

A Short Explanation of Hamiltonian Monte Carlo

Matt Simpson

February 20, 2017

1 Introduction

This document is a short high level explanation of Hamiltonian Monte Carlo (HMC) for myself. It should be useful for a Stan talk I plan to give to the Spatio-Temporal Working Group. It has been organized into an outline with short, bite sized chunks.

Useful videos:

- HMC: <https://www.youtube.com/watch?v=pHsuIaPbNbY>
- Stan: <https://www.youtube.com/watch?v=xWQpEAYI5s8>
- Everything you should have learned about MCMC : <https://www.youtube.com/watch?v=DJ0c7Bm5Djk&feature=youtu.be&t=4h40m10s>

2 Hamiltonian Monte Carlo in Theory

HMC is a specific class of auxillary variable Markov chain Monte Carlo (MCMC) algorithms. Suppose we wish to sample from the probability density $\pi(\mathbf{q})$, e.g. a Bayesian posterior density. A MCMC algorithm constructs a Markov chain with transition density $t(\mathbf{q}'|\mathbf{q})$ such that $\mathbf{q}^{(k)}$ converges in distribution to $\pi(\mathbf{q})$ as $k \rightarrow \infty$. Auxillary variable methods expand the parameter space to include some extra variable \mathbf{p} with joint density $\pi(\mathbf{q}, \mathbf{p})$, then construct a Markov chain for (\mathbf{q}, \mathbf{p}) jointly. HMC constructs \mathbf{p} in a very special way.

2.1 HMC definition

Suppose that $\pi(\mathbf{q}) > 0$ for all $\mathbf{q} \in \mathbb{R}^n$, and write $\pi(\mathbf{q}) = e^{-V(\mathbf{q})}$. Then construct the conditional auxillary variable density $\pi(\mathbf{p}|\mathbf{q}) = e^{-T(\mathbf{q}, \mathbf{p})}$ with $\pi(\mathbf{p}|\mathbf{q}) > 0$ for all $\mathbf{q}, \mathbf{p} \in \mathbb{R}^n$. Then the joint density is $\pi(\mathbf{q}, \mathbf{p}) = e^{-H(\mathbf{q}, \mathbf{p})}$ where $H(\mathbf{q}, \mathbf{p}) = V(\mathbf{q}) + T(\mathbf{q}, \mathbf{p})$.

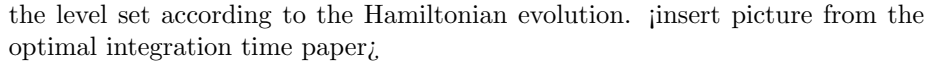
The function $H(\mathbf{q}, \mathbf{p})$ is called a *Hamiltonian* with *position* vector \mathbf{q} and *momentum* vector \mathbf{p} . The function $T(\mathbf{q}, \mathbf{p})$ is called the *kinetic energy* while $V(\mathbf{q})$ is called the *potential energy*. The Hamiltonian corresponds to the total energy of the system. The Hamiltonian evolution in time of the system of

positions and momenta that conserves total energy is defined by $2n$ ordinary differential equations given by Hamilton's equations:

$$\frac{d\mathbf{p}}{dt} = -\frac{\partial H}{\partial \mathbf{q}}; \quad \frac{d\mathbf{q}}{dt} = +\frac{\partial H}{\partial \mathbf{p}}.$$

The theoretical HMC transition density $t(\mathbf{q}^{(k+1)}, \mathbf{p}^{(k+1)} | \mathbf{q}^{(k)}, \mathbf{p}^{(k)})$ can then be sampled from as follows:

1. Sample $\mathbf{p}' \sim \pi(\mathbf{p} | \mathbf{q}^{(k)})$.
2. Run the Hamiltonian evolution forward in time from $(\mathbf{q}^{(k)}, \mathbf{p}')$ for some amount of time to obtain $(\mathbf{q}^{(k+1)}, \mathbf{p}^{(k+1)})$.

Note that the theoretical HMC algorithm accepts 100% of the time. Graphically, you can picture this algorithm as a two step process: 1) a random walk step on the level sets of $\pi(\mathbf{p}, \mathbf{q})$, then 2) a deterministic step that moves around the level set according to the Hamiltonian evolution. 

3 Hamiltonian Monte Carlo in Practice

In practice, the theoretical algorithm's performance depends on two things: the kinetic energy $T(\mathbf{q}, \mathbf{p})$, and how far forward in time the Hamiltonian evolution is computed. A good kinetic energy allows the Markov chain to move around level sets more quickly and efficiently (think a well-tuned random walk). When the kinetic energy is well chosen, the performance of the algorithm depends on how long the Hamiltonian evolution is run. Running the Hamiltonian evolution for too short of a time results in highly autocorrelated Markov chains. Running it longer leads to much better chains, but in practice this comes at a higher computational cost so there is a trade-off.

In fact, this is the third component of HMC in practice: the Hamiltonian evolution is almost never analytically tractable, so it must be approximated using numerical integration methods. This, in turn, requires a Metropolis correction, along with some constraints on the kinetic energy and tweaks to the basic algorithm in order to ensure that the HMC-Metropolis algorithm has the right stationary distribution.

In practice, the HMC algorithm as actually implemented is as follows:

1. Sample $\mathbf{p}' \sim \pi(\mathbf{p} | \mathbf{q}^{(k)})$.
2. Approximate running the Hamiltonian evolution forward in time from $(\mathbf{q}^{(k)}, \mathbf{p}')$ for some amount of time to obtain $(\mathbf{q}'', \mathbf{p}'')$.
3. Apply necessary tweaks to ensure the algorithm has the right stationary density to obtain proposal $(\mathbf{q}^{(prop)}, \mathbf{p}^{(prop)})$ (commonly set $\mathbf{p}^{(prop)} = -\mathbf{p}''$ & $\mathbf{q}^{(prop)} = \mathbf{q}''$, but it depends on the approximation & kinetic energy).

4. Compute the Metropolis ratio and accept $\mathbf{q}^{(k+1)} = \mathbf{q}^{(prop)}$ or reject and set $\mathbf{q}^{(k+1)} = \mathbf{q}^{(k)}$ as usual.

A good HMC algorithm will accept a high percentage of the time (80% or more) and the traceplots will look nearly iid. The computational cost per iteration is often higher than a good Gibbs sampler, but the often near-perfect mixing typically more than makes up for that. To construct an HMC algorithm for your model, you need the following:

1. No discrete parameters. If you have discrete parameters, integrate them out.
2. No constrained parameters. If you have constrained parameters, transform to the unconstrained space.
3. A function for computing the log density of the unconstrained parameters.
4. A function for computing the gradient of the log density.
5. Functions for computing higher order derivatives for some exotic HMC variants.
6. Choose a good kinetic energy.
7. Choose a good Hamiltonian approximation.
8. Choose a good integration time.

Many of these are hard. Stan allows you to do (3) straightforwardly with a modeling language similar to BUGS or JAGS. It handles (2) by allowing you to specify constraints on parameters, then it transforms for you. Unfortunately, you still must do (1) on your own, but for many problems it is not too hard. The big innovation in Stan is an autodifferentiation package that numerically computes the gradient for you, accomplishing (4), and the developers are currently working on (5). Stan also has an adaptive phase that chooses the kinetic energy, Hamiltonian approximation, and integration time in concert, depending on your target density, making (6)–(8) straightforward. Stan exposes some knobs of this adaptation process and the developers have worked on useful diagnostics for evaluating how well an HMC algorithm is working, so you can play with these things to improve the efficiency of your sampler. For many common problems the default settings work great.

4 Working with Stan

Because of the nature of HMC and the specifics of how Stan is implemented, there are various things you can do in order to make Stan run faster or construct better HMC algorithms. All of these are documented in better detail in the manual, but I'll briefly mention all of them. See also this brief guide to Stan's warnings <http://mc-stan.org/misc/warnings.html>.

4.1 Center and Scale Your Data

This is the number one piece of advice I have for implementing things in Stan. Because of how HMC works, Stan can be dramatically faster when you center and scale your response and any covariates. Everything else in this section I would say is more for advanced users who are trying to fit more complex models in Stan. But even for simple models, centering and scaling your data can dramatically speed up Stan.

For example, consider the simple regression model

$$\mathbf{y} \sim \text{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2\mathbf{I})$$

If the scales of the columns of \mathbf{X} are drastically different, this can make Stan very slow. But consider the equivalent model

$$\tilde{\mathbf{y}} \sim \text{N}(\tilde{\mathbf{X}}\tilde{\boldsymbol{\beta}}, \sigma^2\mathbf{I})$$

where $\tilde{\mathbf{y}} = (\mathbf{y} - \bar{y})/\text{SD}(\mathbf{y})$, and for the j th column of $\tilde{\mathbf{X}}$, $\tilde{\mathbf{x}}_j = (\mathbf{x}_j - \bar{x}_j)/\text{SD}(\mathbf{x}_j)$, $j = 2, 3, \dots, p$ where \mathbf{X} has p columns, and the first column corresponds to the intercept term. This model will give the same inferences about each $\beta_j = \tilde{\beta}_j \text{SD}(\mathbf{y})/\text{SD}(\mathbf{x}_j)$, $j = 2, 3, \dots, p$, and similarly for β_1 , which is a more complex function of the means and sds of \mathbf{y} , the \mathbf{x}_j s, and of $\tilde{\beta}_1$.

4.2 Center and Scale Your Parameters

For similar reasons it can be a good idea to center and scale your parameters, especially parameters that have normal conditional distributions. For example, consider the following model written in terms of the *centered parameterization*:

$$y \sim \text{N}(\lambda, \tau^2), \quad \lambda \sim \text{N}(\mu, \sigma^2).$$

An alternative is the *noncentered parameterization*:

$$y \sim \text{N}(\mu + \sigma\tilde{\lambda}, \tau^2), \quad \tilde{\lambda} \sim \text{N}(0, 1).$$

These models are equivalent with $\lambda = \mu + \sigma\tilde{\lambda}$, but often the noncentered parameterization will result in faster HMC algorithms. And sometimes the opposite is true. The basic intuition both here is the same as for Gibbs samplers: the more information you have from the data about the parameter, the more likely the centered parameterization is optimal, but the less information you have the more likely the noncentered parameterization is optimal.

4.3 Other Reparameterizations

In general, Stan and HMC more generally can have trouble with certain complex geometries in the target distribution. To fully understand this stuff you need differential geometry, but there are some more rules of thumb. For example, HMC can have trouble with fat tails, so reparameterization can help there. Some examples (not just fat tails):

1. Use the gamma-normal mixture representation of the student's t distribution.
2. Use the normal distribution on logs instead of the lognormal distribution.
3. Use the Cholesky factor of a covariance matrix instead of directly using the covariance matrix.

4.4 Vectorize and Looping Concerns

Stan is written in C++\verb and loops are very fast. Stan has also been vectorized so that the following two model definitions are equivalent:

```
data{
  vector[N] y;
}
parameters{
  real mu;
  real<lower = 0> sigma;
}
model{
  y ~ normal(mu, sigma);
  mu ~ normal(0.0, 10.0);
  sigma ~ normal(0.0, 10.0); // half-normal prior w/ constraint above
}

and

data{
  vector[N] y;
}
parameters{
  real mu;
  real<lower = 0> sigma;
}
model{
  for(n in 1:N){
    y[n] ~ normal(mu, sigma);
  }
  mu ~ normal(0.0, 10.0);
  sigma ~ normal(0.0, 10.0); // half-normal prior w/ constraint above
}
```

Stan does not care which of the two ways you write this model, so you might think that using C++'s blazing fast for loops is the way to go. However, because of how the autodifferentiation package works, the opposite is true. Using for loops means computing the log density will be faster, but vectorizing means computing the gradient of the log density will be faster, and this latter computation is

usually the largest computational bottleneck. So vectorize whenever possible if you are fitting a complex model.

If you have to use loops, there are some tricks you can use. In Stan, matrices are stored in column-major order, so it's faster to loop over columns first (i.e. more slowly). For example

```
for(c in 1:ncol){
  for(r in 1:nrow){
    x[r, c] = // some stuff
  }
}
```

Also some things are faster when working with arrays than with vectors or matrices and vice versa. Read the manual!

4.5 Sufficient Statistics

When possible, parameterize distributions in terms of sufficient statistics instead of the underlying data or parameters. This should speed things up. For example instead of

$$y_i \stackrel{iid}{\sim} \text{Ber}(p) \text{ for } i = 1, 2, \dots, N$$

use

$$z = \sum_{i=1}^N y_i \sim \text{Bin}(N, p).$$

4.6 Exploit Conjugacy

If $y \sim [y|\phi]$ with $\phi \sim [\phi]$ but $[\phi|y]$ is known, use $\phi \sim [\phi|y]$ instead. For example instead of

```
y ~ bin(n, theta);
theta ~ beta(a, b);
```

use

```
theta ~ beta(a + sum(y), b + n - sum(y));
```