emcee: The MCMC Hammer

Daniel Foreman-Mackey[1,2], David W. Hogg[2,3], Dustin Lang[4,5], Jonathan Goodman[6]

ABSTRACT

We introduce a stable, well tested Python implementation of the affine-invariant ensemble sampler for Markov chain Monte Carlo (MCMC) proposed by Goodman & Weare (2010). The code is open source and has already been used in several published projects in the astrophysics literature. The algorithm behind emcee has several advantages over traditional MCMC sampling methods and it has excellent performance as measured by the autocorrelation time (or function calls per independent sample). One major advantage of the algorithm is that it requires hand-tuning of only 1 or 2 parameters compared to $\sim N^2$ for a traditional algorithm in an N-dimensional parameter space. In this document, we describe the algorithm and the details of our implementation. Exploiting the parallelism of the ensemble method, emcee permits *any* user to take advantage of multiple CPU cores without extra effort. The code is available online at http://dan.iel.fm/emcee under the MIT License.

Subject headings: methods: data analysis — methods: numerical — methods: statistical

[1]To whom correspondence should be addressed: danfm@nyu.edu

[2]Center for Cosmology and Particle Physics, Department of Physics, New York University, 4 Washington Place, New York, NY, 10003, USA

[3]Max-Planck-Institut für Astronomie, Königstuhl 17, D-69117 Heidelberg, Germany

[4]McWilliams Center for Cosmology, Department of Physics, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213

[5]Princeton University Observatory, Princeton, NJ, 08544, USA

[6]Courant Institute, New York University, 251 Mercer St., New York, NY 10012, United States

Note: If you want to get started immediately with the emcee package, start at Appendix A on page 15 or visit the online documentation at http://dan.iel.fm/emcee. If you are sampling with emcee and having low-acceptance-rate or other issues, there is some advice in Section 4 starting on page 9.

1. Introduction

Probabilistic data analysis—including Bayesian inference—has transformed scientific research in the past decade. Many of the most significant gains have come from numerical methods for approximate inference, especially Markov chain Monte Carlo (MCMC). For example, many problems in cosmology and astrophysics[1] have directly benefited from MCMC because the models are often expensive to compute, there are many free parameters, and the observations are usually low in signal-to-noise.

Probabilistic data analysis procedures involve computing and using either the posterior probability density function (PDF) for the parameters of the model or the likelihood function. In some cases it is sufficient to find the maximum of one of these, but it is often necessary to understand the posterior PDF in detail. MCMC methods are designed to sample from—and thereby provide sampling approximations to—the posterior PDF efficiently even in parameter spaces with large numbers of dimensions. This has proven useful in too many research applications to list here but the results from the NASA Wilkinson Microwave Anisotropy Probe (WMAP) cosmology mission provide a dramatic example (for example, Dunkley *et al.* 2005).

Arguably the most important advantage of Bayesian data analysis is that it is possible to *marginalize* over nuisance parameters. A nuisance parameter is one that is required in order to model the process that generates the data, but is otherwise of little interest. Marginalization is the process of integrating over all possible values of the parameter and hence propagating the effects of uncertainty about its value into the final result. Often we wish to marginalize over all nuisance parameters in a model. The exact result of marginalization is the marginalized probability function $p(\Theta|D)$ of the set (list or vector) of model parameters Θ given the set of observations D

$$p(\Theta|D) = \int p(\Theta, \alpha|D) \, d\alpha \quad , \tag{1}$$

---

[1]The methods and discussion in this document have general applicability, but we will mostly present examples from astrophysics and cosmology, the fields in which we have most experience

where α is the set (list or vector) of nuisance parameters. Because the nuisance parameter set α can be very large, this integral is often extremely daunting. However, a MCMC-generated sampling of values (Θ_t, α_t) of the model and nuisance parameters from the joint distribution $p(\Theta, \alpha | D)$ automatically provides a sampling of values Θ_t from the marginalized PDF $p(\Theta | D)$.

In addition to the problem of marginalization, in many problems of interest the likelihood or the prior is the result of an expensive simulation or computation. In this regime, MCMC sampling is very valuable, but it is even *more* valuable if the MCMC algorithm is efficient, in the sense that it does not require many function evaluations to generate a statistically independent sample from the posterior PDF. The methods presented here are designed for efficiency.

Most uses of MCMC in the astrophysics literature are based on slight modifications to the Metropolis-Hastings (M–H) method (introduced below in Section 2). Each step in a M–H chain is proposed using a compact proposal distribution centered on the current position of the chain (normally a multivariate Gaussian or something similar). Since each term in the covariance matrix of this proposal distribution is an unspecified parameter, this method has $N[N+1]/2$ tuning parameters (where N is the dimension of the parameter space). To make matters worse, the performance of this sampler is very sensitive to these tuning parameters and there is no fool-proof method for choosing the values correctly. As a result, many heuristic methods have been developed to attempt to determine the optimal parameters in a data-driven way (for example, Gregory 2005; Dunkley *et al.* 2005; Widrow *et al.* 2008). Unfortunately, these methods all require a lengthy "burn-in" phase where shorter Markov chains are sampled and the results are used to tune the hyperparameters. This extra cost is unacceptable when the likelihood calls are computationally expensive.

The problem with traditional sampling methods can be visualized by looking at the simple but highly anisotropic density

$$p(\mathbf{x}) \propto f\left(-\frac{(x_1 - x_2)^2}{2\,\epsilon} - \frac{(x_1 + x_2)^2}{2}\right) \tag{2}$$

which would be considered difficult (in the small-ϵ regime) for standard MCMC algorithms. In principle, it is possible to tune the hyperparameters of a M–H sampler to make this sampling converge quickly, but if the dimension is large and calculating the density is computationally expensive the tuning procedure becomes intractable. Also, since the number of parameters scales as $\sim N^2$, this problem gets much worse in higher dimensions. Equation (2) can, however, be transformed into the much easier problem of sampling an isotropic density

by an *affine transformation* of the form

$$y_1 = \frac{x_1 - x_2}{\sqrt{\epsilon}}, \qquad y_2 = x_1 + x_2 \quad . \tag{3}$$

This motivates affine invariance: an algorithm that is *affine invariant* performs equally well under all linear transformations; it will therefore be insensitive to covariances among parameters.

Extending earlier work by Christen (2007), Goodman & Weare (2010, hereafter GW10) proposed an affine invariant sampling algorithm (Section 2) with only two hyperparameters to be tuned for performance. Hou *et al.* (2012) were the first group to implement this algorithm in astrophysics. The implementation presented here is an independent effort that has already proved effective in several projects (Sanders & Fabian 2013; Reis et al. 2013; Weisz et al. 2013; Cieza et al. 2013; Akeret et al. 2012; Huppenkothen et al. 2012; Monnier et al. 2012; Morton 2012; Crossfield et al. 2012; Roškar et al. 2012; Bovy et al. 2012; Brown et al. 2012; Brammer et al. 2012; Bussmann et al. 2012; Bovy et al. 2012; Lang & Hogg 2012; Bovy et al. 2012; Olofsson et al. 2012; Dorman et al. 2012). In what follows, we summarize the algorithm from GW10 and the implementation decisions made in emcee. We also describe the small changes that must be made to the algorithm to parallelize it. Finally, in the Appendices, we outline the installation, usage and troubleshooting of the package.

2. The Algorithm

A complete discussion of MCMC methods is beyond the scope of this document. Instead, the interested reader is directed to a classic reference like MacKay (2003) and we will summarize some key concepts below.

The general goal of MCMC algorithms is to draw M samples $\{\Theta_i\}$ from the posterior probability density

$$p(\Theta, \alpha | D) = \frac{1}{Z} p(\Theta, \alpha) \, p(D | \Theta, \alpha) \quad , \tag{4}$$

where the prior distribution $p(\Theta, \alpha)$ and the likelihood function $p(D | \Theta, \alpha)$ can be relatively easily (but not necessarily quickly) computed for any particular value of (Θ_i, α_i). The normalization $Z = p(D)$ is independent of Θ and α once we have chosen the form of the generative model. This means that it is possible to sample from $p(\Theta, \alpha | D)$ without computing Z — unless one would like to compare the validity of two different generative models. This is important because Z is generally very expensive to compute.

Once the samples produced by MCMC are available, the marginalized constraints on Θ can be approximated by the histogram of the samples projected into the parameter subspace

spanned by Θ. In particular, this implies that the expectation value of a function of the model parameters $f(\Theta)$ is

$$\langle f(\Theta) \rangle = \int p(\Theta|D)\, f(\Theta)\, \mathrm{d}\Theta \approx \frac{1}{M} \sum_{i=1}^{M} f(\Theta_i) \quad . \tag{5}$$

Generating the samples Θ_i is a non-trivial process unless $p(\Theta, \alpha, D)$ is a very specific analytic distribution (for example, a Gaussian). MCMC is a procedure for generating a random walk in the parameter space that, over time, draws a representative set of samples from the distribution. Each point in a Markov chain $X(t_i) = [\Theta_i, \alpha_i]$ depends only on the position of the previous step $X(t_{i-1})$.

The Metropolis-Hastings (M–H) Algorithm   The simplest and most commonly used MCMC algorithm is the M–H method (Algorithm 1; MacKay 2003; Gregory 2005; Press *et al.* 2007; Hogg, Bovy & Lang 2010). The iterative procedure is as follows: (1) given a position $X(t)$ sample a proposal position Y from the transition distribution $Q(Y; X(t))$, (2) accept this proposal with probability

$$\min\left(1, \frac{p(Y|D)}{p(X(t)|D)} \frac{Q(X(t); Y)}{Q(Y; X(t))}\right) \quad . \tag{6}$$

The transition distribution $Q(Y; X(t))$ is an easy-to-sample probability distribution for the proposal Y given a position $X(t)$. A common parameterization of $Q(Y; X(t))$ is a multivariate Gaussian distribution centered on $X(t)$ with a general covariance tensor that has been tuned for performance. It is worth emphasizing that if this step is accepted $X(t+1) = Y$; Otherwise, the new position is set to the previous one $X(t+1) = X(t)$ (in other words, the position $X(t)$ is *repeated in the chain*).

The M–H algorithm converges (as $t \rightarrow \infty$) to a stationary set of samples from the distribution but there are many algorithms with faster convergence and varying levels of implementation difficulty. Faster convergence is preferred because of the reduction of computational cost due to the smaller number of likelihood computations necessary to obtain the equivalent level of accuracy. The inverse convergence rate can be measured by the autocorrelation function and more specifically, the integrated autocorrelation time (see Section 3). This quantity is an estimate of the number of steps needed in the chain in order to draw independent samples from the target density. A more efficient chain has a shorter autocorrelation time.

The stretch move   GW10 proposed an affine-invariant ensemble sampling algorithm informally called the "stretch move." This algorithm significantly outperforms standard M–H

Algorithm 1 The procedure for a single Metropolis-Hastings MCMC step.

1: Draw a proposal $Y \sim Q(Y; X(t))$
2: $q \leftarrow [p(Y) \, Q(X(t); Y)] / [p(X(t)) \, Q(Y; X(t))]$ // *This line is generally expensive*
3: $r \leftarrow R \sim [0, 1]$
4: **if** $r \leq q$ **then**
5:     $X(t+1) \leftarrow Y$
6: **else**
7:     $X(t+1) \leftarrow X(t)$
8: **end if**

---

methods producing independent samples with a much shorter autocorrelation time (see Section 3 for a discussion of the autocorrelation time). For completeness and for clarity of notation, we summarize the algorithm here and refer the interested reader to the original paper for more details. This method involves simultaneously evolving an ensemble of K *walkers* $S = \{X_k\}$ where the proposal distribution for one walker k is based on the current positions of the $K - 1$ walkers in the *complementary ensemble* $S_{[k]} = \{X_j, \forall j \neq k\}$. Here, "position" refers to a vector in the N-dimensional, real-valued parameter space.

To update the position of a walker at position X_k, a walker X_j is drawn randomly from the remaining walkers $S_{[k]}$ and a new position is proposed:

$$X_k(t) \rightarrow Y = X_j + Z\left[X_k(t) - X_j\right] \tag{7}$$

where Z is a random variable drawn from a distribution $g(Z = z)$. It is clear that if g satisfies

$$g(z^{-1}) = z \, g(z), \tag{8}$$

the proposal of Equation (7) is symmetric. In this case, the chain will satisfy detailed balance if the proposal is accepted with probability

$$q = \min\left(1, \ Z^{N-1} \frac{p(Y)}{p(X_k(t))}\right) \quad, \tag{9}$$

where N is the dimension of the parameter space. This procedure is then repeated for each walker in the ensemble *in series* following the procedure shown in Algorithm 2.

GW10 advocate a particular form of $g(z)$, namely

$$g(z) \propto \begin{cases} \dfrac{1}{\sqrt{z}} & \text{if } z \in \left[\dfrac{1}{a}, a\right], \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where a is an adjustable scale parameter that GW10 set to 2.

Algorithm 2 A single stretch move update step from GW10

1: **for** $k = 1, \ldots, K$ **do**
2:      Draw a walker X_j at random from the complementary ensemble $S_{[k]}(t)$
3:      $z \leftarrow Z \sim g(z)$, Equation (10)
4:      $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
5:      $q \leftarrow z^{N-1}\,p(Y)/p(X_k(t))$ // *This line is generally expensive*
6:      $r \leftarrow R \sim [0, 1]$
7:      **if** $r \leq q$, Equation (9) **then**
8:          $X_k(t+1) \leftarrow Y$
9:      **else**
10:        $X_k(t+1) \leftarrow X_k(t)$
11:      **end if**
12: **end for**

The parallel stretch move It is tempting to parallelize the stretch move algorithm by simultaneously advancing each walker based on the state of the ensemble instead of evolving the walkers in series. Unfortunately, this subtly violates detailed balance. Instead, we must split the full ensemble into two subsets ($S^{(0)} = \{X_k, \forall k = 1, \ldots, K/2\}$ and $S^{(1)} = \{X_k, \forall k = K/2 + 1, \ldots, K\}$) and simultaneously update all the walkers in $S^{(0)}$ — using the stretch move procedure from Algorithm 2 — based *only* on the positions of the walkers in the other set ($S^{(1)}$). Then, using the new positions $S^{(0)}$, we can update $S^{(1)}$. In this case, the outcome is a valid step for all of the walkers. The pseudocode for this procedure is shown in Algorithm 3. This code is similar to Algorithm 2 but now the computationally expensive inner loop (starting at line 2 in Algorithm 3) can be run in parallel.

The performance of this method — quantified by the autocorrelation time — is comparable to the serial stretch move algorithm but the fact that one can now take advantage of generic parallelization makes it extremely powerful.

3. Tests

Judging the convergence and performance of an algorithm is a non-trival problem and there is a huge associated literature (see, for example, Cowles & Carlin 1996, for a review). In astrophysics, spectral methods have been used extensively (for example Dunkley *et al.* 2005). Below, we advocate for one such method: the autocorrelation time. The autocorrelation time is especially applicable because it is an affine invariant measure of the performance.

First, however, we should take note of another extremely important measurement: the

---

Algorithm 3 The parallel stretch move update step

1: **for** $i \in \{0, 1\}$ **do**
2:   **for** $k = 1, \ldots, K/2$ **do**
3:     // *This loop can now be done in parallel for all k*
4:     Draw a walker X_j at random from the complementary ensemble $S^{(\sim i)}(t)$
5:     $X_k \leftarrow S_k^{(i)}$
6:     $z \leftarrow Z \sim g(z)$, Equation (10)
7:     $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
8:     $q \leftarrow z^{n-1}\,p(Y)/p(X_k(t))$
9:     $r \leftarrow R \sim [0, 1]$
10:     **if** $r \leq q$, Equation (9) **then**
11:       $X_k(t + \frac{1}{2}) \leftarrow Y$
12:     **else**
13:       $X_k(t + \frac{1}{2}) \leftarrow X_k(t)$
14:     **end if**
15:   **end for**
16:   $t \leftarrow t + \frac{1}{2}$
17: **end for**

---

acceptance fraction a_f. This is the fraction of proposed steps that are accepted. There appears to be no agreement on the optimal acceptance rate but it is clear that both extrema are unacceptable. If $a_f \sim 0$, then nearly all proposed steps are rejected, so the chain will have very few independent samples and the sampling will not be representative of the target density. Conversely, if $a_f \sim 1$ then nearly all steps are accepted and the chain is performing a random walk with no regard for the target density so this will also not produce representative samples. As a rule of thumb, the acceptance fraction should be between 0.2 and 0.5 (for example, Gelman, Roberts, & Gilks 1996). For the M–H algorithm, these effects can generally be counterbalanced by decreasing (or increasing, respectively) the eigenvalues of the proposal distribution covariance. For the stretch move, the parameter a effectively controls the step size so it can be used to similar effect. In our tests, it has never been necessary to use a value of a other than 2, but we make no guarantee that this is the optimal value.

Autocorrelation time The autocorrelation time is a direct measure of the number of evaluations of the posterior PDF required to produce independent samples of the target density. GW10 show that the stretch-move algorithm has a significantly shorter autocorrelation time on several non-trivial densities. This means that fewer PDF computations

are required—compared to a M–H sampler—to produce the same number of independent samples.

The autocovariance function of a time series $X(t)$ is

$$C_f(T) = \lim_{t \to \infty} \mathrm{cov}\left[f\left(X(t+T)\right), f\left(X(t)\right)\right]. \tag{11}$$

This measures the covariances between samples at a time lag T. The value of T where $C_f(T) \to 0$ measures the number of samples that must be taken in order to ensure independence. In particular, the relevant measure of sampler efficiency is the integrated autocorrelation time

$$\tau_f = \sum_{T=-\infty}^{\infty} \frac{C_f(T)}{C_f(0)} = 1 + 2 \sum_{T=1}^{\infty} \frac{C_f(T)}{C_f(0)}. \tag{12}$$

In practice, one can estimate $C_f(T)$ for a Markov chain of M samples as

$$C_f(T) \approx \frac{1}{M-T} \sum_{m=1}^{M-T} \left[f(X(T+m)) - \langle f \rangle\right] \left[f(X(m)) - \langle f \rangle\right]. \tag{13}$$

We advocate for the autocorrelation time as a measure of sampler performance for two main reasons. First, it measures a quantity that *we are actually interested in* when sampling in practice. The longer the autocorrelation time, the more samples that we must generate to produce a representative sampling of the target density. Second, the autocorrelation time is affine invariant. Therefore, it is reasonable to measure the performance and diagnose the convergence of the sampler on densities with different levels of anisotropy.

emcee can optionally calculate the autocorrelation time using the Python module acor[2] to estimate the autocorrelation time. This module is a direct port of the original algorithm (described by GW10) and implemented by those authors in C++.[3]

4. Discussion & Tips

The goal of this project has been to make a sampler that is a useful tool for a large class of data analysis problems—a "hammer" if you will. If development of statistical and data-analysis understanding is the key goal, a user who is new to MCMC benefits enormously by writing her or his own Metropolis–Hastings code (Algorithm 1) from scratch before downloading emcee. For typical problems, the emcee package will perform better than

---

[2]http://github.com/dfm/acor

[3]http://www.math.nyu.edu/faculty/goodman/software/acor

any home-built M–H code (for all the reasons given above), but the intuitions developed by writing and tuning a self-built MCMC code cannot be replaced by reading this document and running this pre-built package. That said, once those intuitions are developed, it makes sense to switch to emcee or a similarly well engineered piece of code for performance on large problems.

Ensemble samplers like emcee require some thought for initialization. One general approach is to start the walkers at a sampling of the prior or spread out over a reasonable range in parameter space. Another general approach is to start the walkers in a very tight N-dimensional ball in parameter space around one point that is expected to be close to the maximum probability point. The first is more objective but, in practice, we find that the latter is much more effective if there is any chance of walkers getting stuck in low probability modes of a multi-modal probability landscape. The walkers initialized in the small ball will expand out to fill the relevant parts of parameter space in just a few autocorrelation times. A third approach would be to start from a sampling of the prior, and go through a "burn-in" phase in which the prior is transformed continuously into the posterior by increasing the "temperature." Discussion of this kind of annealing is beyond the scope of this document.

It is our present view that autocorrelation time is the best indicator of MCMC performance (the shorter the better), but there are several proxies. The easiest and simplest indicator that things are going well is the acceptance fraction; it should be in the 0.2 to 0.5 range (there are theorems about this for specific problems; for example Gelman, Roberts, & Gilks 1996). In principle, if the acceptance fraction is too low, you can raise it by decreasing the a parameter; and if it is too high, you can reduce it by increasing the a parameter. However, in practice, we find that $a = 2$ is good in essentially all situations. That means that when using emcee *if the acceptance fraction is getting very low, something is going very wrong.* Typically a low acceptance fraction means that the posterior probability is multi-modal, with the modes separated by wide, low probability "valleys." In situations like these, the best idea (though expensive of human time) is to split the space into disjoint single-mode regions and sample each one independently, combining the independently sampled regions "properly" (also expensive, and beyond the scope of this document) at the end. In previous work, we have advocated clustering methods to remove multiple modes (Hou *et al.* 2012). These work well when the different modes have *very* different posterior probabilities.

Another proxy for short autocorrelation time is large expected or mean squared jump distance (ESJD; Pasarica & Gelman 2010). The higher the ESJD the better; if walkers move (in the mean) a large distance per chain step then the autocorrelation time will tend to be shorter. The ESJD is not an affine-invariant measure of performance, and it doesn't have a trivial interpretation in terms of independent samples, so we prefer the autocorrelation time

in principle. In practice, however, because the ESJD is a simple expectation value it can be more robustly evaluated on short chains.

With emcee you want (in general) to *run with a large number of walkers*, like hundreds. In principle, there is no reason not to go large when it comes to walker number, until you hit performance issues. Although each step takes twice as much compute time if you double the number of walkers, it also returns to you twice as many independent samples per autocorrelation time. So go large. In particular, we have found that—in almost all cases of low acceptance fraction—increasing the number of walkers improves the acceptance fraction. The one disadvantage of having large numbers of walkers is that the burn-in phase (from initial conditions to reasonable sampling) can be slow; burn-in time is a few autocorrelation times; the total run time for burn-in scales with the number of walkers. These considerations, all taken together, suggest using the smallest number of walkers for which the acceptance fraction during burn-in is good, or the number of samples you want out at the end (see below), whichever is *greater*. A more ambitious project would be to increase the number of walkers after burn-in; this requires thought beyond the scope of this document; it can be accomplished by burning in a set of small ensembles and then merging them into a big ensemble for the final run.

One mistake many users of MCMC methods make is to take *too many* samples! If all you want your MCMC to do is produce one- or two-dimensional error bars on two or three parameters, then you only need dozens of independent samples. With ensemble sampling, you get this from a *single snapshot* or single timestep, provided that you are using dozens of walkers (and we would recommend that you use hundreds in most applications). The key point is that *you should run the sampler for a few (say 10) autocorrelation times.* Once you have run that long, no matter how you initialized the walkers, the set of walkers you obtain at the end should be an independent set of samples from the distribution, of which you rarely need many.

Another common mistake, of course, is to run the sampler for *too few* steps. You can identify that you haven't run for enough steps in a couple of ways: If you plot the parameter values in the ensemble as a function of step number, you will see large-scale variations over the full run length if you have gone less than an autocorrelation time. You will also see that if you try to measure the autocorrelation time (with, say, acor), it will give you a time that is always a significant fraction of your run time; it is only when the correlation time is much shorter (say by a factor of 10) than your run time that you are sure to have run long enough. The danger of both of these methods—an unavoidable danger at present—is that you can have a huge dynamic range in contributions to the autocorrelation time; you might think it is 30 when in fact it is 30 000, but you don't "see" the 30 000 in a run that

is only 300 steps long. There is not much you can do about this; it is generic when the posterior is multi-modal: The autocorrelation time within each mode can be short but the mode–mode migration time can be long. See above on low acceptance ratio; in general when your acceptance ratio gets low your autocorrelation time is very, very long.

There are some cases where emcee won't perform as well as some more specialized sampling techniques. In particular, when the target density is multi-modal, walkers can become "stuck" in different modes. When this happens, the vector between walkers is no longer a good proposal direction. In these cases, the acceptance fraction and autocorrelation time can deteriorate quickly. While this is a fairly general problem, we find that in many applications the effect isn't actually very important. That being said, there are some problems where higher-end machinery (such as Brewer et al. 2011, Hou et al. forthcoming) is necessary (see, for example, Brewer et al. 2012; van Haasteren et al. 2013).

Another limitation to the stretch move and moves like it is that they implicitly assume that the parameters can be assembled into a vector-like object on which linear operations can be performed. This is not (trivially) true for parameters that have non-trivial constraints, like parameters that must be integer-valued or equivalent, or parameters that are subject to deterministic non-linear constraints. Sometimes these issues can be avoided by reparameterization, but in some cases, samplers like emcee will not be useful, or might require clever or interesting improvements. The emcee package is open-source software; please push us changes!

REFERENCES

Akeret, J., Seehars, S., Amara, A., Refregier, A., & Csillaghy, A. 2012, arXiv:1212.1721

Bovy, J., Rix, H.-W., Liu, C., et al. 2012, ApJ, 753, 148

Bovy, J., Rix, H.-W., Hogg, D. W., et al. 2012, ApJ, 755, 115

Bovy, J., Allende Prieto, C., Beers, T. C., et al. 2012, ApJ, 759, 131

Brammer, G. B., Sánchez-Janssen, R., Labbé, I., et al. 2012, ApJ, 758, L17

Brewer B. J., Pártay L. B., Csányi G., 2011, Statistics and Computing, 21, 4, 649-656,
    arXiv:0912.2380

Brewer, B. J., Foreman-Mackey, D., & Hogg, D. W. 2012, arXiv:1211.5805

Brown, J. M., Rosenfeld, K. A., Andrews, S. M., Wilner, D. J., & van Dishoeck, E. F.
    2012, ApJ, 758, L30

Bussmann, R. S., Gurwell, M. A., Fu, H., et al. 2012, ApJ, 756, 134

Christen, J., *A general purpose scale-independent MCMC algorithm*, technical report
    I-07-16, CIMAT, Guanajuato, 2007

Cieza, L. A., Olofsson, J., Harvey, P. M., et al. 2013, ApJ, 762, 100

Cowles, M. K. & Carlin, B. P., 1996, Journal of the American Statistical Association, 91,
    883

Crossfield, I. J. M., Barman, T., Hansen, B. M. S., Tanaka, I., & Kodama, T. 2012, ApJ,
    760, 140

Dorman, C. E., Guhathakurta, P., Fardal, M. A., et al. 2012, ApJ, 752, 147

Dunkley, J., Bucher, M., Ferreira, P. G., Moodley, K., & Skordis, C., 2005, MNRAS, 356,
    925–936

Gelman, A., Roberts, G. O., & Gilks, W. R., in *Bayesian Statistics 5*, ed. J. Bernardo et
    al., Oxford University Press, 599–607

Goodman, J. & Weare, J., 2010, Comm. App. Math. Comp. Sci., 5, 65

Gregory, P. C., *Bayesian Logical Data Analysis for the Physical Sciences*, Cambridge
    University Press, 2005

Hogg, D. W., Bovy, J., & Lang, D., 2010, arXiv:1008.4686 [astro-ph.IM]

Hou, F., Goodman, J., Hogg, D. W., Weare, J., & Schwab, C. 2012, ApJ, 745, 198

Huppenkothen, D., Watts, A. L., Uttley, P., et al. 2012, arXiv:1212.1011

Lang, D., & Hogg, D. W. 2012, AJ, 144, 46

MacKay, D., *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003

Monnier, J. D., Che, X., Zhao, M., et al. 2012, ApJ, 761, L3

Morton, T. D. 2012, ApJ, 761, 6

Olofsson, J., Juhász, A., Henning, T., et al. 2012, A&A, 542, A90

Pasarica, C. & Gelman, A., 2010, Statistica Sinica, 20, 343–364

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 2007

Reis, R. C., Miller, J. M., Reynolds, M. T., et al. 2013, ApJ, 763, 48

Roškar, R., Debattista, V. P., & Loebman, S. R. 2012, arXiv:1211.1982

Sanders, J. S., & Fabian, A. C. 2013, MNRAS, 453

van Haasteren, R., Mingarelli, C. M. F., Vecchio, A., & Lassus, A. 2013, arXiv:1301.6673

Weisz, D. R., Fouesneau, M., Hogg, D. W., et al. 2013, ApJ, 762, 123

Widrow, L. M. and Pym, B. and Dubinski, J., 2008, ApJ, 679, 1239

A. Installation

The easiest way to install emcee is using pip[4]. Running the command

```
% pip install emcee
```

at the command line of a UNIX-based system will install the package in your Python path. If you would like to install for all users, you might need to run the above command with superuser permissions. In order to use emcee, you must also have numpy[5] installed (this can also be achieved using pip on most systems). emcee has been tested with Python 2.7 and numpy 1.6 but it is likely to work with earlier versions of both of these as well.

An alternative installation method is to download the source code from http://dan.iel.fm/emcee and run

```
% python setup.py install
```

in the unzipped directory. Make sure that you have numpy installed in this case as well.

B. Issues & Contributions

The development of emcee is being coordinated on GitHub at http://github.com/dfm/emcee and contributions are welcome. If you encounter any problems with the code, please report them at http://github.com/dfm/emcee/issues and consider contributing a patch.

C. Online Documentation

To learn more about how to use emcee in practice, it is best to check out the documentation on the website http://dan.iel.fm/emcee. This page includes the API documentation and many examples of possible work flows.

---

[4]http://pypi.python.org/pypi/pip

[5]http://numpy.scipy.org