Reversible Jump Markov Chain Monte Carlo: Applications for Model Selection in Linear Regression

Elise Dixon

Department of Statistics, Florida State University

STA5107: Computational Methods in Statistics II

Dr. Anuj Srivastava

April 27, 2023

Introduction

The Metropolis-Hastings algorithm is a simple and yet powerful tool toward approximating or estimating a distribution which is difficult to work with directly. Complex distributions often arise as the result of real-world random variables with unwieldy cumulative distribution functions (CDFs), a high dimensional parameter space, or multiple modes. The algorithm utilizes Monte Carlo Markov Chains (MCMC) to produce sequences which can be interpreted as random samples to approximate a specified probability distribution for which a closed form is not apparent. Though these methods are highly effective in single-variable settings, they are often employed toward estimating multi-dimensional distributions with many parameters; these types of models can become overwhelmingly complicated, as the number of possible subset models increases exponentially with the number of parameters.

A particularly relevant problem toward this type of parameter estimation, where the number of unknown parameters to estimate is itself unknown, is model selection in a high-dimensional parameter space. As by definition, the number of unknown parameters cannot be known in variable selection and, thus, neither can the dimension of the parameter space be known. Reversible Jump Markov Chain Monte Carlo (RJMCMC) is a method toward estimation of such multi-dimensional models where, between iterates, the MCMC estimation must be able to transverse or 'jump' dimension spaces. This method treats the distribution estimation as a union of multidimensional parameter spaces, with shifts allowed depending on state.

The specific problem which will be addressed is the utilization of RJMCMC in model selection for linear regression in which the number of predictors ultimately selected is, of course, unknown. Clearly, RJMCMC is an ideal method for dealing with such a problem. In experimental settings, the possible number of subsets grows exponentially with the number of predictors. However, we will work with pre-ordered variables such that the only unknown parameter to estimate is n, the number of predictors, and show that RJMCMC is an effective tool for determining this unknown parameter.

Methodology

We seek coefficients for the model $y = \sum_{i=1}^{n} x_i b_i + \epsilon$ where the total number of predictors $m = 10$ is less than selected predictors, n, and the number of iterations N = 100,000. In this model, x_i's are predictors, y is the response, and ϵ is the measurement noise. Given k independent observations, we seek a Bayesian solution to the joint estimation of $\{n, b_1, \dots, b_n\}$. In Bayesian estimation, unknown parameters are considered random variables themselves and the posterior density is constructed from the conditional distribution on observed data and such unknown parameters. A countable number of models each has n parameters and the joint posterior distribution of (n, b_n) given y is:

$$f(n, b_n|y) \propto f(y|n, b_n) f(b_n|n) f(n) ,$$

a product of the likelihood $f(y|n, b_n)$ and the joint posterior $f(b_n|n)f(n)$. The likelihood is:

$$f(y|n, b_n) = (\frac{1}{\sqrt{2\pi\sigma_0^2}})^k e^{-\left(\frac{1}{2\sigma_0^2}\right)||y - X_n b_n||^2}$$

the prior on b_n is given as:

$$f(b_n|n) = (\frac{1}{\sqrt{2\pi\sigma_p^2}})^k e^{-\left(\frac{1}{2\sigma_p^2}\right)||b_n - \mu_b||^2}$$

and the prior on n is:

$$f(n) = \frac{1}{m};$$

here, (n, b_n) are current samples from the posterior distribution, $b_n = \{b_1, \dots, b_n\}$.

In RJMCMC, this joint posterior is the target density of the MCMC where the states of the Markov chain may vary in dimension. The Markov chain must have a stationary distribution, $\pi = \pi P$; that is, with enough time steps, the Markov chain converges to a limiting distribution. The chain must be reversible, so that the probability of moving from one state to another is equal to the probability of moving back from the second state to the first; this requirement is to satisfy the detailed balance condition, a critical condition for the convergence of the chain to a stationary distribution. The Markov chain moves through the states by generating a suggested step from the proposal density and acceptance-rejection is performed such that the detailed balance condition is satisfied (Fan and Sisson). This acceptance probability is given as:

$$\rho = \min\left\{1, \frac{f(n^*, b_{n^*}|y)h_2}{f(n, b_n|y)h_1}\right\}$$

$$= \min\{1, \frac{e^{-(E_1 - E_2)}\left(2\pi\sigma_p^2\right)^{\frac{n-n^*}{2}} e^{-\left(\frac{1}{2\sigma_p^2}\right)\left(||b_{n^*} - \mu_b||^2 - ||b_n - \mu_b||^2\right)h_2}}{h_1}\}$$

where $E_1 = \left(\frac{1}{\sqrt{2\pi\sigma_r^2}}\right)||y - X_{n^*} b_{n^*}||^2$ and $E_2 = \left(\frac{1}{\sqrt{2\pi\sigma_r^2}}\right)||y - X_n b_n||^2$.

The steps of the algorithm are computed as follows:

1. Initialize parameters:
    a. Generate a sample X of size (m, k) (observation)
    b. Coefficient vector b
    c. Likelihood parameters and y vector representing the linear model response
2. For data (y, X), implement RJMCMC to sample from the posterior
    a. Generate a candidate n from $f(n)$
    b. If $n^* > n$, generate a random normal vector u with the same variance as coefficient vector b; the candidate coefficient vector is

$$b_{n^*} = \begin{bmatrix} b_n \\ 0 \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

where b_n is length n^*, u_1 is length n, u_2 being the length of $n^* - n$.

c. Compute likelihoods h_1 and h_2:

$$h_1(u) = \left(\frac{1}{\sqrt{2\pi\sigma_r^2}}\right)^{n^*} e^{-\left(\frac{1}{2\sigma_r^2}\right)||u||^2}$$

$$h_2(u) = \left(\frac{1}{\sqrt{2\pi\sigma_r^2}}\right)^{n} e^{-\left(\frac{1}{2\sigma_r^2}\right)||u_1||^2}$$

d. If $n^* < n$, generate a random normal vector u_1 as earlier u; candidate coefficient vector is now

$$b_{n^*} = b_n^1 + u_1, b_n = \begin{bmatrix} {b_n}^1 \\ {b_n}^2 \end{bmatrix}$$

where ${b_n}^1$ is length $n^* - n$, ${b_n}^2$ is length n.

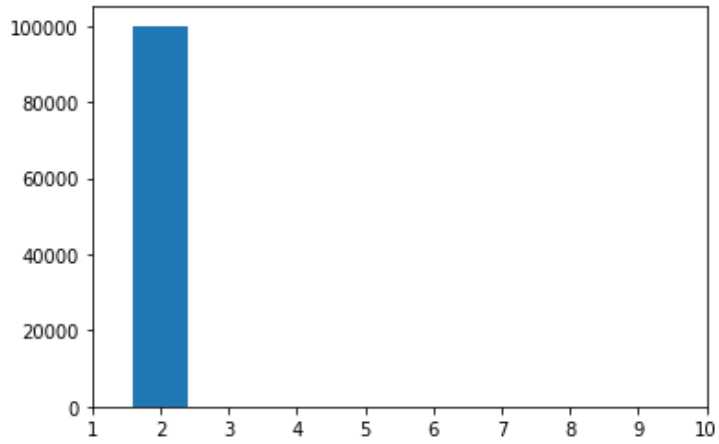
e. Form $u = \begin{bmatrix} u_1 \\ {b_n}^2 \end{bmatrix}$ and compute likelihoods $h_2(u)$, $h_1(u_1)$.

f. Compute acceptance-rejection function enumerated above; if a uniformly distributed random variable in range [0,1] is less than acceptance-rejection value, set (n, b_n) to (n^*, b_{n^*}); otherwise, keep the n generated at the beginning of the iteration and begin the next step.

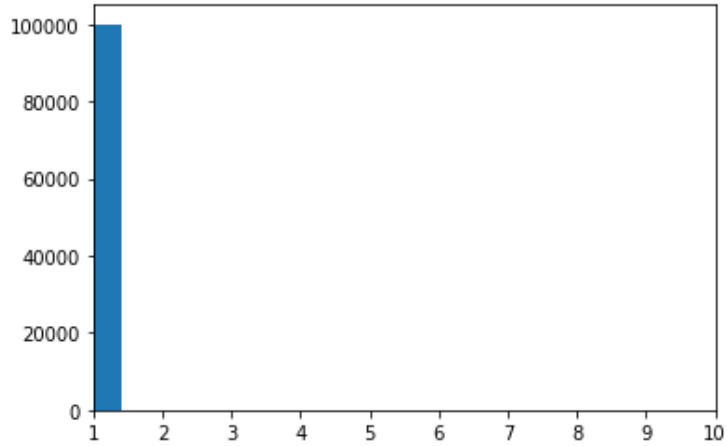
3. Plot a histogram of the observed n's

Results

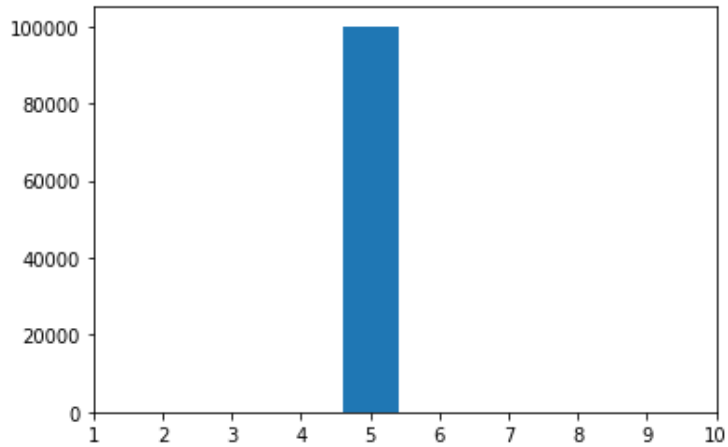
Of the ten runs of 100,000 iterations, 9 of the 10 converged successfully to the correct value of n while one did not. Convergence occurred quite quickly, indicating a high performance of the algorithm towards convergence to the true value of n. Resulting histograms are pictured below:
n = 2:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

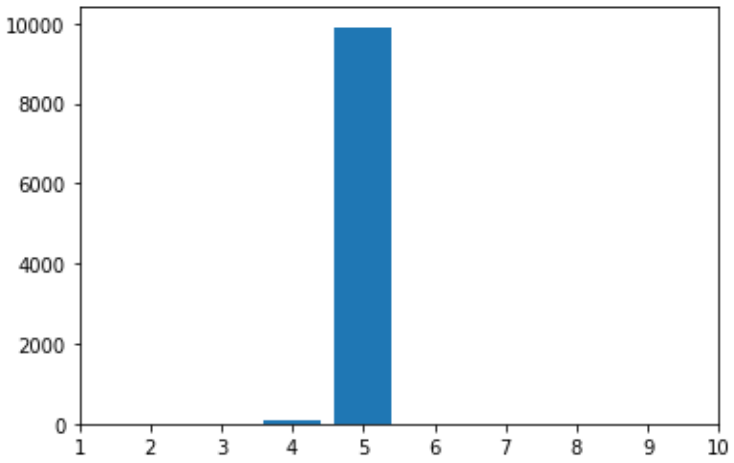
n = 1:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

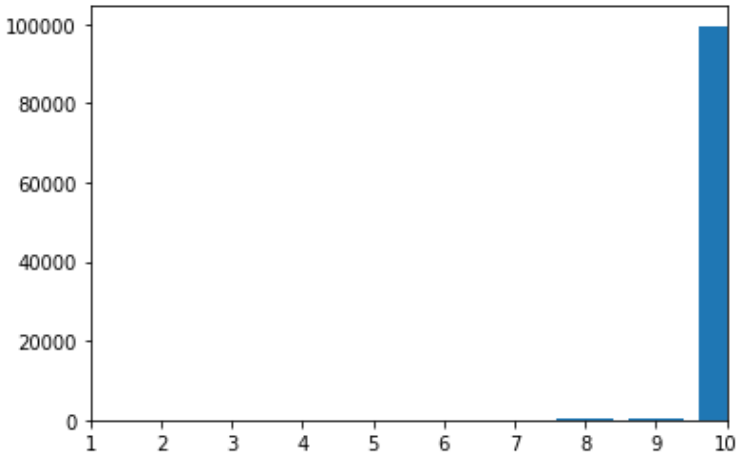
n = 5:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

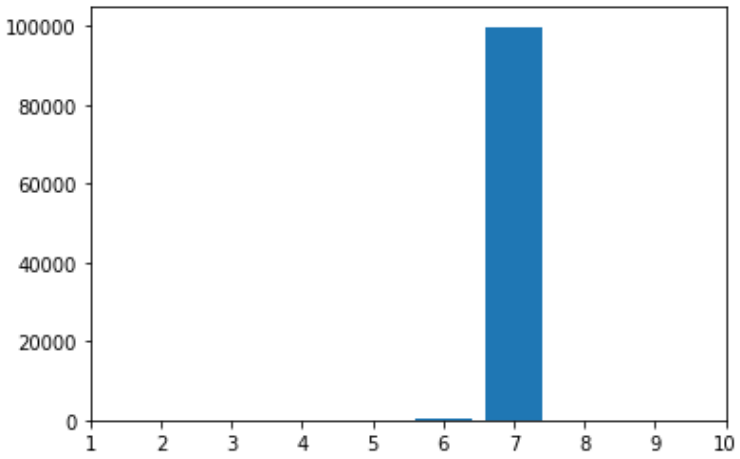
n = 4:


10000
8000
6000
4000
2000
0
1
2
3
4
5
6
7
8
9
10

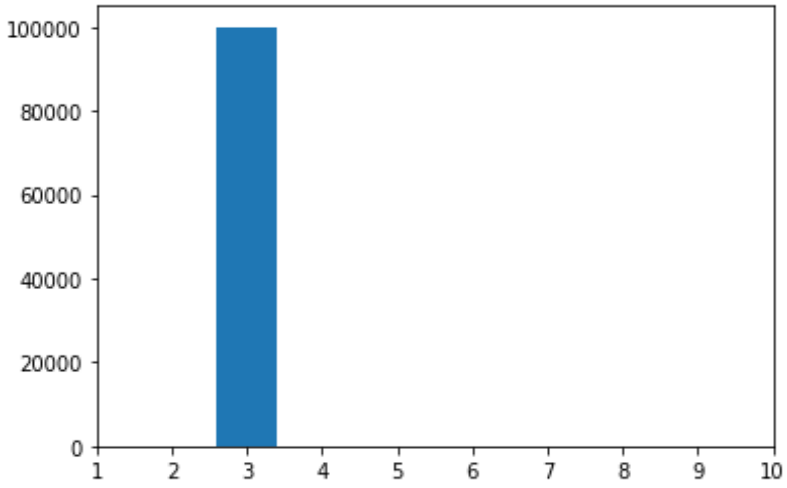
n = 10:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

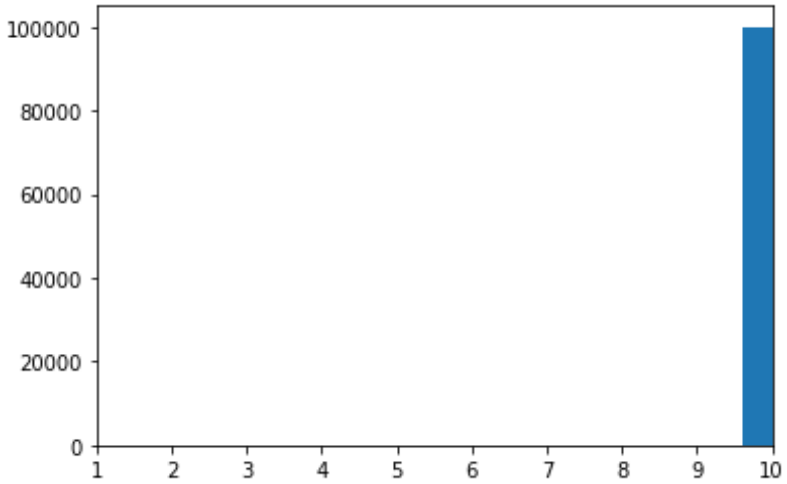
n = 7


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

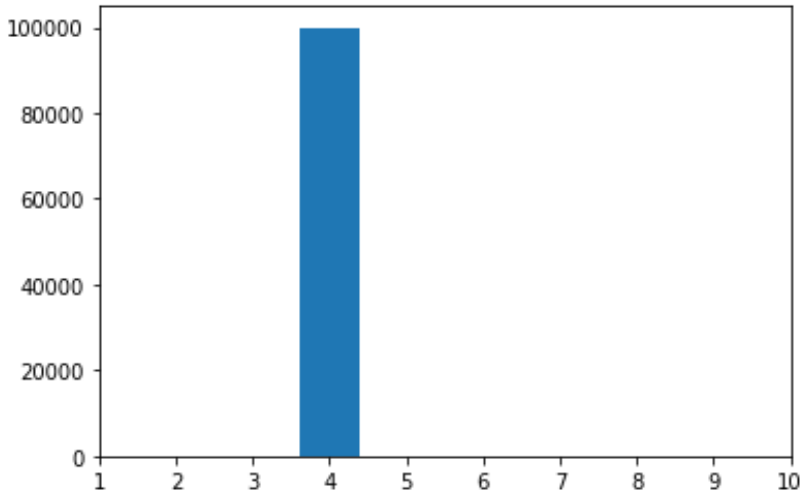
n = 3


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

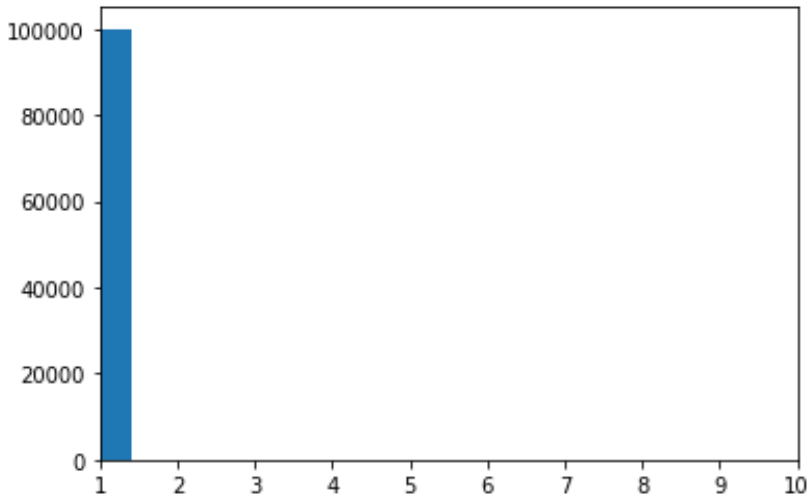
n = 10:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

n = 4:


100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10

n = 1:



Out[42]: 1

Conclusion

The Metropolis-Hastings algorithm is a versatile and powerful MCMC method with wide-ranging applications in Bayesian estimation for complex distributions which are not easily calculable. Often utilized in optimization, parameter estimation, and model selection, these applications are particularly powerful in simplifying complex multi-dimensional parameter and model estimation problems where the number of unknown parameters is itself unknown. Of course, this type of problem presents significant difficulties particularly toward computational expense and efficiency.

The RJMCMC algorithm is a specific formulation of Metropolis-Hastings which utilizes the unique qualities of invariant Markov chains to allow “jumps” in dimension, creating a union of multi-dimensional state spaces which can be sampled as a complex distribution. Fulfillment of the detailed balance condition is a critical aspect of the ability to work with such a state space, as it enables reversibility of the chain. Balancing computational efficiency with accurate determination, the simplicity and generality of RJMCMC enables its applicability to a wide range of complex model and parameter estimation problems. To quantify the uncertainty surrounding another uncertain quantity is a challenging problem in statistical estimation, but RJMCMC provides a starting point from which one may attempt such an analysis.

References

Fan, Yanan and Scott Sisson. “Handbook of Approximate Bayesian Computation.” *Handbooks of Modern Statistical Methods*, Chapman & Hall/CRC, 2019, https://arxiv.org/pdf/1001.2055.pdf. Accessed 30 Apr. 2023.

Green, Peter, and David Hastie. “Model Choice using Reversible Jump Markov Chain Monte Carlo.” Harvard School of Engineering and Applied Sciences, 11 May 2011, https://groups.seas.harvard.edu/courses/cs281/papers/hastie-green-2011.pdf. Accessed 28 Apr. 2023.

Appendix

Python Code:

```
import numpy as np
from matplotlib import pyplot
import math as m
import statistics as stat

#Final Project

#Data Simulation
N = 100000          #Number of repetitions
M = 10              #Predictors
k = 10              #Observations

n = np.zeros((N))
n[0] = np.ceil(np.random.uniform(0,1)*M)
n_0 = int(n[0])

sigma_0 = 0.2
sigma_p = 0.3
sigma_r = 0.2

mu_b = (2*np.ones((int(n[0]),1)))         #Prior betas
b_0 = (mu_b+sigma_p*np.random.randn(n_0, 1))

b = np.zeros((10,N))
#mu_b = 2*np.ones((int(n[0])))          #Prior betas
b_new = (mu_b+sigma_p*np.random.randn(int(n[0]),1))
b[:,0] = np.pad(b_new, ((0,(10-b_new.shape[0])), (0,0))).reshape(10)

X = np.zeros((k,M))
for i in range(0,k):
    X[i,0:M] = 5*np.random.normal(0, 1, k)

y = np.dot(X[:,0:n_0],b_0[0:n_0,0]) + sigma_0*np.random.normal(0, 1, k)

def accept_reject(y, X, n_current, n_star, bn, b_star, sigma_0, sigma_p, h1, h2):
    Xn = X[:,:n_current]
    Xstar = X[:,:n_star]
    E1 = (1/(2*sigma_0**2))*np.linalg.norm(y-Xstar@b_star)**2
    E2 = (1/(2*sigma_0**2))*np.linalg.norm(y-Xn@bn)**2

    neum = (np.exp(-(E1-E2)) * (2 * np.pi * sigma_p**2)**((n_current-n_star)/2) * \
```

```
        np.exp((-1/(2*sigma_p**2)) * \
              (np.linalg.norm(b_star - (2*np.ones((n_star,1))))**2 \
             - np.linalg.norm(bn - 2*np.ones((n_current,1)))**2))) * h2
    frac = neum / h1
    return np.min([1.,frac])

def h1u(u, n_star):
    h_one = ((1/np.sqrt(2*m.pi*sigma_r**2))**n_star)*np.exp((-
          1/(2*sigma_r**2))*(np.linalg.norm(u)**2))
    return(h_one)

def h2u(u, n):
    htwo = ((1/np.sqrt(2*m.pi*sigma_r**2))**n)*np.exp((-
          1/(2*m.pi*sigma_r**2))*(np.linalg.norm(u))**2)
    return(htwo)

#Algorithm
for i in range(1,N):
    #n = np.zeros((N+1))
    n[i] = np.ceil(np.random.uniform(0,1)*M)
    #n0 = int(n[i-1])

    n_star = np.ceil(np.random.uniform(0,1)*M)
    n_star = int(n_star)
    b_star = np.zeros((10))

    bn = b[:,i-1]
    n_current = int(n[i-1])



    if n_star >= n_current:
        u = np.random.normal(0,sigma_r, n_star)
        u1 = u[0:n_current]
        b_star[0:n_current] = bn[:n_current]
        b_star = b_star[:n_star] + u
        h1 = h1u(u, n_star)
        h2 = h2u(u1, n_current)

    else:
        u1 = np.random.normal(0, sigma_r, n_star).reshape(n_star,1)
        b1n = bn[:n_star].reshape(n_star,1)
        b2n = bn[n_star:n_current].reshape(n_current-n_star,1)
        u = np.vstack((b1n,b2n))
```

```
        b_star = b1n + u1

        h2 = h2u(u, n_current)
        h1 = h1u(u1, n_star)

    #AR ratio
    rho = accept_reject(y=y, X=X, n_current=n_current, n_star=n_star,
                        bn=bn[:n_current],
                        b_star=b_star[:n_star], sigma_0=sigma_0,
                        sigma_p=sigma_p, h1=h1, h2=h2)

    #AR
    U = np.random.uniform(0,1)

    if U < rho:
        n[i] = n_star
        b[0:len(b_star),i] = b_star
    else:
        n[i] = n_current
        b[0:len(bn),i] = bn

pyplot.bar(*np.unique(n, return_counts=True))
pyplot.xlim((1,10))
```