## Problem 1

1. To generate bivariate random sample with given joint density, I have written a function “gib\_sample” using the following algorithm.
2. Generate a random sample of size one from binomial and one from beta distribution for initial and
3. Again, generate a random sample of size one from binomial and one from beta distribution for and using previous and as the parameter for this step
4. Repeat step (b) for times
5. Finally, return sample excluding first (burn) observation from the sample

This function gib\_sample has five parameters-

a = *bivariate density function parameter*

b = *bivariate density function parameter*

n = *bivariate density function parameter*

sample\_size = *Size of Sample that we want to generate*

burn = *Number of observations that we want to exclude (Default 500)*

# -\*- coding: utf-8 -\*-

"""

Created on Wed Nov 23 21:08:43 2016

@author: Kanak

"""  
##################################################################  
# Problem 1 Part 1  
##################################################################

The following codes were written to import some python package.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.pyplot as pltbar

from mpl\_toolkits.mplot3d import Axes3D

from math import exp

from scipy.stats import chi2

from scipy.stats import binom

from scipy.special import gamma  
  
def gib\_sample(a, b, n, sample\_size = 1, burn = 500):

At first, I have added sample size and burning size to find the chain size of the sample. Then, for the initial random number, one random sample was generated from binomial distribution with parameter and 0.5. Here I have considered 0.5 because at the beginning we do not have any information about the probability of binomial distribution. Again, for , one random sample was generated from beta distribution with parameter and . After that, I have created an empty list “sam” to store generated sample and append the initial and in it.

chain = sample\_size + burn

x = np.random.binomial(n, 0.5, size=None)

y = np.random.beta(x + a, n - x +b, size=None)

sam = []

sam.append([x, y])

Now, within a loop I have done the same steps repeatedly until I get the desired sample size. Finally, I have returned the sample excluding first “burn” number of observations.

for \_ in range(1, chain):

x = np.random.binomial(n, y, size=None)

y = np.random.beta(x+a, n-x+b, size=None)

sam.append([x, y])

return(sam[burn:chain])

After generating the sample, I have calculated the probability for sample with the given pdf. To do so, I have written two functions “ncr” to calculating and then “prob\_func” for calculating probability using given pdf.

The functions “ncr” will take two parameter *n* and *r*. Then it will use the following equation to find .

First, minimum of *r* and *n-r* was calculated and assigned to *r* and if this is equal 0 then it will return 1 otherwise, it will calculate the product of all numbers between *n-r* and *n* and store it as “numer”. After that it will calculate *r!* and store it at “denom”. Finally, it will return ratio of this two number.

def ncr(n, r):

r = min(r, n-r)

if r == 0: return 1

numer = 1

for i in range(n, n-r, -1):

numer \*= i

denom = 1

for i in range(1, r+1):

denom \*= i

return numer//denom

To calculate the probability I have used the following pdf-

,

The function “prob\_func” will take data and parameters as input and it will use the above probability function to calculate the probability and finally it will return data with probability column.

def prob\_func(data, a, b, n):

length = len(x)

for i in range(0, length):

data[i].append(ncr(n,data[i][0])\*(data[i][1]\*\*(data[i][0]+a-1)

)\*((1-data[i][1])\*\*(n-data[i][0]+b-1)))

return(data)

It is found online that the marginal distribution of X is a Beta-Binomial distribution. The function “marginalpdf” was used to calculate the probability using Beta-Binomial distribution and it will take data and parameters related to pdf as input and it will use the following probability function to calculate the probability and finally it will return the probability column.

def marginalpdf(x, n, a, b):

fx = ncr(n, x)\*gamma(a+b)\*gamma(x+a)\*gamma(n-x+b)/

(gamma(a)\*gamma(b)\*gamma(a+b+n))

return fx

Here I called “gib\_sample” function to generate 10000 bivariate observation from the given probability fucntion.

burn = 500

a = 5

b = 10

n = 25

x = gib\_sample(sample\_size = 10000, a = a, b = b, n = n)

Here I called “prob\_func” function to calculate the probability for the generated sample and stored it in “xp” as a data frame. And then, assign the column name for the data frame as "x", "y", "p".

xp = pd.DataFrame(prob\_func(x, a, b, n))

xp.columns = ["x", "y", "p"]

Here, a three dimensional plot was drawn because it is a bivariate distribution. To draw the graph I have used “matplotlib” library. In this graph I have used bivariate sample in *x* and *y* axis and the probability in the *z* axis. Object “add\_subplot” will create a 3D plot area with 12 by 10 size and “scatter” will create the 3D scatter diagram. Finally, it will show the figure and to reset plot options I have used clf(), cla(), close().

fig = plt.figure(figsize=(12,10))

ax = fig.add\_subplot(111, projection='3d')

z = xp["p"]

x = xp["x"]

y = xp["y"]

ax.scatter(x, y, z, c='r', marker='.')

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Probability')

plt.show()

plt.clf()

plt.cla()

plt.close()

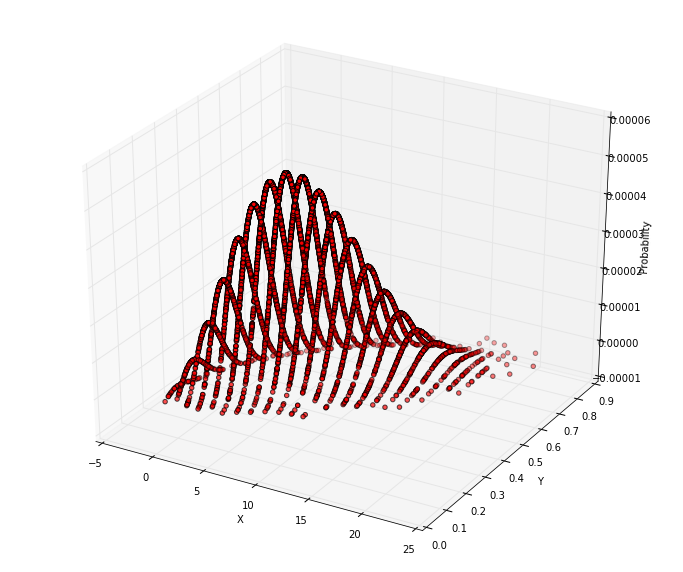


Figure 1: Bivariate Probability Density Plot

In the following, I have calculated the empirical marginal distribution for *X* as we know that the conditional distribution of *X* is Birnomial(*n*, *y*). Then, using Beta-Binomial distribution, marginal probability has been calculated using “marginalpdf” function to compare both empirical marginal distribution and true marginal distribution. The empirical marginal distribution is stored in “bp” and true marginal distribution is stored in “bpm”.

xran = np.array(range(0, n))

bp = []

for i in xran:

bp.append(np.mean(binom.pmf(i, n, xp["y"])))

bpm = []

for i in xran:

bpm.append(marginalpdf(i, n, a, b))

To compare both marginal distribution, I have plotted the probability. To create the figure, first I have created two plot area in a row with size 10 by 6 and both figure will share *x* and *y* axis. Then, the first figure shows the empirical marginal distribution and the second figure shows the true marginal distribution. Finally, to reset plot options I have used clf(), cla(), close()

fig, axs=pltbar.subplots(1,2, figsize=(10, 6), sharex='col', sharey='row')

width = 1

axs[0].bar(xran, bp, width, color="blue")

axs[1].bar(xran, bpm, width, color="blue")

pltbar.show()

pltbar.clf()

pltbar.cla()

pltbar.close()

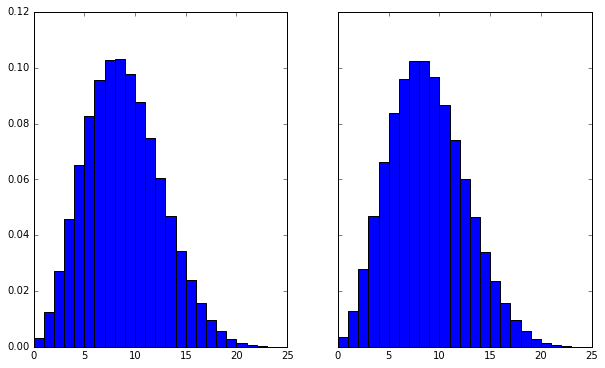


Figure 2: Marginal distribution

Again, I have calculated mean and variable for the simulated marginal distribution and true marginal distribution of *X* to compare the two distribution. The output is given below.

meanbp = np.sum(xran\*bp)

varbp = np.sum(xran\*\*2\*bp) - meanbp\*\*2

meanbpm = np.sum(xran\*bpm)

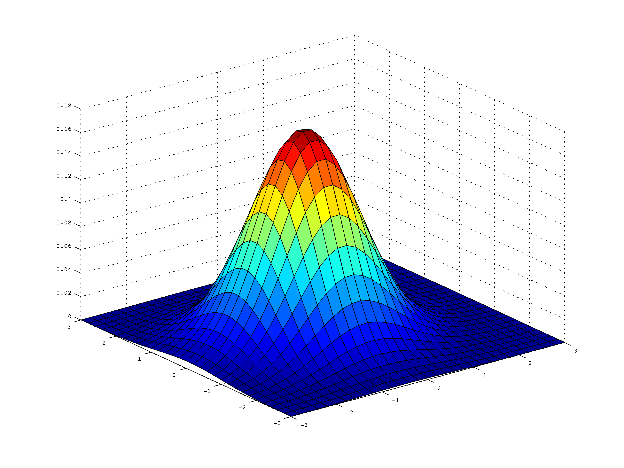
varbpm = np.sum(xran\*\*2\*bpm) - meanbpm\*\*2

#Output

# Simulated Mean: 8.36658079489 , Simulated Var: 13.7656244926 ,

# Marginal Dist Mean: 8.33329397009 , Marginal Dist Var: 13.8885608603

**Comment**: The 3D scatter plot has been used to represent density plot. Although it should be like a surface plot (for example following figure), but due to time I couldn’t figure out the figure.



Still, the above figure gives us an idea of the probability density plot. It seems that the generated sample represents the distribution. Again, from the marginal distribution plot, we can also say that the generated sample represents the true distribution.

## Problem 2

1. To monitor convergence of chain using Gelman-Rubin method for the Rayleigh distribution, I have used the same logit described in book Example 9.8. First, function “rlpdf” has been written to generate probability for Rayleigh distribution. Second, function “rand\_rayleigh” has been used to generate random sample from Rayleigh Distribution using Metropolis-Hastings algorithm. Finally, function “gelman\_rubin” has been used to calculate Gelman-Rubin statistic.

########################################################################

# Problem 2

########################################################################

For calculating pdf of Rayleigh distribution, “rlpdf” function has been used. This function will take two parameters, *x* random number and parameter . Then, it will use the following function to calculate pdf.

,

“rlpdf” function will return nothing if the value of *x* is less than 0 or if the value of less than or equal to 0.

def rlpdf(x, sigma):

if x < 0 or sigma <= 0: return

return((x/sigma\*\*2)\*exp(-x\*\*2/(2\*sigma\*\*2)))

“rand\_rayleigh” function has been used to generate random sample from Rayleigh Distribution using Metropolis-Hastings algorithm. The function has three parameter, “*n”* number of sample need to generate, “sigma” distribution parameter and “x1” the initial random number (default None). If we do not input any initial value, it will generate a random sample from chi-square distribution with df = 1 and it will consider this value as the initial random sample. Here, Chi-square distribution is considered as the proposal distribution.

def rand\_rayleigh(n, sigma, x1 = None):

if not x1 == None:

x = []

x.append(x1)

else:

x = []

x.append(np.random.chisquare(1))

Here, it will generate uniform random number of size *n* and assign this to *u*.

u = np.random.uniform(size = n)

After getting initial random number, I have repeated the following procedure.

1. Generate *Y* from chi-square distribution with df = .
2. If , we have accepted Y and append to *X*; otherwise is considered as .

for i in range(1, n):

xt = x[i-1]

y = np.random.chisquare(xt)

num = rlpdf(y, sigma)\*chi2.pdf(xt, y)

den = rlpdf(xt, sigma)\*chi2.pdf(y, xt)

if (u[i] <= num / den):

x.append(y)

else:

x.append(xt)

return (np.array(x))

“gelman\_rubin” function has been used to calculate given in Section 9.4 by Rizzo.

def gelman\_rubin(x):

At first, I have checked whether the given data is an array or not. If it is not an array type data, it will try to convert the data as an array and if it is not possible it will return a message.

if not isinstance(x, np.ndarray):

try:

x = np.array(x)

except:

print ("x is not possible to convert as array")

return

Then, it will find the number of observations in each chain and assign it to *n*. np.mean function will calculate mean by row, that is, mean of each chain. Then, using mean of each chain, I have calculated the between sequence variance and stored it in “*b*” and estimated the pooled variance by calculating mean of each sequence variance. Finally, was calculated and returned.

n = x.shape[1]

x\_means = np.mean(x, axis = 1)

b = n \* np.var(x\_means)

x\_w = np.var(x, axis = 1)

W = np.mean(x\_w)

v\_hat = W\* (n-1) / n + (b/n)

r\_hat = np.sqrt(v\_hat / W)

return(r\_hat)

Here, some parameter have been initialized. *n* for number of random number, *b* burn-in length, *sigma* Rayleigh distribution parameter, *x0* some initial values.

n = 15000

b = 500

sigma = 20

x0 = [0.5, 5, 10, 20, 50, 100, 300, 500]

indexlow = 1

indexup = 2000

In the following, I called the “rand\_rayleigh” function to generate sample of size *n* = 15000 using M-H algorithm. The line graph shows that at the first few iteration there is a trend and very high volatility in the sequence. We consider those iteration as a burning sequence.

x = rand\_rayleigh(n, sigma)

plt.plot(range(indexlow, indexup), x[indexlow:indexup])

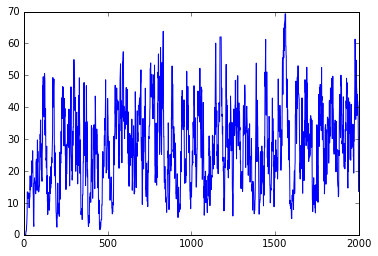


Figure 3: Line chart for the random sample

To monitor the convergence of the chain, I have generated eight sequence with different initial value given in *x0* and stored it in *x*. After that, I have converted those list of sequence as an array because the functions and methods that I have used in the following is based on the array type data.

x = []

for i in range(0, len(x0)):

x.append(rand\_rayleigh(n, sigma, x0[i]))

x = np.array(x)

“np.cumsum” function is used to find the cumulative sum of the sequences of array. Then, I have calculated the average of the cumulative sum.

cumsumx = np.cumsum(x, axis = 1)

ncol = np.array(range(1, x.shape[1]+1))

cumsumx = cumsumx / ncol

I have plotted all the chains to see whether it is converges or not.

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0, len(x0)):

axs[i].plot(range(0, n-b), cumsumx[i, b:n])

plt.show(

plt.clf()

plt.cla()

plt.close()

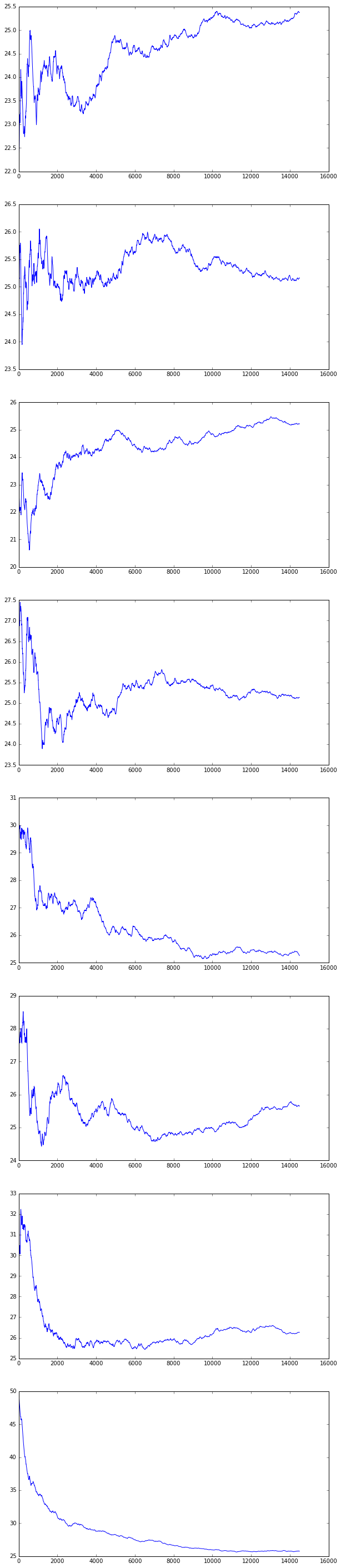
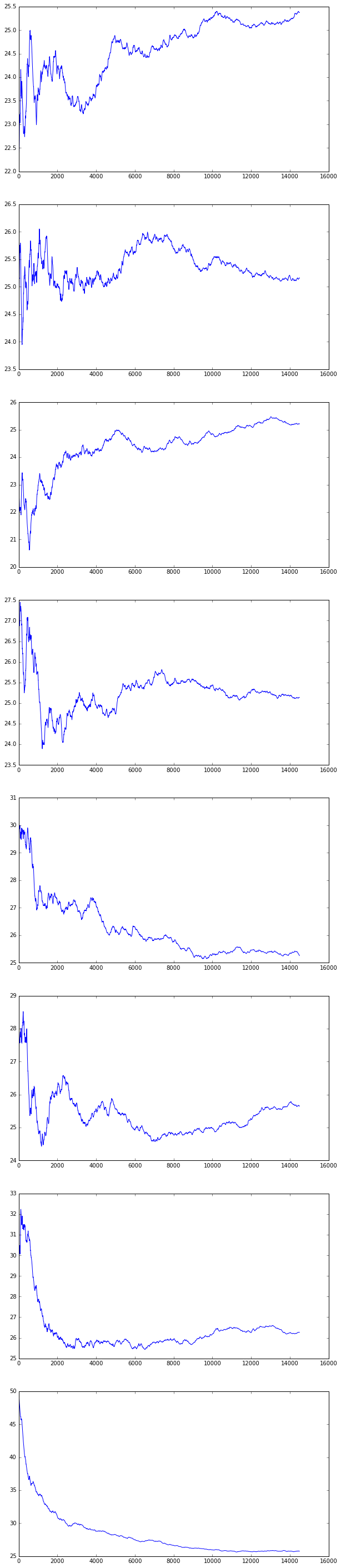


Figure 4: Line chart for each chain

Now, I have calculated the G-R statistic for the sequence of random sample to monitor the convergence of the sequence. Then, I have calculated for which number the G-R statistic is less than 1.2 to find the burning steps. That means, when the sequence states to converge.

rhat = []

xrange = np.array(range(b, n))

for i in xrange:

rhat.append(gelman\_rubin(cumsumx[:,0:i]))

rhat = np.array(rhat)

h = 1.2

lst = min(xrange[rhat<1.2])

print("Burning numbers", lst)

# Burning numbers 721

Here, I have plotted the G-R statistics with sequence number and indicated the 1.2 value with red lines that indicated that after that point the sequence starts to converge.

plt.plot(xrange,rhat)

plt.axvline(x=lst, c="red", linewidth=0.5,zorder=0)

plt.axhline(y=h, c="red", linewidth=0.5,zorder=0)

plt.show()

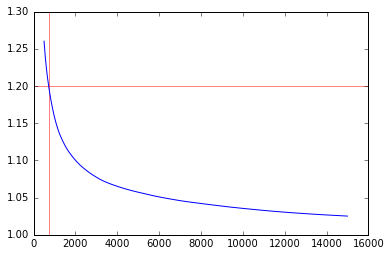


Figure 5: G-R Convergence plot

n = 15000

b = 10

sigma = [10, 20, 30]

x0 = [5, 10, 20, 50, 100]

listn = []

for j in range(0, len(sigma)):

x = []

for i in range(0, len(x0)):

x.append(rand\_rayleigh(n, sigma[j], x0[i]))

x = np.array(x)

cumsumx = np.cumsum(x, axis = 1)

ncol = np.array(range(1, x.shape[1]+1))

cumsumx = cumsumx / ncol

rhat = []

xrange = np.array(range(b, n))

for i in xrange:

rhat.append(gelman\_rubin(cumsumx[:,0:i]))

rhat = np.array(rhat)

h = 1.2

lst = min(xrange[rhat<1.2])

listn.append(lst)

plt.plot(xrange,rhat)

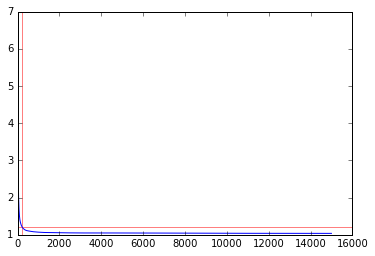
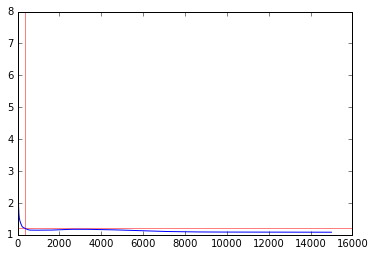
plt.axvline(x=lst, c="red", linewidth=0.5,zorder=0)

plt.axhline(y=h, c="red", linewidth=0.5,zorder=0)

plt.show()

print("Burning numbers", lst)

print(listn)

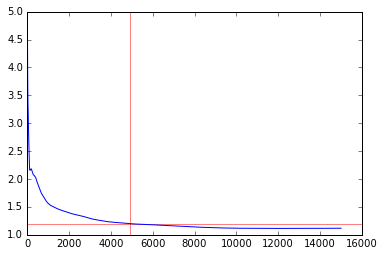


Figure 6: Convergence graph for different sigma value

**Comment**: From the Figure 4, it is seen that if we take larger initial value for the given sigma = 20, it converges faster. However, it is not the same for different sigma value. Figure 5 shows that for sigma = 20 and the different initial value the chain converges (G-R statistics less than 1.2) at 721 iteration.

It is seen that the convergence depends on the parameters of the distribution as well as the initial value. For sigma = 10, 20, 30 and the initial value = 5, 10, 20, 50, 100, we have found that the sequence converges at [227, 362, 4935] respectively for each sigma value. That indicates that as the sigma increases the burning steps increases with the given initial value. It is also found that convergence steps changes with the initial value (results not shown here)