# $MCMC$ autostopping algorithm returning independent samples

Report for Kinney Lab, CSHL

Kush Coshic

July 14, 2017

# Contents

# 1 Introduction

Our goal in a Bayesian computation is to obtain a set of *independent* draws $\theta^s$; $s = 1, ...., S$ from the *posterior* distribution, with enough draws $S$ such that quantities of interest can be estimated with reasonable accuracy.

MCMC (Markov-chain Monte Carlo) is a simulation strategy that is based on a Markov-chain construct; also referred to as Markov chain simulation. It's a general method based on drawing values of the Bayesian parameter $\theta$ from approximate distributions and then correcting those draws to better approximate the target *posterior* distribution, $p(\theta|y)$

In probability theory, a Markov process is a stochastic process that obeys the *Markov* property (i.e. one can predict the next state, solely with the conditional information on the present state). A Markov-chain is a Markov process with a discrete state space.

Due to the *Markov* property, samples returned from an MCMC simulation contain an inherent *correlation*. This is a general problem faced with all Markov chain based simulation methods, like MCMC.

2 ways to address this could be:

- Run the chain for a long time so that we have enough samples to compensate for the inherent correlations
- Draw a set of *effectively independent* samples from the posterior

Clearly the second method is more useful, one being able to approximate the posterior distribution with a relatively smaller number of samples (the computational power used to extract the independent samples would usually be much lesser than the cumulative computational requirement for doing subsequent computations with the (large) correlated samples from the posterior distribution).

This article discusses a general prescription for computing *effectively independent* samples from the *posterior* distribution, as discussed in the paper by Gelman & Rubin [2]. Subsequently, the authors wrote an excellent book *Bayesian Data Analysis* [1] in which Part III of the book discusses this discussion.

This aim of the algorithm is returning a minimum number of effectively independent samples as required by the user. A Metropolis-MCMC simulation is used to move around the posterior distribution, starting with multiple chains (5 in this example) and continuing the simulation till convergence around the posterior is approximately achieved.

# 2 The Code

Consider the posterior to be a Bivariate Normal distribution with correlated coordinates, as given by:

$$f(x) = (2\pi)^{-\frac{k}{2}} |\mathbf{\Sigma}|^{-\frac{1}{2}} e^{-\frac{1}{2}\ (x-\mu)'\ \mathbf{\Sigma}^{-1}\ (x-\mu)} \tag{1}$$

where,

- $k(= 2)$ denotes the dimension of the distribution
- $x$ denotes a general k dimensional vector
- $|\Sigma|$ denotes determinant of the Covariance matrix
- $\mu$ denotes the k dimensional, maxima of the distribution

```
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

Importing relevant packages.

```
# x_start array contains starting points to be used for the Markov chains
x_start = np.array([[0.,0.],[-0.9,0.9],[-0.9,-0.9],[0.9,-0.9],[0.9,0.9]])

dimension = len(x_start[0])

# Multivariate Normal distribution to be sampled, f(x)

```

```
x0=np.matrix([2.5,2.5])            # defining the center of f(x)

# cov is the covariance matrix used to define f(x)
cov=np.matrix([[1.,3./5],[3./5,2.]])
det=np.linalg.det(cov)             # determinant
inv=np.linalg.inv(cov)             # inverse of cov
k=dimension                        # dimension

# Defining f(x)
def f(x):
    x=np.ravel(x)
    x=np.matrix(x)
    return (((2*np.pi)**(-k/2.))*((det)**(-1/2))*np.exp(-(1./2)*(x-x0)*inv*(x-x0).T))
```

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{k/2}\,\mathbf{det}^{1/2}} e^{\frac{-1}{2}(\mathbf{x}-\mathbf{x0}).\mathbf{inv}.(\mathbf{x}-\mathbf{x0})^T} \tag{2}$$

I used a bivariate normal distribution with a covariance term (i.e. coordinates are not independent) to check the efficiency of the code. Adding a covariance term would make it more difficult for the code to construct independent samples, as apart from the correlations due to the Markov chain construct there is now an inherent correlation between the coordinates; making it more difficult to extract independent samples.

```
n_eff_list = []
mcmc_step_list = []
n_eff_list_y = []

# Defining a function stepper(x) that returns a random vector around x
# The stepper parametrizes the jumping distribution for the Metropolis MCMC

covv=1.5*cov       # covariance matrix for the stepper function
mean=[0.,0.]       # mean vector for the stepper function

def stepper(x):
    x=np.ravel(x)
    dx = np.random.multivariate_normal([0.,0.], covv, 1)
    return x+dx


# Parameters to be entered by the user:

# minimum number of independent samples required
n_eff_min = 1000

mcmc_step = 1000    # MCMC iterations step-size
R_max = 1.1         # maximum allowed value for R, (Potential scale reduction factor)
choice = dimension*['']
```

Note that, *mcmc_step* needs to be a multiple of 4 (because of warm-up and subsequent slicing into 2 arrays)

The array *choice* is used to ensure the simulation keeps running till autocorrelation reaches zero, which might not happen if a small $n\_eff\_min$ value is used.

The program starts by running $n\_eff\_min$ number of MCMC iterations for each of the chains. However if the autocorrelation doesn't reach zero, another *mcmc_step* number of iterations are made for each chain. Till the autocorrelation doesn't reach zero this process would keep continuing.

## Description of the algorithm

- Essentially the function *run_mcmc* computes everything; the MCMC sampling, subsequent autocorrelation computations, confirming approximate convergence, and computing the final array of independent samples.
- The while loop keeps running till:
  - number of independent samples obtained are atleast equal to $n\_eff\_min$
  - $R < R\_max$
  - The $3^{rd}$ condition helps avoid the autocorrelation error (when enough MCMC iterations are not done so that the autocorrelation never reaches zero)

- In line 74 of the code (shown below), the for loop generates mcmc samples for each dimension; and records the data in *x_recorded*
- warming up discards $1^{st}$ half of the data in *x_recorded* and stores the remaining in *x_warmup*
- Every chain is split into 2 and made seperate chains (in our example the total chains change from 5 to 10). These are the final mcmc samples to be used for extracting independent samples, and are recorded into *x_result*
- All variables like *psi_dot_j_bar*, *psi_dot_dot_bar* etc. follow exactly as described in the book. For e.g. *psi_dot_j_bar* is an array that stores (for each dimension) the $\bar{\psi}_{.j}$.
- I have initially assigned each of these as a null array [ ], and append() values for each dimension/coordinate
- All functions, such as variance, *var_dagger*, Variogram at each lag $t$, autocorrelation etc. have been defined and implemented accordingly, as mentioned in the book [1], pg 281-287.
- Inside the while loop is a for loop (in the variable d) that computes for each of the coordinates (=dimension) the functions defined below,
- For each scalar estimand $\psi$, we label the simulations as $\psi_{ij}(i = 1, ..., n; j = 1, ..., m)$ and compute **B** and **W**, the Between and Within sequence variances. ($n$ denotes length of each chain, $m$ denotes number of chains)

$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\psi}_{.j} - \bar{\psi}_{..})^2 \ , \quad where \ \ \bar{\psi}_{.j} = \frac{1}{n}\sum_{i=1}^{n}\psi_{ij} \ , \quad \bar{\psi}_{..} = \frac{1}{m}\sum_{j=1}^{m}\bar{\psi}_{.j} \tag{3}$$

$$W = \frac{1}{m}\sum_{j=1}^{m}s_j^2, \quad where \ s_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\psi_{ij} - \bar{\psi}_{.j})^2 \tag{4}$$

The marginal posterior variance is defined as,

$$\boxed{\hat{var}^{+}(\psi|y) = \frac{n-1}{n}W + \frac{1}{n}B} \tag{5}$$

The motivation for defining it this way is given in [1] pg. 284.

Convergence of the simulation is monitored by estimating a *scale reduction factor*, defined by

$$\hat{R} = \sqrt{\frac{\hat{var}^{+}(\psi|y)}{W}} \tag{6}$$

which declines to 1 as $n \to \infty$. We chose $R = 1.1$ as the maximum bar for assessing convergence.

For defining an effective sample size for correlated simulation draws is to consider the statistical efficiency of the average of the simulations $\bar{\psi}_{..}$, as an estimate of the posterior mean $E(\psi|y)$. This seems a reasonable choice, although it might be inappropriate if one is interested in the accurate representation of events in the tail of the distribution.

- An estimate of the effective sample size is obtained using the asymptotic formula for the variance of the average of a correlated sequence:

$$lim_{n\to\infty} mn \ var(\bar{\psi}_{..}) = \big(1 + 2\sum_{t=1}^{\infty}\rho_t\big)var(\psi|y) \tag{7}$$

where $\rho_t$ is the autocorrelation of the sequence $\psi$ at lag $t$.

Lets derive Eq. (7),

For simplicity ignore the $2^{nd}$ index in $\psi$ on the left side of Eq. (7). So, consider we need to prove:

$$lim_{n\to\infty} n\ var(\bar{\psi}) = \left(1 + 2\sum_{t=1}^{\infty}\rho_t\right) var(\psi|y) \tag{8}$$

with $\bar{\psi} = \frac{1}{n}\sum_{i=1}^{n}\psi_i$. So,

$$\begin{aligned}
var(\bar{\psi}) &= <\bar{\psi}>^2 - <\bar{\psi}>^2 \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} <\psi_i\psi_j> - \frac{1}{n^2}\sum_{i=1}^{n} <\psi_i>^2 \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} [<\psi_i\psi_j> - <\psi>^2] \\
&= \frac{2}{n^2}\sum_{i=1}^{n}\sum_{t=1}^{n-i} [<\psi_i\psi_{i+t}> - <\psi>^2] + \frac{1}{n^2}\sum_{i=1}^{n^2} [<\psi_i>^2 - <\psi>^2] \\
&= \frac{2}{n^2}\sum_{i=1}^{n}\sum_{t=1}^{n-i} \rho_t\ var(\psi) + \frac{1}{n^2}\sum_{i=1}^{n} var(\psi) \\
&= \frac{2}{n^2}\sum_{i=1}^{n}\sum_{t=1}^{n-i} \rho_t\ var(\psi) + \frac{1}{n} var(\psi)
\end{aligned} \tag{9}$$

Assuming $\rho_t \to 0$ as $t \to \infty < n$ , we can approximate the $2^{nd}$ summation in Eq. (9) to run till infinity. So,

$$var(\bar{\psi}) = \frac{1}{n}\left(1 + 2\sum_{t=1}^{\infty}\rho_t\right) var(\psi|y) \tag{10}$$

which proves Eq. (8). Now we can add the $2^{nd}$ summation index $j$ (and multiply by $m$) to get Eq. (7). Further, we define an $n_{eff}$ such that,

$$\begin{aligned}
\frac{var(\psi)}{n_{eff}} \equiv var(\bar{\psi}) &\approx \frac{1}{n} var(\psi)\left[1 + 2\sum_{t=1}^{\infty}\rho_t\right] \\
&\Rightarrow \boxed{n_{eff} = \frac{n}{1 + 2\sum_{t=1}^{\infty}\rho_t}}
\end{aligned} \tag{11}$$

- If the n simulation draws from each of the m chains were independent, then $var(\bar{\psi}_{..})$ would simply be $\frac{1}{mn} var(\psi|y)$ and the sample size would be mn. However in the presence of correlation we can define the **effective sample size** as:

$$n_{eff} = \frac{mn}{1 + 2\sum_{t=1}^{\infty}\rho_t} \tag{12}$$

- Unfortunately, simply summing all of the autocorrelations to estimate $n_{eff}$ will have the sample correlation too noisy for large values of t. Therefore we compute a partial sum, starting from lag0 and continuing until the sum of autocorrelation estimates for 2 successive lags $\hat{\rho}_{2t'} + \hat{\rho}_{2t'+1}$ is negative; where

$$\hat{n}_{eff} = \frac{mn}{1 + 2\sum_{t=1}^{T}\hat{\rho}_t} \tag{13}$$

where T is the first odd positive integer for which $\hat{\rho}_{T+1} + \hat{\rho}_{T+2}$ is negative.

Now, to compute the effective samples size (or the number of effectively independent samples) we need an estimate of the sum of the correlations $\rho$. For this first we compute the *variogram* $V_t$ at each lag $t$:

$$V_t = \frac{1}{m(n-t)} \sum_{j=1}^{m} \sum_{i=t+1}^{n} (\psi_{i,j} - \psi_{i-t,j})^2 \tag{14}$$

Now we want to express correlations in terms of $V_t$. The correlation $\rho_t$ between two samples $\psi_i$ and $\psi_{i-t}$ is defined by,

$$\rho_t \equiv corr(\psi_i, \psi_{i-t}) = \frac{cov(\psi_i, \psi_{i-t})}{\sqrt{var(\psi_i)}\sqrt{var(\psi_{i-t})}} = \frac{cov(\psi_i, \psi_{i-t})}{var(\psi)} \tag{15}$$

and,

$$cov(\psi_i, \psi_{i-t}) \ = < (\psi - \bar{\psi_i})(\psi - \bar{\psi}_{i-t}) > = < \psi_i \psi_{i-t} > - \bar{\psi}^2 \tag{16}$$

Further,

$$\begin{aligned} < (\psi_i - \psi_{i-t})^2 > &= < \psi_i^2 + \psi_{i-t}^2 - 2\psi_i \psi_{i-t} > \\ &= 2 < \psi^2 > - 2 < \psi_i \psi_{i-t} > \\ &= 2\big( < \psi^2 > - \bar{\psi}^2 - \rho_t \ var(\psi)\big) \end{aligned} \tag{17}$$

Using Eqs. (15) and (16), Eq. (17) becomes,

$$\begin{aligned} < (\psi_i - \psi_{i-t})^2 > &= 2(\ var(\psi) - \rho_t \ var(\psi)) \\ &= 2(1 - \rho_t) \ var(\psi) \end{aligned} \tag{18}$$

Using this we can rewrite Eq. (14) as,

$$\hat{\rho}_t = 1 - \frac{V_t}{2 \ \hat{var}^{\dagger}} \tag{19}$$

where we have manifested the ensemble average in Eq. (18) as a sum over all chains (index $j$), in Eq. (14). The point to note is that, for doing this the variance term in Eq. (18) is to be replaced with $\hat{var}^{\dagger}$, and we get Eq. (19).

- Now to get the set of effectively independent samples, we draw samples from *x_result*, but skipping every $\frac{mn}{n_{eff}}$ samples in between. Finally we record the final set of independent samples in **samples**

Beginning with the main portion of the code, defining the *run_mcmc* function:

```
def run_mcmc(f,x_start,stepper,n_eff_min,R_max,mcmc_step):
    n_effective = np.zeros(dimension)
    n_eff_min_list = np.zeros(dimension)
    acceptance_list=[]
    for i in range(dimension):
        n_eff_min_list[i] = n_eff_min
    R_max_list = np.zeros(dimension)
    for i in range(dimension):
        R_max_list[i] = R_max

    z=0
    R=0
    mcmc_step_remember = mcmc_step
    x_recorded = np.zeros((mcmc_step,dimension,len(x_start)))
```

The function *run_mcmc* takes as arguments; $f$ (distribution to be sampled), *x_start* (starting points for the MCMC chains), *stepper* (the stepper function which parametrizes jumping conditions for the MCMC), $n\_eff\_min$ (minimum number of independent samples required), $R\_max$ (maximum allowed Potential scale reduction factor), *mcmc_step* (MCMC iterations step-size).

Initializing variables in lines 51-61. In line 62, I've stored *mcmc_step* in a new variable *mcmc_step_remember* since *mcmc_step* would get updated with every subsequent while loop execution. Line 63 defines a 3-dimensional array that stores all the MCMC iterations. Its $1^{st}$ axis corresponds to every subsequent iteration, $2^{nd}$ axis corresponds to the 2 coordinates in the functional space of the distribution f(), and each element of the $3^{rd}$ axis corresponds to a Markov chain (5 in this example).

```
64      while (np.any(np.array(n_effective)<n_eff_min_list) or np.any(R>R_max_list) and all([i==
        '' for i in choice])):
65          # MCMC results stored in a 3-d array, 3rd dimension corresponding to every new chain
66          #choice = dimension*['']
67          if z==0:                            # this if else is there to make sure the samples
        from the previous run are not wasted
68              mcmc_step_0=0
69              x_recorded = np.zeros((mcmc_step,dimension,len(x_start)))
70          else:
71              x_recorded_template = np.zeros((mcmc_step,dimension,len(x_start)))          #
        this if-else is used to store the already computed MCMC samples, before moving to
        further iterations (if n_eff condition not satisfied). This prevents unecessary wastage
        of computational power.
72              x_recorded_template[:mcmc_step_0,:,:]=x_recorded[:,:,:]
73              x_recorded = x_recorded_template
74          for i in range(len(x_start)):
75              x_current = x_start[i]
76              acceptances = np.zeros(mcmc_step)
77              for k in range(mcmc_step_0,mcmc_step):
78                  x_new = stepper(x_current)
79                  if np.random.uniform(0,1) < f(x_new)/f(x_current):
80                      x_current = x_new
81                      acceptances[k] = 1
82                  x_recorded[k,:,i] = x_current
83              acceptance_list.append(1.0*acceptances.sum()/len(acceptances))
84              #print 'acceptance fraction = %f'%(1.0*acceptances.sum()/len(acceptances))
```

Note that the indentation is since the while loop is inside the def() function for *run_mcmc*.

The while loop stops executing till the number of effective independent samples are greater or equal to $n\_eff\_min$ and $R > R\_max$. The third condition inside the while loop is to keep the loop running till autocorrelations reach a zero, which otherwise might not happen if we input a smaller value for *mcmc_step*. Note that in the code the conditions are in the form of lists, to incorporate all the dimensions.

The variable $z$ keeps track of the number of times the while loop has been executed and started back (it starts with $z = 0$).

When $z = 0$ the shape of *x_recorded* is defined so as to accomodate only the first set of *mcmc_step* (=1000 in this example) MCMC iterations. Else every time a new array *x_recorded_template* is defined which can accomodate the set of MCMC iterations for the updated *mcmcstep* steps. The earlier iterations are stored in Lines 72,73.

The for loop in Line 74 is for performing the MCMC iterations for each of the chains. In Line 77, the for loop only runs for the remaining (=1000 in this example) empty elements in *x_recorded*. At the end of the overall while loop, each time *mcmc_step* gets stored in *mcmc_step_0* and *mcmc_step* gets incremented by *mcmc_step_remember* (=1000).

Line 78 computes an MCMC iteration around the current vector (*x_current*). Line 79 is the Metroplis argument, and finally all the iterations are stored in *x_recorded*.

```
85      # Warm up period (extracting only second half of the iterations)
86      x_warmup = np.zeros((mcmc_step/2,dimension,len(x_start)))
87      x_warmup[:,:,:] = x_recorded[mcmc_step/2:,:,:]
88
89      # Splitting the chains
90      x_result = np.zeros((mcmc_step/4,dimension,2*len(x_start)))
91      x_result[:,:,0:len(x_start)] = x_warmup[0:mcmc_step/4,:,:]
92      x_result[:,:,len(x_start):2*len(x_start)] = x_warmup[mcmc_step/4:mcmc_step/2,:,:]
93
94
95      n=mcmc_step/4.0          # number of iterations in each chain
96      m=2.0*(len(x_start))     # number of chains
97
98
99      # Assessing mixing using between and within sequence variances
```

```
100
101         psi = []
102         for i in range(dimension):
103             psi.append(x_result[:,i,:])
104         psi = np.array(psi)
105         # this was done to divide the data for each dimension, for convenience
```

Next, lines 86,87 perform the warmup step, i.e. remove the $1^{st}$ half of all the iterations in *x_recorded* and update *x_recorded* with the remaining iterations.

Then lines 90-92 perform the splitting of each chain into 2, and subsequently store them in a new variable *x_result*.

In lines 95,96 I've defined new variables $n$,$m$ respectively as the number of iterations in each chain and the number of chains. So, *x_result* is an $(n, 2, m)$ dimensional array.

Lines 101-104 just divides *x_result* into the x and y components separately as the 2 components of an array *psi*.

```
106         psi_dot_j_bar = []
107         psi_dot_dot_bar = []
108         B_term = []
109         B = []                                  # Between sequence variance
110         s_j_square_term = []
111         s_j_square = []
112         W = []                                  # Within sequence variance
113         var_dagger = []
114         R = []
115         psi_i_comma_j = []
116         psi_i_minus_t_comma_j = []
117
118         rho_array_set = []
119         t_array_set = []
120         n_effective_list = []
121         t_array = np.arange(int(n))
122
123
124         # for every subsequent dimension the values are appended into the arrays
125         for d in range(dimension):
126             psi_dot_j_bar.append((1/n)*np.sum(psi[d], axis=0))
127
128             psi_dot_dot_bar.append((1/m)*np.sum(psi_dot_j_bar[d]))
129
130             B_term.append(psi_dot_j_bar[d] - psi_dot_dot_bar[d])
131
132             B.append((n/(m-1.0))*np.sum(B_term[d]**2))
133
134             s_j_square_term.append(np.zeros((int(n),int(m))))
135
136             for j in range(int(m)):          # performing for all chains
137                 s_j_square_term[d][:,j] = (psi[d][:,j] - psi_dot_j_bar[d][j])**2.0
138
139             s_j_square.append((1.0/(n-1))*np.sum(s_j_square_term[d], axis=0))
140
141             W.append((1.0/(m))*np.sum(s_j_square[d]))
142
143
144             # variance
145             var_dagger.append(((n-1)/(n))*W[d] + (1/n)*B[d])
146
147             #var_dagger
148
149             # potential scale reduction
150             R.append(np.sqrt(var_dagger[d]/W[d]))
151
152             #R
153             # Variogram
154             def V(t):
155                 psi_i_comma_j = psi[d][t+1-1:,:]
156
157                 psi_i_minus_t_comma_j = psi[d][0:int(n)-t,:]
158
159                 t=float(t)
```

```
160                 return (1.0/(m*(n-t)))*np.sum(np.sum((psi_i_comma_j - psi_i_minus_t_comma_j)
    **2.0,axis=0))
161
162             # autocorrelation
163             def rho(t):
164                 return 1.0-(V(t)/(2.0*var_dagger[d]))
165
166
167             rho_array=np.zeros(int(n))
168             for t in range(int(n)):
169                 rho_array[t]= rho(int(t))
170             rho_array_set.append(rho_array)
171             # checking where the autocorrelation goes to zero for the first time, append
    that iteration number in choice array for every subsequent dimension
172             # The while loop in the beginning, makes the code run until all k (=dimension)
    elements in choice[] are not null, which implies autocorrelation reached zero for each
    of them.
173             for i in range(int(n)):
174                 check = rho(i) + rho(i+1)
175                 if check < 0:
176                     #choice.append(i*1.0)
177                     choice[d]=i*1.0
178                     break
179
180             # if choice[d] is not null, autocorrelation did reach zero and we could use the
    corresponding iteration number to calculate the value for n_effective obtained for the
    simulation
181             if choice[d] != '':
182                 # effective sample size
183                 n_effective[d]=(m*n)/(1+2.0*np.sum(rho_array[1:choice[d]]))
184             else:
185                 break
186
187         mcmc_step_0 = mcmc_step                     # record current number of mcmc runs in
    mcmc_step_0, for the next run
188         mcmc_step += mcmc_step_remember             # increase number of mcmc runs to be
    performed next in the next while loop run
189         z += 1.0                                    # to keep track of the number of while
    loop runs (=z after run_mcmc ends)
```

Lines 106-121 initializes variables/arrays for subsequent use. Each of them have meaning as defined in the description above. For e.g. *psi_dot_j_bar* is an array that stores (for each dimension) the $\bar{\psi}_{.j}$ values.

In line 125, the for loop runs for all the dimensions (=2 in this example), and lines 126-170 computes all the variables as defined above in the description above. In line 173,174 the autocorrelations values are checked to find the $i^{th}$ one such that $\rho(i) + \rho(i+1) < 0$ for the first time. When this happens, the for loop breaks and $i$ stores the corresponding value which is then used as the upper limit for the summation in Eq.9., that computes the effective number of independent samples.

In line 177, $choice[d] = i * 1.0$ (where d denotes the dimension which is been considered in the loop, line 125) if such an $i$ is found, i.e. autocorrelation reached zero. Else the third condition in the main while loop will execute again with an incremented *mcmc_step* value.

→ Note that there is still some bug here. Try with a small value for *mcmc_step* (=40 say), the autocorrelation error still pops up, indicating that the loop is not executed again when the autocorrelations do not reach zero. One might want to put an *assert* statement for it.

Finally line 187 stores the current value of *mcmc_step* in *mcmc_step_* and increments *mcmc_step* in line 188. The $z$ value is incremented in line 189 (to keep track of the number of times the main while loop is executed).

```
190     # Taking the minimum value for n_effective
191     n_effective_final = np.amin(n_effective)
192
193     # no. of samples to be skipped before recording the next independent sample
194     samples_skip = 1.0*x_result.shape[2]*x_result.shape[0]/n_effective_final
195     samples_skip=np.ceil(samples_skip)
196
197     # extract and store independent samples in samples_final
198     if int(x_result.shape[0]/samples_skip)==x_result.shape[0]/samples_skip:
```

```
199         samples_final = np.zeros((int((x_result.shape[0])/samples_skip),x_result.shape[1],
        x_result.shape[2]))
200     else:
201         samples_final = np.zeros((int(np.ceil((x_result.shape[0])/samples_skip)),x_result.
        shape[1],x_result.shape[2]))
202     # Broadcasting independent samples into samples_final
203     samples_final[:,:,:] = x_result[0::samples_skip,:,:]
204     # note that at this point the simulation data from each chain is recorded separately, it
         will be condensed in samples,
205
206     # putting all points of different chains into 1 single array which can be referred to as
         the final array of our independent samples
207     samples = np.zeros((samples_final.shape[0]*samples_final.shape[2],dimension))
208     for i in range(samples_final.shape[2]):
209         samples[i*samples_final.shape[0]:(i+1)*samples_final.shape[0],:]=samples_final[:,:,i
        ]
210
211     return n_effective,R,z,x_recorded.shape,x_result.shape,x_result,samples_final,samples,
        acceptance_list,x_recorded,choice
212
213
214 run = run_mcmc(f,x_start,stepper,n_eff_min,R_max,mcmc_step)
```

The number of effective independent samples in all the dimensions (=2 in our example) are stored in $n\_effective$, and the minimum of those is stored in $n\_effective\_final$. Lines194,195 compute the number of samples to be skipped successively, for extracting the independent samples. This is stored in *samples_skip*.

Ideally one just needs to now perform the skipping and get the independent samples, however there might occur a broadcast array error at line 203, because of the division by *samples_skip*. To deal with this I used an if-else statement, lines 198-203.

Finally in lines 207-209, I have taken the independent samples from all the different chains and put them together in *samples*, which now contains the final set of independent samples computed.

I've returned the computations in line 211, for the output of the defined function *run_mcmc*.

In line 214, I've stored the execution statement for *run_mcmc* inside a variable *run*.

```
215 samples_final=run[6]
216
217 samples_final_mean = np.zeros(dimension)
218 samples_final_mean_variance = np.zeros(dimension)
219 for i in range(len(samples_final_mean)):
220     samples_final_mean[i] = np.mean(samples_final[:,i,:])                    # sample mean
221     samples_final_mean_variance[i] = np.var(samples_final[:,i,:])            # sample variance
222
223 #print samples_final_mean,samples_final_mean_variance
224 variance = (samples_final_mean_variance[0]**2.0 + samples_final_mean_variance[1]**2.0)
        **(1/2.0)
225
226 # calculate z values
227 mu = np.mean(run[7],axis=0)              # variance in samples
228 var = np.var(run[7],axis=0)              # variance in samples
229 N = run[7].shape[0]                      # no. of samples
230 z = (mu-0)/np.sqrt(var/N)                # z-value of the samples
231 print z
```

where I have computed the array for the sample mean, variance and $z$ value.

```
232 # Checking how close the covariance matrix obtained from the independent samples is to what'
        s defined in f(x)
233 np.allclose(np.cov(run[7].T),cov,.06,.06)
234
235 # covariance matrix calculated from the samples
236 np.cov(run[7].T)
```

The output for line 236 should be close to what is defined in line 16, definition for $f()$. As we increase the number of independent samples computed, it would get closer and closer; as expected. This acts as a test for how good the samples are.

In the next code snippet I have plotted the autocorrelation for the independent samples. The 2 curves represent the 2 dimensions and as expected It typically fluctuates around zero.

```
237 # # Autocorrelation plots in the final set of effectively independent samples
238 # As we should expect, the autocorrelation plot should stay around the zero value (whichi it
      does perfectly if mcmc_step is taken large)
239
240 # Define for convenience
241 n = samples_final.shape[0]*1.0                    # no. of iterations for every chain
242 m = samples_final.shape[2]*1.0                    # no. of final chains
243 sample_x = samples_final[:,0,:]                   # samples x-dimension data
244 sample_y = samples_final[:,1,:]                   # samples y-dimension data
245
246 def V(t):                                         # same as defined inside run_mcmc
247     sample_x_i_comma_j = sample_x[t+1-1:,:]
248     sample_x_i_minus_t_comma_j = sample_x[0:int(n)-t,:]
249     t=float(t)
250     return (1/(m*(n-t)))*np.sum(np.sum((sample_x_i_comma_j - sample_x_i_minus_t_comma_j)
      **2.0,axis=0))
251
252 def rho(t):                                       # same as defined inside run_mcmc
253     return 1-(V(t)/(2.0*samples_final_mean_variance[0]))
254
255 rho_array = np.zeros(int(n))
256 t_array = np.arange(int(n))
257
258 for t in range(int(n)):
259     rho_array[t]= rho(int(t))                     # array containing rho values for the x
      coordinate
260
261
262 # Doing the same for the y-coordinate
263 def V_y(t):                                       # the same as above, but for the y
      coordinate
264     sample_y_i_comma_j = sample_y[t+1-1:,:]
265     sample_y_i_minus_t_comma_j = sample_y[0:int(n)-t,:]
266     t=float(t)
267     return (1/(m*(n-t)))*np.sum(np.sum((sample_y_i_comma_j - sample_y_i_minus_t_comma_j)
      **2.0,axis=0))
268
269 def rho_y(t):
270     return 1-(V_y(t)/(2.0*samples_final_mean_variance[1]))
271
272 rho_array_y = np.zeros(int(n))
273 t_array = np.arange(int(n))
274
275 for t in range(int(n)):
276     rho_array_y[t]= rho_y(int(t))                 # array containing rho values for the y
      coordinate
277
278
279 plt.figure(2,figsize=(8,5))
280 plt.plot(t_array,rho_array,'-',color='b')                    # autocorrelation plot for x
      coordinate
281 plt.plot(t_array,rho_array_y,'-',color='r')                  # autocorrelation plot for y
      coordinate
282 plt.xlabel('x')
283 plt.ylabel('autocorrelation')
284 plt.show()
285
286 # The 2 curves seem correlated which it should since f(x) has the 2 coordinates correlated.
```

Here I have used the same definition for autocorrelations (7), except that since now we don't have the problem of between and within variances so I have just replaced *var_dagger* with the sample variance *samples_final_mean_variance* inside the definition for *rho*() (lines 253,270 for each of the 2 dimensions).

## Comparison tests

For testing the effectiveness of the independent samples, I compared with arbitrarily chosen correlated samples (for this I just picked up data of the same length, from *x_result* which contains all the MCMC iteration data without the extraction of independent samples). The following is the code,

```
# The _1 indicates correlated samples
samples_final_1 = np.zeros((samples_final.shape[0],run[5].shape[1],run[5].shape[2]))

samples_final_1[:,:,:] = run[5][0:samples_final.shape[0],:,:]

samples_final_mean1 = np.zeros(dimension)
samples_final_mean1_variance = np.zeros(dimension)
for i in range(len(samples_final_mean1)):
    samples_final_mean1[i] = np.mean(samples_final_1[:,i,:])
    samples_final_mean1_variance[i] = np.var(samples_final_1[:,i,:])
```

After that I computed autocorrelation plots for the correlated samples.

Below, I've plotted the autocorrelation vs each lag $t$ plot for both the indpendent samples and the chosen correlated samples,

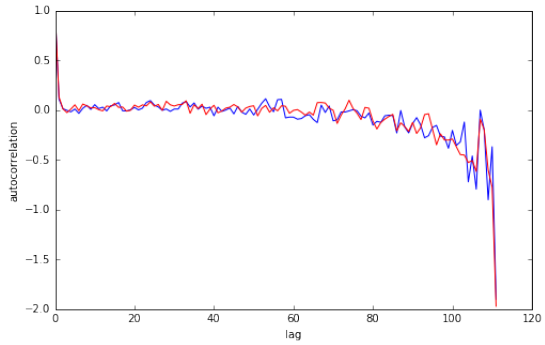


Figure 1: independent sample

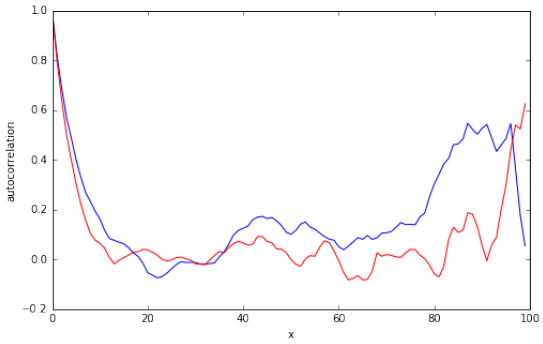


Figure 2: correlated samples

As you can see, the plot for independent samples sways around zero autocorrelation almost perfectly while for the correlated samples the deviations are significant.

A top view of the samples is shown below. The samples in both cases are the yellow dots.

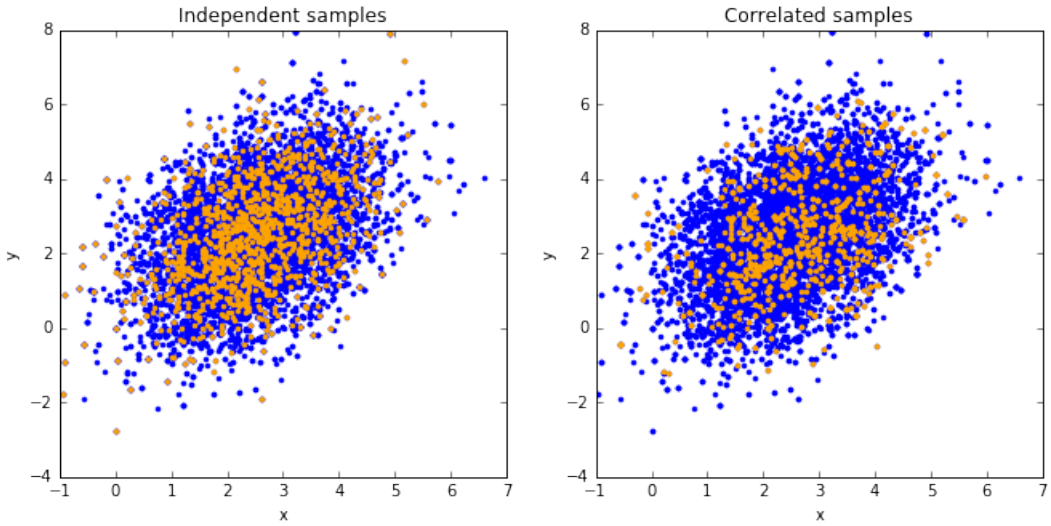


Figure 3: Top view plot of samples. Blue dots are the entire *x_result* data, Yellow dots are the samples

From this plot it's hard to say if the independent samples would be better, however the correlated samples do seem to exhibit clumping; which makes sense.

Finally, as a more direct and comprehensive comparison, I computed the sample covariance for both the samples and compared to *cov*, the covariance term in the original distribution which we sampled.

```
1
2 # covariance matrix calculated from independent samples
3 cov_independent = np.cov(run[7].T)
4 difference_independent = cov_independent - cov
5
6 # covariance matrix calculated from correlated samples
7 cov_correlated = np.cov(samples_1.T)
8 difference_correlated = cov_correlated - cov
```

The results were,

$$difference\_independent = \begin{pmatrix} 0.02159163 & -0.00358995 \\ -0.00358995 & 0.10220646 \end{pmatrix}$$

$$difference\_correlated = \begin{pmatrix} 0.02711125 & -0.05234385 \\ -0.05234385 & 0.33637548 \end{pmatrix}$$

As we can see the difference is greater for the correlated samples (particularly the off diagonal elements that are a measure of the covariance).

This is the most direct comparsion test to illustrate the effectiveness of independent samples.

For accessing the iPython files, I have created a repository on Github. The link is,
https://github.com/kcoshic/MCMC-autostopping-algorithm-returning-independent-samples

# References


[1] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*, volume 2. CRC press Boca Raton, FL, 2014.

[2] Andrew Gelman and Donald B Rubin. Inference from iterative simulation using multiple sequences. *Statistical science*, pages 457–472, 1992.