to the randomly generated values from the target.

In ks-test, D-value is greater than critical value at a confidence level of 0.95.

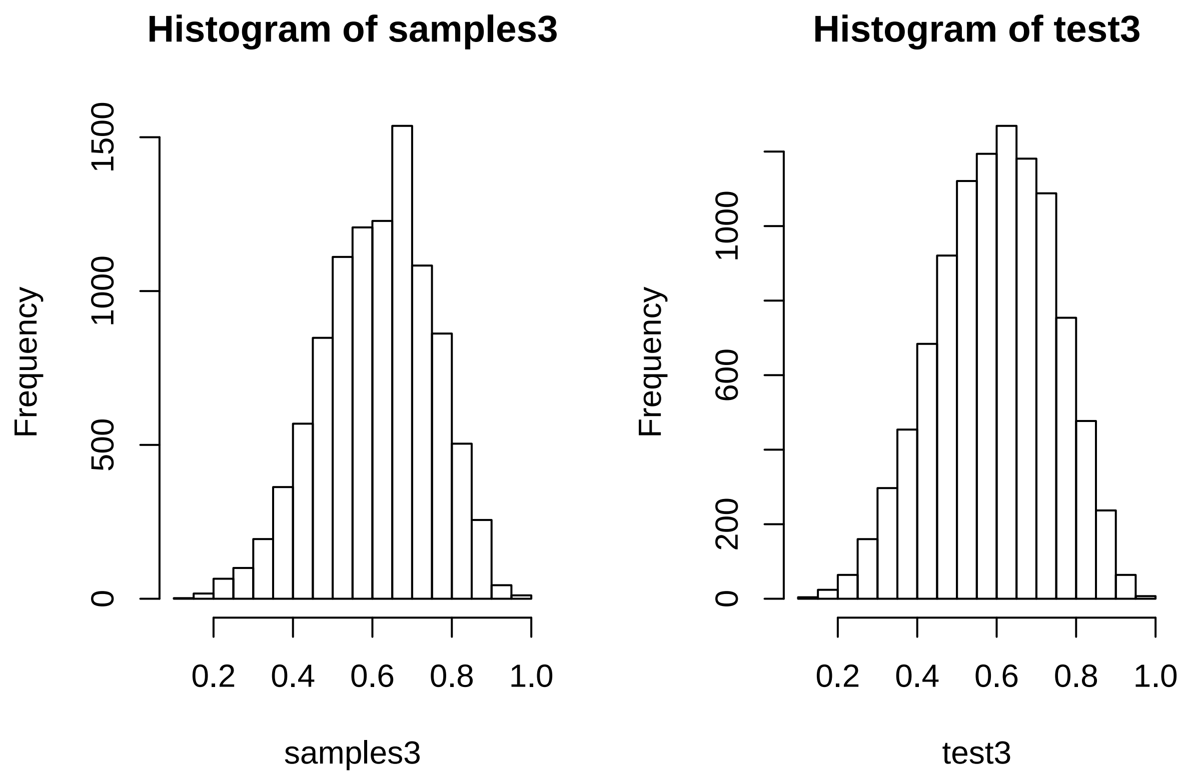
```
samples3 <- run_metropolis_MCMC(2.5, start, 10000)
acceptance_rate3 <- 1 - mean(duplicated(samples3))
print(acceptance_rate3)
```

```
## [1] 0.3936606
```

```
 par(mfrow=c(1,3))  # 1 row, 3 columns
 plot(samples3); acf(samples3); hist(samples3)  # plot commands
```



```
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test3 = rbeta(10000,6,4)
par(mfrow=c(1,2))
hist(samples3)
hist(test3)
```



- the Kolmogorov Smirnov statistic

```
ks.test(samples3, pbeta, 6, 4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  samples3
## D = 0.041799, p-value = 1.332e-15
## alternative hypothesis: two-sided
```

(3) c = 10

When c = 2.5, acceptance rate ~ 0.4.

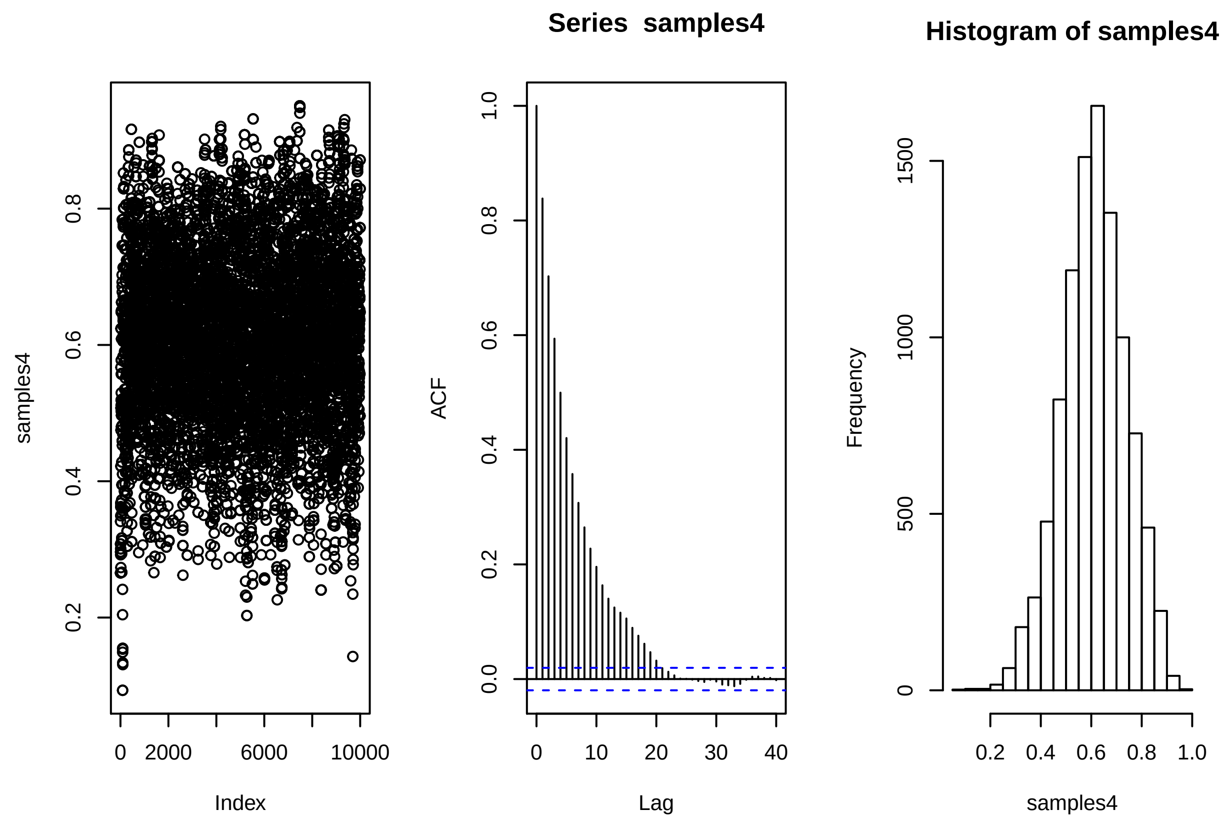
From the histogram we can see that the sample has a narrower shape than the randomly generated values from the target.

In ks-test, D-value is greater than critical value at a confidence level of 0.95.

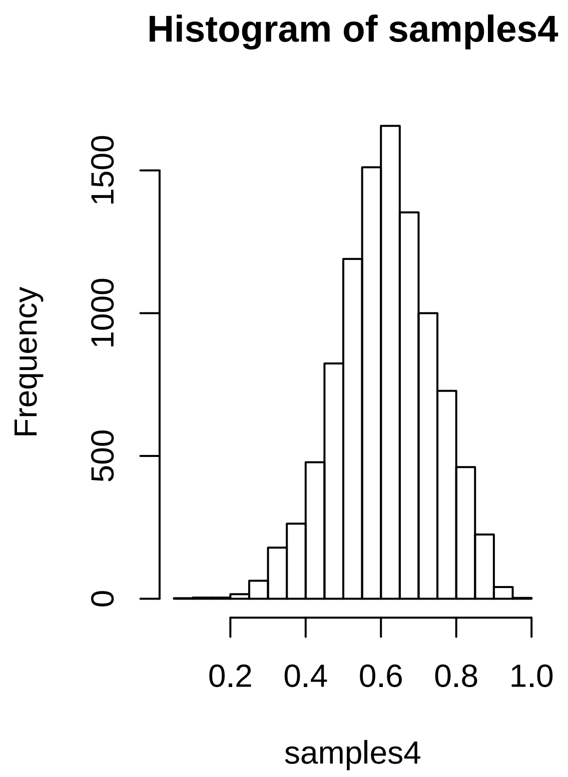
```
samples4 <- run_metropolis_MCMC(10, start, 10000)
acceptance_rate4 <- 1 - mean(duplicated(samples4))
print(acceptance_rate4)
```

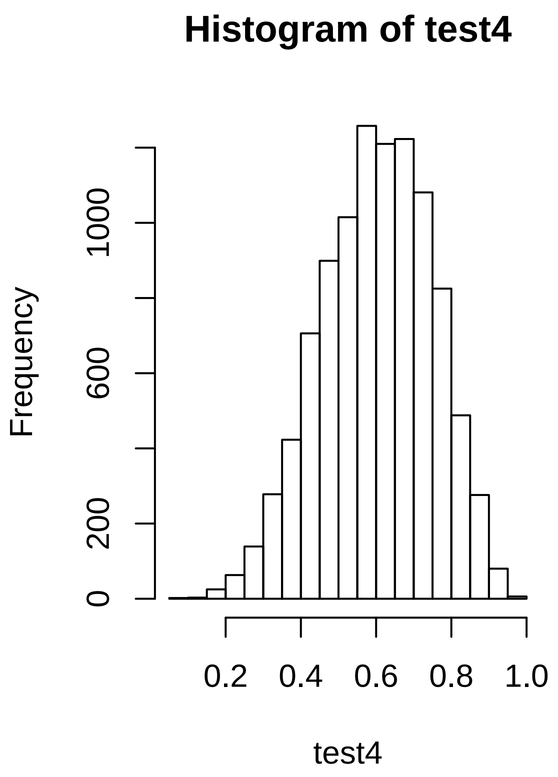
```
## [1] 0.5811419
```

```
 par(mfrow=c(1,3))  # 1 row, 3 columns
 plot(samples4); acf(samples4); hist(samples4)  # plot commands
```



```
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test4 = rbeta(10000,6,4)
par(mfrow=c(1,2))
hist(samples4)
hist(test4)
```





- the Kolmogorov Smirnov statistic

```
ks.test(samples4, pbeta, 6, 4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  samples4
## D = 0.071777, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

(2) adding burn-in time

Since acceptance rate for c = 0.1 is too low, it should have a later burn-in time than c = 2.5 and c = 10. acceptance for c = 10 is the highest among the three values, D-value for c = 2.5 is the smallest.

Therefore, we expect a smaller burn-in value for c = 2.5 and c = 10.

```
burnIn2 <- 0.7 * 10000
burnIn3 <- 0.5 * 10000
burnIn4 <- 0.3 * 10000
draws1 = samples1[-(1:burnIn3)]
draws2 = samples2[-(1:burnIn2)]
draws3 = samples3[-(1:burnIn3)]
draws4 = samples4[-(1:burnIn4)]
acceptance1 <- 1 - mean(duplicated(draws1))
acceptance2 <- 1 - mean(duplicated(draws2))
acceptance3 <- 1 - mean(duplicated(draws3))
```

```
acceptance4 <- 1 - mean(duplicated(draws4))
```

```
print(c(acceptance2, acceptance3, acceptance4))
```

```
## [1] 0.04331889 0.39092182 0.57820311
```

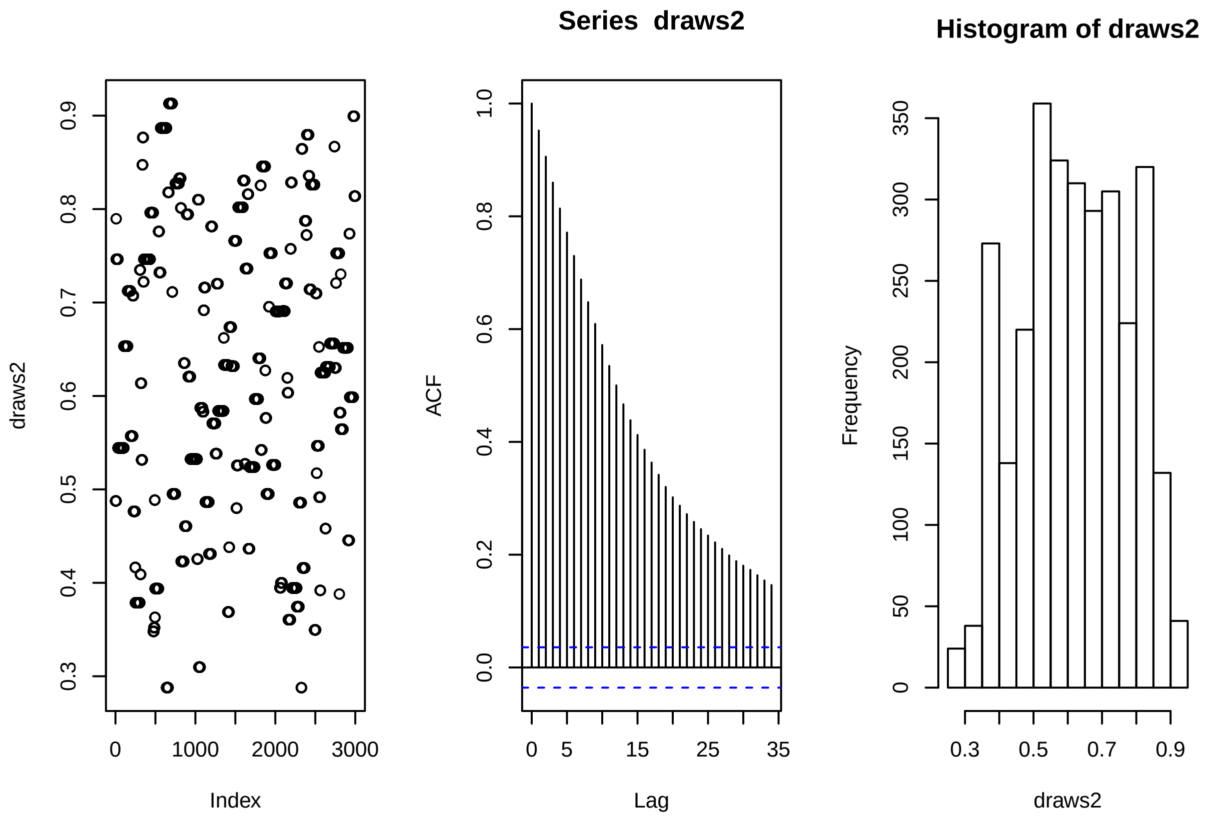
```
print(c(1 - mean(duplicated(samples3[-(1:3000)])),
        1 - mean(duplicated(samples3[-(1:5000)])),
        1 - mean(duplicated(samples3[-(1:8000)]))))
```

```
## [1] 0.3939437 0.3909218 0.3813093
```

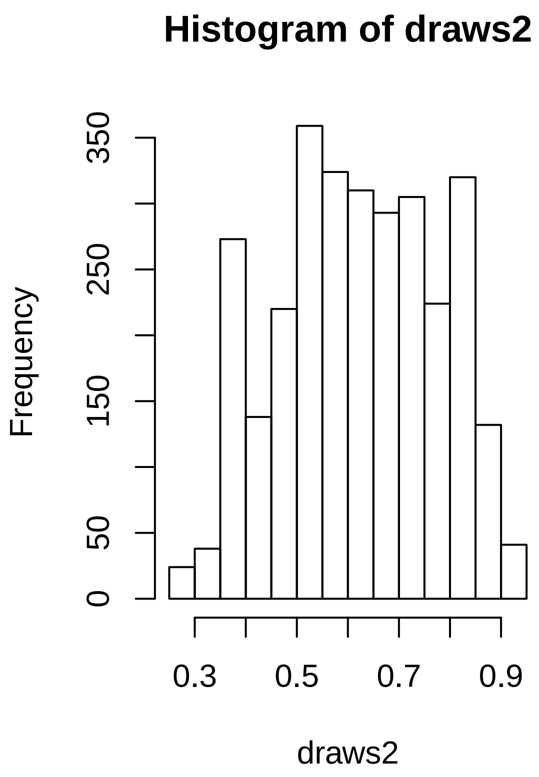
We can see that after adding a burn-in time, the acceptance rate for c = 2.5 and c = 10 does not change much.

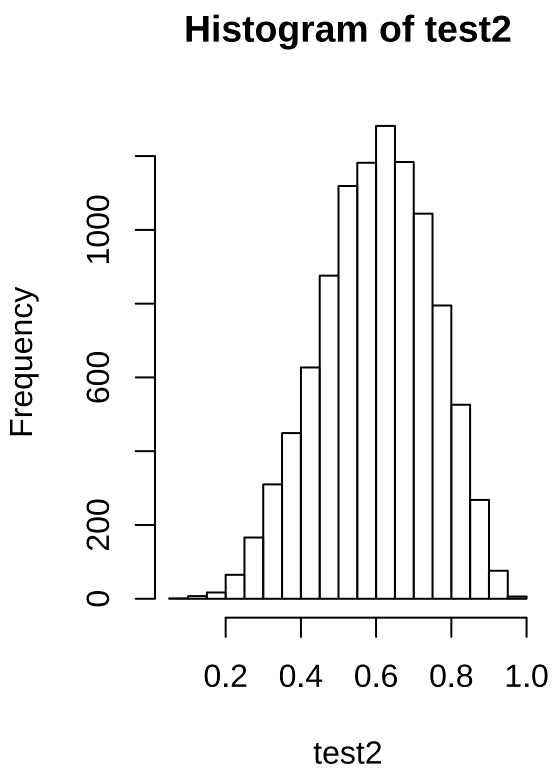
(a) c = 0.1

```
 par(mfrow=c(1,3))  # 1 row, 3 columns
 plot(draws2); acf(draws2); hist(draws2)  # plot commands
```



```
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test3 = rbeta(300,6,4)
par(mfrow=c(1,2))
hist(draws2)
hist(test2)
```





- the Kolmogorov Smirnov statistic

```
print(ks.test.critical.value(3000, 0.95))
```

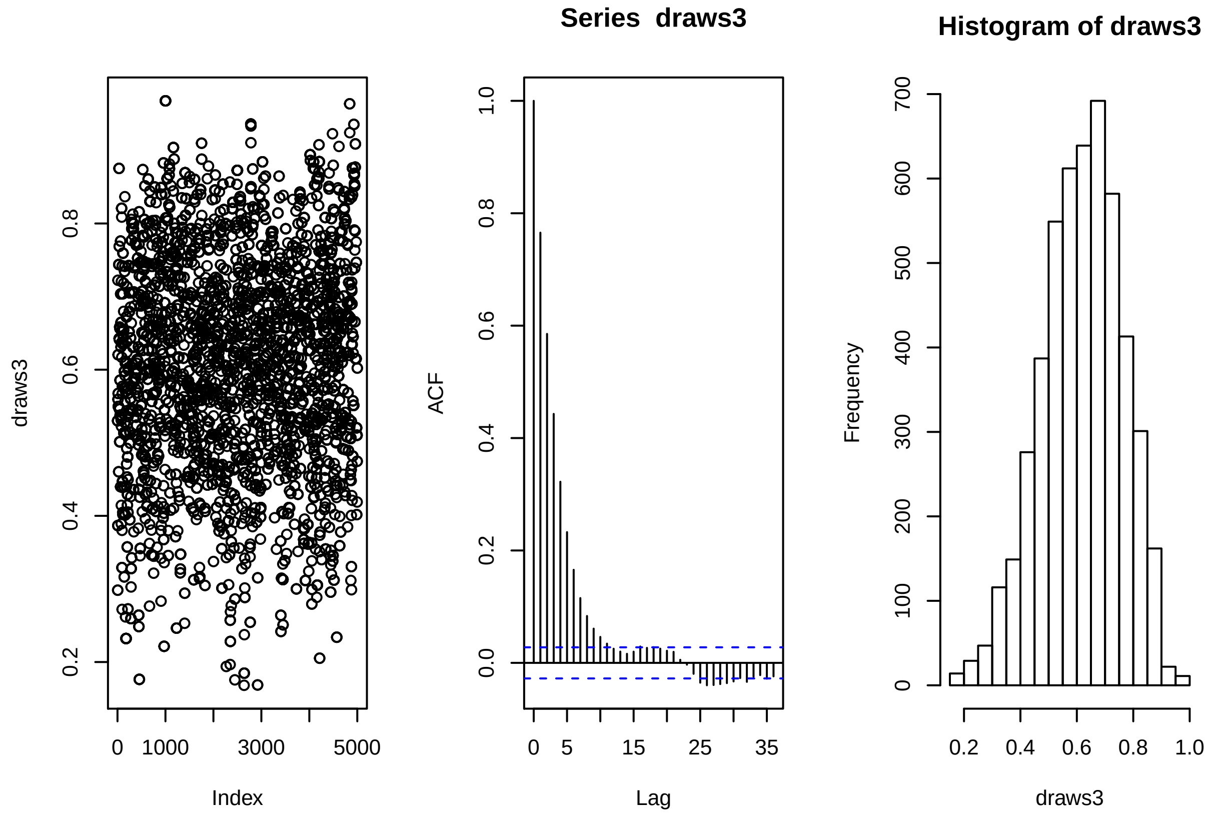
```
## [1] 0.0247954
```

```
ks.test(draws2, pbeta, 6, 4)
```

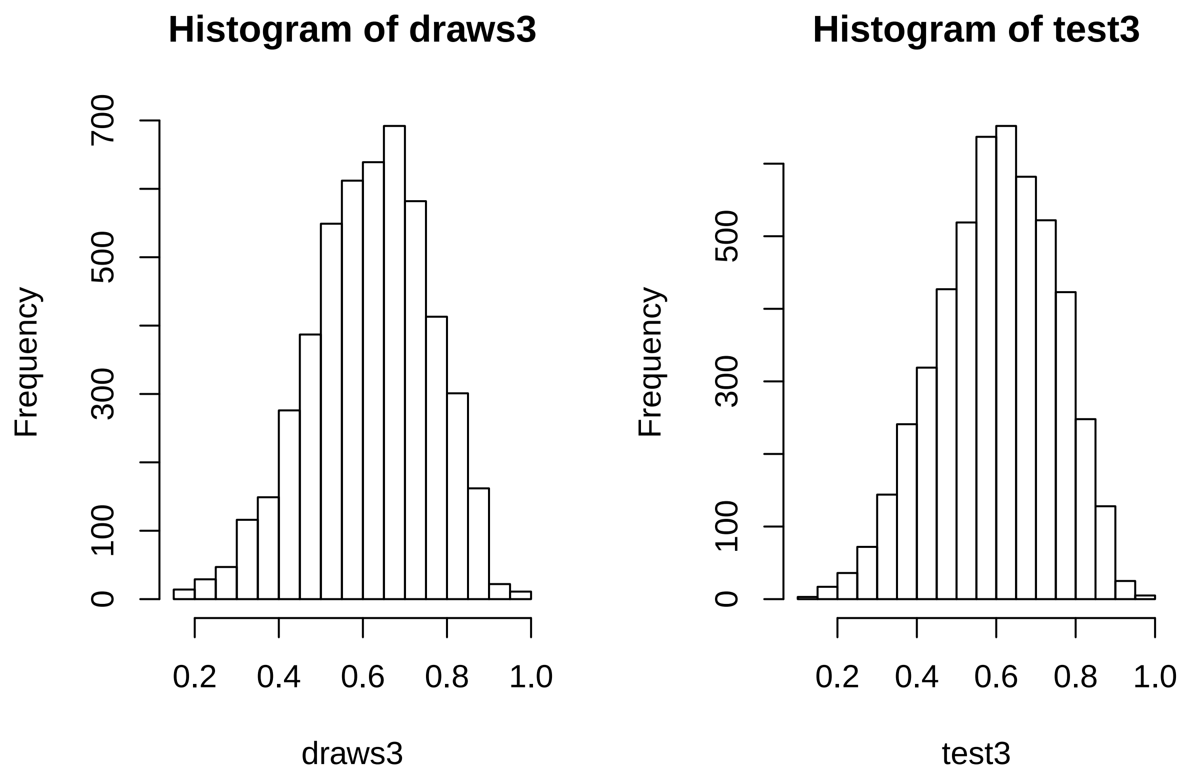
```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws2
## D = 0.099896, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

(b) c = 2.5

```
 par(mfrow=c(1,3))   # 1 row, 3 columns
 plot(draws3); acf(draws3); hist(draws3)   # plot commands
```



```
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test3 = rbeta(5000,6,4)
par(mfrow=c(1,2))
hist(draws3)
hist(test3)
```



- the Kolmogorov Smirnov statistic

```
print(ks.test.critical.value(5000, 0.95))
```

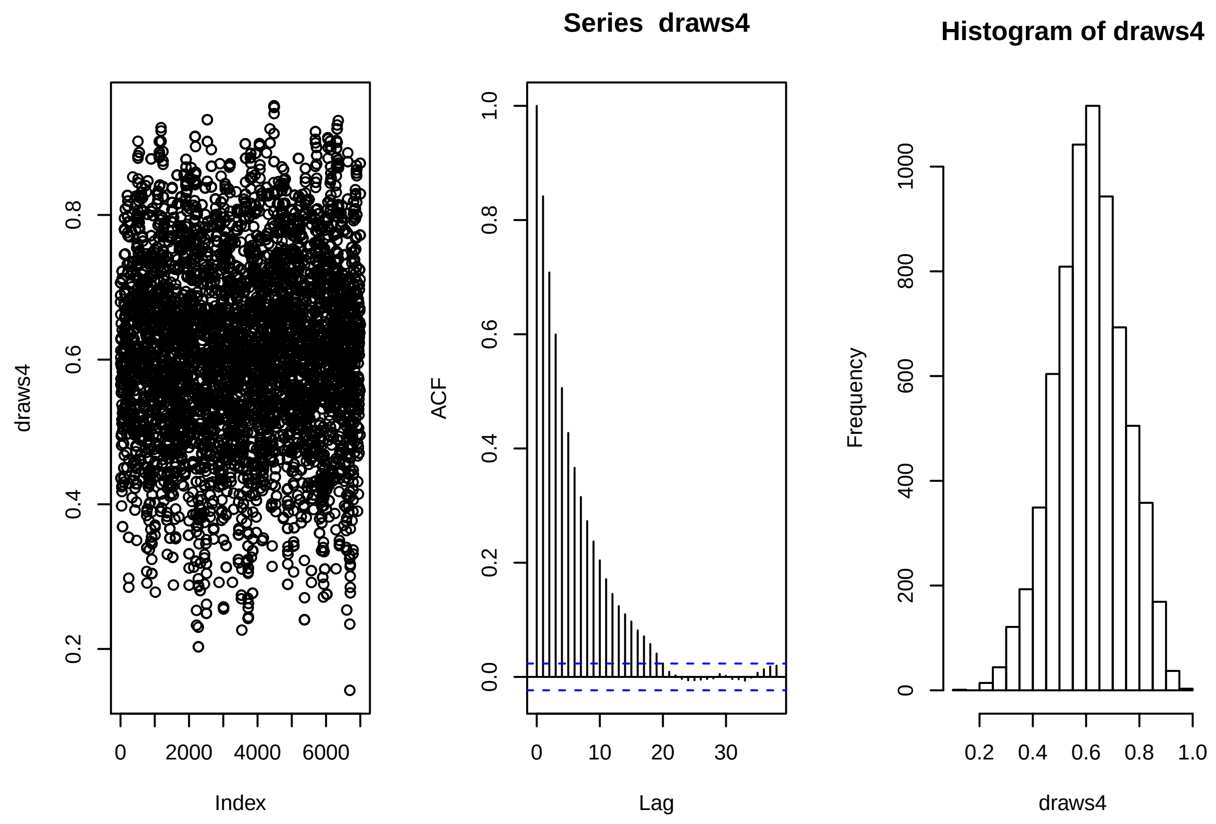
```
## [1] 0.01920643
```

```
ks.test(draws3, pbeta, 6, 4)
```

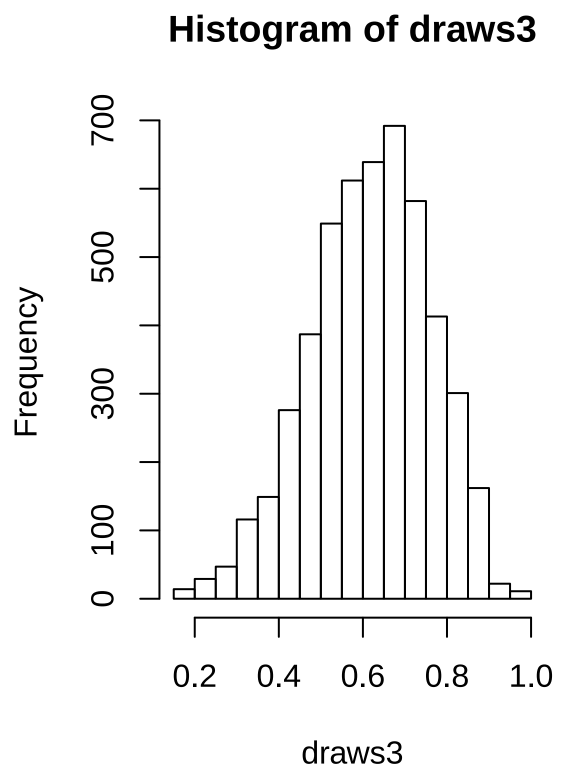
```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws3
## D = 0.05426, p-value = 3.253e-13
## alternative hypothesis: two-sided
```

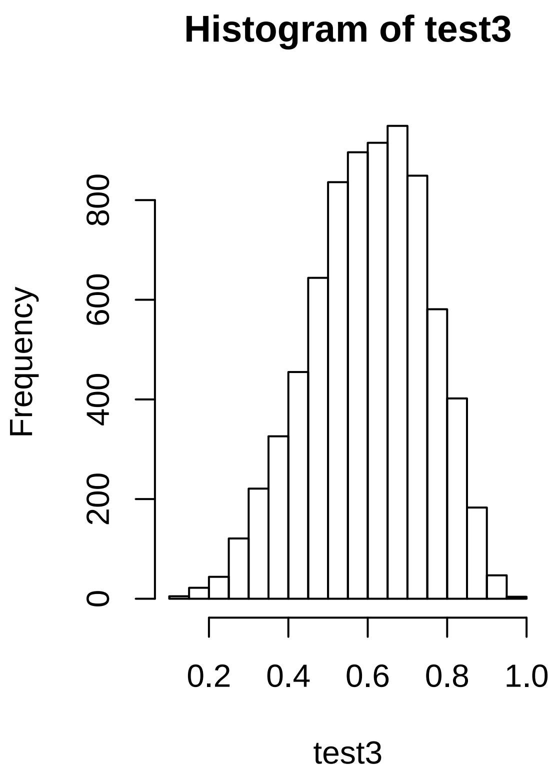
(c) c = 10

```
 par(mfrow=c(1,3))   # 1 row, 3 columns
 plot(draws4); acf(draws4); hist(draws4)   # plot commands
```



```
# compare with beta distribution with shape 1 = 6 and shape 2 = 4
test3 = rbeta(7500,6,4)
par(mfrow=c(1,2))
hist(draws3)
hist(test3)
```





- the Kolmogorov Smirnov statistic

```
print(ks.test.critical.value(7000, 0.95))
```

```
## [1] 0.0162324
```

```
ks.test(draws4, pbeta, 6, 4)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws4
## D = 0.069267, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

Based on histograms and D-value from KS-test, c = 2.5 has the most similar shape to beta(6,4) and the smallest difference between D-Value and the critical value.

Part II Gibbs sampling

November 4, 2018

This part is originally written by Dai Zhen,and revised by Boxin Zhao.

```
In [2]: import random,math
        import numpy as np
        import scipy
        from scipy import stats
        from numpy import vstack
        from numpy import concatenate
        from pynverse import inversefunc
        import matplotlib
        import matplotlib.pyplot as plt
        import seaborn as sb
        from pylab import rcParams
        %matplotlib inline
```

1 Description of algorithm

Our algorithm is based on two sampling techniques: the inverse probability transform sampling and the Gibbs sampling.

First, we want to get a way to sample points from a truncated exponential distribution. To do this, we apply the inverse probability transform by picking points from the uniform distribution and then compose it with the inverse cdf of the truncated exponential distribution. Let F denote the cdf of our desired distribution and U be the uniform distribution. Then $F^{-1}(U)$ will have the same probability distribution as F. This is due to the fact that for any z, $P(F^{-1}(U) \leq z) = P(U \leq F(z)) = F(z)$. This step is realized by the function inverse_transform(B,y), which samples n points from an truncated exponential distribution with parameter y and whose range lies in $[0, B]$.

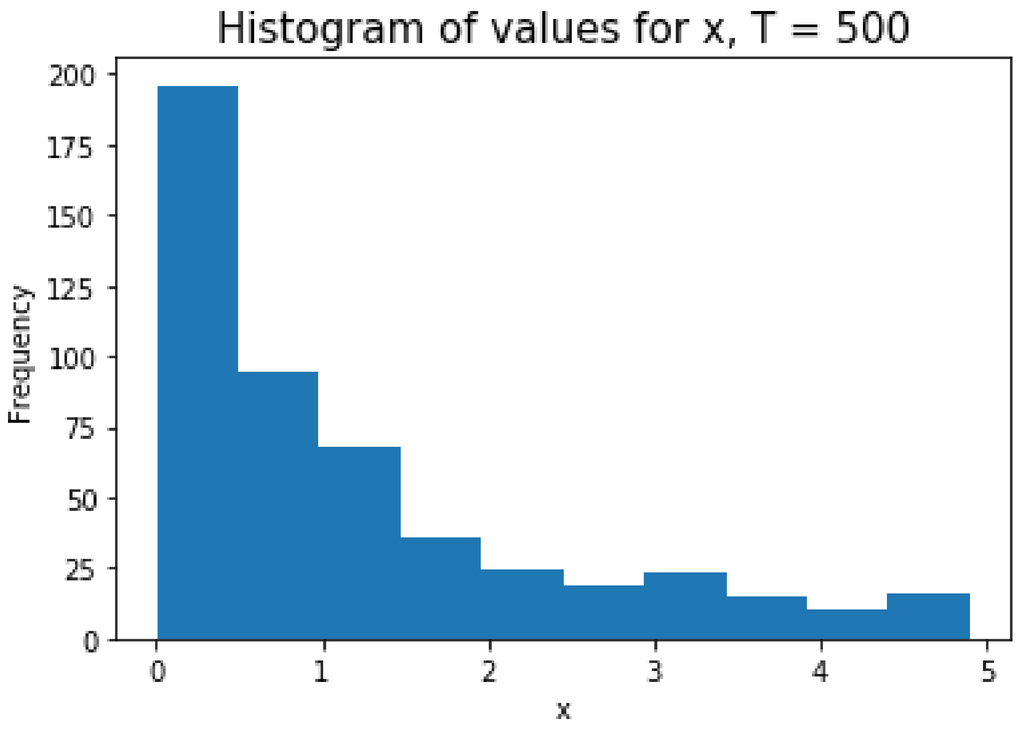
```
In [3]: def conditional_dist(x, y):
            # computes the conditional distribution of x given y.
            p = y * exp(-y * x)
            # this is the conditional distribution of X given Y=y.
            return p

        def cdf(x, y, B):
            # computes the cdf for conditional distribution function.
            integeral_val = scipy.integrate.quad(conditional_dist, 0, B)
            # The integral of the conditional distribution from 0 to B.
            const = 1/integeral_val  # normalizing constant.
            cdf = const * conditional_dist
            # to scale the conditional distribution so that it integrates to 1.
            return cdf

        def inverse_transform(B, y):
            # implements the inverse probability transform.
            cdf = np.random.uniform(0, 1)  # randomly pick a number in [0,1].
            x = - math.log(1 - cdf * (1 - math.exp(-B * y)))/y
            # apply the inverse of the cdf of the conditional distribution to it.
            return x
```

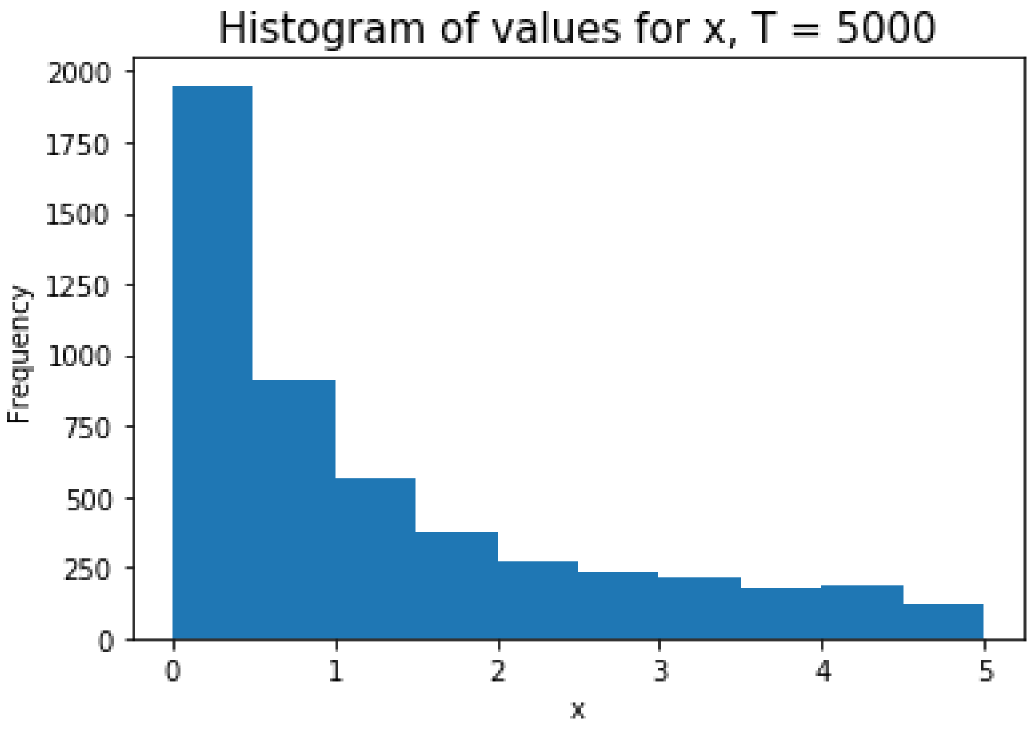
Second, we apply the Gibbs sampling to get an estimation of the joint distribution of X and Y. By assumption, the conditional distribution of x given y is the exponential distribution with parameter y and the conditional distribution of y given x is the exponential distribution with parameter x. Now, we first take one random number from 0 to B as the start value of y. Then given the value of y, we can sample a point from the conditional distribution of X given Y = y, which is the truncated exponential distribution to get a value of x. Then, given the value of x, we can again sample a point from the conditional distribution of Y given X = x, which is again an truncated exponential distribution to get a new value for y. We repeat the step above for T times to get T+1 samples of x and y. By the property of Gibbs sampling, we know that our sample will converges to the true joint distribution of X and Y. The above is implemented by the function gibbs(B,T), where [0,B] is the support for X and Y, and T is the number of iterations.

```
In [4]: def gibbs(B,T):
            # B is the upper bound for X and Y, and T is the number of iterations.
            start_y = np.random.rand(1)*B
            # Take a starting value of Y randomly from [0,B].
            start_x = inverse_transform(B, start_y)
            # Draw a starting value of X from the
            # conditional distribution of X given Y=start_y.
            start = concatenate((start_x, start_y), axis = None)
            # Put the starting values of X and Y together.
            result = np.matrix(start)
            # Create a matrix to save the value.
            current_x = start_x[0]
            # Set the current value to be the starting value.
            current_y = start_y[0]
            # Set the current value to be the starting value.
            for i in range(T):
                # run for T iterations.
                current_y = inverse_transform(B, current_x)
                # Update the value of Y by draw a point from the
                # conditional distribution based on the current value of X.
                current_x = inverse_transform(B, current_y)
                # Update the value of X by draw a point from the
                # conditional distribution based on the current value of Y.
                current = concatenate((current_x, current_y), axis = None)
                # Put the current values of X and Y together.
                result = vstack((result, current))
                # Save the current value to our matrix of samples.
            return result
```

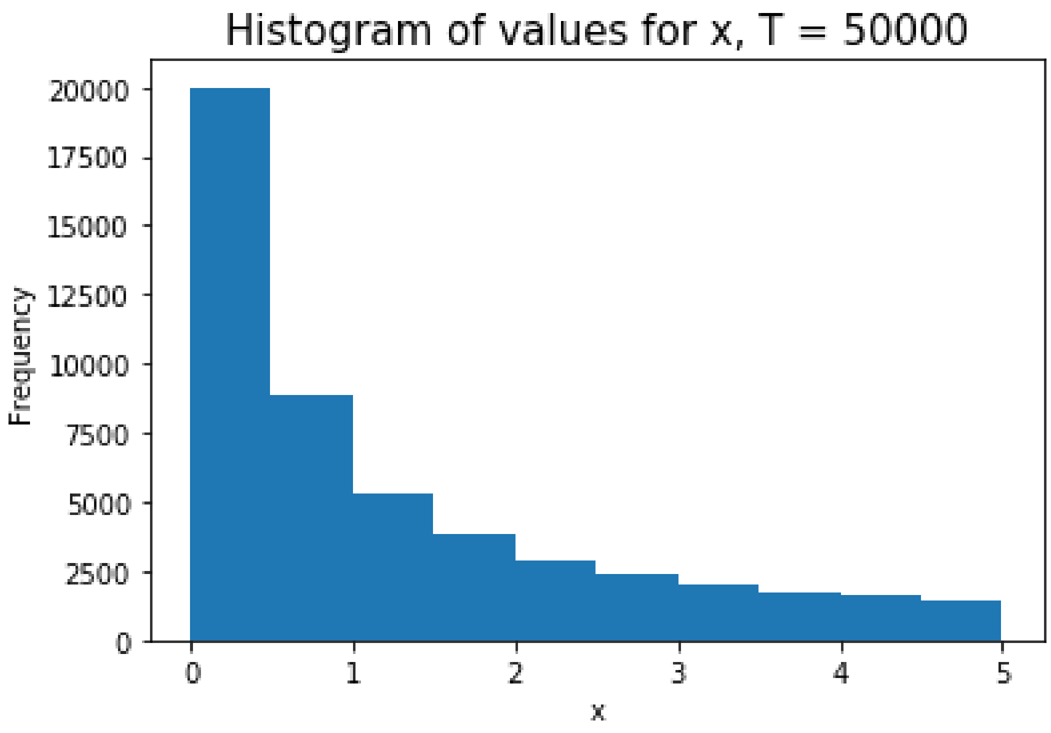
2 Plot the sample

```
In [5]: sample1 = gibbs(B = 5, T = 500)
        #Do Gibbs sampling to get 500 samples

        plt.hist(sample1[:,0])
        plt.title('Histogram of values for x, T = 500', fontsize = 15)
        plt.xlabel('x')
        plt.ylabel("Frequency")
        plt.show()


        sample2 = gibbs(B = 5, T = 5000)
        #Do Gibbs sampling to get 5000 samples

        plt.hist(sample2[:,0])
        plt.title('Histogram of values for x, T = 5000', fontsize = 15)
        plt.xlabel('x')
        plt.ylabel("Frequency")
        plt.show()

        sample3 = gibbs(B = 5, T = 50000)
        #Do Gibbs sampling to get 50000 samples

        plt.hist(sample3[:,0])
        plt.title('Histogram of values for x, T = 50000', fontsize = 15)
        plt.xlabel('x')
        plt.ylabel("Frequency")
```


Histogram of values for x, T = 500
Frequency
200
175
150
125
100
75
50
25
0
0
1
2
3
4
5
x


Histogram of values for x, T = 5000
Frequency
2000
1750
1500
1250
1000
750
500
250
0
0
1
2
3
4
5
x

```
Out[5]: Text(0,0.5,'Frequency')
```



3 Estimation of the mean of X

```
In [7]: est1_x = np.mean(np.asarray(sample1[:,0]))
        # estimate of the mean of X using 500 samples
        print("estimate of the mean of X using 500 samples is: %f" % est1_x)

        est2_x = np.mean(np.asarray(sample2[:,0]))
        # estimate of the mean of X using 5000 samples
        print("estimate of the mean of X using 5000 samples is: %f" % est2_x)

        est3_x = np.mean(np.asarray(sample3[:,0]))
        # estimate of the mean of X using 50000 samples
        print("estimate of the mean of X using 50000 samples is: %f" % est3_x)
```

```
estimate of the mean of X using 500 samples is: 1.198195
estimate of the mean of X using 5000 samples is: 1.274148
estimate of the mean of X using 50000 samples is: 1.267107
```

Part III K-Means

This part is originally written by Zhengfang Yang, and revised by Boxin Zhao.

1.The algorithm

Here we write a k-means classifier and test it on different data sets. In the algorithm, we run a few iterations with different initial cluster centers, and choose the best result in each iteration with respect to the sum of distance.

The distance between points is Euclidean distance. The distance function is

```
distance <- function (a, b) {
  dis <- sqrt (sum ((a - b) ^ 2))
  return (dis)
}
```

The algorithm of K-means with given k and given starting points is

(i) **Initialize** the coordinates of the starting points to be the centers of the k clusters.

(ii) **Check** each data point, if the distance to other cluster center is smaller than the distance to the current cluster center, change this point to the cluster with the nearest center, and update the centers of clusters.

(iii) **Repeat** step (ii) until no point changes its cluster. The current state is the result.

(iv) **Return** the labels of cluster that each point belongs to.

The code is showed as follows, together with comment of details.

```
K_means_fixed_start <- function (k, dataset, starting_points) {
  # The data size
  N <- length (dataset[, 1])

  # The dimension of each data point
  dimension <- length (dataset[1, ])

  # The center of each class
  center <- matrix (nrow = k, ncol = dimension)

  # The sum of coordinates of each class
  coordinate_sum <- matrix (nrow = k, ncol = dimension)

  # The number of points in each class
  cnt <- rep(1, k)

  # The class label for each point
  label <- rep(k + 1, N)

  # Initialize: assign starting_points to be centers of the k classes
  number <- 1
  for (i in starting_points) {
      label[i] <- number
      number <- number + 1
  }
  for (i in 1: N) {
      if (label[i] != k + 1) {
```

```r
        center[label[i], ] <- dataset[i, ]
        coordinate_sum[label[i], ] <- dataset[i, ]
      }
  }

  # Keep updating clusters and center coodinates
  # until no point changes it cluster
  is_changed <- TRUE
  INF <- 9999999;

  while (is_changed) {
    is_changed = FALSE

    # update the cluster for each point
    for (i in 1: N) {
      if (label[i] <= k)
        min_dis <- distance(dataset[i, ], center[label[i], ])
      else
        min_dis <- INF

    # check the distance to each cluster center
    for (j in 1: k) {
      if (distance(dataset[i, ], center[j, ]) < min_dis) {
        min_dis <- distance(dataset[i, ], center[j, ])

        # if there is a nearer cluster center
        if (j != label[i]) {
          # update cluster center coordinates
          if (label[i] <= k) {
            coordinate_sum[label[i],] <- coordinate_sum[label[i],] - dataset[i,]
            cnt[label[i]] <- cnt[label[i]] - 1
            center[label[i], ] <- coordinate_sum[label[i], ] / cnt[label[i]]
          }

          coordinate_sum[j, ] <- coordinate_sum[j, ] + dataset[i, ]
          cnt[j] <- cnt[j] + 1
          center[j, ] <- coordinate_sum[j, ] / cnt[j]

          # the cluster of this point is changed
          # then keep looping
          is_changed <- TRUE
          label[i] <- j
          }
        }
      }
    }
  }

  # return the labels of cluster that each point belongs to
  return (label)
}
```

Different starting point may result in different cluster result. Which result shall we choose? We measure the result by the sum of distance between each point and their corresponding cluster center.

```
compute_cluster_error <- function (k, dataset, label) {
  dimension <- length (dataset[1, ])
  N <- length (dataset[, 1])
  coordinate_sum <- matrix (0, nrow = k, ncol = dimension)
  center <- matrix (0, nrow = k, ncol = dimension)
  cnt <- rep(0, k)

  # compute the coordinate of each cluster center
  for (i in 1: N) {
    cnt[label[i]] <- cnt[label[i]] + 1
    coordinate_sum[label[i], ] <- coordinate_sum[label[i], ] + dataset[i, ]
  }
  for (i in 1: k)
    center[i, ] <- coordinate_sum[i, ] / cnt[i]

  # add up the distance
  error <- 0
  for (i in 1: N)
    error = error + sum((dataset[i, ] - center[label[i], ]) ^ 2)

  return (error)
}
```

A good cluster result should be "compact", thus the sum of distance should be small. Therefore, we compute the sum of distance for each result, and choose the one with the **least** sum of distance.

The following code shows the complete algorithm. We run a few iterations, and in each iteration we generate different starting points. We get the cluster result with funtion *K_means_fixed_start()*, and compute its *cluster_error*, i.e. the sum of distance between each point and its cluster center. Then we choose the result with the least *cluster_error*.

```
K_means <- function (data.train, k, trials) {
  N <- length (data.train[, 1])
  min_error <- 999999999
  best_ans <- rep (1, N)

  error_list <- c ()

  for (i in 1: trials) {
    starting_points <- sample(1: N, size = k, replace = FALSE)
    label <- K_means_fixed_start (k, as.matrix (data.train), starting_points)
    error <- compute_cluster_error (k, as.matrix (data.train), label)
    if (min_error > error) {
      min_error <- error
      best_ans <- label
    }
  }

  return (best_ans)
}
```

When the true labels are given, we can compare the true labels and our cluster labels to measure the performance of K-means. Since we may not have the same name of each cluster as the true labels, we define the correctness in the following way:

Suppose the data points are $\{X_k\}_{1\leq k\leq N}$, with true labels $\{y_k\}_{1\leq k\leq N}$, and our result of cluster labels are

$\{\hat{y}_k\}_{1\le k\le N}$. Then the number of pairs (X_i, X_j) where $1 \le i < j \le N$ is $\frac{N(N-1)}{2}$. For each pair (X_i, X_j), the true label $y_i = y_j$ means they are in the same cluster. If we have $\hat{y}_i = \hat{y}_j$, then our prediction for this pair is correct. Also, the true label $y_i \ne y_j$ means they are in different clusters. If we have $\hat{y}_i \ne \hat{y}_j$, then our prediction for this pair is correct.

Therefore, we can write our definition of correctness as the following formula shows:

$$Correctness \triangleq \frac{N(N-1)}{2} \sum_{1\le i<j\le N} (\mathbb{I}_{y_i=y_j,\hat{y}_i=\hat{y}_j} + \mathbb{I}_{y_i\ne y_j,\hat{y}_i\ne\hat{y}_j}).$$

The code is

```r
compute_correctness <- function(my_label, true_label) {
  N <- length(my_label)

  # total number of pairs
  tot <- N * (N - 1) / 2

  # the number of correct pairs we find
  correct <- 0

  for (i in 1: (N - 1))
    for (j in (i + 1): N) {
      # point i and point j are in the same cluster
      if (my_label[i] == my_label[j] && true_label[i] == true_label[j])
        correct <- correct + 1
      # point i and point j are in different clusters
      if (my_label[i] != my_label[j] && true_label[i] != true_label[j])
        correct <- correct + 1
    }

  return (correct / tot)
}
```

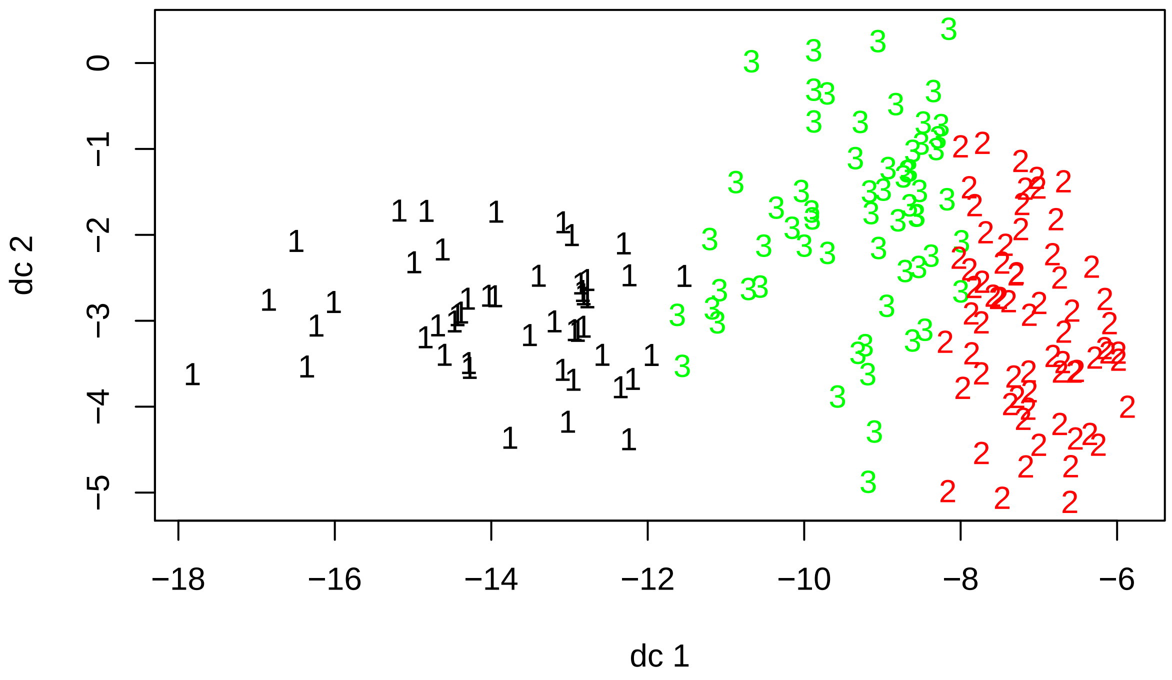
2. Test

We test our algorithm on rattle project in R. We know in advance that $k = 3$. First we test out algorithm without any preprocess.

```r
data (wine, package = "rattle")
data.train <- wine[-1]
true_label <- as.numeric(wine[, 1])

my_label <- K_means(data.train, k = 3, trials = 20)
compute_correctness(my_label, true_label)
```

```
## [1] 0.7186568
```

```r
plotcluster(data.train, my_label)
```



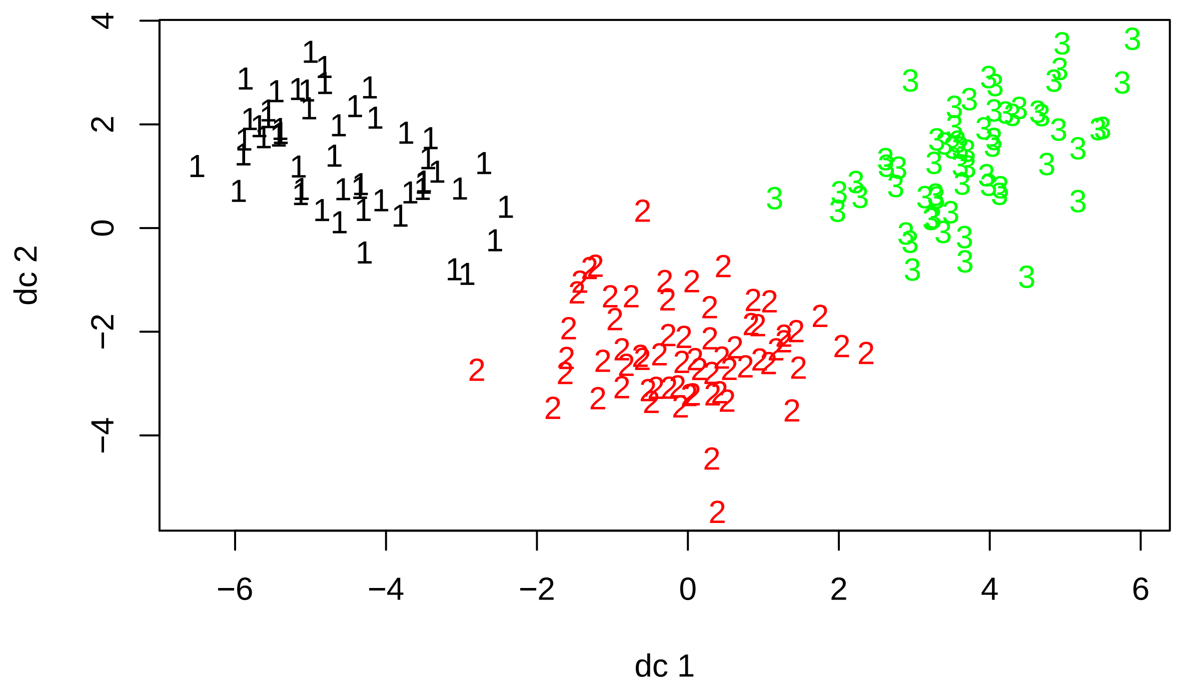
The clusters seems not perfectly separated - cluster 2 and cluster 3 are closed to each other. And our correct rate is not very high, only 72% approximately.

The scale of features has a strong influence on our algorithm. Because the features with a big scale contribute a lot to the Euclidean distance, while the small scale features have a much smaller influence on the cluster result. However, when we run clustering algorithm on this data set, there is no reason to believe that features with a big scale should contribute more to the final result than the features with a small scale do. Therefore, we can rescale our data so that each feature contributes to the Euclidean distance in the equal scale.

```
data.train <- scale(wine[-1])
# this line of code executes a linear transformation to each column
# so that each column (each feature) will be: mean = 0, variance = 1

my_label <- K_means(data.train, k = 3, trials = 20)
compute_correctness(my_label, true_label)
```

```
## [1] 0.9542944
```

```
plotcluster(data.train, my_label)
```



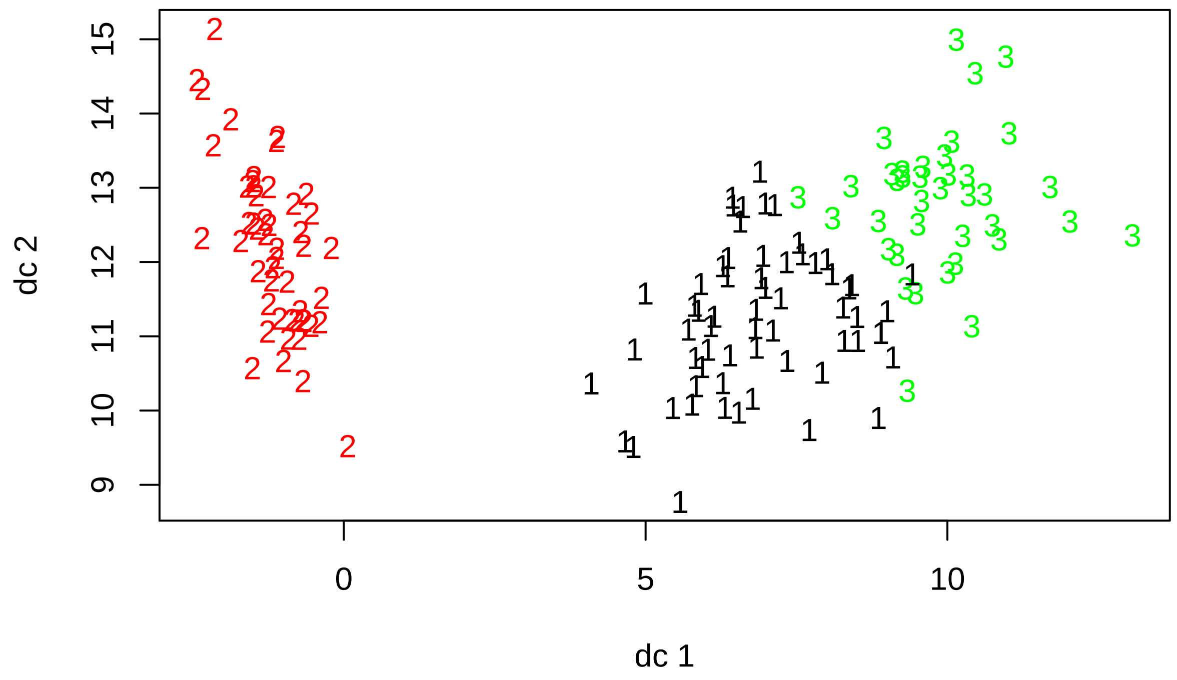
We can see that there is a significant improvement in the result. The correct rate exceeds 95%. From the cluster plot we can also see that the three clusters are well separated. It means that the rescaling makes sense and works well.

Then we repeat these steps on *iris* data set. The result of original data is

```
data(iris)
data.train <- iris[-5]
true_label <- as.numeric(iris[, 5])
k <- 3
my_label <- K_means(data.train, k, trials = 20)
compute_correctness(my_label, true_label)
```

```
## [1] 0.8797315
```

```
plotcluster(data.train, my_label)
```

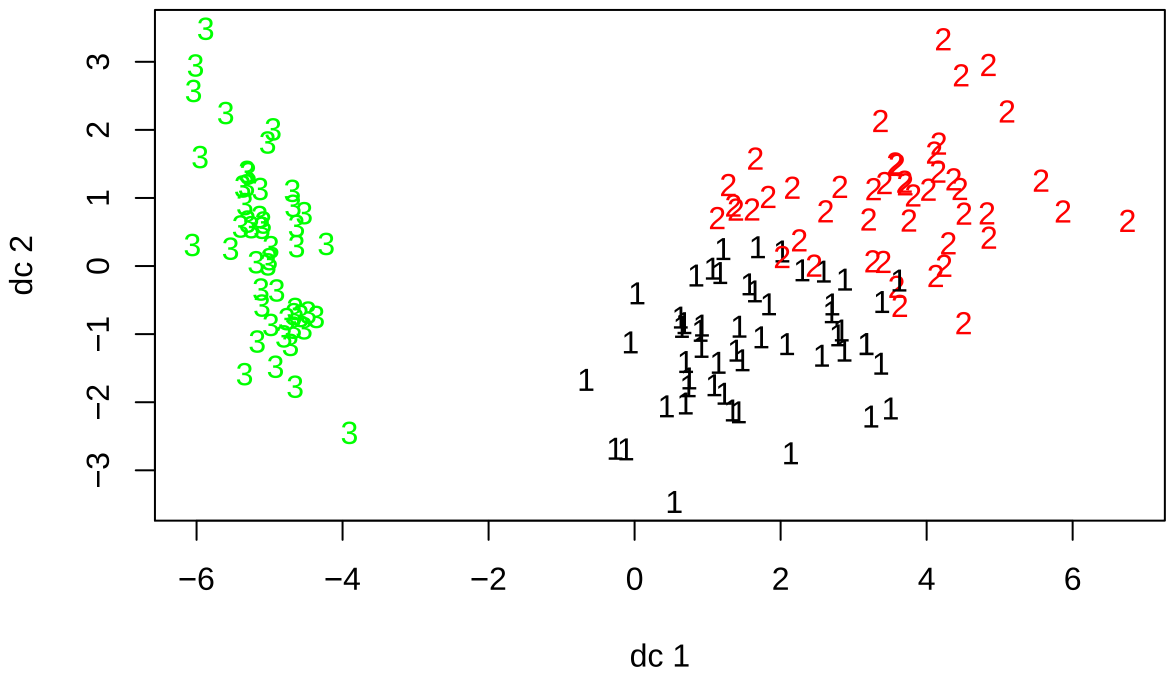


And the result of rescaled data

```r
data.train <- scale(iris[-5])
my_label <- K_means(data.train, k, trials = 20)
compute_correctness(my_label, true_label)
```

```
## [1] 0.8322148
```

```r
plotcluster(data.train, my_label)
```



The correct rate is above 80%. K-means still works, but not perfect.

It is notable that the result of original data is slightly better than the result of rescaled data, i.e. the rescaling does not work this time.

It is not surprising because the scale of features in the original data does not have a big variation, so the rescaling does not make a big difference. From the cluster plot we can see that there are two clusters that are closed to each other. It is questionable that whether clustering is the best model for this data set, because there are only 5 variables. It may not be easy to get an accurate clustering result when the number of variables is small.