The Hamiltonian Monte Carlo: an Excellent Algorithm for Simulating Bayesian Posterior Distributions

Jonathan Brinkerhoff, Minghui Xu, Nancy Blacklock
Bayesian Statistics
California State University, Fullerton

Markov chain Monte Carlo methods for sampling from probability distributions are a useful tool in Bayesian analysis. The Hamiltonian Monte Carlo algorithm combines Hamiltonian Dynamics and MCMC methods to move quickly and with computational efficiency towards the target distribution.

The Hamiltonian, once known as the Hybrid Monte Carlo, was created as a method for simulating lattice models in the study of subatomic particles. (Duane et al.) The word "hybrid" refers to the algorithm's integration of Molecular Dynamics and Langevin Monte Carlo methods. Molcular Dynamics is a simulation method which is used for "analyzing the physical movements of atoms and molecules. ("Molecular Dynamics") The algorithm was repurposed for continuous statistical models by Radford M. Neal in 1996 in the "Monte Carlo Implementation". The repurposed Hamiltonian Monte Carlo, or HMC, has improved efficiency compared to the Metropolis-Hastings algorithm, but regular implementation of the Hamiltonian had to wait for advancements in computational power. The "HMC is more computationally costly than Metropolis or Gibbs sampling. But its proposals are also more efficient" (McElreath) Generating individual samples using the HMC requires more computational resources but the Hamiltonian can describe posterior distributions with thousands of parameters. Attempting the same using a random walk algorithm is ill-advised as the "Earth would be swallowed by the Sun before your chain produces a reliable approximation of the posterior." (McElreath) Free and open-source programs, like the statistical software Stan, which was released in 2012, make the algorithm even more accessible by performing the mathematical calculations that power the algorithm.

Multi-parameter models of posterior distributions can be difficult to sample from using the Metropolis-Hastings or Gibbs algorithms; both are random walk MCMC methods which can be slow to come to convergence in some high parameter spaces. Models with multiple parameters tend to have highly correlated, high probability regions which are structurally difficult to explore as "both Metropolis and Gibbs make too many dumb proposals of where to go next. So they get stuck." (McElreath) Neither Gibbs nor the Metroplois-Hastings algorithms handle correlation between parameters as well as the Hamiltonian. The Molecular Dynamics components of the Hamiltonian algorithm utilize the structures caused by correlation to more efficiently search the parameter space.

An interesting issue raised in "Statistical Rethinking" is the idea of concentration of measure. McElreath asks us to consider the dimensional distribution of the mode of a distribution. In 2D, McElreath describes the mode of a Gaussian distribution as the top of the hill. In 3D we could imagine the hill filled with dirt, the total volume of dirt increases as we move downward and away from the peak. The dirt represents the probability volume which increases as we move downward through the hill of the distribution. This "means that the combination of parameter values that maximizes posterior probability, the mode, is not actually a region of parameter values that are highly plausible." (McElreath) The structure of the HMC algorithm focuses on the entire posterior at one time, which will help us from getting stuck in narrow areas of the parameter space.

The HMC algorithm uses a physics simulation based on the Hamiltonian function which describes the total energy for a closed system of particles. Total energy in a closed system is defined as the sum of its kinetic and potential energies. The Hamiltonian function, $H(\boldsymbol{q}, \boldsymbol{p})$, describes a system's total energy in terms of each particle's position and momentum.

The Hamiltonian denoted here by, $H(q_k, p_k)$, is a function of the independent variables of momentum, p_k, and position, q_k. Taking the differential and performing some algebra gives us Hamilton's canonical equations describing the motion in a system:

$$\dot{q}_k = \frac{\partial H}{\partial p_k}$$

$$\dot{p}_k = -\frac{\partial H}{\partial q_k}$$

for $k = 1, ..., d$ where the vectors q and p are both d-dimensional. "The description of motion by these equations is termed Hamiltonian dynamics." (Thornton and Marion)

Instead of random walk behavior, the Hamiltonian uses Hamiltonian dynamics in its search for the target distribution. In this section, we will use the symbol, $H(q,p)$ to represent the joint target distribution of our variables of interest, q, and a new, randomly selected, momentum variable, p. Using Hamiltonian dynamics we can view $H(q,p)$ as the total energy of a conservative system and therefore the sum of the kinetic and potential energy of that system. Here we will represent the potential energy as a function of our variables of interest, $U(q)$, and the kinetic energy, $K(p)$, as a function of the introduced momentum.

$$H(q,p) = U(q) + K(p)$$

One option for $U(q)$ is the negative log probability density of q but other functions may be used. The kinetic energy function, $K(p)$ is defined as:

$$K(p) = \frac{1}{2} p^T M^{-1} p$$

If we assume that M is a diagonal covariance with elements $m_1, ..., m_d$, we can write the kinetic energy as:

$$K(p) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i}$$

We can then describe the joint distribution as Hamilton's equations:

$$\frac{dq_i}{dt} = [M^{-1}p]_i$$
$$\frac{dp_i}{dt} = -\frac{\partial U}{\partial q_i}$$

This allows us to use the leapfrog method to approximate the solution to the system of differential equations:

$$p_i(t + \epsilon/2) = p_i(t) = (\epsilon/2) \frac{\partial U}{\partial q_i}(q(t))$$

$$q_i(t + \epsilon) = q_i(t) + \epsilon \frac{p_i(t + \epsilon/2}{m_i}$$

$$p_i(t + \epsilon) = p_i(t + \epsilon/2) = (\epsilon/2) \frac{\partial U}{\partial q_i}(q(t + \epsilon))$$

Where ϵ is the step size for the leapfrog steps. The leapfrog method is a modification of Euler's method of approximation, the "leapfrog performs half steps for momentum at the very beginning and very end of the trajectory, and the time labels of the momentum values computed are shifted by $\frac{\epsilon}{2}$". (Brooks et al.) Leapfrog integration is the same as updating the momentum and position "at different interleaved time points staggered in such a way that they "leapfrog" over each other". ("Leapfrog Integration")

The leapfrog steps help to propel us towards our target but they are not the first step of the HMC algorithm. The iterative steps of the HMC algorithm begin with the random and invented momentum variable.

In this section we will adopt the notation from "Bayesian Data Analysis" for the variables of the Hamiltonian algorithm to help with readability.

Our target joint posterior distribution is $p(\theta, \phi|y) = p(\phi)p(\theta|y)$ where ϕ is our momentum variable and θ our parameter of interest.

The Hamiltonian Algorithm: A Single Iteration in Search of θ

1. Randomly select and update the momentum, usually using a normal distribution with mean zero and covariance equal to the mass matrix, M, where the mass matrix is defined as the covariance of the momentum distribution, $p(\phi)$.

$$\phi \sim N(0, M)$$

2. The second step of the HMC algorithm is where we see the leapfrog portion of the algorithm updating both the parameter of interest, θ, and the momentum, ϕ, "each in relation to the other". (Gelman et al.) A half step for the momentum is taken using the gradient of the (in this example) log posterior density of the parameter of interest.

$$\phi \leftarrow \phi + \frac{1}{2}\epsilon\frac{d\ log\ p(\theta|y)}{d\theta}$$

Then the momentum is used to update the parameter of interest.

$$\theta \leftarrow \theta + \epsilon M^{-1}\phi$$

Finally the gradient of the parameter of interest is used to half update the momentum again.

$$\phi \leftarrow \phi + \frac{1}{2}\epsilon\frac{d\ log\ p(\theta|y)}{d\theta}$$

These steps within the leapfrog are repeated for L leapfrog steps "each scaled by a factor ϵ. M, L, and ϵ are tunable portions of the algorithm.

3. The new values of θ and ϕ are compared to the values from the start of the leapfrog iterations using a ratio of posterior distributions in the accept-reject step:

$$r = \frac{p(\theta^*|y)p(\phi^*)}{p(\theta^{t-1}|y)p(\phi^{t-1})}$$

4. The parameter of interest is set using the uniform distribution:

$$\begin{cases} \theta^* & \text{with probability } min(r, 1) \\ \theta^{t-1} & \text{otherwise} \end{cases}$$

These steps are repeated until convergence and number of desired samples are obtained.

One of the downsides to the Hamiltonian Monte Carlo algorithm is the amount of tuning that is required for each application. The recommendations from "Bayeisan Data Analysis" are to begin with the scale parameters for the momentum variables and to set them using "some crude estimate of the scale of the target distribution". (Gelman et al.) Next we use the product of the leapfrog steps, L, and the scale factor, ϵ, equal to 1. Doing this "roughly calibrates the HMC algorithm to the 'radius' of the target distribution" to ensure the algorithm traverses the entire target distribution. (Gelman et al.) Another option is to randomly vary the leapfrog variables from iteration to iteration in order to encourage variable movement lengths.

Autocorrelation can be very low with the HMC however it is not guaranteed. The size of leapfrog steps and their step size have a significant impact of the amount of autocorrelation. It is possible to hinder the algorithm with such poor choices of leapfrog variables that the algorithm becomes trapped within a portion of parameter space. This type of problem is called "the U-turn problem". (McElreath) If you are coding a Hamiltonian algorithm you may try the recommendations for the leapfrog parameters discussed above or, programs like Stan will select leapfrog variables based on a warmup phase performed at the start of every run. Stan specifically also uses an additional algorithm "which uses the shape of the posterior to infer when the path is turning around ... to adaptively set the number of leapfrog steps" (Gelman et al.)

Recommendations for the acceptance rate range between 65% to 95%, the acceptance rate should be considered in conjunction with additional review of the chain. Trace plot and trank plots are visualizations that can be used to verify the health of the chain. We look for stationarity, good mixing, and convergence. Trank plots are visualizations "of the distribution of the ranked samples" and are recommended for readability when you are plotting many chains together. (McElreath)

A common problem with the Hamiltonian is caused by using flat priors. Flat priors can cause wildly implausible values and wide intervals in the posterior. This can happen particularly in cases with very small amounts of data information. This problem will show in unhealthy trace and trank plots. It may also be seen in "broad, flat regions on the posterior density". (McElreath) Changing to a weakly informative prior should stop unlikely values from unbalancing the chain; verify with new trace and trank plots.

There are a few different forms of the Hamiltonian including the adaptive algorithm for setting leapfrog steps described above. Gibbs Sampling and Riemannian adaptations can be used to increase the efficiency of the Hamiltonian. Adjustments to the algorithm should be determined based on the needs of the investigation.

It is helpful to review the background and structure of the Hamiltonian algorithm in order to better understand the optional additions to the algorithm as well as to improve the efficiency of the tuning process. However, we highly recommend using one of the many packages available in R such as the rethinking package, brms, rstanarm, or blavaan depending on the structure of your model. The packages are run based on the principles we have investigated with many additional options and are much easier than writing your own program for each investigation.

Before we take the easy (and 100% acceptable) route by plugging information into a package interface, we should look at an example in action.

Leapfrog Method

We first use a circle to represent the real trajectory, and then use the leapfrog method to simulate whether the predicted trajectory is the same as the real trajectory.

Firs, we draw a circle which radius is 1 and starting point is (0,1).

```
r = 1; a = −3.14159265/2
q = 0;p = 1

#real circle
t = seq(0, 6, by = 0.01)
```

```
q = cos(a + t)
p = -sin(a + t)

plot(q, p, type = "l", xlim = c(-2, 2), ylim = c(-2,2),
     xlab = "q(position)", ylab = "p(momentum)",
     main = "real circle")
```

Then, use leapfrog method to simulate our predicted trajectory in GIF. Here, we setup stepsize is 0.3. Following is the algorithm implemented in R.

```
epsilon = 0.3; L = 20
q_leapfrog = vector(length = L + 1)
p_leapfrog = vector(length = L + 1)
q_leapfrog[1] = 0
p_leapfrog[1] = 1

for (i in 2:(L+1))
{
  p_half = p_leapfrog[i - 1] - epsilon/2 * q_leapfrog[i - 1]
  q_leapfrog[i] = q_leapfrog[i - 1] + epsilon * p_half
  p_leapfrog[i] = p_half - epsilon/2 * q_leapfrog[i]
}

plot(q_leapfrog, p_leapfrog, xlim = c(-2, 2), ylim = c(-2, 2),
     xlab = "q(position)", ylab = "p(momentum)", type = "l",
     main = "leapfrog")

library(animation)
saveGIF(
  {
    ani.options(interval = 0.1)
    for (i in 1:(L+1))
    {
      plot(q, p, type = "l", xlim = c(-2, 2), ylim = c(-2,2),
           xlab = "q(position)", ylab = "p(momentum)",
           col = "red")

      lines(q_leapfrog[1:i], p_leapfrog[1:i], xlim = c(-2, 2),
            ylim = c(-2, 2), xlab = "q(position)",
            ylab = "p(momentum)", type = "b",col = "blue")
    }
  }
  ,movie.name = "leapfrog.gif")
```

Figure 1 shows the results using the leapfrog method with a stepsize of $\epsilon = 0.3$, which are indistinguishable from the true trajectory, at the scale of this plot. In Figure 2, the results of using the leapfrog method with $\epsilon = 1.2$ are shown (still with 20 steps, so almost four cycles are seen, rather than almost one). With this larger stepsize, the approximation error is clearly visible, but the trajectory still remains stable (and will stay stable indefinitely). Only when the stepsize approaches $\epsilon = 2$ do the trajectories become unstable.

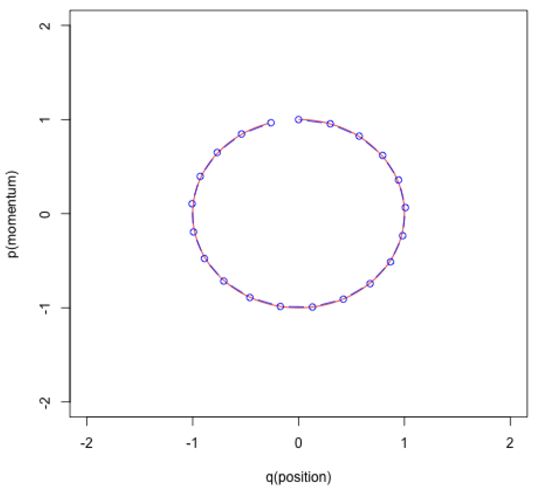

Figure 1: Results using leapfrog method for approximating Hamiltonian dynamics, when $H(q,p) = q^2/2 + p^2/2$. The initial state was q = 0, p = 1. The stepsize was $\epsilon = 0.3$. Twenty steps of the simulated trajectory are shown, along with the true trajectory (in red).

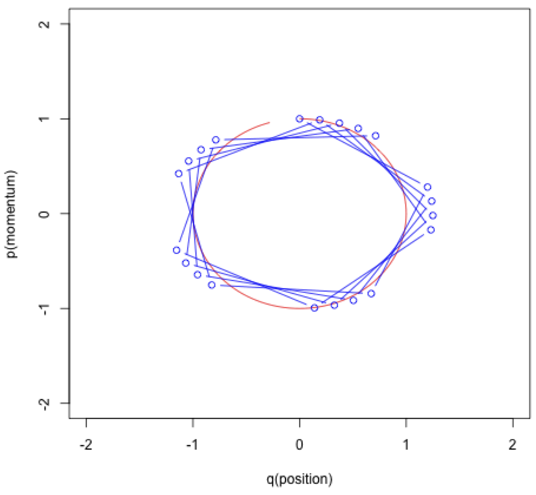

Figure 2: Results using stepsize $\epsilon = 1.2$. Twenty steps of the simulated trajectory are shown, along with the true trajectory (in red).

For these useful methods, as the step size ϵ goes to 0, the error also goes to 0, so an upper bound on the error applies to any differentiable function of the state, that is, if the error of (q, p) does not exceed ϵ^2, then the error of H(q, p) does not exceed order ϵ^2.

Implementation of HMC algorithm

An implementation of the HMC algorithm in R is given below. Its first two arguments are functions — U, which returns the potential energy of q, and grad_U, which returns the partial derivative (gradient) of U given q. The other parameters are the step size of the leapfrog algorithm - ϵ, the number of steps of the trajectory - L, and the current position - current_q, (where the trajectory starts). The momentum variable is sampled within the function, discarded at the end of the function, and the second position q is returned. Kinetic energy is assumed to have the simplest form $K(p) = p^2/2$ (i.e. all m_i are 1). In this program, vector operations are used to update all components of q and p simultaneously.

```
    HMC = function (U, grad_U , epsilon , L, current_q)
{
  q = current_q
  p = matrix(rnorm(length(q),0,1), ncol = 1)
  current_p = p
  # Make a half step for momentum at the beginning
  p = p - epsilon * grad_U(q) / 2

  # Alternate full steps for position and momentum
  for (i in 1:L)
  {
    # Make a full step for the position
    q = q + epsilon * p
    # Make a full step for the momentum, except at end of trajectory
    if (i!=L) p = p - epsilon * grad_U(q)
  }

  # Make a half step for momentum at the end.
  p = p - epsilon * grad_U(q) / 2

  # Negate momentum at end of trajectory to make the proposal symmetric
  p = -p

  # Evaluate potential and kinetic energies at start and end of trajectory
  current_U = U(current_q)
  current_K = sum(current_p^2) / 2
  proposed_U = U(q)
  proposed_K = sum(p^2) / 2

  # Accept or reject the state at end of trajectory, returning either
  # the position at the end of the trajectory or the initial position
  if (runif(1) < exp(current_U - proposed_U + current_K - proposed_K))
  {
    return (q)  # accept
  }
  else
  {
    return (current_q)  # reject
  }
}
```

A Familiar Example

For problem 5 in homework 2, we sampled 50 values from a normal population, then used MC sampling to sample from the joint posterior density of μ and σ^2 via the following model.

$$(\frac{\sum(y_i-\mu)}{\sigma^2}, \frac{-(n+2)}{\sigma^2}+\frac{\sum(y_i-\mu)^2}{2(\sigma^2)^2})$$

$$y_i \sim N(\mu,\sigma^2), i=1,...,n$$

$$p(\mu,\sigma^2) \propto \sigma^{-2}$$

$$\sigma^2|data \sim s.inv\chi^2(n-1,s^2)$$

$$\mu|\sigma^2, data \sim N(\bar{y},\frac{s^2}{n})$$

We found that the MAP estimates of μ and σ^2 were close to the true population parameter values. Here, we revisit the problem in the context of Hamiltonian Monte Carlo (HMC). To implement HMC, we need to calculate the gradient of the log-posterior, which is fairly simple in this case. The posterior is given to be

$$p(\mu,\sigma^2|data) = (\sigma^2)^{-(n+2)}e^{-\frac{\sum(y_i-\mu)^2}{2\sigma^2}}$$

The gradient, then is

$$(\frac{\sum(y_i-\mu)}{\sigma^2}, \frac{-(n+2)}{\sigma^2}+\frac{\sum(y_i-\mu)^2}{2(\sigma^2)^2})$$

The gradient is used to update a momentum vector ϕ, which accelerates the target distribution traversal process.

The HMC algorithm implemented here is described is pages 301-303 of the text (Gelman). The algorithm works in this case as follows: A momentum vector, ϕ, is drawn from it's target distribution; bivariate normal, in this case, as we are simulating two parameters. Next, the leapfrog process is initiated. The gradient of the log-posterior is evaluated at the initial values of μ and σ^2, this value is scaled by a scaling factor ϵ, multiplied by 0.5, and added to ϕ to make a half-step update of ϕ. A full update is made of μ and σ^2, by adding the product of the of the inverse of the mass matrix and ϕ, scaled by a factor ϵ, to the prior values of μ and σ^2. Then, another half-step update of ϕ is performed in the same way as before, but by evaluating the log-posterior at the updated values of μ and σ^2. This leapfrog process runs as many times as is specified by the user. At the end of the leapfrog process, the accept/reject step determines whether to keep the update parameters; we set $\mu^\star$, $\sigma^{2^\star}$ and $\phi^\star$ to the post-leapfrog μ and σ^2 ϕ, and $\mu^{(t-1)}$, $\sigma^{2^{(t-1)}}$, and $\phi^{(t-1)}$

to the pre-leapfrog μ σ^2 and ϕ, and the accept reject step is performed as follows.

$$r = \frac{p(\mu^\star, \sigma^{2^\star}|y)p(\phi^\star)}{p(\mu^{(t-1)}, \sigma^{2^{(t-1)}}|y)p(\phi^{(t-1)})}$$

$$\mu^t, \sigma^{2^t} = \begin{cases} \mu^\star, \sigma^{2^\star} & \text{with probability min(r,1)} \\ \mu^{(t-1)}, \sigma^{2^{(t-1)}} & \text{otherwise} \end{cases}$$

Note that there are three tuning parameters: the mass matrix, which is the covariance matrix of ϕ's target distribution, L, the number of leapfrog steps, and ϵ, the scaling factor. The algorithm is very sensitive to the tuning parameters, and small adjustments can result in dramatic spikes in autocorrelation and convergence failure. The following notes on tuning are taken from a STAN reference manual. "If ϵ is too large, the leapfrog integrator will be inaccurate and too many proposals will be rejected. If ϵ is too small, too many small steps will be taken by the leapfrog integrator leading to long simulation times per interval. Thus the goal is to balance the acceptance rate between these extremes. If L is too small, the trajectory traced out in each iteration will be too short and sampling will devolve to a random walk. If L is too large, the algorithm will do too much work on each iteration. If the mass matrix σ is poorly suited to the covariance of the posterior, the step size ϵ will have to be decreased to maintain arithmetic precision while at the same time, the number of steps L is increased in order to maintain simulation time to ensure statistical efficiency" (STAN Reference Manual).

Following is the algorithm implemented in R.

```
library(mvtnorm)
library(matlib)

n = 50
y = rnorm(n,7,2)#generate 50 values from normal

mu = sig = numeric()
mu[1] = 10 #initial values
sig[1] = 10
epsilon = .175 #scaling factor
L = 7 #number of leapfrog steps, must be greater than 1

post = function(mu, sig){
  return(sig^(-(n+2)) * exp(-sum((y-mu)^2)/(2*sig)))
}
```

```
gradient = function(mu, sig){
  dmu = sum(y-mu)/sig
  dsig = -(n+2)/sig + sum((y-mu)^2)/(2*sig^2)
  return(c(dmu, dsig))
}

mass_matrix = diag(c(1,1))

accept = 0
B = 2000

for(i in 1:B){
  #step 1, draw a phi
  Momentum_init = rmvnorm(1,mean=c(0,0),sigma = mass_matrix)
  #step 2, update theta and phi (first leapfrog step)
  #a, use gradient to make a half step of phi
  Momentum_star = Momentum_init + .5 * epsilon * gradient(mu[i],sig[i])
  #b, use momentum to update position vector theta (alpha and beta)
  mu_star = mu[i] + epsilon * (inv(mass_matrix)%*%t(Momentum_star))[1]
  sig_star = sig[i] + epsilon * (inv(mass_matrix)%*%t(Momentum_star))[2]
  #continue to leapfrog L times
    for(l in 2:L){
      #c again use gradient to make a half step of phi
      Momentum_star = Momentum_star + .5 * epsilon * gradient(mu_star,sig_star)
      #b, use momentum to update position vector theta (alpha and beta)
      mu_star = mu_star + epsilon * (inv(mass_matrix)%*%t(Momentum_star))[1]
      sig_star = sig_star + epsilon * (inv(mass_matrix)%*%t(Momentum_star))[2]
    }
  #make final half step of phi
  Momentum_star = Momentum_star + .5 * epsilon * gradient(mu_star,sig_star)

  #accept/reject step
  r = (post(mu_star,sig_star)*dmvnorm(Momentum_star,mean=c(0,0),sigma=mass_matrix))/
    (post(mu[i],sig[i])*dmvnorm(Momentum_init,mean=c(0,0),sigma=mass_matrix))

  if(runif(1) < min(1, r)){
    mu[i+1] = mu_star
    sig[i+1]= sig_star
    #Momentum = Momentum_star
    accept = accept + 1
  }else{ mu[i+1] = mu[i]
         sig[i+1]= sig[i]}
}
```

This algorithm, if tuned properly, produces output similar to that obtained via MC sampling from the known marginal distributions of μ and σ^2. Following is the output of a reasonably well tuned run, generating 2000 values.

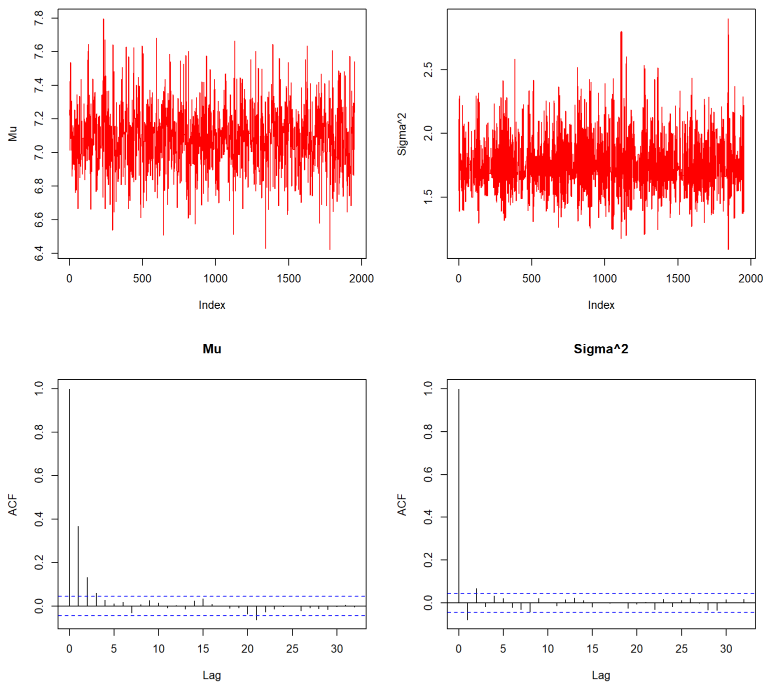

Figure 3: Trace plots and ACF plots of draws from a reasonably well-tuned HMC algorithm.

Next are two zoomed in trace plots, comparing HMC to MC.

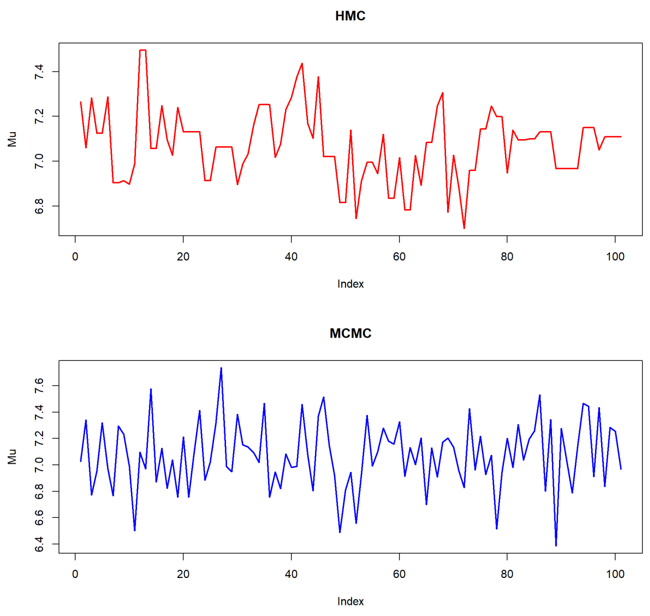

Figure 4: Trace plot of HMC sample of μ (top) does not exhibit random walk behavior, and closely resembles trace plot obtained via sampling directly from the marginal distribution of μ (bottom).

Unlike Metropolis-Hastings, Hamiltonian Monte Carlo does not exhibit random walk behavior. Even with less than ideal tuning, the trace plot of the HMC derived μ sample looks quite a bit like the trace plot of the sample obtained directly from μ's known marginal distribution; significant thinning is not required, though there is still some autocorrelation.

Conclusion

Hamiltonian Monte Carlo (HMC) is a computationally efficient approach to Monte Carlo sampling that uses principles of Hamiltonian mechanics to accelerate the Metropolis-Hastings process. Whereas Metropolis-Hastings exhibits random walk behavior in it's traversal of the target distribution, HMC does not, and samples generated via the Hamiltonian method usually do not require significant thinning to achieve independence. That said, tuning the HMC algorithm can be difficult, as it is very sensitive to three tuning parameters, so it is advisable to use pre-built packages in implementing HMC, such as STAN and hmclearn.

Works Cited

Brooks, Steve, et al. Handbook of Markov Chain Monte Carlo.
Boca Raton, Crc Press/Taylor & Francis, 2011.

Duane, Simon, et al. “Hybrid Monte Carlo.”
Physics Letters B, vol. 195, no. 2, 3 Sept. 1987, pp. 216–222,
https://doi.org/10.1016/0370-2693(87)91197-x.

Gelman, Andrew, et al. Bayesian Data Analysis. Boca Raton, Fl, Chapman
& Hall/Crc, 2014.

“Hamiltonian Function — Physics — Britannica.” Www.britannica.com,
www.britannica.com/science/Hamiltonianfunction.Mcelreath, Richard.

“Leapfrog Integration.” Wikipedia, 6 Apr. 2020, en.wikipedia.org/wiki/Leapfrog_integration.

“Molecular Dynamics.” Wikipedia, Wikimedia Foundation,
en.wikipedia.org/wiki/Molecular_dynamics.

Statistical Rethinking : A Bayesian Course with Examples in R and Stan.
Boca Raton, Crc Press/Taylor & Francis Group, 2016.

Thornton, Stephen T, and Jerry B Marion. Classical Dynamics of Particles
and Systems. Andover Cengage Learning, 2014.

STAN Reference Manual - 14.2 HMC Algorithm Parameters
https://mc-stan.org/docs/$2_19/reference-manual/hmc-algorithm-parameters.html$