Statistics, Data Analysis, and Machine Learning for Physicists
Timothy Brandt
Spring 2020

Lectures 12 and 13

In this lecture we'll discuss Markov Chain Monte Carlo (MCMC): its theoretical basis, its assumptions, and the conditions under which it is useful. MCMC is an approach to the problem of computing a posterior probability density. Bayes' theorem states that

$$p(\text{model}|\text{data}) = \frac{p(\text{data}|\text{model})p(\text{model})}{p(\text{data})} \propto p(\text{data}|\text{model}). \tag{1}$$

The numerator is the product of the likelihood and the prior, which we assume you can calculate for any given model. The denominator is a normalizing constant, the integral of the probability of the data under all possible models. It may be very difficult to compute.

To get a little bit of intuition, recall the example from the first lecture. You take a test for a rare disease and it comes back positive. We want the probability that you are sick given the positive test. Bayes' theorem says that

$$p(\text{sick}|+) = \frac{p(+|\text{sick})p(\text{sick})}{p(+)}. \tag{2}$$

The numerator is easy: it's the true positive rate times my prior probability that I have the disease. The denominator is harder. In this case, we can compute the probability of a positive test by enumerating all of the possibilities: you can either be sick or healthy and get a positive test either by a false positive or a true positive. Enumerating all of the possibilities (or, in the continuous case, by doing an integral) can become incredibly difficult in a higher-dimensional problem. MCMC is often used in problems with 10 or more dimensions; these integrals may be intractable even numerically. MCMC is an approach that does not require the calculation of the normalizing constant.

If I cannot compute the normalization constant, then I cannot compute the posterior probability distribution. This is very bad. If you suggest a model, I can tell you its likelihood, but I cannot tell you how much of the probability distribution is near that model. I cannot know whether this is an especially good model, or much worse than some other models far away in parameter space. Knowing this would require the normalization of the posterior probability distribution.

If we can't (or choose not to) compute the full posterior probability distribution by normalizing it, the next best thing is to be able to draw samples from this probability distribution. That isn't the same thing: I could draw a sample from the posterior, but still be unable to tell you what the posterior probability density is for a given set of model parameters.

One way to draw samples from the posterior distribution,

$$p(\text{model}|\text{data}) \propto p(\text{data}|\text{model})p(\text{model}) \equiv \boldsymbol{\pi}, \tag{3}$$

is to use Monte Carlo sampling to draw many random models, and to weight each model by the product of the likelihood and the prior. This is called *importance sampling*. You then divide each weight by the total weight of all of your samples to normalize. Importance sampling works fine in some cases, but often the posterior probability distribution is concentrated in a small volume of parameter space. This can become a serious problem if there are many dimensions. For example, in a 5-dimensional parameter space, suppose that the 1σ contour encloses 10% of the range of each parameter. That doesn't sound so small, and in one or two dimensions, it would be easy to integrate numerically. However, if I randomly draw points, only one in 10^5 (on average) will be within the 1σ contour! To map out this contour with any fidelity, I might want $\sim$100

points inside. That means 10^7 draws for importance sampling. This scales very badly if the dimensionality increases. For a 10-dimensional parameter space where the 1σ contour encloses 10% of the range of each parameter, I would need $\sim 10^{12}$ draws to get 100 points inside the 1σ contour. Actually, it's even worse, since a 10-dimensional hypersphere of radius r doesn't have volume $(2r)^{10}$, but rather $\pi^5/5!r^{10}$, a factor of $\sim$400 lower. So I would need something like 10^{14} points, which would require a supercomputer.

MCMC is a way of sampling a posterior probability distribution when most of the posterior probability is concentrated in a small fraction of the volume of parameter space. This is usually the case when the dimensionality is high, $\gtrsim 5$. In other words, in most applications, the vast majority of possible models are poor fits to the data. For lower dimensionality it is often faster to use importance sampling or to just do the integral and normalize the posterior by brute force. I would generally recommend one of these choices if your dimensionality is three or less.

MCMC abandons the idea that we will have independent models, truly independent samples of our probability distribution. To achieve a faster convergence, we must accept that our samples from the posterior will be correlated. In effect, we will take a random walk through the posterior probability space, and our path through that space will serve as a map.

MCMC stands for Markov Chain Monte Carlo, and those words mean the following:

- A Markov chain is a sequence of points where a given point depends on the previous point, but not on the rest of the chain. In other words, if I find myself at point $\mathbf{x}$ in parameter space, my odds of going to $\mathbf{x}'$ as my next step do not depend on where I was before I got to $\mathbf{x}$ or how I ended up at that point.
- Monte Carlo means that I step around parameter space randomly.

These two things together make something like a random walk, but a random walk that is guided by $\boldsymbol{\pi}$, the product of the likelihood and the prior. Once I find myself at point $\mathbf{x}$, the probability of moving to $\mathbf{x}'$ next is the transition probability $p(\mathbf{x}'|\mathbf{x})$. I will assume that this transition probability is fixed throughout my chain (that the chain is stationary). In what follows, I will assume that there are a finite (though potentially very large) number of $\mathbf{x}$ in the parameter space of the models. The results generalize to the continuous case.

Suppose I am at a given point $\mathbf{x}_i$ and that my parameter space contains N models. N can be large, but I am assuming it to be finite. In practice, this could be something as trivial as saying that the models are only defined at values that can be represented in the computer using floating point. There are then N possibilities for the next step in the chain, so N transition probabilities

$$M_j \equiv p(\mathbf{x}_j|\mathbf{x}_i) \tag{4}$$

to move from point $\mathbf{x}_i$ to point $\mathbf{x}_j$ with $\sum_j M_j = 1$ (since I have to go somewhere for the next step). There is no requirement that $i \neq j$; it is perfectly fine to remain at the same position. The full matrix $\mathbf{M}$ has one of these columns for each of the N points, giving an $N \times N$ stochastic matrix. Again, N can be fantastically large, so this matrix isn't really good for anything except formal proofs.

What we want from MCMC is to have our desired distribution $\boldsymbol{\pi}$ be a stationary distribution of the transition probability matrix $\mathbf{M}$. What this means is that if I have a vector consisting of the model weights given by the target distribution $\boldsymbol{\pi}$, then

$$\mathbf{M}\boldsymbol{\pi} = \boldsymbol{\pi} = \mathbf{M}^k\boldsymbol{\pi} \quad \forall k \geq 0. \tag{5}$$

In other words, assume that a given point in my Markov chain is drawn from the target distribution: I can represent it as a probability for each state equal to the probability in the distribution I wish to sample. The vector $\boldsymbol{\pi}$ contains the normalized probability for each $\mathbf{x}$. Once I am drawing from my desired distribution, I want to keep drawing from my desired distribution forever. I can guarantee that the target distribution will be a stationary distribution of $\mathbf{M}$ if I can make $\mathbf{M}$ satisfy *detailed balance*:

$$\boldsymbol{\pi}(\mathbf{x}')p(\mathbf{x}|\mathbf{x}') = \boldsymbol{\pi}(\mathbf{x})p(\mathbf{x}'|\mathbf{x}). \tag{6}$$

Proof: the probability that I am at $\mathbf{x}'$ now and go to $\mathbf{x}$ next is the product of $\boldsymbol{\pi}(\mathbf{x}')$ (I assume I am now drawing from $\boldsymbol{\pi}$) times the transition probability to $\mathbf{x}$, $p(\mathbf{x}|\mathbf{x}')$. The probability of my next step being $\mathbf{x}$ is the sum (or integral, in the continuous case) over all possible $\mathbf{x}'$,

$$p(\mathbf{x}) = \sum_{\mathbf{x}'} \boldsymbol{\pi}(\mathbf{x}')p(\mathbf{x}|\mathbf{x}'). \tag{7}$$

Now I use detailed balance to substitute the argument of the summation (I'm still summing over $\mathbf{x}'$, though):

$$p(\mathbf{x}) = \sum_{\mathbf{x}'} \boldsymbol{\pi}(\mathbf{x})p(\mathbf{x}'|\mathbf{x}). \tag{8}$$

The term $\boldsymbol{\pi}(\mathbf{x})$ factors out of the sum:

$$p(\mathbf{x}) = \boldsymbol{\pi}(\mathbf{x}) \sum_{\mathbf{x}'} p(\mathbf{x}'|\mathbf{x}). \tag{9}$$

Now, the sum is the probability of moving to state $\mathbf{x}'$ from $\mathbf{x}$, summed over all possible $\mathbf{x}'$. It is the probability of being *somewhere* for the next step. This probability has to be one; the transition probabilities must be normalized. So, the next step will obey $p(\mathbf{x}) = \boldsymbol{\pi}(\mathbf{x})$.

We have proved that if we can construct the matrix of transition probabilities, then once we start sampling from the target distribution, we will always sample from the target distribution. To see how we converge from an arbitrary starting distribution, we will keep analyzing the matrix $\mathbf{M}$. We have already proved that $\boldsymbol{\pi}$ is an eigenvector of this matrix with eigenvalue 1. We'll now assume without proof that $\mathbf{M}$ has a full set of N eigenvectors, and that these eigenvectors span the full N-dimensional state space. This is normally the case, though it is possible to have pathological situations where it doesn't work. Assuming I have a full set of eigenvectors, I can write an initial point $\mathbf{x}^{(0)}$ as a weighted sum of the eigenvectors $\mathbf{y}_i$, with eigenvalues λ_i,

$$\mathbf{x}^{(0)} = \sum_i w_i \mathbf{y}_i. \tag{10}$$

Now, if I apply the matrix $\mathbf{M}$ k times, I have

$$\mathbf{M}^k \mathbf{x}^{(0)} = \sum_i w_i \mathbf{M}^k \mathbf{y}_i = \sum_i w_i \lambda_i^k \mathbf{y}_i. \tag{11}$$

If any of the $\lambda_i > 1$, then the eigenvector with the largest eigenvalue will blow up and what started as a normalized probability vector ($\mathbf{x}^(0)$) with all elements ≥ 0 will not be normalized any longer. This isn't possible for a valid matrix of transition probabilities, which should give me exactly one location (or a probability distribution that sums to one) for each step of the chain. So, all eigenvalues must be $\leq$1. There is no requirement that the elements of the eigenvectors be nonnegative; these eigenvectors will generally not be probability distributions. The only remaining question is whether the stationary distribution $\boldsymbol{\pi}$ is unique (the eigenvalue 1 is non-degenerate). The answer is generally yes (again stated without proof). In that case, as long as I have $w_i > 0$ for the invariant distribution $\boldsymbol{\pi}$ in my initial state (i.e. I haven't managed to somehow choose a state completely orthogonal to the target distribution), all of the other eigenvectors die off ($\lambda_i^k \to 0$ as $k \to \infty$) and I will be left sampling from the distribution I want. The time for these transient terms to die away is called the *burn-in* time of the chain.

So, regardless of where I start, I will converge to the target distribution. I will need to estimate the burn-in time and throw away the initial samples; this is usually something that we do after the fact. We see how long it takes for the typical values of the parameters to approach their long-term averages and throw out the samples before that.

So far, this is all academic. To see how we can use this in practice, let's write down a transition probability matrix that does indeed satisfy detailed balance. This one is called the Metropolis-Hastings algorithm, and has long been a common approach in MCMC. More recently other techniques have become more efficient and

more commonly used. Metropolis-Hastings decomposes the transition probability into two steps: a proposal step, and an acceptance step. The proposal step consists of drawing a new set of parameters by stepping away from the current set. The acceptance step consists of deciding whether or not to take the proposed increment in parameters.

In the following discussion, I will assume that the transition probability is symmetric, in the sense that the probability of proposing a step from a to b is the same as the probability of proposing a step from b to a. The following discussion becomes more complicated if this assumption is not satisfied. The probability to accept a step from $\mathbf{x}$ to $\mathbf{x}'$ should then be

$$p(\text{accept}) = \min\left(1, \frac{\boldsymbol{\pi}(\mathbf{x}')}{\boldsymbol{\pi}(\mathbf{x})}\right) \tag{12}$$

to satisfy detailed balance. Intuitively, if $\boldsymbol{\pi}(\mathbf{x}') > \boldsymbol{\pi}(\mathbf{x})$, then the new step has a higher probability density than the old one, and I always take it (with probability 1). If $\boldsymbol{\pi}(\mathbf{x}') < \boldsymbol{\pi}(\mathbf{x})$, then the new step has a lower probability density than the old step, and I sometimes take it (with probability equal to the ratio of the likelihoods). Notice that I never need a normalized $\boldsymbol{\pi}$, but only ratios of $\boldsymbol{\pi}$. I only need an unnormalized probability density to be able to do this.

To prove that Equation (12) satisfies detailed balance, we substitute the product of the proposal distribution $q(\mathbf{x}'|\mathbf{x})$ and the acceptance probability of Equation (12) into Equation (6):

$$\boldsymbol{\pi}(\mathbf{x}) \times q(\mathbf{x}'|\mathbf{x}) \times \min\left(1, \frac{\boldsymbol{\pi}(\mathbf{x}')}{\boldsymbol{\pi}(\mathbf{x})}\right) \stackrel{?}{=} \boldsymbol{\pi}(\mathbf{x}') \times q(\mathbf{x}|\mathbf{x}') \times \min\left(1, \frac{\boldsymbol{\pi}(\mathbf{x})}{\boldsymbol{\pi}(\mathbf{x}')}\right). \tag{13}$$

Since we assumed that the proposal distribution is symmetric, the q terms cancel out (otherwise you would need to include them in the acceptance probability). In that case, this simplifies to

$$\boldsymbol{\pi}(\mathbf{x}) \times \min\left(1, \frac{\boldsymbol{\pi}(\mathbf{x}')}{\boldsymbol{\pi}(\mathbf{x})}\right) \stackrel{?}{=} \boldsymbol{\pi}(\mathbf{x}') \times \min\left(1, \frac{\boldsymbol{\pi}(\mathbf{x})}{\boldsymbol{\pi}(\mathbf{x}')}\right). \tag{14}$$

Now, there are three possibilities: either $\boldsymbol{\pi}(\mathbf{x}') < \boldsymbol{\pi}(\mathbf{x})$, $\boldsymbol{\pi}(\mathbf{x}') < \boldsymbol{\pi}(\mathbf{x})$, or they are equal. If they are equal then the acceptance probabilities are both unity, and we recover $\boldsymbol{\pi}(\mathbf{x}') = \boldsymbol{\pi}(\mathbf{x})$. If $\boldsymbol{\pi}(\mathbf{x}') < \boldsymbol{\pi}(\mathbf{x})$, then the left-hand side will take the ratio of the probabilities and the right-hand side will take 1. We have

$$\boldsymbol{\pi}(\mathbf{x}) \times \left(\frac{\boldsymbol{\pi}(\mathbf{x}')}{\boldsymbol{\pi}(\mathbf{x})}\right) \stackrel{?}{=} \boldsymbol{\pi}(\mathbf{x}') \times 1, \tag{15}$$

which checks out. By the exact same reasoning detailed balance is also satisfied if $\boldsymbol{\pi}(\mathbf{x}') > \boldsymbol{\pi}(\mathbf{x})$.

So, the acceptance probability is just the ratio of the distributions I want to sample. In python, implementing this is as easy as the following:

```
accept = numpy.random.rand() < likelihood(xnew)/likelihood(xold)
```

where `accept` is a Boolean variable, and equals `True` if we are to take the proposed step. Note that there is no need to actually call a function to take the minimum: if the new likelihood is larger than the old, we will always take the step (we're comparing it to a random number between 0 and 1). Also, where I wrote `likelihood`, I really mean the product of the likelihood and the prior.

Instead of the likelihood, you might write down a log likelihood to avoid overflow errors or just because it's easier. In that case, you would write

`accept = numpy.random.rand() < numpy.exp(loglike(xnew) - loglike(xold))`.

Again, `loglike` is really the combination of the likelihood and prior. The acceptance probability, then, is trivial to write down. The art is in choosing the proposal distributions.

Our discussion was for stationary chains, and to get a nice, simple acceptance probability, we also assumed that the proposal distribution is symmetric. An example of a proposal distribution that does satisfy symmetry is a Gaussian. An example of a proposal that does not satisfy symmetric is a Gaussian with nonzero

mean. In several dimensions, an example of a proposal distribution that satisfies symmetry is one in which you randomly decide one variable to increment at a time. If you incremented the variables in some predetermined order, that would not even satisfy the stationary chain criterion (that the transition probabilities don't change with time). Another example of an asymmetric transition probability is one in which you make a proposed step smaller or asymmetric near a boundary of parameter space. The correct way to handle boundaries is in the acceptance probability: you are free to set the prior of an unphysical model to zero.

As a very simple example, and one that you will use in your homework, take the proposal distribution to be a Gaussian with zero mean and some variance. The proposed step will be the current step plus a random variable drawn from this Gaussian. If you have more than one dimension, then your proposal distribution can consist of two random numbers: one that determines the variable to increment, and a second that determines the size of the proposed step. The characteristic step sizes can (and should) differ in the different dimensions to converge faster. Assuming that `x` is a numpy array of length `npar` representing the (current) parameter values and `sig` is an array representing the standard deviations of the proposal distributions, the Metropolis-Hastings algorithm might look something like this:

```
n_visits = 0
for i in range(N_MCMC):
    n_visits += 1
    par_to_vary = np.random.randint(npar)
    dx = np.zeros(npar)
    dx[par_to_vary] = np.random.normal()*sig[par_to_vary]
    accept = np.random.rand() < likelihood(x + dx)/likelihood(x)
    if accept:
        print(x, n_visits)
        x += dx
        n_visits = 0
```

You would probably replace the print statement with actually writing the values and number of visits to a file. Note that I am only writing each set of values once by doing it right before incrementing. The variable `n_visits` keeps track of how many chain elements had that set of parameter values; it saves me from repeating a line many times in a file. For a 25% acceptance rate, putting that write statement within the accept block should save a factor of $\sim$4 in disk space without sacrificing any information. Also, where I write `likelihood` I really mean the product of the likelihood and prior. But there, in 11 lines, is MCMC, and that is part of the reason why it has become popular.

Choosing the step sizes for the proposal distribution is an important decision and will determine how quickly your chain converges. The difference in performance between and good choice and a poor choice of proposal distribution may be many orders of magnitude in the correlation length of your chain. For now, I will assume that your proposal distribution takes a Gaussian step in one variable at a time; the task is then to choose the standard deviations of these Gaussians. The following procedure (sometimes called *sigma-tuning*) will do that for you:

1. Use a multidimensional optimization routine (or MCMC itself) to find the (approximate) peak of your posterior distribution.

2. Start a short chain near the posterior with guesses for the standard deviations of the proposal distributions for each parameter.

3. Record the acceptance rates of the steps in each parameter. If the acceptance rate in a given parameter is much larger than 25%, increase its standard deviation. If the acceptance rate is much smaller than 25%, reduce the standard deviation.

4. Repeat steps (2)-(3) until your acceptance rates are around 25% in all parameters.

An MCMC chain consists of a long sequence of dependent data points, and also includes an initial "burn-in" time during which the transient behavior dies away and it converges to the posterior distribution. To actually use your chain, you must first discard the burn-in, and then compute the number of quasi-independent points in your chain to assess its convergence. Discarding the burn-in isn't a very rigorous process, and it's better to err on the side of caution and discard more rather than less. If you divide up your chain into many sub-chains, then the first few subchains (while the chain is burning in) will have higher variances and significantly different means than the latter subchains. You could take a running mean of the value of each parameter in your chain find the point at which the running mean matches the long-term average in all parameters. You would then discard all of the points up to then as the burn-in. Remember, if the burn-in time seems to be different in different parameters, you need to discard the maximum number of points given by any of the parameters.

Once you have discarded the burn-in, you need to account for the fact that the data points are not independent: the "effective" number of independent points can be much, much less than the total length of the chain. For simple Monte Carlo, the number of independent points is equal to the number of points drawn. For MCMC, there are no truly independent points, but we can see how far apart our points have to be to lose most of their autocorrelation. There are a few ways to do this, and I will present two:

1. Divide up your chain (after discarding burn-in) into many chains of length n. The variance of any given variable within the chains should be comparable to its variance in the entire chain. In a single variable x, this means
$$\sigma_x^2[\text{entire chain}] \sim \langle \sigma_x^2[\text{subchains}] \rangle \tag{16}$$
The subchains will still have some correlation, so in practice we might choose n for the subchains such that
$$\frac{1}{2}\sigma_x^2[\text{entire chain}] = \langle \sigma_x^2[\text{subchains}] \rangle, \tag{17}$$
where $\langle\rangle$ denotes an average. The number of effective points in the chain is then N, the total number of points, divided by the n for which the subchains satisfy Equation (17).

2. Another, similar thing to do is to compute the autocorrelation function,
$$f(x,n) = \frac{1}{N}\sum_{i=1}^{N} \frac{(x_{i+n} - x_i)^2}{\langle x^2 \rangle - \langle x \rangle^2}. \tag{18}$$
For $n = 0$, $f(x,0) = 0$. Assuming a Gaussian distribution of x, $f(x,n) \to 2$ as $n \to \infty$. You can take the n for which $f(x,n) = 1$ as another estimate of the correlation length of the chain in parameter x.

Very roughly speaking, if you want a 1% measurement of a confidence interval on a parameter from MCMC, you need a chain as long as 10,000 correlation lengths (I assumed Gaussianity to use $\sqrt{N}$). If you only need a 10% measurement of a confidence interval, then a chain as short as ∼100 correlation lenghts may suffice.

MCMC gives a realization of the full posterior probability distribution. This distribution has n dimensions, where n is the number of parameters in your model. It is very, very difficult to visualize a probability distribution in n parameters, or really, in anything more than two parameters. A one-dimensional probability distribution is just given by a probability density–a line plot. You can show a two-dimensional probability density with a density plot (as you did for the first homework). For more dimensions than that, we showed how, in the case of Gaussianity, you can describe the shape of the distribution using a covariance matrix.

To get a sense of what we will do with the output of MCMC, first imagine the covariance matrix
$$\mathbf{C} = \begin{bmatrix} \sigma_1^2 & \dots & \mathrm{Cov}(x_k, x_1) & \dots \\ \vdots & \ddots & & \dots \\ \mathrm{Cov}(x_1, x_k) & & \sigma_k^2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{19}$$

The covariance is symmetric, $\mathrm{Cov}(x_k, x_1) = \mathrm{Cov}(x_1, x_k)$, so I really only need the lower (or upper) triangular part of this matrix. The diagonal components are the second moments of the marginalized posterior probability distributions of the parameters. The top-left term, σ_1^2, is the second moment of the posterior distribution for parameter 1 integrated (marginalized) over parameters 2 through n. The off-diagonal elements are the first moments of the posterior distributions of, e.g., $(x_1 - \langle x_1 \rangle)(x_k - \langle x_k \rangle)$, marginalized over all parameters other than x_1 and x_k. The covariance matrix then has $n(n+1)/2$ unique entries, and each one is a single number.

The most common way that the results of an MCMC chain are plotted is with something similar to the covariance matrix, but with the full $n(n-1)/2$ distributions plotted rather than just their moments. For the diagonal elements, these are just one-dimensional distributions, which you can plot with histograms. For the off-diagonal elements you need to use density plots. There is a nice python package called `corner` that will do most of this for you, and I strongly recommend that you begin there. You can easily install it yourself using `pip`, the python package index. Schematically, a corner plot looks just like the covariance matrix shown above, but with histograms where σ_1^2, σ_k^2, etc. are written, and with two-dimensional density plots where, e.g., $\mathrm{Cov}(x_1, x_k)$ is written. The upper-right of the plot will be blank (hence the name *corner plot*).

This discussion of MCMC hasn't really touched on its usefulness. MCMC takes a huge penalty at the start: it is drawing dependent samples from a probability distribution, rather than independent samples. In order for it to beat regular Monte Carlo, very roughly speaking, *the fraction of points drawn by regular Monte Carlo that lie within the 1σ confidence interval should be smaller than the inverse of the MCMC chain correlation length.* Basically, regular Monte Carlo is wasteful: most of the points might lie well outside the 1σ contour. MCMC is also wasteful. This is a test to see which algorithm is *less* wasteful. And don't forget, especially when you have $\leq$3 dimensions, you are probably better off just integrating the posterior directly with some kind of grid-based approach (as you did for Homework 1). MCMC really shines for high-dimensional problems (say, $\gtrsim$5), where the fraction of the allowable parameter space lying within a 1σ contour could be very small indeed.

Finally, don't forget that MCMC provides samples from the posterior probability distribution defined by the product of your likelihood and your prior. It's not magic: a poor likelihood model will lead to a meaningless posterior. This is even more true for MCMC than for something like χ^2 fitting. At least for χ^2 fitting, you could easily see whether your best-fit model was formally a good fit; you had the null test that $\chi^2_{\mathrm{dof}} \approx 1$. You don't have that for MCMC. You might be able to plug in your best-fit model as determined by MCMC and estimate whether that model is a good fit (and if you can, you should!). Again, just because someone has done MCMC correctly doesn't mean the posterior is right. It only means that the posterior they have calculated is indeed the product of their adopted prior and likelihood function. People assume Gaussianity in their likelihood functions all of the time, and it can lead to grossly incorrect results if this assumption is not warranted. Concealing Gaussianity assumptions beneath MCMC doesn't change that fact.

Now, if you're ready to actually use MCMC in your work, it's probably best to leave the Metropolis-Hastings algorithm behind and to use a package written specifically for long MCMC chains. The standard one in astronomy is called `emcee`, though there is nothing specific to astronomy in the algorithm. It could just as well be applied to any problem where you can write down your likelihood and prior. You may install `emcee` using `pip`, feed it an initial guess and a likelihood function that includes your prior, and get the chain and corner plots out at the end.