STA 6106 Homework 3 Anmol Panchal

Import all the following packages:

```
import numpy as np
import pandas as pd
import scipy.stats
import matplotlib as mp
import plotly
import matplotlib.pyplot
import seaborn as sns
import emcee
import math
from numpy import linalg as la
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import time
from scipy.stats import rayleigh
```

Problem 1: 1

Metropolis-Hastings algorithm is another sampling algorithm to sample from high dimensional, difficult to sample directly (due to intractable integrals) distributions or functions.

It's an MCMC algorithm, just like Gibbs Sampling. It's MC (Markov Chain) because to get the next sample, one only need to consider the current sample. It's MC (Monte Carlo) because it generates random sample which we could use to compute integrals or numerical results, for example in the probability distribution setting, the integrals we may want to compute are the expected value, mode, median, etc.

We are using rayleigh distribution to sample from drawing 1000 random variates :

```
def f(x):
   return rayleigh.rvs(1000)
```

We used the algorithm as discussed in class taking f(x) as rayleigh distribution now we define the function for metropolis hastings algorithm as given below:
Metropolis algorithm (symmetric proposal distribution)

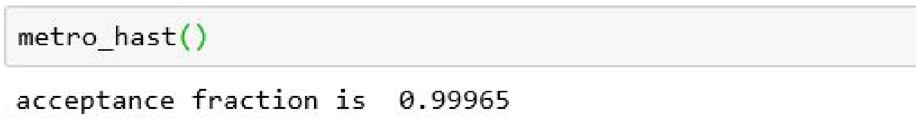
Let f(x) be a rayleigh function that is proportional to the desired probability distribution P(x) (a.k.a. a target distribution).

Initially, choose an arbitrary point x{0} to be the first sample, and choose an arbitrary probability density that suggests a candidate for the next sample value x_next, given the previous sample value x.

For the Metropolis algorithm, function need to be symmetric.
For each iteration t:
Generate : Generate a candidate x for the next sample by picking from the distribution.
Calculate the acceptance ratio alpha =f(x')/f(x_{t}), which will be used to decide whether to accept or reject the candidate which will be calculated in my function with the help of **counter/float(N).**

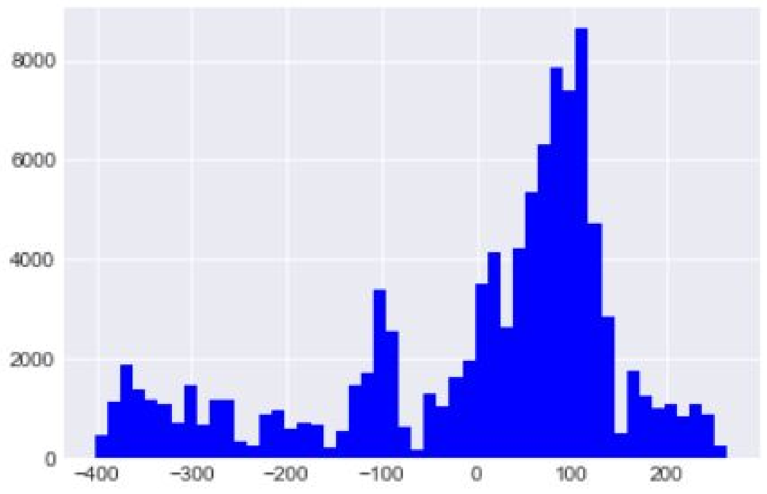
Accept or Reject :
Generate a uniform random number u on [0,N-1].
If alpha accept the candidate by setting **x[i+1] = x_next & counter = counter + 1**
If alpha reject the candidate and set **x[i+1] = x[i]** instead.
Here alpha is **np.random.random_sample() < min(1, f(x_next)/f(x[i])):**

```
def metro_hast():
    N = 100000
    x = np.arange(N,dtype=np.float)

    x[0] = 0.2
    counter = 0
    for i in range(0, N-1):

        x_next = np.random.normal(x[i], 1.)
        if np.random.random_sample() < min(1, f(x_next)/f(x[i])):
            x[i+1] = x_next
            counter = counter + 1
        else:
            x[i+1] = x[i]

    print("acceptance fraction is ", counter/float(N))

    pl.hist(x, bins=50, color='blue')
    pl.show()
```



```
metro_hast()
```

```
acceptance fraction is  0.99965
```



Code:

```
# Metropolis hasting algorithm  for Rayleigh distribution
import numpy as np
import matplotlib.pyplot as pl

def f(x):
    return rayleigh.rvs(1000)

def metro_hast():
    N = 100000
    x = np.arange(N,dtype=np.float)

    x[0] = 0.2
    counter = 0
    for i in range(0, N-1):

        x_next = np.random.normal(x[i], 1.)
        if np.random.random_sample() < min(1, f(x_next)/f(x[i])):
            x[i+1] = x_next
            counter = counter + 1
        else:
            x[i+1] = x[i]

    print("acceptance fraction is ", counter/float(N))

    pl.hist(x, bins=50, color='blue')
    pl.show()
```

Problem 1: 2

I will use Gibb Sampler to draw sampler from a Bivariate Gaussian with mu of [2, 2] and sigma/covariance matrix of [[1, 0], [0, 1]]

```
mus = np.array([2, 2])
sigmas = np.array([[1, 0], [0, 1]])
```

Suppose p(x, y) is a p.d.f. or p.m.f. that is difficult to sample from directly.

The Gibbs sampler proceeds as follows:
1. set x and y to some initial starting values
2. then sample x|y, then sample y|x,
then x|y, and so on.

Set (x0, y0) to some starting value.
1. Sample x1 ~ p(x|y0), that is, from the conditional distribution
X | Y = y0.
Current state: (x1, y0)
Sample y1 ~ p(y|x1), that is, from the conditional distribution
Y | X = x1.
Current state: (x1, y1)
2. Sample x2 ~ p(x|y1), that is, from the conditional distribution
X | Y = y1.
Current state: (x2, y1)
Sample y2 ~ p(y|x2), that is, from the conditional distribution
Y | X = x2.
Current state: (x2, y2)
.
.
Repeat iterations 1 and 2, M times.

```
def gibbs_sampling(mus, sigmas, iter=10000):
    samples = np.zeros((iter, 2))
    y = np.random.rand() * 10

    for i in range(iter):
        x = p_x_given_y(y, mus, sigmas)
        y = p_y_given_x(x, mus, sigmas)
        samples[i, :] = [x, y]

    return samples
```

Where p_x_given_y is P X|Y and p_y_given_x is P Y|X :

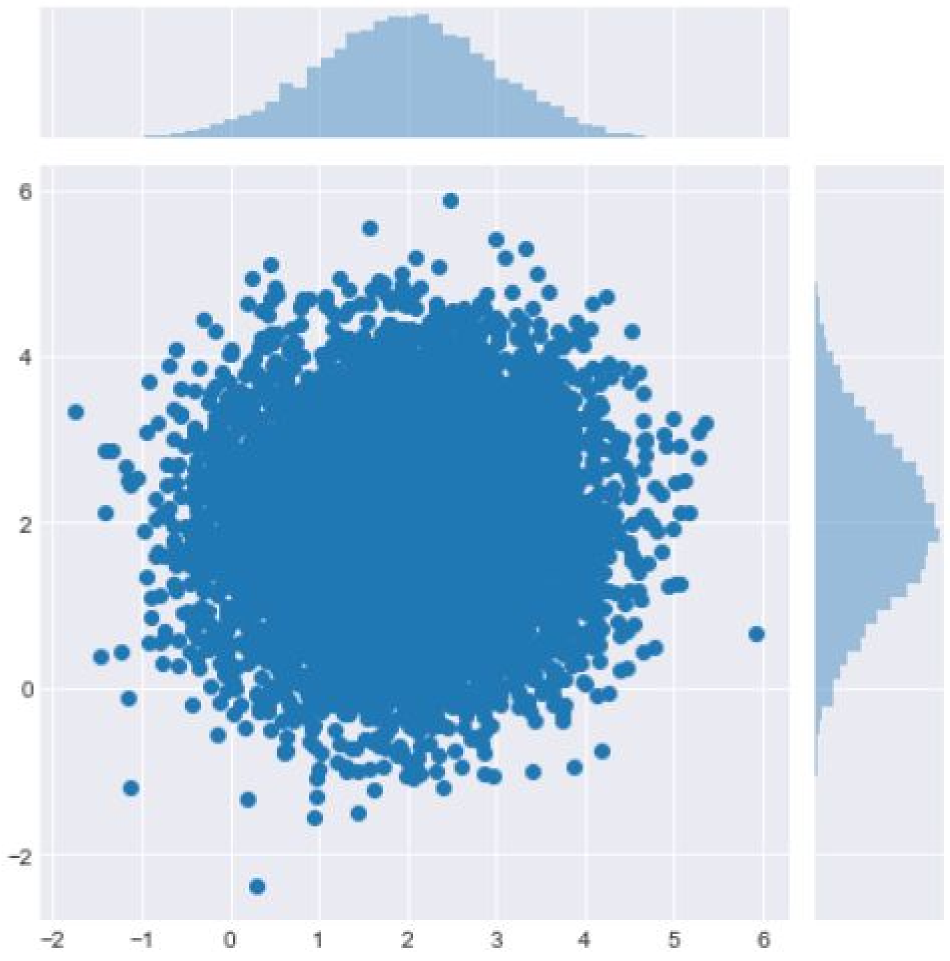
```
def p_x_given_y(y, mus, sigmas):
    mu = mus[0] + sigmas[1, 0] / sigmas[0, 0] * (y - mus[1])
    sigma = sigmas[0, 0] - sigmas[1, 0] / sigmas[1, 1] * sigmas[1, 0]
    return np.random.normal(mu, sigma)


def p_y_given_x(x, mus, sigmas):
    mu = mus[1] + sigmas[0, 1] / sigmas[1, 1] * (x - mus[0])
    sigma = sigmas[1, 1] - sigmas[0, 1] / sigmas[0, 0] * sigmas[0, 1]
    return np.random.normal(mu, sigma)
```

Code:

```
import numpy as np
import seaborn as sns


def p_x_given_y(y, mus, sigmas):
    mu = mus[0] + sigmas[1, 0] / sigmas[0, 0] * (y - mus[1])
    sigma = sigmas[0, 0] - sigmas[1, 0] / sigmas[1, 1] * sigmas[1, 0]
    return np.random.normal(mu, sigma)


def p_y_given_x(x, mus, sigmas):
    mu = mus[1] + sigmas[0, 1] / sigmas[1, 1] * (x - mus[0])
    sigma = sigmas[1, 1] - sigmas[0, 1] / sigmas[0, 0] * sigmas[0, 1]
    return np.random.normal(mu, sigma)


def gibbs_sampling(mus, sigmas, iter=10000):
    samples = np.zeros((iter, 2))
    y = np.random.rand() * 10

    for i in range(iter):
        x = p_x_given_y(y, mus, sigmas)
        y = p_y_given_x(x, mus, sigmas)
        samples[i, :] = [x, y]

    return samples


if __name__ == '__main__':
    mus = np.array([2, 2])
```

```
sigmas = np.array([[1, 0], [0, 1]])

samples = gibbs_sampling(mus, sigmas)
sns.jointplot(samples[:, 0], samples[:, 1])
```

Output:



Problem 3: 1-2 - 9.8 & 9.9 from Rizzo

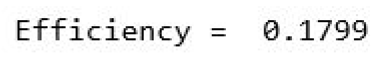
We need to draw random samples from the following distributions
1- The standard uniform distribution
2- A proposal distribution p(x) that we choose to be N(0,σ)
3- The target distribution g(x) which is proportional to the posterior probability

We define our values for n, h, a,b,lik,prior prob (binomial), sigma, theta, niters(number of iterations), samples (as niters +1)

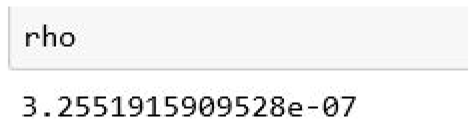
```
n = 100
h = 61
a = 10
b = 10
lik = st.binom
prior = st.beta(a, b)
sigma = 0.3

naccept = 0
theta = 0.1
niters = 10000
samples = np.zeros(niters+1)
samples[0] = theta
```

Now we run a for loop for calculating theta_p , rho with u as random uniform distribution for given no. of iterations.
Where rho equals to minimum of 1, target distribution with (lik, prior, n, h, theta_p) / target distribution (lik, prior, n, h, theta)
Then we accept the uniform distribution value if less than rho and calculate the efficiency based on that.

```
for i in range(niters):
    theta_p = theta + st.norm(0, sigma).rvs()
    rho = min(1, target(lik, prior, n, h, theta_p)/target(lik, prior, n, h, theta ))
    u = np.random.uniform()
    if u < rho:
        naccept += 1
        theta = theta_p
    samples[i+1] = theta
nmcmc = len(samples)//2
print ("Efficiency = ", naccept/niters)
```



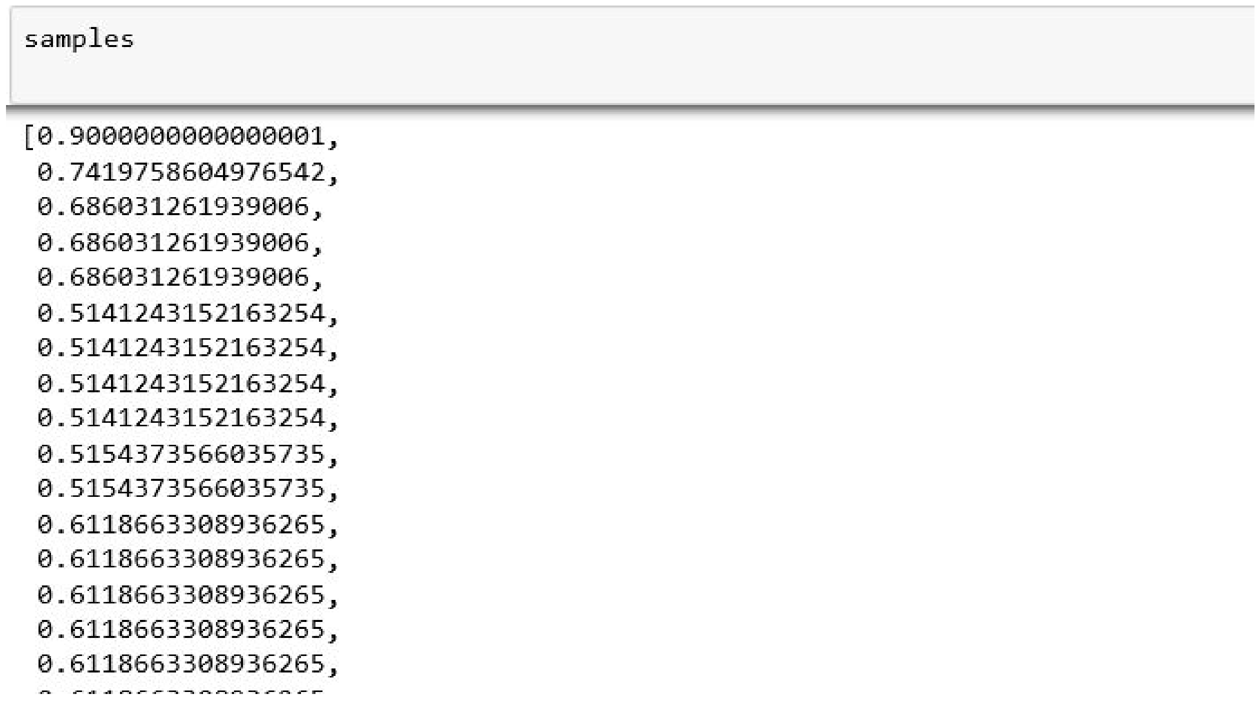
```
Efficiency =  0.1799
```


```
rho
```

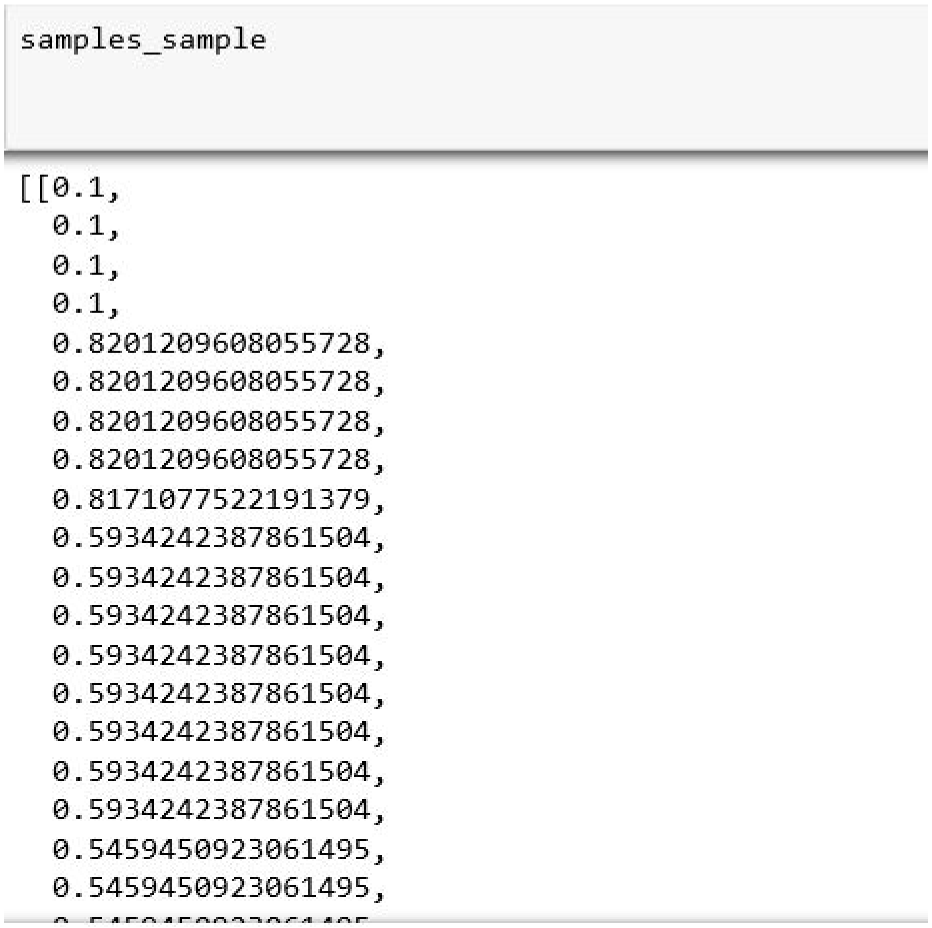
```
3.2551915909528e-07
```

Now we have to define a sampler function to calculate the gelmen-rubin diagonistic for convergence of our above function to run it multiple times to check for convergence.
So we define function sampler to sample for niters - no of iterations, with n , h, theta, lik,prior,sigma values.
So that we get samples to run a diagnostic check by using gelmen-rubin method to check for convergence.

```
def sampler(niters, n, h, theta, lik, prior, sigma):
    samples = [theta]
    while len(samples) < niters:
        theta_p = theta + stats.norm(0, sigma).rvs()
        rho = min(1, target(lik, prior, n, h, theta_p)/target(lik, prior, n, h, theta ))
        u = np.random.uniform()
        if u < rho:
            theta = theta_p
        samples.append(theta)
    return samples
samples_sample = [sampler(niters, n, h, theta, lik, prior, sigma) for theta in np.arange(0.1, 1, 0.2)]
```


```
samples
```

```
[0.9000000000000001,
 0.7419758604976542,
 0.686031261939006,
 0.686031261939006,
 0.686031261939006,
 0.5141243152163254,
 0.5141243152163254,
 0.5141243152163254,
 0.5141243152163254,
 0.5154373566035735,
 0.5154373566035735,
 0.6118663308936265,
 0.6118663308936265,
 0.6118663308936265,
 0.6118663308936265,
 0.6118663308936265,
```



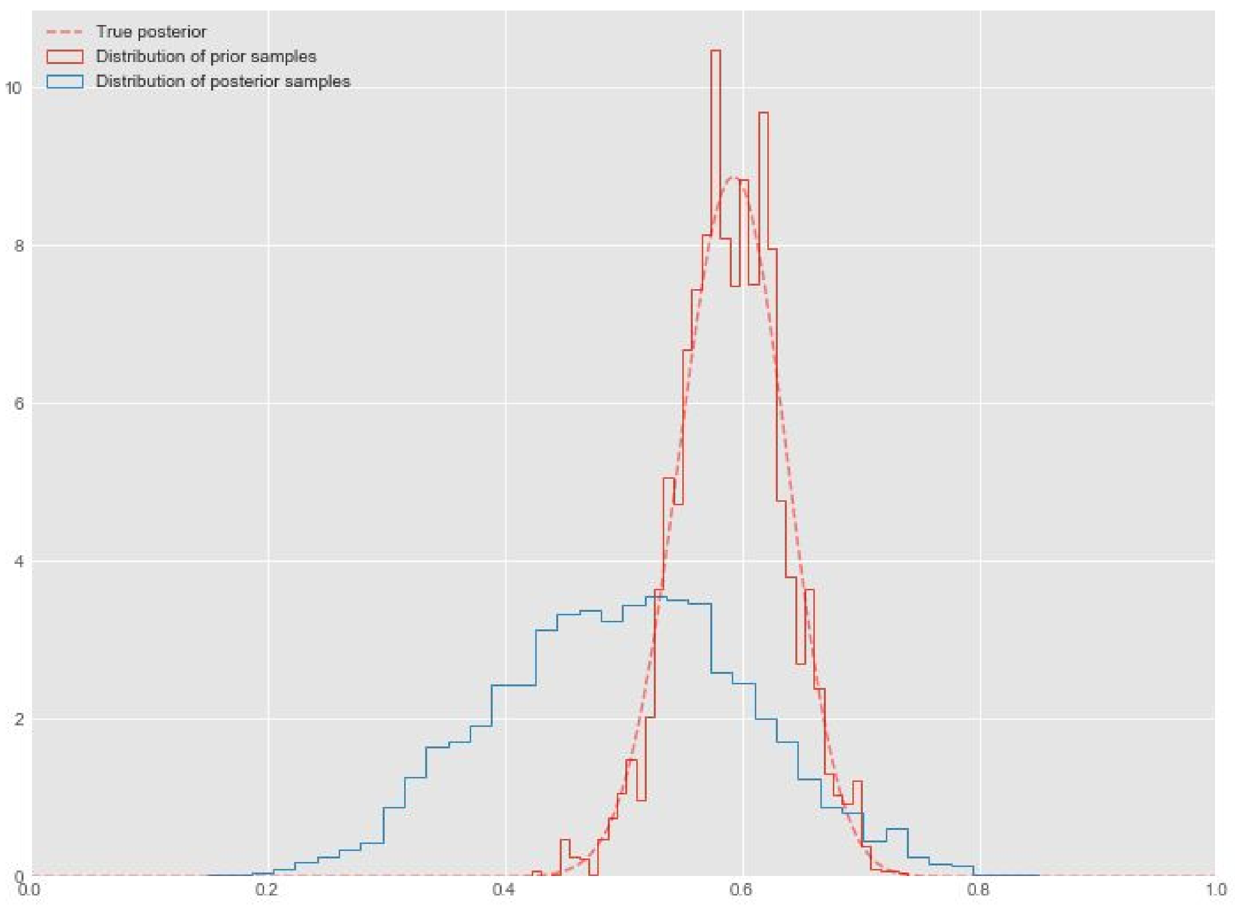
```
samples_sample
```

```
[[0.1,
  0.1,
  0.1,
  0.1,
  0.8201209608055728,
  0.8201209608055728,
  0.8201209608055728,
  0.8201209608055728,
  0.8171077522191379,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5934242387861504,
  0.5459450923061495,
  0.5459450923061495,
```

Now we plot the graphs for posterior samples distribution and prior samples distribution with a true posterior curve.

post = st.beta(h+a, n-h+b)

plt.figure(figsize=(12, 9))
plt.hist(samples[nmcmc:], 40, histtype='step', normed=True, linewidth=1, label='Distribution of prior samples');
plt.hist(prior.rvs(nmcmc), 40, histtype='step', normed=True, linewidth=1, label='Distribution of posterior samples');
plt.plot(thetas, post.pdf(thetas), c='red', linestyle='--', alpha=0.5, label='True posterior')
plt.xlim([0,1]);
plt.legend(loc='best');



```
# Gelmen-Rubin Diagnostic
mean1 = np.mean(u)
B = n * ((mean1)**2)
var_theta = (1 - 1/n) * u + 1/n*B
print("Gelmen-Rubin Diagnostic: ", np.sqrt(var_theta/u))
```

```
Gelmen-Rubin Diagnostic:  1.085379342795298
```

Now to check the convergence via graph to check the above generated measure.
Gelman Rubin Diagnostic(R^)
If our MH MCMC Chain reaches a stationary distribution, and we repeat the exercise multiple times, then we can examine if the posterior for each chain converges to the same place in the distribution of the parameter space.

Steps:

Run M>1 Chains of length NC×N .

Discard the first N draws of each chain, leaving N iterations in the chain.
Calculate the within and between chain variance.

Within chain variance:
W=1M∑j=1Ms2j

where s2j is the variance of each chain (after throwing out the first N draws).
Between chain variance:
B=NM−1∑j=1M(θj¯−θ¯¯)2

where θ¯¯ is the mean of each of the M means.

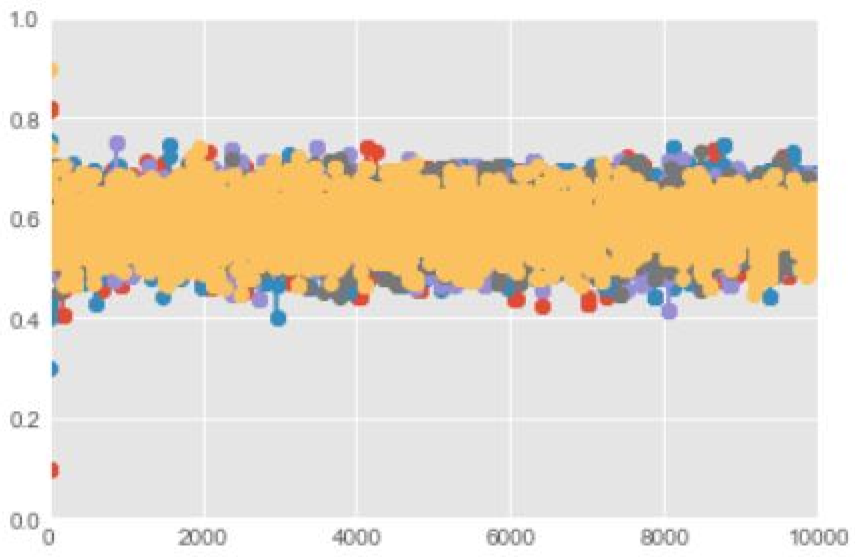
Calculate the estimated variance of θ as the weighted sum of between and within chain variance.
var^(θ)=(1−1N)W+1NB

Calculate the potential scale reduction factor.
R^=var^(θ)W−−−−−−√

We want this number to be close to 1. Why? This would indicate that the between chain variance is small. This makes sense, if between chain variance is small, that means both chains are mixing around the stationary distribution. Gelmen and Rubin show that when R^ is greater than 1.1 or 1.2, we need longer burn-in.

Let's run N (niters) chains:

```
for samples in samples_sample:
    plt.plot(samples, '-o')
plt.xlim([0, niters])
plt.ylim([0, 1]);
```



The above plot shows the convergence and we further increase the number of iterations we will be able to see more clearly.

Code:

```
def target(lik, prior, n, h, theta):
    if theta < 0 or theta > 1:
        return 0
    else:
        return lik(n, theta).pmf(h)*prior.pdf(theta)

n = 100
h = 61
a = 10
b = 10
lik = st.binom
prior = st.beta(a, b)
sigma = 0.3

naccept = 0
theta = 0.1
niters = 10000
samples = np.zeros(niters+1)
samples[0] = theta
for i in range(niters):
    theta_p = theta + st.norm(0, sigma).rvs()
    rho = min(1, target(lik, prior, n, h, theta_p)/target(lik, prior, n, h, theta ))
    u = np.random.uniform()
    if u < rho:
        naccept += 1
        theta = theta_p
    samples[i+1] = theta
nmcmc = len(samples)//2
```

```
print ("Efficiency = ", naccept/niters)


def sampler(niters, n, h, theta, lik, prior, sigma):
    samples = [theta]
    while len(samples) < niters:
        theta_p = theta + stats.norm(0, sigma).rvs()
        rho = min(1, target(lik, prior, n, h, theta_p)/target(lik, prior, n, h, theta ))
        u = np.random.uniform()
        if u < rho:
            theta = theta_p
        samples.append(theta)
    return samples
samples_sample = [sampler(niters, n, h, theta, lik, prior, sigma) for theta in np.arange(0.1, 1, 0.2)]

# Gelmen-Rubin Diagnostic
mean1 = np.mean(u)
B = n * ((mean1)**2)
var_theta = (1 - 1/n) * u + 1/n*B
print("Gelmen-Rubin Diagnostic: ", np.sqrt(var_theta/u))
post = st.beta(h+a, n-h+b)

plt.figure(figsize=(12, 9))
plt.hist(samples[nmcmc:], 40, histtype='step', normed=True, linewidth=1, label='Distribution of prior
samples');
plt.hist(prior.rvs(nmcmc), 40, histtype='step', normed=True, linewidth=1, label='Distribution of posterior
samples');
plt.plot(thetas, post.pdf(thetas), c='red', linestyle='--', alpha=0.5, label='True posterior')
plt.xlim([0,1]);
plt.legend(loc='best');

for samples in samples_sample:
    plt.plot(samples, '-o')
plt.xlim([0, niters])
plt.ylim([0, 1]);
```

Problem 4: 1

We are going to first import our mtcars_hw3 dataset which has mpg, hp as columns whose bootstrap bias and std. Error with statistic as correlation is evaluated as follows:

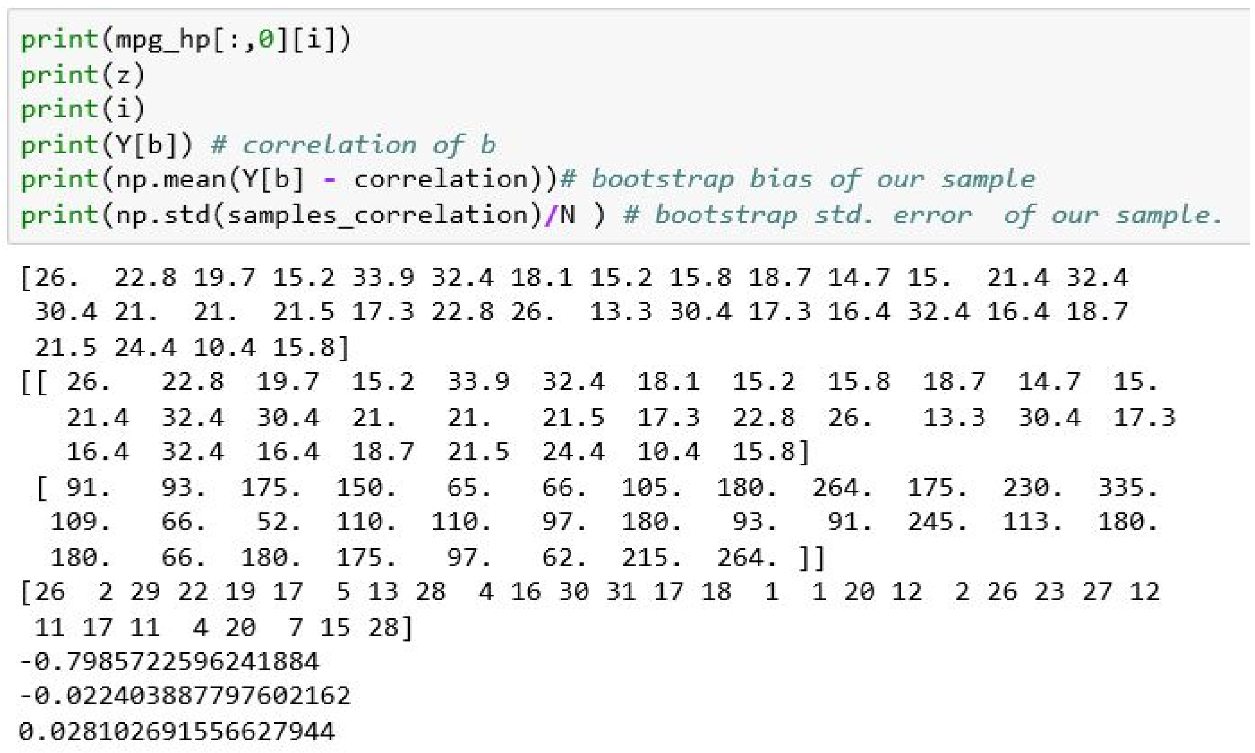
```
mpg_hp = dataset[['mpg', 'hp']]
mpg_hp = np.array(mpg_hp)
corr = dataset[['mpg', 'hp']].corr()
corr = np.array(corr)
```

Now this above evaluates correlation between two vectors ‘mpg and ‘hp’.
The bootstrap method can be used to estimate a quantity of a population. This is done by repeatedly taking small samples, calculating the statistic, and taking the average of the calculated statistics. Now below we are declaring Boot = 1000 for number of iterations to sample and corr[0][1] stating the index and the entire function will calculate the bootstrap estimate.

```
correlation = corr[0][1]
Boot = 1000
N = len(X)
Y = np.zeros(Boot)
arrays = np.arange(1,N)
#Bootstrap method to calculate the estimate of standard error
for b in range(Boot):
    i = np.random.choice(arrays, size = N, replace = True)
    mpg = mpg_hp[:,0][i]
    hp = mpg_hp[:,1][i]
    z = np.array((mpg,hp))
    samples_correlation = np.array(pd.DataFrame(z.T).corr())
    Y[b] = samples_correlation[0][1]
```

Now we will print the **estimate bias** that is **-0.022** and which is given by **sample mean - correlation**. And we also calculate the **std. Error** by : **Std deviation(sample) / N** which comes out to be **0.0281.**

```
print(mpg_hp[:,0][i])
print(z)
print(i)
print(Y[b]) # correlation of b
print(np.mean(Y[b] - correlation))# bootstrap bias of our sample
print(np.std(samples_correlation)/N ) # bootstrap std. error  of our sample.
```



```
print(mpg_hp[:,0][i])
print(z)
print(i)
print(Y[b]) # correlation of b
print(np.mean(Y[b] - correlation))# bootstrap bias of our sample
print(np.std(samples_correlation)/N ) # bootstrap std. error  of our sample.
```

```
[26.  22.8 19.7 15.2 33.9 32.4 18.1 15.2 15.8 18.7 14.7 15.  21.4 32.4
 30.4 21.  21.  21.5 17.3 22.8 26.  13.3 30.4 17.3 16.4 32.4 16.4 18.7
 21.5 24.4 10.4 15.8]
[[ 26.   22.8  19.7  15.2  33.9  32.4  18.1  15.2  15.8  18.7  14.7  15.
   21.4  32.4  30.4  21.   21.   21.5  17.3  22.8  26.   13.3  30.4  17.3
   16.4  32.4  16.4  18.7  21.5  24.4  10.4  15.8]
 [ 91.   93.  175.  150.   65.   66.  105.  180.  264.  175.  230.  335.
  109.   66.   52.  110.  110.   97.  180.   93.   91.  245.  113.  180.
  180.   66.  180.  175.   97.   62.  215.  264. ]]
[26  2 29 22 19 17  5 13 28  4 16 30 31 17 18  1  1 20 12  2 26 23 27 12
 11 17 11  4 20  7 15 28]
-0.7985722596241884
-0.022403887797602162
0.028102691556627944
```

Code:

```
import scipy.stats
from scipy.stats import pearsonr
import numpy as np
dataset = pd.read_csv(r"C:\Users\anmol\Downloads\mtcars_hw3.csv")
mpg_hp = dataset[['mpg', 'hp']]
mpg_hp = np.array(mpg_hp)
corr = dataset[['mpg', 'hp']].corr()
corr = np.array(corr)
correlation = (corr[0][1])
Boot = 1000
N = len(X)
Y = np.zeros(Boot)
arrays = np.arange(1,N)
#Bootstrap method to calculate the estimate of standard error
for b in range(Boot):
   i = np.random.choice(arrays, size = N, replace = True)
   mpg = mpg_hp[:,0][i]
   hp = mpg_hp[:,1][i]
   z = np.array((mpg,hp))
   samples_correlation = np.array(pd.DataFrame(z.T).corr())
   Y[b] = samples_correlation[0][1]
print(mpg_hp[:,0][i])
print(z)
print(i)
print(Y[b]) # correlation of b
print(np.mean(Y[b] - correlation))# bootstrap bias of our sample
print(np.std(samples_correlation)/N ) # bootstrap std. error  of our sample.
```

Problem 4: 2

We are going to first import our mtcars_hw3 dataset which has mpg, hp as columns whose jackknife bias and std. Error with statistic as correlation is evaluated as follows:

Defining a function for correlation using pandas corr() for each value of the vector comprising of ‘mpg and ‘hp’.

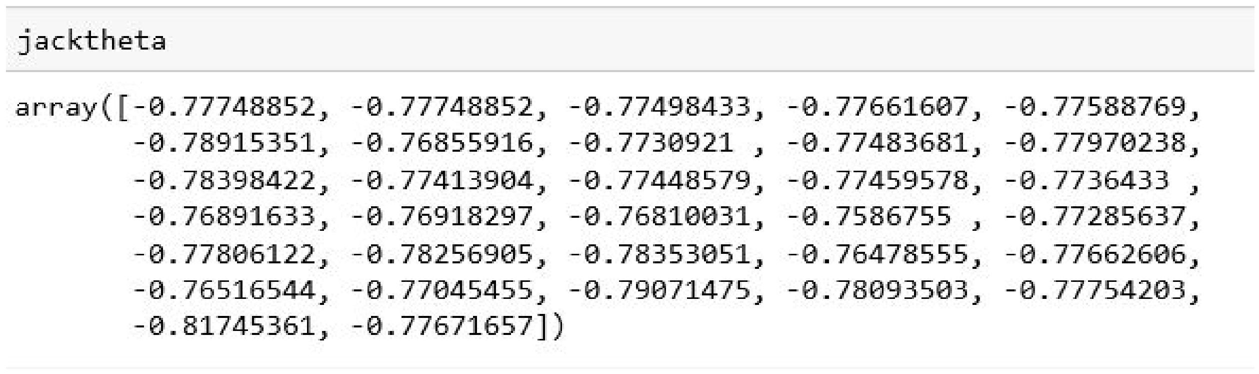
```
def correlation_coeff(a):
    c = pd.DataFrame(x).corr()
    c = np.array(c)
    answer = c[0][1]
    return np.array(ans)
```

No determines the length of the (mpg,hp) vector.

```
No = len(mpg_hp)
mpg = mpg_hp[:,0]
hp = mpg_hp[:,1]
```

Now below we calculate jackknife estimate so that we can calculate jackknife bias (jackbias) and std. error(jack_se). Here we use **np.corrcoef()** to evaluate correlation coefficient.

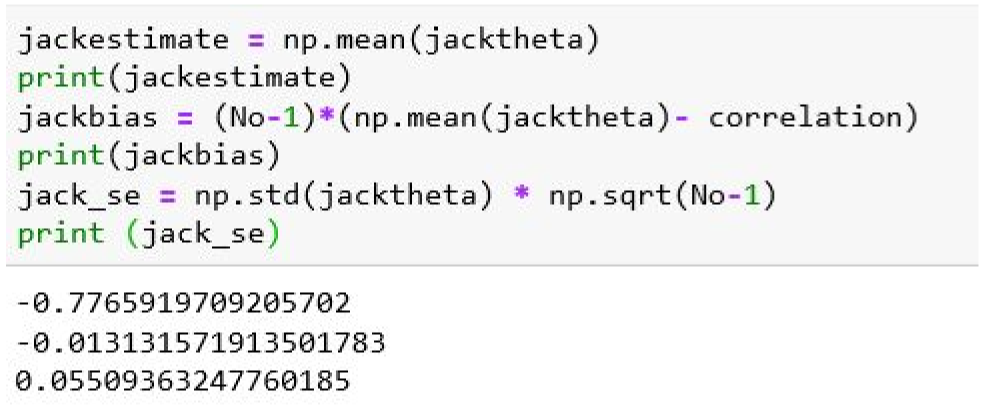
```
jacktheta = np.zeros(No)
for i in range(No):
    idx = np.arange(No)
    jacktheta[i] = np.corrcoef(mpg[idx!=i],hp[idx!=i])[0][1]
jackestimate = np.mean(jacktheta)
print(jackestimate)
jackbias = (No-1)*(np.mean(jacktheta)- correlation)
print(jackbias)
jack_se = np.std(jacktheta) * np.sqrt(No-1)
print (jack_se)
```



```
jacktheta
```

```
array([-0.77748852, -0.77748852, -0.77498433, -0.77661607, -0.77588769,
       -0.78915351, -0.76855916, -0.7730921 , -0.77483681, -0.77970238,
       -0.78398422, -0.77413904, -0.77448579, -0.77459578, -0.7736433 ,
       -0.76891633, -0.76918297, -0.76810031, -0.7586755 , -0.77285637,
       -0.77806122, -0.78256905, -0.78353051, -0.76478555, -0.77662606,
       -0.76516544, -0.77045455, -0.79071475, -0.78093503, -0.77754203,
       -0.81745361, -0.77671657])
```

The above output shows values of jackknife estimate vector given by **jacktheta.**



```
jackestimate = np.mean(jacktheta)
print(jackestimate)
jackbias = (No-1)*(np.mean(jacktheta)- correlation)
print(jackbias)
jack_se = np.std(jacktheta) * np.sqrt(No-1)
print (jack_se)
```

```
-0.7765919709205702
-0.013131571913501783
0.05509363247760185
```

Here **-0.0131** is our jackknife bias and **0.055** as jackknife std. Error.

Code:

```
def correlation_coeff(a):
    c = pd.DataFrame(x).corr()
    c = np.array(c)
    answer = c[0][1]
    return np.array(ans)

No = len(mpg_hp)
mpg = mpg_hp[:,0]
hp = mpg_hp[:,1]
jacktheta = np.zeros(No)
for i in range(No):
    idx = np.arange(No)
    jacktheta[i] = np.corrcoef(mpg[idx!=i],hp[idx!=i])[0][1]
jackestimate = np.mean(jacktheta)
print(jackestimate)
jackbias = (No-1)*(np.mean(jacktheta)- correlation)
print(jackbias)
jack_se = np.std(jacktheta) * np.sqrt(No-1)
print (jack_se)
```

Problem 5 :

In this problem we are going to calculate the bootstrap-t interval for same mtcars dataset. The simplest in terms of implementation is a percentile bootstrap interval: generate B bootstrap samples of the original data and calculate B statistics from these samples. Then take the alpha/2 and 1-alpha/2 quantiles of the statistic to form a confidence interval of level alpha. However, this interval does not behave as well as other intervals, as we will show:awe can calculate the confidence intervals.

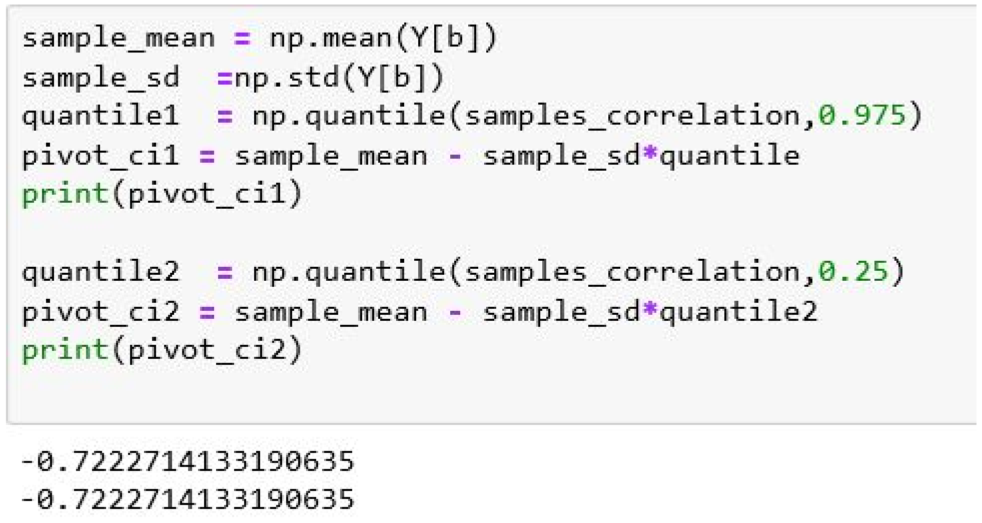
```
dataset = pd.read_csv(r"C:\Users\anmol\Downloads\mtcars_hw3.csv")
mpg_hp = dataset[['mpg', 'hp']]
mpg_hp = np.array(mpg_hp)
```

Then generate B=1000 bootstrap samples of correlation between mpg and hp.
This is done by first ordering the statistics, then selecting values at the chosen percentile for the confidence interval. The chosen percentile in this case is called alpha.

```
corr = dataset[['mpg', 'hp']].corr()
corr = np.array(corr)
correlation = (corr[0][1])
Boot = 1000
N = len(X)
Y = np.zeros(Boot)
arrays = np.arange(1,N)

#Bootstrap method to calculate the estimate of standard error
for b in range(Boot):
    i = np.random.choice(arrays, size = N, replace = True)
    mpg = mpg_hp[:,0][i]
    hp = mpg_hp[:,1][i]
    z = np.array((mpg,hp))
    samples_correlation = np.array(pd.DataFrame(z.T).corr())
    Y[b] = samples_correlation[0][1]
```

For example, if we were interested in a confidence interval of 95%, then alpha would be 0.95 and we would select the value at the 2.5% percentile as the lower bound and the 97.5% percentile as the upper bound on the statistic of interest.Then, for an alpha = .05 / 95% confidence interval, look at the .025 and .975 quantiles of the bootstrap statistics in the vector.

The quantity pivot shows that its distribution is approximately the same for each value of theta.



```
sample_mean = np.mean(Y[b])
sample_sd  =np.std(Y[b])
quantile1  = np.quantile(samples_correlation,0.975)
pivot_ci1 = sample_mean - sample_sd*quantile
print(pivot_ci1)

quantile2  = np.quantile(samples_correlation,0.25)
pivot_ci2 = sample_mean - sample_sd*quantile2
print(pivot_ci2)
```

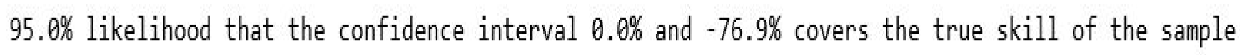
```
-0.7222714133190635
-0.7222714133190635
```

For example, if we calculated 1,000 statistics from 1,000 bootstrap samples, then the lower bound would be the 25th value and the upper bound would be the 975th value, assuming the list of statistics was ordered.

In this, we are calculating a non-parametric confidence interval that does not make any assumption about the functional form of the distribution of the statistic. This confidence interval is often called the empirical confidence interval. In below code we use the same logic using Y[b] samples generated in bootstrap function in problem 4-1

```
# confidence intervals
alpha = 0.95
p = ((1.0-alpha)/2.0) * 100
lower = max(0.0, np.percentile(Y[b], p))
p = (alpha+((1.0-alpha)/2.0)) * 100
upper = min(1.0, np.percentile(Y[b], p))
print('%.1f%% likelihood that the confidence interval %.1f%% and %.1f%% covers the true skill of the sample' % (alpha*100, lower*100, upper*100))
```

Output:



```
95.0% likelihood that the confidence interval 0.0% and -76.9% covers the true skill of the sample
```

Problem 2 : 1

Lets import some modules which will be required to run the code for this problem:

```
%matplotlib inline
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
```

Now we declare the group sizes which are given as [120,18,20,34] and its respective probabilities. We keep seed =1 and randomly select theta from range(0,1) since probability ‘p’ is 0<p<1. And ‘dg’ gives us the sum of group size multiplied with their respective probability which helps us to draw that number of random samples.

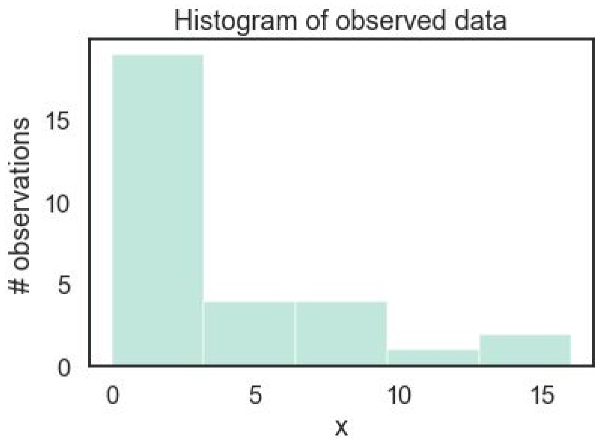
```
sizes = np.r_[125, 18, 20, 34]
np.random.seed(1)
theta = np.random.rand(10)
t = np.r_[2+theta, (1-theta), theta]/4
dg = sizes[0]*t[0] + sizes[1]*t[1] + sizes[2]*t[2] + sizes[3]*t[3]
dg = np.floor(dg)
print (dg)
>>>117.0
```

```
data = np.random.multinomial(117,t*0.1)
```

So above given line draws are desired multinomial distribution.

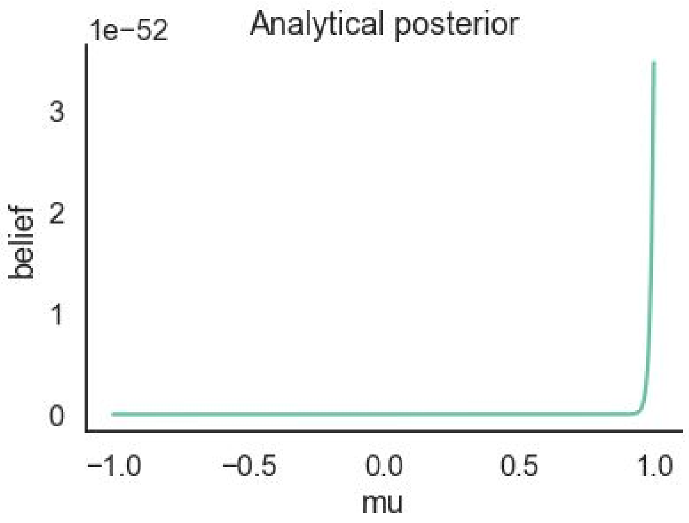
Now we will see histogram of our observations generated below:

```
ax = plt.subplot()
sns.distplot(data, kde=False, ax=ax)
_ = ax.set(title='Histogram of observed data', xlabel='x', ylabel='# observations');
```



Next, we have to define our model. In this simple case, we will assume that this data is normal distributed, i.e. the likelihood of the model is normal. As you know, a normal distribution has two parameters -- mean μ and standard deviation σ.
For simplicity, we'll assume we know that σ=1 and we'll want to infer the posterior for μ. For each parameter we want to infer, we have to chose a prior. For simplicity, lets also assume a Normal distribution as a prior for μ.
μ~Normal(0,1)
x|μ~Normal(x;μ,1)
We actually compute the posterior analytically. That's because for a normal likelihood with known standard deviation, the normal prior for mu is conjugate (conjugate here means that our posterior will follow the same distribution as the prior), so we know that our posterior for μ is also normal.

```
def calc_posterior_analytical(data, x, mu_0, sigma_0):
    sigma = 1.
    n = len(data)
    mu_post = (mu_0 / sigma_0**2 + data.sum() / sigma**2) / (1. / sigma_0**2 + n / sigma**2)
    sigma_post = (1. / sigma_0**2 + n / sigma**2)**-1
    return norm(mu_post, np.sqrt(sigma_post)).pdf(x)

ax = plt.subplot()
x = np.linspace(-1, 1, 500)
posterior_analytical = calc_posterior_analytical(data, x, 0., 1.)
ax.plot(x, posterior_analytical)
ax.set(xlabel='mu', ylabel='belief', title='Analytical posterior');
sns.despine()
```



Now on to the sampling logic for using MCMC sampler we need starting parameter position. Let it be :

```
mu_current = 1.
```

The Metropolis sampler is very simple and just takes a sample from a normal distribution (no relationship to the normal we assume for the model) centered around your current mu value (i.e. mu_current) with a certain standard deviation (proposal_width) that will determine how far you propose jumps (here we're use scipy.stats.norm):

```
proposal = norm(mu_current, proposal_width).rvs()
```

We quantify fit by computing the probability of the data, given the likelihood (normal) with the proposed parameter values (proposed mu and a fixed sigma = 1). This can easily be computed by calculating the probability for each data point using scipy.stats.normal(mu, sigma).pdf(data) and then multiplying the individual probabilities

```
likelihood_current = norm(mu_current, 1).pdf(data).prod()
likelihood_proposal = norm(mu_proposal, 1).pdf(data).prod()

# Compute prior probability of current and proposed mu
prior_current = norm(mu_prior_mu, mu_prior_sd).pdf(mu_current)
prior_proposal = norm(mu_prior_mu, mu_prior_sd).pdf(mu_proposal)

# Nominator of Bayes formula
p_current = likelihood_current * prior_current
p_proposal = likelihood_proposal * prior_proposal
```

Eventually we'll get to mu = 0 (or close to it) from where no more moves will be possible. However, we want to get a posterior so we'll also have to sometimes accept moves into the other direction. The key trick is by dividing the two probabilities, so we get an acceptance probability. You can already see that if p_proposal is larger, that probability will be > 1 and we'll definitely accept.

```
p_accept = p_proposal / p_current
accept = np.random.rand() < p_accept

if accept:
   # Update position
   cur_pos = proposal
```

This simple procedure gives us samples from the posterior.

We can show this by computing the acceptance ratio over the normalized posterior and seeing how it's equivalent to the acceptance ratio of the unnormalized posterior (lets say μ0 is our current position, and μ is our proposal):

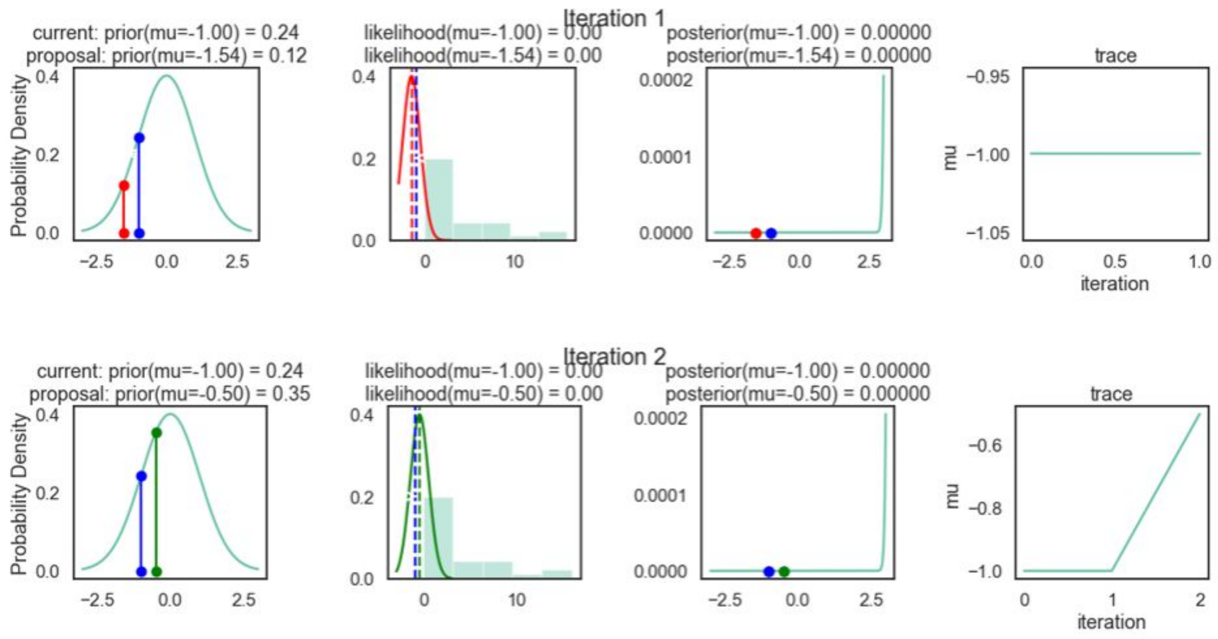
$$\frac{\frac{P(x|\mu)P(\mu)}{P(x)}}{\frac{P(x|\mu_0)P(\mu_0)}{P(x)}} = \frac{P(x|\mu)P(\mu)}{P(x|\mu_0)P(\mu_0)}$$

In words, dividing the posterior of proposed parameter setting by the posterior of the current parameter setting, P(x) that quantity we can't compute -- gets canceled out. So you can intuit that we're actually dividing the full posterior at one position by the full posterior at another position. That way, we are visiting regions of high posterior probability relatively more often than those of low posterior probability.
Now putting all this together we will define a sampler function below using above mentioned theory.

```
def sampler(data, samples=4, mu_init=.5, proposal_width=.5, plot=False, mu_prior_mu=0,
mu_prior_sd=1.):
    mu_current = mu_init
    posterior = [mu_current]
    for i in range(samples):
        # suggest new position
        mu_proposal = norm(mu_current, proposal_width).rvs()

        # Compute likelihood by multiplying probabilities of each data point
        likelihood_current = norm(mu_current, 1).pdf(data).prod()
        likelihood_proposal = norm(mu_proposal, 1).pdf(data).prod()

        # Compute prior probability of current and proposed mu
        prior_current = norm(mu_prior_mu, mu_prior_sd).pdf(mu_current)
        prior_proposal = norm(mu_prior_mu, mu_prior_sd).pdf(mu_proposal)

        p_current = likelihood_current * prior_current
        p_proposal = likelihood_proposal * prior_proposal

        # Accept proposal?
        p_accept = p_proposal / p_current

        # Usually would include prior probability, which we neglect here for simplicity
        accept = np.random.rand() < p_accept
        if plot:
            plot_proposal(mu_current, mu_proposal, mu_prior_mu, mu_prior_sd, data, accept, posterior, i)

        if accept:
            # Update position
            mu_current = mu_proposal

        posterior.append(mu_current)

    return posterior

# Function to display
def plot_proposal(mu_current, mu_proposal, mu_prior_mu, mu_prior_sd, data, accepted, trace, i):
    from copy import copy
    trace = copy(trace)
    fig, (ax1, ax2, ax3, ax4) = plt.subplots(ncols=4, figsize=(16, 4))
    fig.suptitle('Iteration %i' % (i + 1))
    x = np.linspace(-3, 3, 5000)
    color = 'g' if accepted else 'r'

    # Plot prior
    prior_current = norm(mu_prior_mu, mu_prior_sd).pdf(mu_current)
    prior_proposal = norm(mu_prior_mu, mu_prior_sd).pdf(mu_proposal)
    prior = norm(mu_prior_mu, mu_prior_sd).pdf(x)
    ax1.plot(x, prior)
    ax1.plot([mu_current] * 2, [0, prior_current], marker='o', color='b')
    ax1.plot([mu_proposal] * 2, [0, prior_proposal], marker='o', color=color)
    ax1.annotate("", xy=(mu_proposal, 0.2), xytext=(mu_current, 0.2),
                 arrowprops=dict(arrowstyle="->", lw=2.))
    ax1.set(ylabel='Probability Density', title='current: prior(mu=%.2f) = %.2f\nproposal: prior(mu=%.2f) =
%.2f' % (mu_current, prior_current, mu_proposal, prior_proposal))

    # Likelihood
    likelihood_current = norm(mu_current, 1).pdf(data).prod()
    likelihood_proposal = norm(mu_proposal, 1).pdf(data).prod()
    y = norm(loc=mu_proposal, scale=1).pdf(x)
    sns.distplot(data, kde=False, norm_hist=True, ax=ax2)
    ax2.plot(x, y, color=color)
    ax2.axvline(mu_current, color='b', linestyle='--', label='mu_current')
    ax2.axvline(mu_proposal, color=color, linestyle='--', label='mu_proposal')
    #ax2.title('Proposal {}'.format('accepted' if accepted else 'rejected'))
    ax2.annotate("", xy=(mu_proposal, 0.2), xytext=(mu_current, 0.2),
                 arrowprops=dict(arrowstyle="->", lw=2.))
    ax2.set(title='likelihood(mu=%.2f) = %.2f\nlikelihood(mu=%.2f) = %.2f' % (mu_current,
1e14*likelihood_current, mu_proposal, 1e14*likelihood_proposal))

    # Posterior
    posterior_analytical = calc_posterior_analytical(data, x, mu_prior_mu, mu_prior_sd)
    ax3.plot(x, posterior_analytical)
    posterior_current = calc_posterior_analytical(data, mu_current, mu_prior_mu, mu_prior_sd)
    posterior_proposal = calc_posterior_analytical(data, mu_proposal, mu_prior_mu, mu_prior_sd)
    ax3.plot([mu_current] * 2, [0, posterior_current], marker='o', color='b')
    ax3.plot([mu_proposal] * 2, [0, posterior_proposal], marker='o', color=color)
    ax3.annotate("", xy=(mu_proposal, 0.2), xytext=(mu_current, 0.2),
                 arrowprops=dict(arrowstyle="->", lw=2.))
    #x3.set(title=r'prior x likelihood $\propto$ posterior')
    ax3.set(title='posterior(mu=%.2f) = %.5f\nposterior(mu=%.2f) = %.5f' % (mu_current, posterior_current,
mu_proposal, posterior_proposal))

    if accepted:
        trace.append(mu_proposal)
    else:
        trace.append(mu_current)
    ax4.plot(trace)
    ax4.set(xlabel='iteration', ylabel='mu', title='trace')
    plt.tight_layout()
    #plt.legend()
```

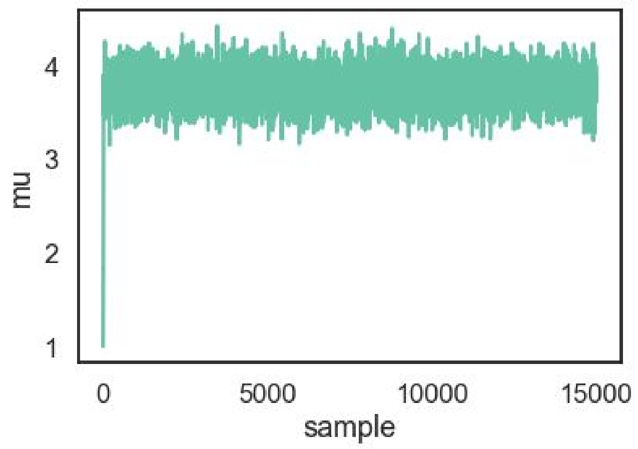
To visualize the sampling, we'll create plots for some quantities that are computed. Each row below is a single iteration through our Metropolis sampler.The first columns is our prior distribution -- what our belief about μ is before seeing the data. You can see how the distribution is static and we only plug in our μ proposals. The vertical lines represent our current μ in blue and our proposed μ in either red or green (rejected or accepted, respectively). The 2nd column is our likelihood and what we are using to evaluate how good our model explains the data. You can see that the likelihood function changes in response to the proposed μ. The blue histogram which is our data. The solid line in green or red is the likelihood with the currently proposed mu. Intuitively, the more overlap there is between likelihood and data, the better the model explains the data and the higher the resulting probability will be. The dotted line of the same color is the proposed mu and the dotted blue line is the current mu.
The 3rd column is our posterior distribution. Here I am displaying the normalized posterior but as we found out above, we can just multiply the prior value for the current and proposed μ 's by the likelihood value for the two μ 's to get the unnormalized posterior values (which we use for the actual computation), and divide one by the other to get our acceptance probability. The 4th column is our trace (i.e. the posterior samples of μ. we're generating) where we store each sample irrespective of whether it was accepted or rejected (in which case the line just stays constant).

```
np.random.seed(123)
sampler(data, samples=2, mu_init=-1., plot=True);
```



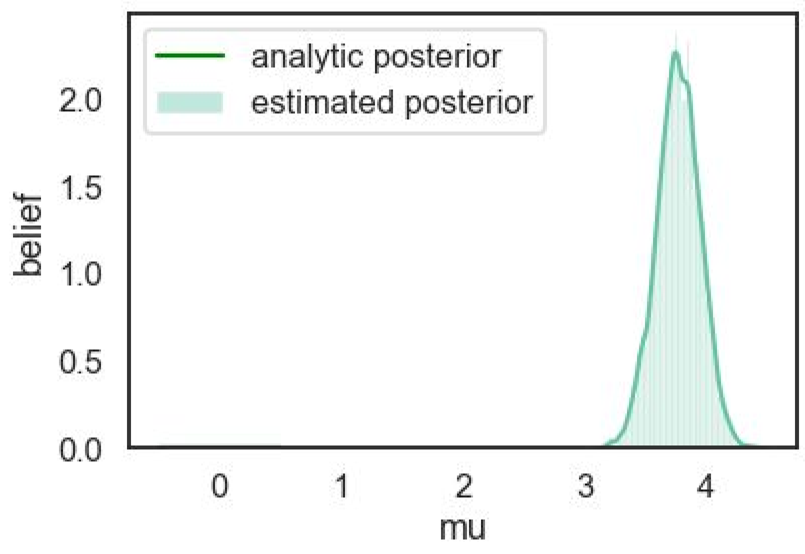
To get a sense of what this produces, lets draw a lot of samples and plot them.

```
posterior = sampler(data, samples=15000, mu_init=1.)
fig, ax = plt.subplots()
ax.plot(posterior)
_ = ax.set(xlabel='sample', ylabel='mu');
```



This is usually called the trace. To now get an approximation of the posterior (the reason why we're doing all this), we simply take the histogram of this trace. It's important to keep in mind that although this looks similar to the data we sampled above to fit the model, the two are completely separate. The below plot represents our belief in mu. In this case it just happens to also be normal but for a different model, it could have a completely different shape than the likelihood or prior.

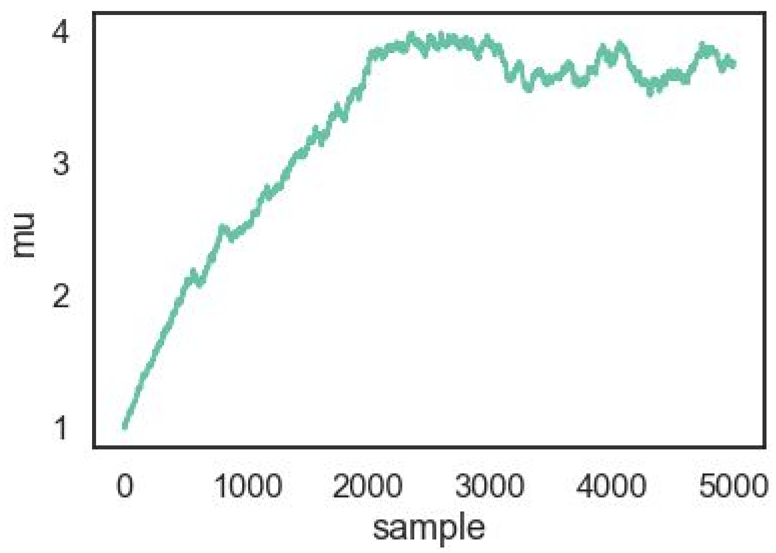
```
ax = plt.subplot()

sns.distplot(posterior[500:], ax=ax, label='estimated posterior')
x = np.linspace(-.5, .5, 500)
post = calc_posterior_analytical(data, x, 0, 1)
ax.plot(x, post, 'g', label='analytic posterior')
_ = ax.set(xlabel='mu', ylabel='belief');
ax.legend();
```



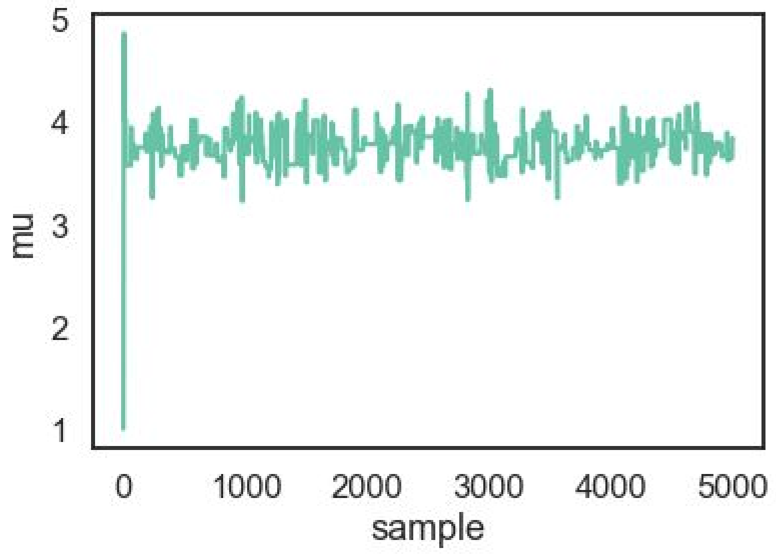
As you can see, by following the above procedure, we get samples from the same distribution as what we derived analytically.

Above we set the proposal width to 0.5. That turned out to be a pretty good value. In general you don't want the width to be too narrow because your sampling will be inefficient as it takes a long time to explore the whole parameter space and shows the typical random-walk behavior:

```
posterior_small = sampler(data, samples=5000, mu_init=1., proposal_width=.01)
fig, ax = plt.subplots()
ax.plot(posterior_small);
_ = ax.set(xlabel='sample', ylabel='mu');
```

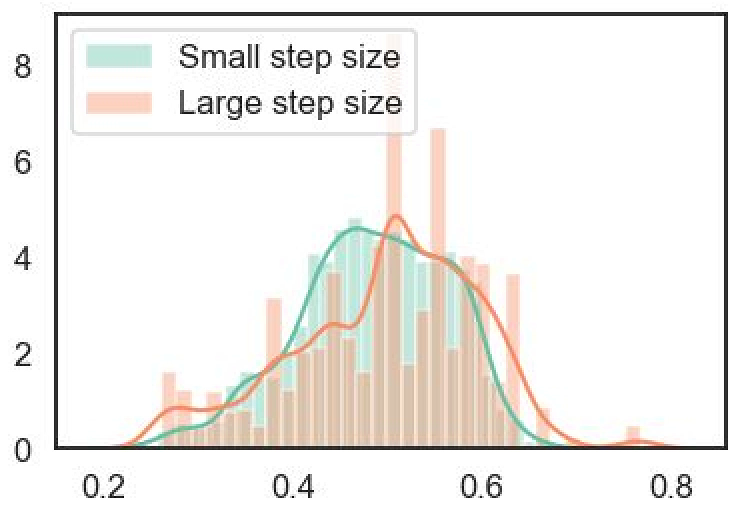


```
posterior_large = sampler(data, samples=5000, mu_init=1., proposal_width=3.)
fig, ax = plt.subplots()
ax.plot(posterior_large); plt.xlabel('sample'); plt.ylabel('mu');
_ = ax.set(xlabel='sample', ylabel='mu');
```



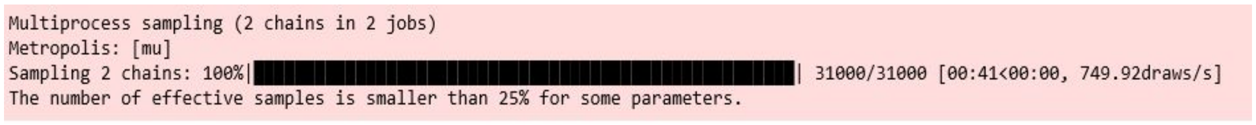
```
sns.distplot(posterior_small[1000:], label='Small step size')
sns.distplot(posterior_large[1000:], label='Large step size');
_ = plt.legend();
```

The above code will plot posterior distribution using seaborn distplot.The below image as you can see if we have more samples then this will eventually look like the true posterior.

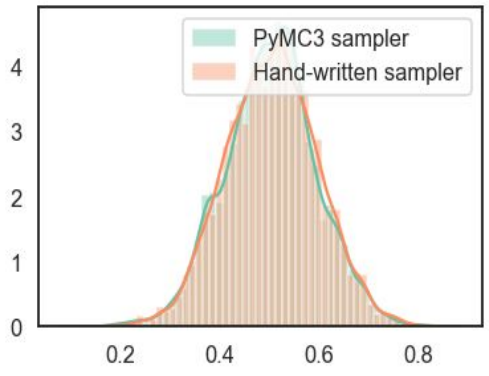


Now you can easily imagine that we could also add a sigma parameter for the standard-deviation and follow the same procedure for this second parameter. In that case, we would be generating proposals for mu and sigma but the algorithm logic would be nearly identical. Or, we could have data from a very different distribution like a Binomial and still use the same algorithm and get the correct posterior. That's pretty huge benefit of probabilistic programming. Just define the model you want and let MCMC take care of the inference. For example, the below model can be written in PyMC3 quite easily. Below we also use the Metropolis sampler (which automatically tunes the proposal width) and see that we get identical results. For more information, see the PyMC3 documentation.

```
import pymc3 as pm

with pm.Model():
    mu = pm.Normal('mu', 0, 1)
    sigma = 1.
    returns = pm.Normal('returns', mu=mu, sd=sigma, observed=data)

    step = pm.Metropolis()
    trace = pm.sample(15000, step)

sns.distplot(trace[2000:]['mu'], label='PyMC3 sampler');
sns.distplot(posterior[500:], label='Hand-written sampler');
plt.legend();
```



```
Multiprocess sampling (2 chains in 2 jobs)
Metropolis: [mu]
Sampling 2 chains: 100%|██████████████████████████████████████████████████| 31000/31000 [00:41<00:00, 749.92draws/s]
The number of effective samples is smaller than 25% for some parameters.
```



Problem 2:2

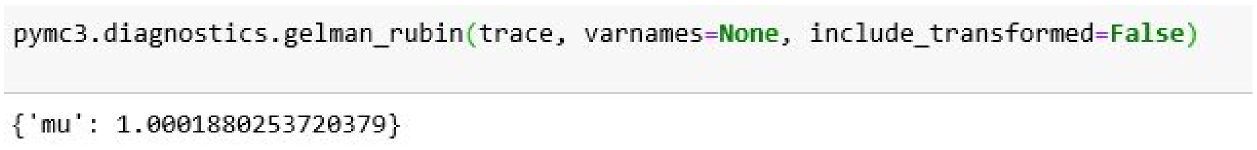
We will continue with the Problem 2-1 to check for gelman-rubin diagnostic measure R^ < 1.2 for its target distribution.

pymc3.diagnostics.gelman_rubin(mtrace, varnames=None, include_transformed=False)

Returns estimate of R for a set of traces.
The Gelman-Rubin diagnostic tests for lack of convergence by comparing the variance between multiple chains to the variance within each chain. If convergence has been achieved, the between-chain and within-chain variances should be identical. To be most effective in detecting evidence for nonconvergence, each chain should have been initialized to starting values that are dispersed relative to the target distribution.

Parameters:
mtrace (MultiTrace or trace object) – A MultiTrace object containing parallel traces (minimum 2) of one or more stochastic parameters.
varnames (list) – Names of variables to include in the rhat report
include_transformed (bool) – Flag for reporting automatically transformed variables in addition to original variables (defaults to False).
Notes
The diagnostic is computed by:
R^=V^/W where W is the within-chain variance and V^ is the posterior variance estimate for the pooled traces. This is the potential scale reduction factor, which converges to

unity when each of the traces is a sample from the target posterior. Values greater than one indicate that one or more chains have not yet converged.



```
pymc3.diagnostics.gelman_rubin(trace, varnames=None, include_transformed=False)
```

```
{'mu': 1.0001880253720379}
```

The above mu gives the R^ for (mu) which you can see the chain is converging but with R^ < 1.2

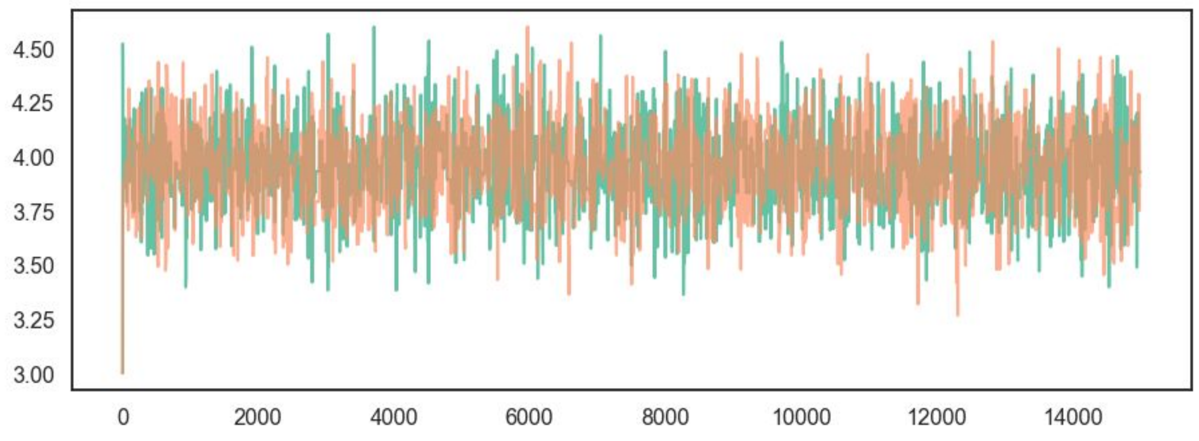
Now we are declaring two chains using sampler function from Problem 2-1 as follows:

```
chain1 = sampler(data,samples=15000,mu_init=3,proposal_width=2,
            mu_prior_mu=mu_prior,mu_prior_sd = sigma_prior)
chain2 = sampler(data,samples=15000,mu_init=3,proposal_width=2,
            mu_prior_mu=mu_prior,mu_prior_sd = sigma_prior)
```

Now we will plot these chains and check for convergence of these chains at regular intervals.

```
plt.figure(figsize=(15,5))
plt.plot(np.arange(15001),chain1)
plt.plot(np.arange(15001),chain2,alpha=.7)
```

```
[<matplotlib.lines.Line2D at 0x2d43dc41e48>]
```



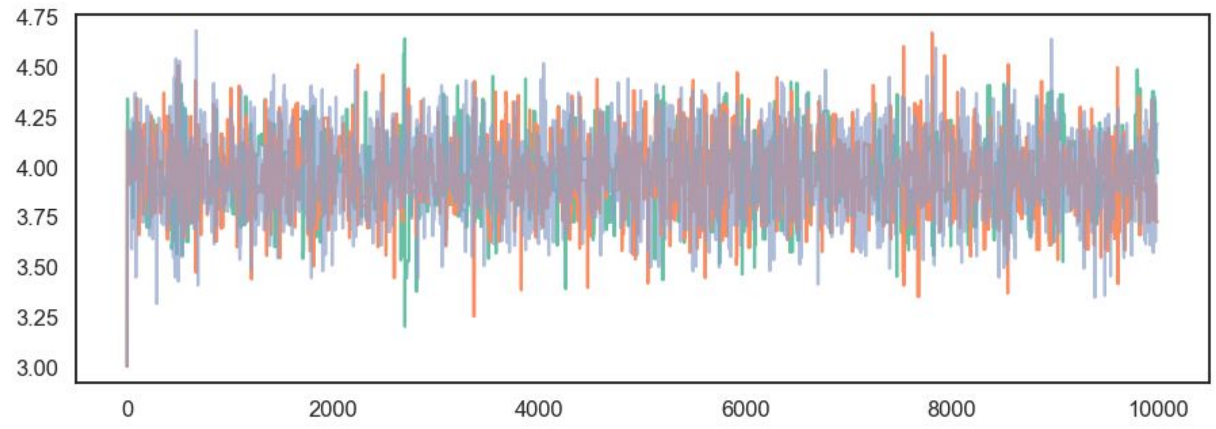
Now we will also check with 3 chains with different widths:

```
chain1 = sampler(data,samples=10000,mu_init=3,proposal_width=3,
          mu_prior_mu=mu_prior,mu_prior_sd = sigma_prior)
chain2 = sampler(data,samples=10000,mu_init=3,proposal_width=2,
          mu_prior_mu=mu_prior,mu_prior_sd = sigma_prior)
chain3 = sampler(data,samples=10000,mu_init=3,proposal_width=1,
          mu_prior_mu=mu_prior,mu_prior_sd = sigma_prior)
```

Let's check its plot:

```
plt.figure(figsize=(15,5))
plt.plot(np.arange(10001),chain1)
plt.plot(np.arange(10001),chain2)
plt.plot(np.arange(10001),chain3,alpha=.7)
```

```
[<matplotlib.lines.Line2D at 0x2d4416faa20>]
```



We can see that all chains are converging and hence we can see the same from the gelman rubin diagnostic measure also.