Lecture 8

MCMC Practicalities

Contents

1 Goals and Layout

In this chapter, we will explore some of the practical details of running and analyzing MCMC computations. It is best to have a specific problem in mind to anchor our discussion - so we shall consider a bivariate normal distribution as our target distribution.

We shall first develop a Metropolis MCMC method for the problem, and then consider issues including:

- burn-in
- thinning
- traceplots
- convergence diagnostics

Finally, we shall look at two elements of error analysis namely,

- auto-correlation
- block averaging

2 MCMC Simulation

2.1 Target Distribution

The multivariate normal distribution has the form:

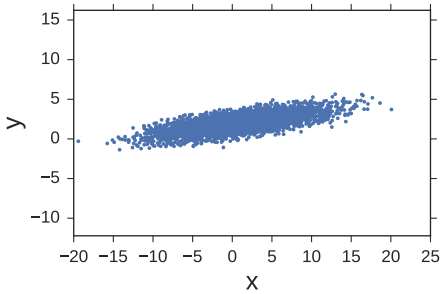
$$f_{\mathbf{x}}(x_1, \ldots, x_k) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right), \tag{1}$$

where $\mathbf{x}$ is a k-dimensional column vector and $|\boldsymbol{\Sigma}|$ is the determinant of the symmetric covariance matrix $\boldsymbol{\Sigma}$. When $k = 2$, we have a bivariate normal distribution, that we saw earlier.

Let us consider a specific bivariate distribution,

$$\boldsymbol{\mu} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} 25 & 3.5 \\ 3.5 & 1 \end{bmatrix} \tag{2}$$

This corresponds to the bivariate distribution with $\sigma_x = 5$, $\sigma_y = 1$, and $\rho = 0.7$.



2.2 Metropolis MCMC

While there are direct MC methods for sampling multivariate normal distributions, let us write a standard Metropolis MCMC routine to learn how to analyze MCMC simulations.

Python functions for the following tasks are presented in section A.2.1:

(i) the distribution to sample (`MultiDNorm`),

(ii) the proposal function (`proposal`),

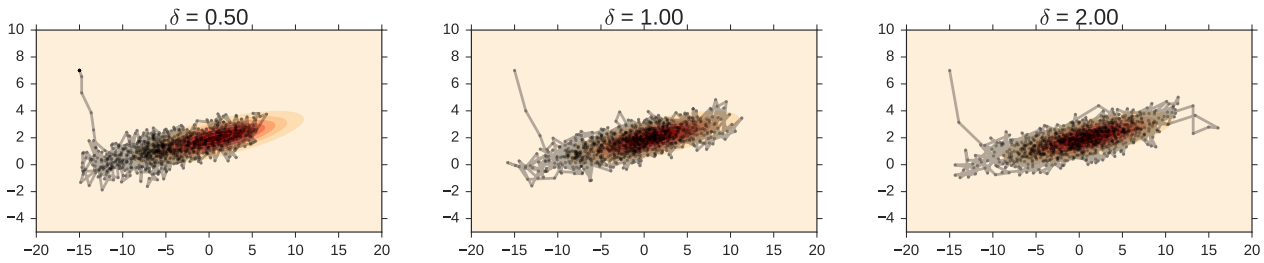
(iii) acceptance functions (`metropolis_accept`), and

(iv) the driver routine to glue everything together (`driver`).

As observables, we shall monitor three quantities: the two components x and y, and the mean squared distance from the origin $a(x, y) = x^2 + y^2$.

The objective is to estimate the mean values $\langle x \rangle$ (μ_x=1), $\langle y \rangle$ (μ_y=2), and $\langle a \rangle$ (≈ 31).

We can visualize the samples for different values of the maximum step size δ from the same starting point (-15, 7).



As δ increases from 0.5 to 2.0, the "coverage" of the distribution improves, as do the estimates of the three observables shown in the table below. Recall that expected means are $\langle x \rangle = 1$, $\langle y \rangle = 2$, and $\langle a \rangle \approx 31$.

δ	accept	$\langle x \rangle$	$\langle y \rangle$	$\langle a \rangle$
0.5	0.77	0.54	2.04	23.09
1.0	0.72	1.06	2.03	31.56
2.0	0.51	1.08	2.01	38.59

Notice that the acceptance ratio falls as δ increases. We have not furnished error-bars for $\langle x \rangle$, $\langle y \rangle$, and $\langle a \rangle$, because we don't know how to calculate them yet.

3 Practical Issues

It is now time to take on four common practical issues that we mentioned earlier. The goal of this section is to understand the nature of typical trade-offs involved in an MCMC simulation.

- Burn-in
- Thinning

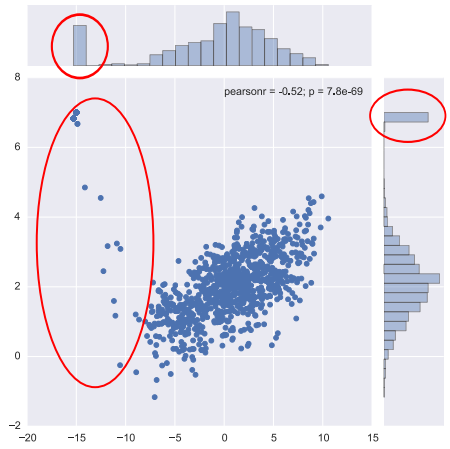
- Traceplots
- Convergence Diagnostics

Many of these choices are "fuzzy"; they fall in the category of things that you can do better with "experience". Fuzzy does not imply unimportant.

3.1 Burn-in

Burn-in is the practice of discarding initial samples to forget the starting point.

In the example, we started from an initial state that was not in the most important part of the distribution. Consequently, it took a few steps for the chain to travel from the initial state to the heart of the distribution. For instance, with $\delta = 0.5$, this is reflected in the joint and marginal PDFs as a spurious peak near the starting point.



Where does this dependence on the initial state stem from?

Recall, the state after n applications of the transition probability matrix (TPM) $\mathbf{W}$ is:

$$\boldsymbol{\pi}_n = \mathbf{W}^n \boldsymbol{\pi}_0. \tag{3}$$

$\mathbf{x}_n$ becomes independent of $\mathbf{x}_0$ only after sufficiently large n. The rate of convergence, which dictates what exactly "large n" means, depends on the second largest eigenvalue λ_2 of $\mathbf{W}$. However, for most real problems λ_2 is not known beforehand.

Guidance on how long the burn-in should be can be obtained from traceplots orconvergence diagnotics, which are discussed subsequently.

What happens if you discard too much or too little?

The trade-off is straightforward; if you discard too much, you risk throwing away good data that you worked hard for. If you discard too little, you risk contamination from samples that do not correspond to the target distribution

3.2 Thinning

MCMC generates a large number of samples.

For long runs, especially for high dimensional problems, we can run into serious storage problems, if we record all samples. In any case, consecutive states are often strongly correlated, and therefore not very informative.

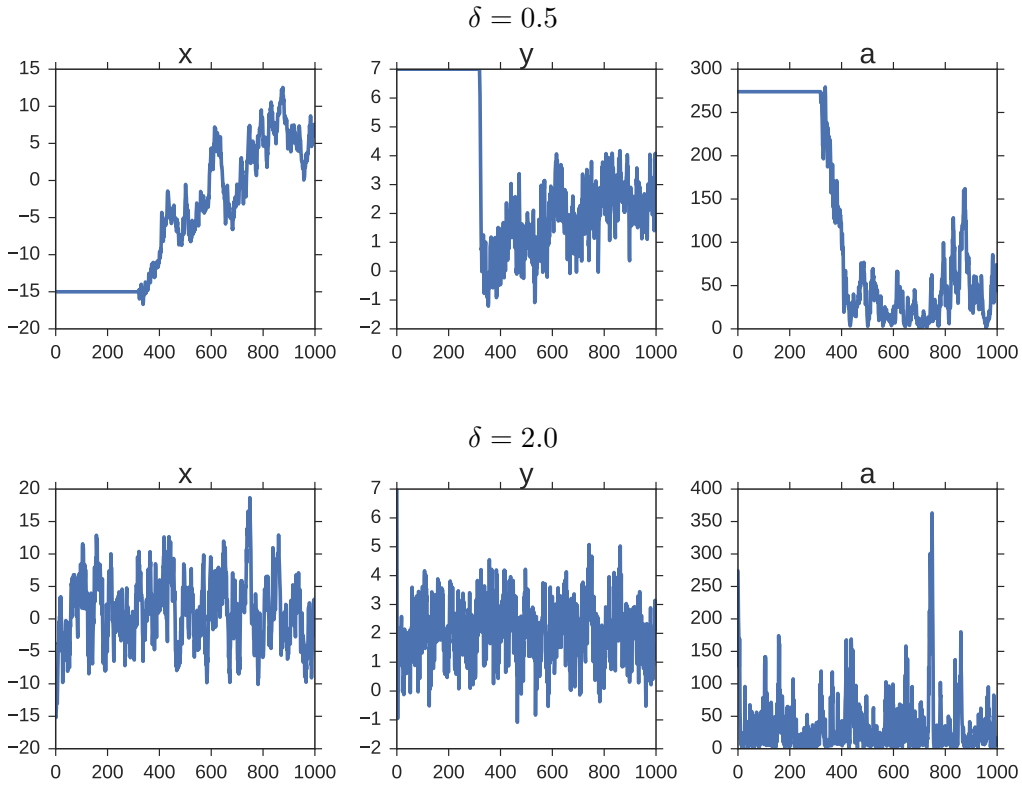
An analogy might help. Suppose you are tasked with monitoring the temperature at a particular location for an entire day. How often do you record the temperature? Every millisecond? Probably not, since temperature doesn't change meaningfully over that timescale. How about a minute, or 10 minutes, an hour, 6 hours?

There is a meaningful trade-off between storing too much (millisecond) and too little (6 hours) data. A wise choice (say 30 minutes) seeks to balance these constraints. The exact choice depends on taste, computer capacity, and target application.

3.3 TracePlots

The idea behind traceplots is simple. We plot important variables in the problem, and visually inspect if they appear "well-mixed". If the variables span the domain repeatedly over simulation run time, then that is evidence for good mixing.

This is a qualitative measure. It often reveals problems with parameter choices immediately. As extreme cases, let us consider $\delta = 0.5$ (poor mixing) and $\delta = 2.0$ (good mixing).



In this particular case, examining the traceplots for $\delta = 0.5$ and $\delta = 2.0$ would have suggested (i) increasing the burn-in time for the former, or (ii) choosing a larger δ.

If you can run the simulation for a really long time, then many of these considerations are immaterial. These are primarily efficiency considerations. In the really long run, they don't matter. But "in the long run, we are all dead"; so efficiency is paramount.

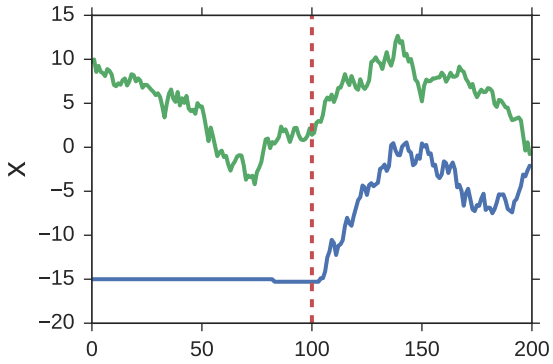
3.4 Convergence Diagnostics

There are many diagnostics to test whether an MCMC run has converged. **None of them are perfect**. They cannot prove beyond a shadow of doubt that your MCMC simulation has converged.[1]

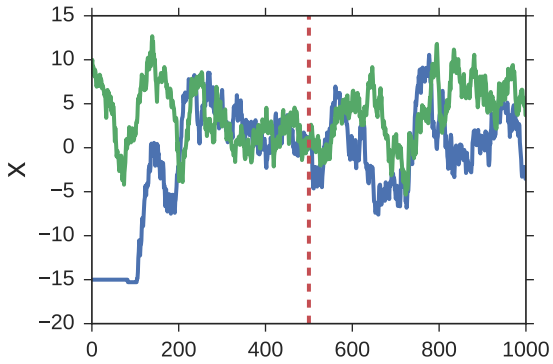
However, they can often diagnose poor mixing. Perhaps the oldest and most famous of these convergence heuristics is the *Gelman-Rubin* diagnostic, which we shall consider in some detail.

The idea is to run multiple MCMC simulations using different initial points. We want to know "Is the run is long enough to forget the effect of the starting point?"

Let us try to couch this discussion in the context of the previous example. Consider a traceplot of x for $\delta = 0.5$ starting from two different points ([-15, 7] and [10, -2]).



After the $2M = 200$ snapshots,[2] the curves still seem to carry some memory of the initial condition. The same simulations carried out to $2M = 1000$ snapshots indicate that the two chains have "merged". If we are asked to guess the initial state from only the last M snapshots, it would be quite difficult.



The dashed red line marks the midpoint M of the simulation. For the Gelman-Rubin diagnostic, we discard left half as burn-in, and entertain the right half as the production run.

3.4.1 Gelman-Rubin Diagnostic

This diagnostic provides a quantitative metric to assess convergence. At its heart lies the following intuition:

[1] In a sense, these diagnostics are a quantitative expression of what we look for in traceplots. It would not be unfair to say, "I diagnose poor mixing for myself using traceplots, but when I have to report it in a paper, I use these convergence diagnostics to convince referees that I have done the work."

[2] $M = 100$, the extra factor of 2 will make more sense shortly

Once a simulation runs long enough, the "variance" between the MCMC chains starting from different initial conditions becomes small.

This begs the obvious question, "small" compared to what? For the Gelman-Rubin diagnoistic, the answer is "the variance within-chain".

$$W = \text{within-chain variance}$$
$$B = \text{the inter-chain variance}$$

If $B \geq MW$, then effect of starting point has not subsided. When $W \gg B/M$, all the chains have escaped the influence of starting points and traversed to the target distribution.[3]

As the length of a simulation ($2M$) increases, B/M decreases. Thus, we can ensure $W > B/M$ by running a sufficiently long simulation.

The original Gelman-Rubin method formalizes this by computing a metric called the *potential scale reduction factor* or PSRF $\hat{R}$. As M increases, $\hat{R}$ approaches 1 from above. In practice, if $\hat{R} = 1.0 - 1.2$, we conclude that the simulation is long enough.

Let us first list the steps involved. Then, we shall look at an example.

The process begins by selecting a scalar variable of interest A.[4]

(i) **Run** $n \geq 2$ chains of length $2M$ from over-dispersed starting values.

- discard the first M samples in each chain
- relabel the retained values as A_{ij} for $i = 1, 2, ..., M$, and $j = 1, 2, ..., n$.

(ii) Calculate the **within-chain variance** W.

$$\bar{A}_j = \frac{1}{M} \sum_{i=1}^{M} A_{ij} \tag{4}$$

$$s_j^2 = \frac{1}{M-1} \sum_{i=1}^{M} (A_{ij} - \bar{A}_j)^2 \tag{5}$$

$$W = \frac{1}{n} \sum_{j=1}^{n} s_j^2 \tag{6}$$

$\bar{A}_j$ and s_j^2 are the mean and variance of j^{th} chain. W is the average of s_j^2. Typically, it underestimates the true variance of the stationary distribution, since our chains may not have explored the stationary distribution well. Traceplots in early regime may appear like they are "trending" towards where they want to get eventually.

(iii) Calculate the **between-chain variance** B. It uses the independence of the n chains to find the

[3] The factor of M is pesky; but it has a reason for existence, as explained shortly.

[4] In our example, this could be the variable x, y or a. Multiple variables can also be selected. Sticking with a single variable simplifies the description.

mean A^*, and the variance of the mean B/M.[5]

$$A^* = \frac{1}{n}\sum_{j=1}^{n} \bar{A}_j \tag{7}$$

$$B = \frac{M}{n-1}\sum_{j=1}^{n} (\bar{A}_j - A^*)^2 \tag{8}$$

(iv) Calculate the estimated **variance of the parameter**

$$\sigma_A^2 = \left(1 - \frac{1}{M}\right) W + \frac{1}{M} B. \tag{9}$$

(v) Calculate the **PSRF**.

$$\hat{R} = \sqrt{\frac{\sigma_A^2}{W}} = \sqrt{\left(1 - \frac{1}{M}\right) + \frac{1}{M}\frac{B}{W}}. \tag{10}$$

As $M \to \infty$, $W \gg B$, and $\hat{R} \to 1$.

(vi) Check if **PSRF = 1.0 - 1.2**; if greater, we surmise that convergence has not been attained. In such cases, we need to run our chains longer to improve convergence to the stationary distribution.

Example: Reconsider the bivariate Gaussian, with $A = x$ (the first coordinate), and $\delta = 0.5$. Compute the Gelman-Rubin for $2M = 200$.

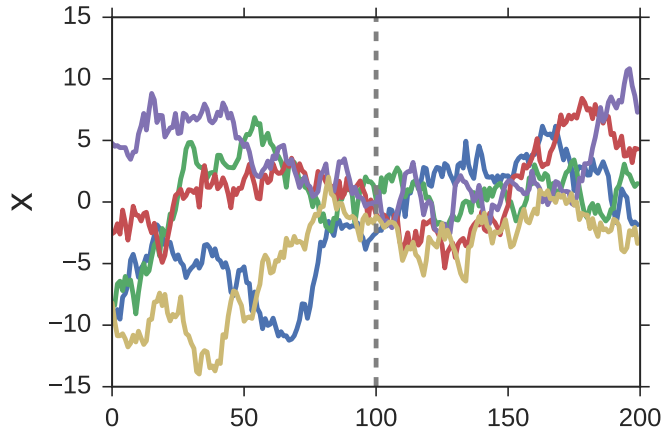
Solution: The python code for implementing the MCMC, and calculating the GR are in sections A.2.1 and A.2.2, respectively.

We modified the driver routine to pick random initial points.

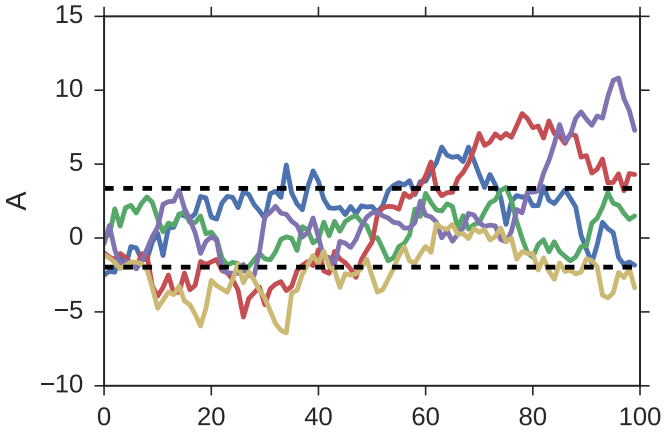
```
np.random.seed(1234)
n     = 5
thin  = 10
M     = 100
thin  = 10
delta = 0.5
```

For $n = 5$ chains, the complete trajectories are shown in the picture below. Here $A = x$.

[5] Recall that $\sigma_{\bar{z}}^2 = \sigma_z^2/n$ where $\bar{z}$ is the mean of n iid random variables z. Thus, $\sigma_z^2 = n\sigma_{\bar{z}}^2$. If the M samples within a chain were independent, then the variance B would be M times the variance of the mean.



Recall that Gelman-Rubin only considers the right half.[6] The figure below shows only the part used in subsequent analysis.



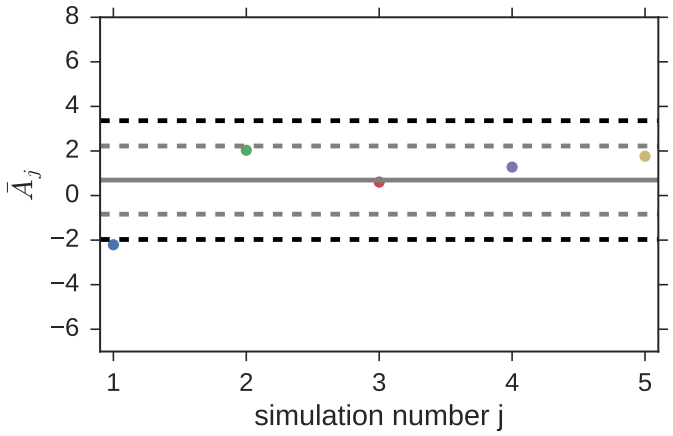
The variances for individual chains,

$$\begin{bmatrix} s_1^2 \\ s_2^2 \\ s_3^2 \\ s_4^2 \\ s_5^2 \end{bmatrix} = \begin{bmatrix} 4.17 \\ 1.91 \\ 16.23 \\ 10.53 \\ 2.70 \end{bmatrix}.$$

The mean of the five s_j^2 is the within-chain variance, $W = 7.11$. It characterizes within-chain spread. In the figure above, the separation between the two dashed black lines is $2\sqrt{W}$.

The averages of the 5 simulations $\bar{A}_j$ are shown in the figure below. They are appropriately color-coded so that the green dot corresponds to the green chain in the figure above.

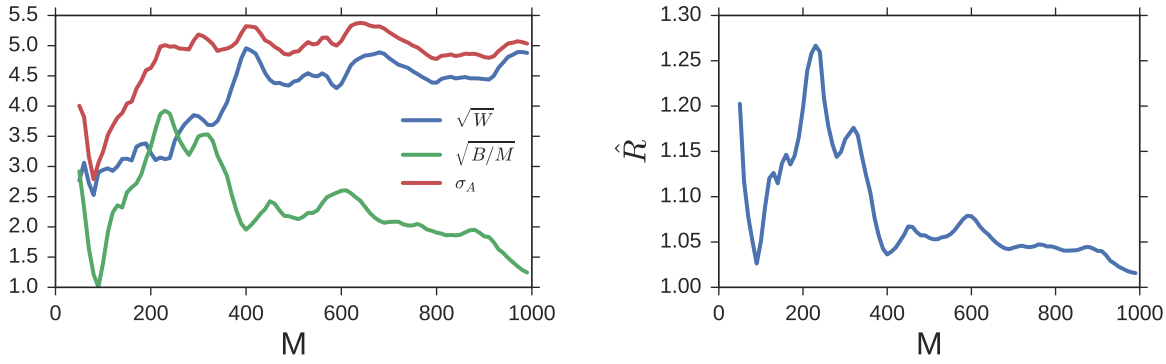
[6] x has been relabeled as A



The mean of the 5 $\bar{A}_j$ is $A^* = 0.69$. It is shown by a solid gray line. The variance of $\bar{A}_j$ is $B/M = 2.34$. The dashed gray lines depict $A^* \pm \sqrt{B/M}$. Overlaid on top of this are dashed black lines which represent $A^* \pm \sqrt{W}$ from the figure above.

Using eqns 9 and 10, we find $\sigma_A^2 = 9.38$, $\hat{R} = 1.15$. In this case, we would surmise that the chain has converged.

We can repeat these calculation for simulation of different lengths. The figure below uses simulations of length $2M = 100 - 2000$.



As simulation time increases $W > B/M$. Notice that beyond a certain point ($M \approx 300 - 400$), the variation in σ_A subsides, and it settles to a value near 5.0. Notice that the single point calculation with $2M = 200$ which yielded $\hat{R} = 1.15$ was a strange artifact. So it is often advisable to consider plots like the ones shown above to get a fuller picture.

Based on the criterion of the shrink factor, the diagnosis is that $M \approx 400$ or higher results in stable converged MCMC simulations.

Criticisms of the Gelman-Rubin diagnostic include:

- burden of finding over-dispersed starting configurations is on the user
- somewhat inefficient, since a large number of early iterations $(M \times n)$ are discarded, compared with a single long run
- the method internally relies on some uncontrolled normal approximations
- since the criterion is based on scalars (A), it does not do a good job of picking up on correlations; in such cases the true convergence is slower than the estimate

4 Error Analysis

In direct sampling or MC, we draw independent samples of random variables.[7] Sticking with our example, the quantities we are after are the means of the two co-ordinates $\langle x \rangle$, $\langle y \rangle$, and $\langle a \rangle$.

In direct sampling, with M independent samples, we have for property A,

$$\sigma^2(\langle A \rangle) = \frac{\sigma^2(A)}{M}.$$

However, in an MCMC calculation consecutive samples are not independent. What are the consequences of this correlation for error estimation?

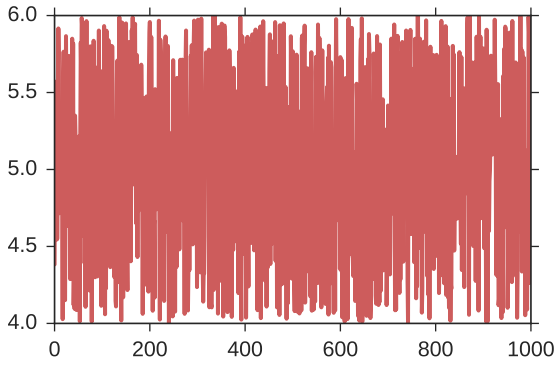
4.1 Auto-correlation

What would happen if we applied the error formula meant for independent variables to correlated variables? Let us consider a simple example to illustrate this.

Let us generate a "time series" with $M = 1000$ *independent* samples of a random number with mean value of 5 and some noise using the following python code.

```
M = 1000
x = np.random.uniform(4, 6, M)
```

The mean and standard deviation of x were 5.02, and 0.59, respectively. These samples are shown in the figure below.



The **auto-correlation function** ϕ of a "time series" series A_i with $i = 0, 1, ..., M-1$, is defined as,

$$\phi_i = \frac{\langle A_i A_0 \rangle - \langle A \rangle^2}{\langle A^2 \rangle - \langle A \rangle^2} \tag{11}$$

where $\langle A \rangle$ is the average, and the denominator is the variance of the samples. If the sample mean $\langle A \rangle = 0$,[8] then the definition simplifies to:

$$\phi_i = \frac{\langle A_i A_0 \rangle}{\langle A^2 \rangle}, \qquad \text{if } \langle A \rangle = 0. \tag{12}$$

[7] and hence, of the properties that depend on these random variables

[8] if the data is shifted by the sample mean, for instance

The term with lag i, viz. $\langle A_i A_0 \rangle$, requires some discussion. In computing it, we consider all sets of observations that are separated by a lag i. As an example,

$$\langle A_1 A_0 \rangle = \frac{\sum_{j=0}^{M-1} A_{j+1} A_j}{M} = \frac{A_1 A_0 + A_2 A_1 + ... + A_M A_{M-1}}{M}.$$

Hence,

$$\langle A_i A_0 \rangle = \frac{A_i A_0 + A_{i+1} A_1 + ... + A_M A_{M-i}}{M - i + 1}. \tag{13}$$

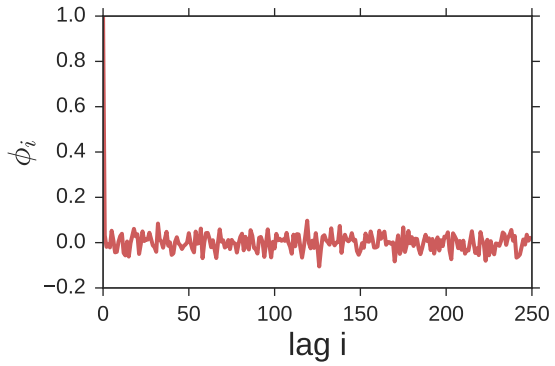
This calculation has been coded into a python program available in sec. A.2.3.

4.1.1 Independent Samples

We can use the program to compute the autocorrelation function of the $M = 1000$ independent samples drawn above.

```
ac = autocorr(x, int(M/4))
```

For this data series, the standard deviation $\sigma_x \approx 0.59$, and the mean $\langle x \rangle = 5.02 \pm 0.02$. The error bars on $\langle x \rangle$ are obtained from the standard deviation of the mean, which is given by $\sigma_x/\sqrt{M} = 0.59/\sqrt{1000} \approx 0.02$. This gives a "$2\sigma$" range for $\langle x \rangle = 4.98 - 5.06$.



We see that the autocorrelation function (ACF) quickly decays from 1 to zero. This is a hallmark of an uncorrelated process.

If successive samples are correlated, then the ACF takes its own sweet time to fall to zero.[9] The amount of time it takes to fall to zero or "decorrelate" is a measure of how correlated the data is.

4.1.2 Correlated Sample

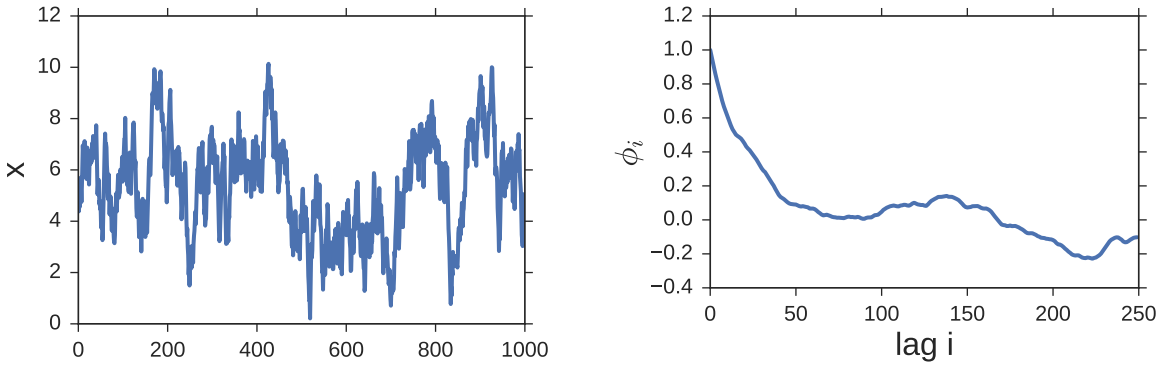
Let us now generate a set of correlated samples. We are using an autoregressive model, but you don't have to know anything about it to follow the discussion below.[10]

```
x[0] = 0.0
for i in range(1,M):
    x[i] = x[i-1] * 0.95 + np.random.uniform(-1, 1)
x = x + 5.0 * np.ones(x.shape)
```

[9] as we shall see in an example shortly

[10] If you compare traceplots from direct MC and MCMC simulations, the former would look like the "independent samples" example, and the latter would look like the time-series obtained from an autoregressive process.

The time-series, and its autocorrelation function are shown below.



The series fluctuates around "5" in a qualitatively different manner from the uncorrelated series. The ACF decays to 0 around i = 50 - 100. One way to interpret the ACF is that samples that are separated by this distance (50 - 100) are essentially independent. It quantifies the persistence or memory of the correlated samples.

As before, the standard deviation of the samples is $\sigma_x = 1.92$, and naively using, $\sigma_{\langle x \rangle} = \sigma_x/\sqrt{N} \approx 0.06$, we get $\langle x \rangle = 5.36 \pm 0.12$. This 2σ range for the mean between 5.24 and 5.48 is not the "correct" bound for the mean value of "5". That is, the error bar is underestimated if we assume correlated samples are independent. The take-home message is:

> assuming samples to be uncorrelated, when they are actually correlated, leads you to assign errorbars that are smaller than they ought to be.

4.2 Block Averaging

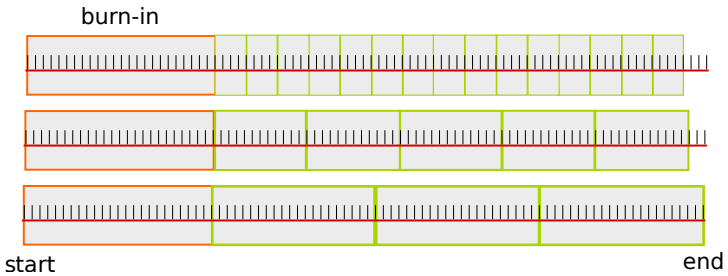
The correct expression for error for correlated samples is,

$$\sigma^2(\langle A \rangle) = \frac{\sigma^2(A)}{M}\left[1 + 2\sum_i (1 - i/M)\phi_i\right], \tag{14}$$

The blue part is the same as the error of independent samples. The additional term in square brackets, goes to 1, when $\phi = 0$ (uncorrelated variables).

Detailed error analysis of MCMC samples would require us to estimate the ACF. As we have seen in the examples above, the estimated ACF is itself noisy, which further needs to be accounted for. Personally, I recommend a simpler practical method called block averaging to estimate $\sigma^2(\langle A \rangle)$.

In block averaging, we discard the burn-in period, and retain M correlated samples. We chop this portion of the dataseries into N_b blocks of different sizes b so that $bN_b \approx M$. This is shown in the picture below.



In this schematic, the top row corresponds to $b = 4$, and $N_b = 15$. The last row corresponds to $b = 21$, and $N_b = 3$. Note that $bN_b \approx M = 63$. Unused data-points near the end don't matter much.

We shall write down the algorithm for implementing block-averaging first, and then consider why and how it works. A python program which implements the algorithm is in sec. A.2.4.

4.2.1 Algorithm

1. Define blocks of length $b = 1, 2, 3, \ldots, M$. The upper limit defined by the length of the simulation run.
2. For each block length b, let
   - $\bar{A}_j$, $j = 1, ..., N_b$ be the block average of the j^{th} block.
   - $\langle A \rangle_b$ be the mean of the $\bar{A}_j$s.
3. Compute the variance estimator:

$$\sigma^2_{\langle A \rangle_b} = \frac{1}{N_b - 1} \sum_{j=1}^{N_b} \bar{A}_j^2 - \langle A \rangle_b^2.$$

4. Plot $\sigma^2_{\langle A \rangle_b}$ versus b. The plot initially rises, and then reaches a plateau.

The **plateau value** is our best estimate of the true variance $\sigma^2(\langle A \rangle)$.

Why does block averaging work?

If the A_i are correlated, the correlation between block averages $\bar{A}_j$ should decrease as block size b increases. That is the $\bar{A}_j$s for $b = 2$ are going to be more correlated than the $\bar{A}_j$ for $b = 100$, where $j = 1, 2, ...N_b$.

Once, the block size exceeds a (unknown) correlation time present in the data, block averages will be independent. The algorithm tries to sniff out the minimum block length for which this happens. This block size is related to the period over which the ACF goes to zero (50 - 100 for the autoregressive model example).

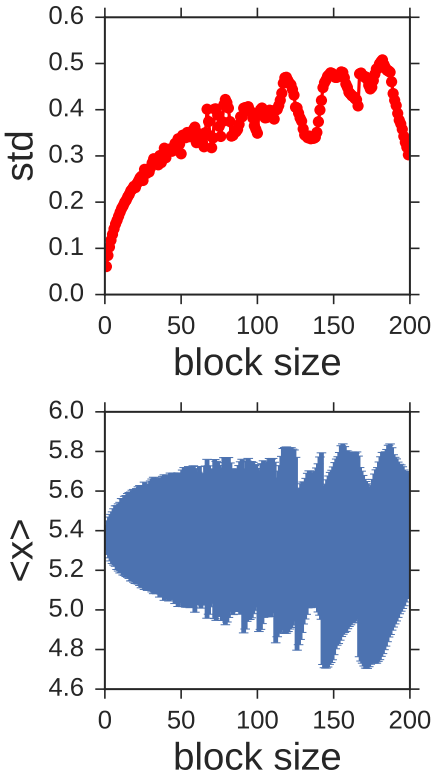
However, there is a trade-off involved. If the block size b is,

- too short: there is no improvement over correlated data and error is over-estimated
- too long: small number of blocks available leads to unreliable variance calculation.

Let us use the program `blockAverage` (sec. A.2.4) to compute an error-bar for $\langle x \rangle$.

```
blockAverage(x, True, 100)
```

The program yields $\langle x \rangle = 5.38 \pm 0.36$, which correctly reflects the lower certainty of correlated variables. Estimates of $\sigma_{\langle x \rangle_b}$ and $\langle x \rangle_b$ yielded by the program as a function of b are plotted below.



5 Problems

5.1 Thought Questions

(i) Consider the formula for the between-chain variance B in the Gelman-Rubin diagnostic. How would you "physically" interpret the B and B/M?

5.2 Numerical Problems

(i) Three Peaks

Consider the three peaks problem considered with the Metropolis MCMC algorithm with $A = x^2 + y^2$.

(a) Use traceplots to determine a reasonable size for δ in the proposal step, thinning frequency, and the length of the burn-in period.

(b) According to the Gelman-Rubin heuristic, how long should you run the simulation to ensure convergence?

(c) Plot the auto-correlation function of the samples A? Estimate the decorrelation time in units of Monte Carlo Steps (MCS).

(d) Let $\langle A \rangle = E_\pi[A]$. Using block-averaging estimate $\langle A \rangle$ and attach an errorbar to it.

A Appendix

A.1 Multidimensional and Bivariate Normal

The multivariate normal distribution has the form:

$$f_{\mathbf{x}}(x_1, \ldots, x_k) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right),$$

where $\mathbf{x}$ is a k-dimensional column vector and $|\boldsymbol{\Sigma}|$ is the determinant of the symmetric covariance matrix $\boldsymbol{\Sigma}$.

When $k = 2$, we have a bivariate normal distribution, that we saw earlier.

General 2D normal or Gaussian distribution:

$$f(x, y) = A \exp\left(-BC\right)$$

where,

$$\begin{aligned} A &= \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \\ B &= \frac{1}{2(1-\rho^2)} \\ C &= \frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2} - \frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} \end{aligned}$$

ρ is the correlation coefficient

The relationship between the "matrix" and "explicit" forms for 2D normal distribution is described via:

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{pmatrix}.$$

A.2 Python Code

A.2.1 Metropolis MCMC for Bivariate Normal

```
def MultiDNorm(x, m, S):
    """the target distribution, x = [x1, x2],
    m = mean, and S is the covariance matrix"""

    k = len(x)

    m.reshape(k,1)
    x.reshape(k,1)

    # exponential part
    v = np.exp(-0.5 * np.dot((x-m).T,
              np.dot(np.linalg.inv(S), (x-m))))
    # prefactor
    v = v/np.sqrt((2*np.pi)**k * np.linalg.det(S))
```

```
    return v

def proposal(oldx, delta):
    newx = oldx + np.random.uniform(-delta, delta,
                                    oldx.shape)
    return newx

def metropolis_accept(newx, oldFuncVal, m, S, func):
    accept     = False
    newFuncVal = func(newx, m, S)
    ratio      = newFuncVal/(oldFuncVal + 1.0e-21)

    if ratio > 1.:
        accept = True
    elif np.random.rand() < ratio:
        accept = True

    return accept, newFuncVal

def driver(delta, nsteps=10000, thin=10):
    # target distribution parameters
    mu = np.array([1., 2.]).reshape(2,1)
    Sig = np.array([[25., 3.5], [3.5, 1.]])

    # initialize
    x        = np.array([-15., 7.]).reshape(2,1)
    f        = MultiDNorm(x, mu, Sig)
    AccRatio = 0.0
    NumSucc  = 0
    recz     = np.zeros((int(nsteps/thin), 2))

    # actual chain
    for iMCS in range(nsteps):

        newx = proposal(x, delta)
        accept, newf = metropolis_accept(newx, f, mu, Sig, MultiDNorm)

        if accept:
            NumSucc += 1
            x = newx
            f = newf

        if (iMCS % thin) == 0:
            recz[int(iMCS/thin)] = x.T

    AccRatio = float(NumSucc)/float(nsteps)

    return recz, AccRatio
```

The following routine overlays samples on a contour plot of the target distribution.

```
def plotChainDist(delta):
        """Make pretty plots of samples"""
```

```
    recz, ar = driver(delta)
    plt.plot(recz[:,0],recz[:,1],'k.-', alpha=0.3)

    x = np.linspace(-20,20,100)
    y = np.linspace(-5,10,100)
    X, Y = np.meshgrid(x,y)
    Z = mpl.mlab.bivariate_normal(X, Y, sigmax=5.0, sigmay=1.0,
                                  mux=1.0, muy=2.0, sigmaxy=3.5)
    Z = Z.reshape(X.shape)

    plt.title('$\delta$ = {0:0.2f}'.format(delta), fontsize=24)
    plt.contourf(X, Y, Z, cmap=cm.OrRd)

    Nburn = 100
    meanx, meany = np.mean(recz[Nburn:-1,0]), np.mean(recz[Nburn:-1,1])
    a = recz[Nburn:-1,0]**2 + recz[Nburn:-1,1]**2
    print("Acceptance Ratio = ", ar,
          "mean x, y, a", meanx, meany, np.mean(a))
```

A.2.2 Gelman-Rubin Diagnostics

```
def GelmanRubin(A, M, n):
    """A is a matrix with n columns and M rows
    Aij = ith sample from jth chain"""

    sj2 = np.zeros(n); aj  = np.zeros(n)

    for j in range(n):
        sj2[j] = np.var(A[:,j])
        aj[j]  = np.mean(A[:,j])

    W = np.mean(sj2) # within-chain
    B = M * np.var(aj)
    s = (1. - 1./M)*W + 1./M * B # inter-chain
    R = np.sqrt(s/W)

    return R, s, W, B
```

A.2.3 Autocorrelation of a Time-Series

```
def autocorr(x, NA = 0):
    """Given vector x, computes  <x(t) x(0)>/<x(0) x(0)>
    If NA specified, NA elements of ACF are computed
    Otherwise, NA = N - 1 by default."""

    x = x - np.mean(x) # subtract mean from "x"
    N = len(x)         # size of "x"

    # If NA is not specified

    if(NA == 0):
         NA = N - 1
```

```
    A  = np.zeros((NA))

    # code: clarity > speed
    for i in range(NA):

        x1 = x[0:N-i-1]  # both terms have N - i + 1 terms
        x2 = x[i:N-1]

        A[i] = np.mean(x1 * x2)

    return A/A[0]
```

A.2.4 Block Averaging

```
def blockAverage(datastream, isplot=True, maxBlockSize=0):

    """use block average of correlated timeseries x,
       to provide error bounds for the estimated mean <x>.
       If maxBlockSize = 0, it is set to Nobs/4."""

    Nobs         = len(datastream)  # # observations
    minBlockSize = 1;               # min: 1 obsv/block

    if maxBlockSize == 0:
        maxBlockSize = int(Nobs/4); # max: 4 blocs

    # total number of block-sizes
    NumBlocks = maxBlockSize - minBlockSize

    # mean and variance for each block size
    blockMean = np.zeros(NumBlocks)
    blockVar  = np.zeros(NumBlocks)
    blockCtr  = 0

    #  blockSize is # observations/block
    #  run them through all possibilities

    for blockSize in range(minBlockSize, maxBlockSize):

        # total number of such blocks in datastream
        Nblock    = int(Nobs/blockSize)
        # container for parcelling block
        obsProp   = np.zeros(Nblock)

        # Loop to chop datastream into blocks
        for i in range(1,Nblock+1):
            ibeg = (i-1) * blockSize
            iend =  ibeg + blockSize
            obsProp[i-1] = np.mean(datastream[ibeg:iend])

        blockMean[blockCtr] = np.mean(obsProp)
```

```
        blockVar[blockCtr]  = np.var(obsProp)/(Nblock - 1)
        blockCtr += 1

    v = np.arange(minBlockSize,maxBlockSize)

    if isplot:
        plt.subplot(2,1,1)
        plt.plot(v, np.sqrt(blockVar),'ro-',lw=2)
        plt.xlabel('block size')
        plt.ylabel('std')

        plt.subplot(2,1,2)
        plt.errorbar(v, blockMean, np.sqrt(blockVar))
        plt.ylabel('<x>')
        plt.xlabel('block size')
        plt.show()

    print("<x> = {0:f} +/- {1:f}\n".format(
           blockMean[-1], np.sqrt(blockVar[-1])))

    return v, blockVar, blockMean
```