The Metropolis-Hastings Algorithm

COMPSCI 3016: Computational Cognitive Science
Dan Navarro & Amy Perfors
University of Adelaide

Abstract

This note provides a discussion of the Metropolis-Hastings algorithm. If anything the notes doesn't make sense please contact me (these notes were written by Dan: daniel.navarro@adelaide.edu.au) and I'll try to fix them!

The problem to be solved

The Metropolis-Hastings algorithm is the most popular example of a *Markov chain Monte Carlo* (MCMC) method. The basic problem that it solves is to provide a method for sampling from some generic distribution, $P(x)$. The idea is that in many cases, you know how to write out the equation for the probability $P(x)$, but you don't know how to generate a random number from this distribution, $x \sim P(x)$. This is the situation where MCMC is handy. In fact, for the Metropolis-Hastings algorithm we don't even need to know how to calculate $P(x)$ completely. For example, suppose I've become interested in the distribution shown in Figure 1. This distribution is given by:

$$P(x) = \frac{\exp(-x^2)(2 + \sin(5x) + \sin(2x))}{\int_{-\infty}^{\infty} \exp(-x'^2)(2 + \sin(5x') + \sin(2x'))\ dx'} \tag{1}$$

My problem is that I either don't know how to solve the integral in the denominator, or I'm just too lazy to try. So this means in truth, I only know the distribution up to some constant. That is, I know that:

$$P(x) \propto \exp(-x^2)(2 + \sin(5x) + \sin(2x)) \tag{2}$$

How can I generate samples from this distribution?

The Metropolis-Hastings algorithm

The basic idea behind MCMC is very simple. The idea is to define a Markov chain over possible x values, in such a way that the *stationary distribution* of the Markov chain is in fact $P(x)$. That is, what we're going to do is use a Markov chain to generate a sequence of x values, denoted $(x_0, x_1, x_2, \ldots)$, in such a way that as $n \to \infty$, we can guarantee that $x_n \sim P(x)$. There are many different ways of setting up a Markov chain that has this property, one of which is the Metropolis-Hastings algorithm.

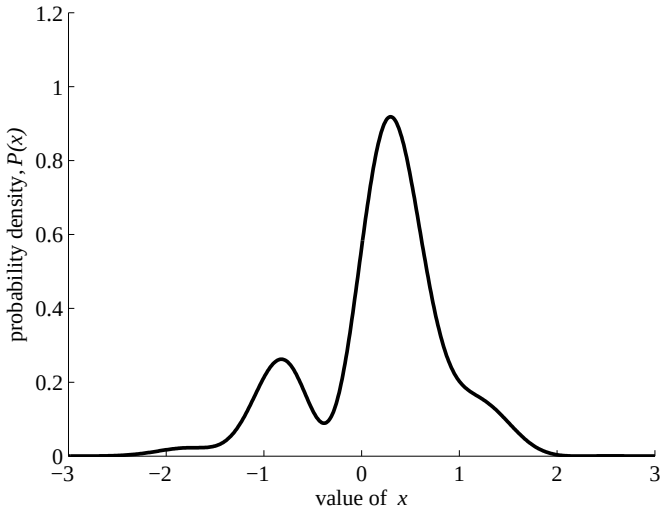

Figure 1. The distribution from which I want to draw samples.

Here's how it works. Suppose that the current state of the Markov chain is x_n, and we want to generate x_{n+1}. In the Metropolis-Hastings algorithm, the generation of x_{n+1} is a two-stage process. The first stage is to generate a *candidate*, which we'll denote x^*. The value of x^* is generated from the *proposal distribution*, denoted $Q(x^*|x_n)$, which depends on the current state of the Markov chain, x_n. There's a few minor technical constraints on what you can use as a proposal distribution, but for the most part it can be anything you like. A very typical way to do this is to use a normal distribution centred on the current state x_n. That is,

$$x^*|x_n \sim \text{Normal}(x_n, \sigma^2) \tag{3}$$

for some standard deviation σ that the user needs to specify.

The second stage is the *accept-reject* step. Firstly, what you need to do is calculate the acceptance probability $A(x_n \rightarrow x^*)$, which is given by:

$$A(x_n \rightarrow x^*) = \min\left(1, \frac{P(x*)}{P(x_n)} \times \frac{Q(x_n|x^*)}{Q(x^*|x_n)}\right) \tag{4}$$

There are two things to pay attention to here. Firstly, notice that the ratio $\frac{P(x*)}{P(x_n)}$ doesn't depend on the normalising constant for the distribution $P(x)$. That is, the integral in Equation 1 is completely irrelevant. So, for the toy problem described in the previous section,

$$\frac{P(x*)}{P(x_n)} = \frac{\exp(-{x^*}^2)(2 + \sin(5x^*) + \sin(2x^*))}{\exp(-{x_n}^2)(2 + \sin(5x_n) + \sin(2x_n))} \tag{5}$$

which is simple to compute. The second thing to pay attention to is the behaviour of the other term, $\frac{Q(x_n|x^*)}{Q(x^*|x_n)}$. What this term does is correct for any biases that the proposal distribution might induce. In this expression, the denominator $Q(x^*|x_n)$ describes the probability

of generating a x^* as the candidate given that the current state is x_n (i.e., what actually happened), whereas the numerator describes the probability that the "opposite" event would have occurred: that is, if the current state had actually been x^*, what is the probability that you would have generated x_n as the candidate value? If the proposal distribution is symmetric, then these two probabilities will turn out to be equal. For example, if the proposal distribution is a normal, then:

$$Q(x^*|x_n) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x_n - x^*)^2\right) \quad (6)$$

$$Q(x_n|x^*) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x^* - x_n)^2\right) \quad (7)$$

clearly, $Q(x^*|x_n) = Q(x_n|x^*)$ for all choices of x_n and x^*, so the ratio $\frac{Q(x_n|x^*)}{Q(x^*|x_n)} = 1$. This special case of the Metropolis-Hastings algorithm is called the *Metropolis algorithm*.

Okay. Having proposed the candidate x^* and calculated the acceptance probability, $A(x_n \rightarrow x^*)$, we now either decide to "accept" the candidate (in which case we set $x_{n+1} = x^*$) or we decide to "reject" the candidate (in which case we set $x_{n+1} = x_n$). To make this decision, we generate a (uniformly distributed) random number between 0 and 1, denoted u. Then:

$$x_{n+1} = \begin{cases} x^* & \text{if } u \leq A(x_n \rightarrow x^*) \\ x_n & \text{if } u > A(x_n \rightarrow x^*) \end{cases} \quad (8)$$

In essence, this is the entirety of the Metropolis-Hastings algorithm! There are quite a few technical issues that attach to this, and if you're interested in using the algorithm for practical purposes I strongly encourage you to do some further reading to make sure you understand the traps in detail, but for now I'll just give you some examples of things that work and things that don't, to give you a bit of a feel for how it works in practice.

Some Examples

Firstly, let's have a look at some MATLAB code (Figure 2) implementing the Metropolis-Hastings algorithm for the toy problem (i.e., sample from the distribution shown in Figure 1). Notice that in addition to the parameter σ, we also need to specify the total number of samples that we are intending to draw `nsamp`, a start point for the chain x_0, and two additional variables `burnin` and `lag` that I'll explain shortly. For the moment, however, let's start the chain *near* the middle of the distribution, but not quite there: we'll use $x_0 = -1$. Then, let's watch what happens to our sampler for three different values of σ, where we run the sampler for 1000 iterations. In one case, we'll set the value too low ($\sigma = .025$) and in another we'll set it too high ($\sigma = 50$), but in a third case we'll get it about right ($\sigma = 1$). The results are shown in Figure 3. For all three values of σ, we have two plots. The top one shows the true target distribution, along with a histogram showing the distribution of samples obtained using the Metropolis sampler. The lower panel plots the actual Markov chain: the sequence of generated values. In the leftmost plots, we see what happens when we choose a good proposal distribution: the chain shown in the lower panel moves rapidly across the whole distribution, without getting stuck in any one place (the acceptance rate here is 47.5%). In the far right panel, we see what happens when the proposal distribution is too wide: the chain gets stuck in one spot for long periods of time.

```
function [X,acc] = MHsimple

    % parameters
    burnin = 0; % number of burn-in iterations
    lag = 1; % iterations between successive samples
    nsamp = 1000; % number of samples to draw
    sig = 50; % standard deviation of Gaussian proposal
    x = -1; % start point

    % storage
    X = zeros(nsamp,1); % samples drawn from the Markov chain
    acc = [0 0]; % vector to track the acceptance rate

    % MH routine
    for i = 1:burnin
        [x,a] = MHstep(x,sig); % iterate chain one time step
        acc = acc + [a 1]; % track accept-reject status
    end
    for i = 1:nsamp
        for j = 1:lag
           [x,a] = MHstep(x,sig); % iterate chain one time step
           acc = acc + [a 1]; % track accept-reject status
        end
        X(i) = x; % store the i-th sample
    end


function [x1,a] = MHstep(x0,sig)

    xp = normrnd(x0,sig); % generate candidate from Gaussian
    accprob = targetdist(xp) / targetdist(x0); % acceptance probability
    u = rand; % uniform random number
    if u <= accprob % if accepted
        x1 = xp; % new point is the candidate
        a = 1; % note the acceptance
    else % if rejected
        x1 = x0; % new point is the same as the old one
        a = 0; % note the rejection
    end


function probX = targetdist(x)

    probX = exp(-x.^2) .* (2 + sin(x*5) + sin(x*2));
```

Figure 2. MATLAB code implementing the Metropolis-Hastings algorithm.

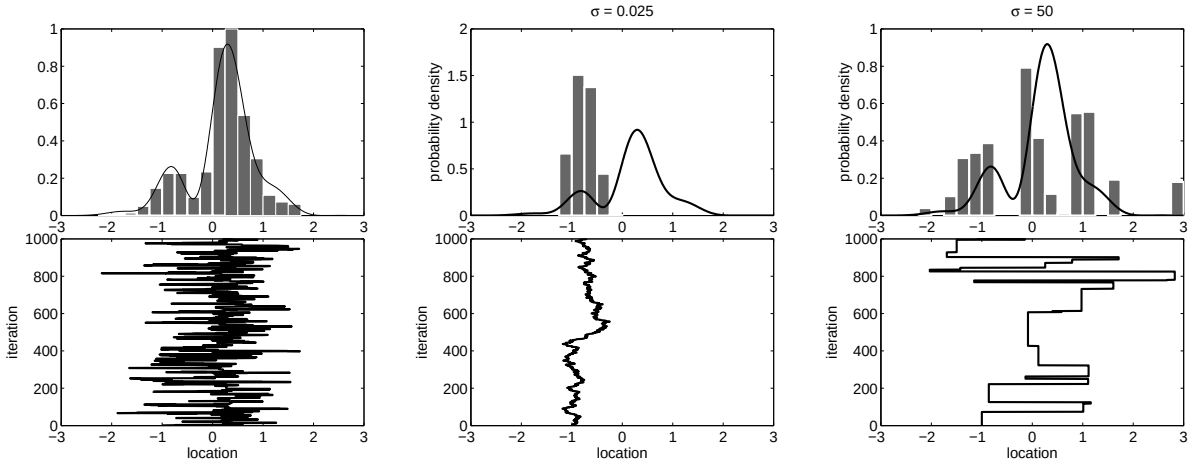

Figure 3. Metropolis samplers for the toy problem, with a good choice of proposal distribution ($\sigma = 1$, left panel), a proposal distribution that is too narrow ($\sigma = .025$, middle panel), and a proposal distribution that is too wide ($\sigma = 50$, right panel).

It does manage to make big jumps, covering the whole range, but because the acceptance rate is so low (2.5%) the distribution is highly irregular. Finally, in the middle panel, if we set the proposal distribution to be too narrow, the acceptance rate is very high (97.7%) so the chain doesn't get stuck in any one spot, but it doesn't cover a very wide range.

This simple example should give you an intuition for why you need to "play around" with the choice of proposal distribution. A good proposal distribution can make a huge difference! However, it's also important to realise that even if you don't get it quite right, you can always solve the problem with brute force. For example, Figure 4 what happens when you run the same three chains for 50,000 iterations rather than just 1000.

Now, at this point I haven't really explained what the `burnin` and `lag` parameters are there for. I don't plan to go into details, but here's the basic idea. First, let's think about the burn-in issue. Suppose you started the sampler at a very bad location. In the examples that I've shown here $x_0 = -1$ is bad, but it's not too bad. So let's pick something nastier: we'll start at $x_0 = -3$. Also, to make the issue visually obvious, we'll use a proposal distribution that is a bit too narrow, say $\sigma = .2$. Figure 5 shows 3 runs of this sampler. As you can see, the sampler spends the first 200 or so iterations slowly moving towards the main body of the distribution. Once it gets there, the samples start to look okay, but notice that the histograms are biased towards the left (i.e., towards the bad start location). A simple way to fix this problem is to let the algorithm run for a while before starting to collect actual samples. The length of time that you spend doing this is called the *burn in period*. To illustrate how it helps, Figure 6 shows what happens when you use a burn in period of 200 iterations for the same sampler.

Finally, I'll mention in passing the role played by the `lag` parameter. In some situations you can be forced into using a proposal distribution that has a very low acceptance rate. When that happens, you're left with an awkward Markov chain that gets stuck in one

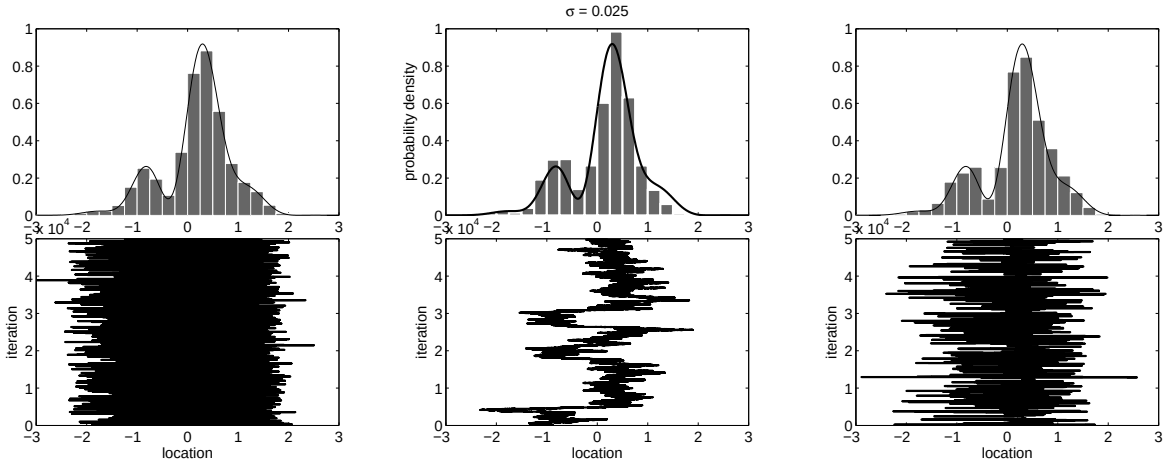

Figure 4. The same three samplers as Figure 3, run for a lot longer.

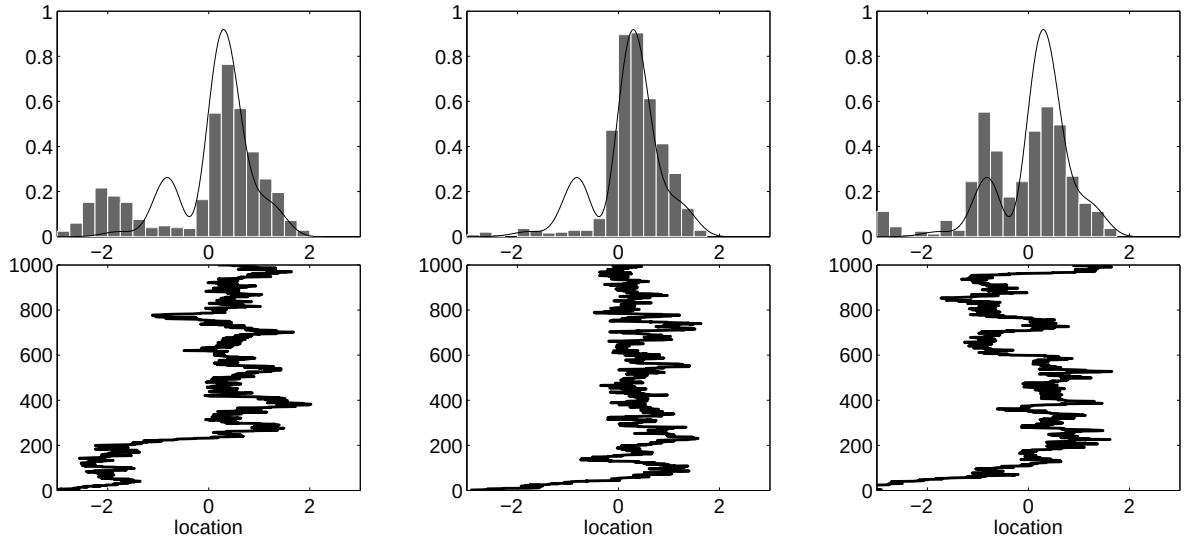

Figure 5. An example of what can happen if you don't have a burn in period, and you happen to start in a bad location ($x_0 = -3$ in this case). These were the first three chains that I ran: notice how in two of the three chains there's a very nasty bias to the left hand side, which is entirely the "fault" of the first 200 or so iterations.

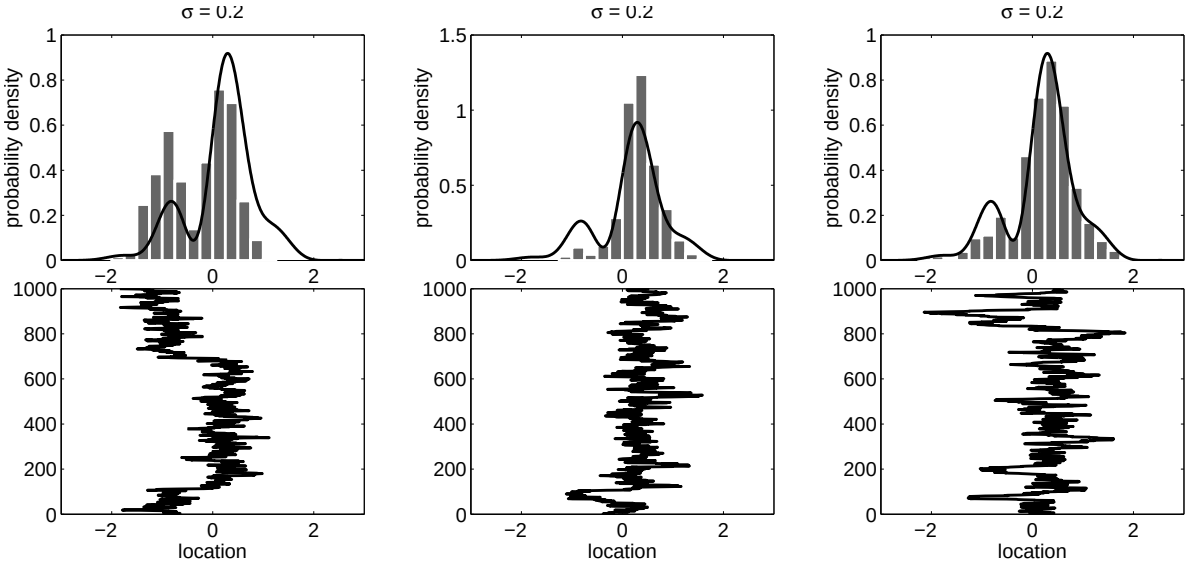

Figure 6. The same sampler as Figure 5, but with a 200 iteration burn-in period. Again, these are the first three chains I ran. The sampler still has the problem that $\sigma = .2$ is a bit too narrow, but the *bias* caused by the bad start point is now gone.

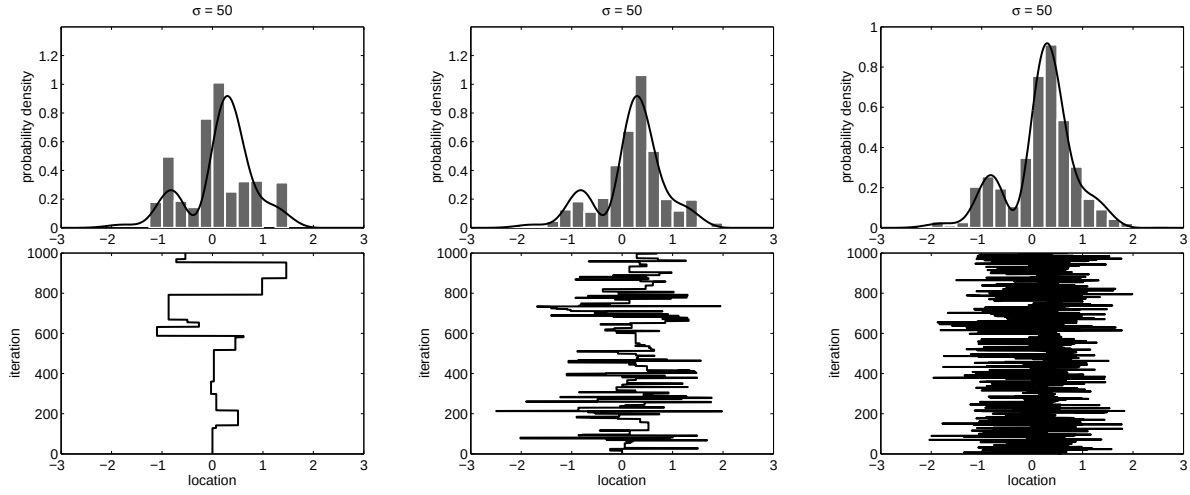

Figure 7. A sampler with a very wide proposal distribution ($\sigma = 50$) with 1000 samples drawn, at a lag of 1 (left), 10 (middle) and 100 (right) each.

location for long periods of time. One thing that people often do in that situation is allow several iterations of the sampler to elapse in between successive samples. This is the *lag* between samples. The effect of this is illustrated in Figure 7.

A Little Warning

The discussion in this note is heavily oversimplified. There are a lot of subtle issues associated with MCMC methods, and in later years I intend to expand this note to discuss some of them. However, given that I only had a few hours to write this, I haven't managed to do that this time around! For those of you who do want to read up on MCMC methods, the classic text is the Gilks et al (1996) book, *Markov Chain Monte Carlo in Practice*. However, there are a lot of very good books, articles and tutorials out there, many of which are freely available online. A quick look suggests that the reference list on the Wikipedia page is pretty decent: `http://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo`.