Hamiltonian Monte Carlo and the NUTS Algorithm

MTH496
Prof. Dootika Vats

Zehaan Naik

Abstract

This project report aims to develop a fundamental understanding of Hamiltonian Monte Carlo (HMC) and the No-U-Turn Sampler (NUTS) as described in Hoffman et al. (2014). Taking a bottom-up approach, the report begins by exploring Markov Chain Monte Carlo (MCMC) methods from scratch, through an analysis of the Metropolis-Hastings (MH) algorithm and deterministic proposals. The report then delves into Hamiltonian Monte Carlo, concluding with an in-depth study of the NUTS algorithm. I explain the mechanics of HMC and NUTS and also highlight their advantages in sampling efficiency and computational performance.

Under Graduate Project - I

Department of Mathematics & Statistics
Indian Institute of Technology Kanpur

Contents

1 Introduction

Understanding the MCMC algorithms serves us first to understand the core concepts that build the foundations of the topic. The first section explores the core concepts in statistics that develop our understanding and context of the significant algorithms we undertake later in our exploration. In this section, I aim to analyse Markov Chains, kernels and their properties that help build a foundation to understand our final goal, which is HMC and the NUTS algorithm (as described in Hoffman et al. (2014)). The aim of this project is to develop a bottom-up understanding of the HMC paradigm and understand how the NUTS algorithm is able to solve the biggest issue with its Leap Frog implementation, which is to be able to tune ϵ and L agnostic of our target density.

1.1 Markov Chains

In this article, the most relevant tools for us are discrete state space and time continuous Markov chains (often referred to using the acronym MC or MCs in the article ahead).

1.1.1 What is a Markov Chain?

A Markov chain or Markov process is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. We can describe these events as a chain of random variables $< X_1, \ldots, X_n >$ for a discrete-time process or $\{X_t : t \in T\}$ for a continuous time process.

For this article, we shall exclusively focus on a discrete-time, continuous-state space Markov chain with the Markov Property defined as:

$$Pr(X_{n+1} \mid X_1, \ldots, X_n) = Pr(X_{n+1} \mid X_n)$$

We shall also assume this process to be Stationary, i.e. $\exists F$:

$$X_{n+1} \mid X_n \sim F \ \forall n$$

1.1.2 Definitions

I'll insert a few definitions of Markov chains, transition kernels and other miscellaneous items that would be useful for our analysis. These definitions come from Neal (2012a) Radford, Chapter 1, and I've mostly retained the same order to keep things consistent.

1. Kernel:

   A transition kernel (or simply, kernel) is defined as follows:

   a map $P : \mathscr{X} \times \mathscr{B}(\mathscr{X}) \to [0, 1]$, s.t.

   i. $\forall A \in \mathscr{B}(\mathscr{X})$

      $P(\dot{,}A)$ is a measurable function on $\mathscr{X}$.

   ii. $\forall x \in \mathscr{X}$

      $P(x, \dot{)}$ is a probability measure on $\mathscr{B}(\mathscr{X})$.

   Hence, we express P as:

$$P(x, A) = Pr(X_{n+1} \in A \mid X_n = x)$$

2. Added properties for the kernel:

   – Non-negative:

$$\forall x \in \mathscr{X}, A \in \mathscr{B}(\mathscr{X})$$

$$P(x, A) \geq 0$$

   – Sub-Markov:

     If P is non-negative and,

$$P(x, A) \leq 1 \ \forall x \in \mathscr{X}, A \in \mathscr{B}(\mathscr{X})$$

   – Markov:

     If P is non-negative and,

$$P(x, S) = 1 \ \forall x$$

     Where S is the state space.

   – Reversible:

     A kernel P is said to be *reversible* wrt. a certain measure μ if:

$$\int \int g(x)h(y)\mu(dx)P(x, dy) = \int \int h(x)g(y)\mu(dx)P(x, dy)$$

Lastly, a Markov kernel P is said to preserve a measure π if:

$$\int \int g(y)\pi(dx)P(x,dy) = \int g(x)\pi(dx)$$

For every bounded function g. Reversibility wrt. π implies preservation of π.

1.1.3 Target Distribution

The target distribution is the probability distribution from which we want to generate our samples. I would often use F or π to refer to our target distribution, which is assumed for this discussion to be continuous concerning λ (Lebesgue measure) and $\forall A \in \mathscr{B}(\mathscr{X})$,

$$F(A) = \int_A f(x)\,dx,$$

where $f(x)$ is the density function of the target distribution. Most MCMC algorithms aim to sample from this target distribution F. However, it often happens that we only know this target distribution up to some normalising constant c, i.e. $f(x) = c\tilde{f}(x)$ is a proper density function, but we only know the un-normalised density $\tilde{f}$.

1.2 Slice Sampling

Algorithms that we shall explore later in this report, like the Metropolis algorithm, can be used to sample from many of the complex, multivariate distributions encountered in statistics. However, to implement the Metropolis algorithm, one must find an appropriate "proposal" distribution that will lead to efficient sampling.

The first algorithm I discuss is "Slice Sampling" from Neal (2003). Suppose we wish to sample from a distribution for a variable, x, taking values in some subset of $\mathscr{R}^n$, whose density is proportional to some function $f(x)$. We can do this by sampling uniformly from the $(n+1)-$dimensional region under the $f(x)$ plot. This idea can be formalised by formalising an auxiliary real variable, y, and defining a joint distribution over x and y that is uniform over the region $U = \{(x,y) : 0 < y < f(x)\}$ below the curve or surface defined by $f(x)$. That is, the joint density for (x,y) is:

$$p(x,y) = \begin{cases} 1/Z, & \text{if } 0 < y < f(x), \\ 0, & \text{otherwise} \end{cases}$$

where, $Z = \int f(x)dx$. The marginal density for x is then:

$$p(x) = \int_0^{f(x)} (1/Z)dy = f(x)/Z$$

To sample from x, we can jointly sample from (x, y) and ignore y. Generating independent points drawn uniformly from U may not be easy, so we might instead define a Markov chain that will converge to this uniform distribution.

The following algorithm describes implementing slice sampling for a simple univariate example.

Algorithm 1 Slice Sampling

1: Choose a starting value x_0 for which $f(x_0) > 0$
2: Sample a $y \sim \text{Unif}(0, f(x_0))$
3: Draw a horizontal line across the curve at this y position
4: Sample a point (x, y) from the line segments within the curve
5: Repeat from step 2 using the new x value

2 Metropolis-Hastings Algorithm

2.1 Motivation

This algorithm uses the F-invariant property of MCs to sample from the target distribution using a known distribution. MH behaves like an accept-reject style algorithm where we take a known Markov kernel, $Q(x, .)$ with a known density function:

$$Q(x, dy) = q(x, y)dy.$$

For example, set $Q(x, .) = \frac{1}{\sqrt{2\pi}} \exp(\frac{-(y-x)^2}{2})$, a simple normal distribution. The MH MCMC algorithm will accept-reject samples from $Q(x, .)$

The ratio that we use to decide whether we accept or reject a proposed value of y is known as the "Hastings Ratio":

$$a(x, y) = \frac{f(y)q(y, x)}{f(y)q(x, y)}.$$

The rough algorithm structure is presented in Algorithm 2:

Algorithm 2 MH Sampling Idea

1: Draw $y \sim Q(x, .)$
2: Set $X_{n+1} = y$ with probability $\min(1, a(x, y))$
3: Else, set $X_{n+1} = X_n$

NOTE: The interesting thing is that we only need to know our target density up to normalising to use the MH paradigm, i.e

$$a(x, y) = \frac{c\tilde{f}(y)q(y, x)}{c\tilde{f}(y)q(x, y)} = \frac{\tilde{f}(y)q(y, x)}{\tilde{f}(y)q(x, y)}.$$

2.2 Theory

Theorem 1. *The MH algorithm defines the following transition kernel.*

$$P(x, A) = \int_A Q(x, dy)a(x, y) + \delta_x(A) \int [1 - a(x, u)]Q(x, du)$$

Proof.

$$\begin{aligned} P(x, A) &= Pr(X_{n+1} \in A \mid X_n = x) \\ &= Pr(X_{n+1} \in A, U \leq a(X_n, Y) \mid X_n = x) + Pr(X_{n+1} \in A, U > a(X_n, Y) \mid X_n = x) \end{aligned}$$

And, following this, it's relatively easy for us to prove:

I.
$$Pr(X_{n+1} \in A, U \leq a(X_n, Y) \mid X_n = x) = \int_A Q(x, dy)a(x, y)$$

II.
$$Pr(X_{n+1} \in A, U > a(X_n, Y) \mid X_n = x) = \delta_x(A) \int [1 - a(x, u)]Q(x, du)$$

And the rest follows. □

Theorem 2. *The MH kernel is F-symmetric and hence F-invariant.*

Proof. We need to show:

$$F(dx)P(x, dy) = F(dy)P(y, dx).$$

Note that this is clearly true if $x = y$. else, $x \neq y$, then,

$$\begin{aligned} F(dx)P(x, dy) &= f(x)q(x, y)\min(1, a(x, y)) \\ &= \min(f(x)q(x, y), f(y)q(y, x)) \\ &= f(y)q(y, x)\ \ \min(1, a(y, x)) \\ &= F(dy)P(y, dx) \end{aligned}$$

Note: This theorem helps us ensure that, if $(X_0 \sim F) \Rightarrow (X_t \sim F\ , \forall t \geq 0)$ □

2.3 Algorithm

Algorithm 3 Metropolis-Hastings

1: Draw $y \sim Q(x, .)$
2: Independently Draw $u \sim \mathcal{U}(0, 1)$
3: **if** $u < \min(1, a(x, y))$ **then**
4: $\quad X_{n+1} \leftarrow y$
5: **else**
6: $\quad X_{n+1} \leftarrow X_n$
7: **end if**

An example of this algorithm is to sample from a T distribution with 73 degrees of freedom. A visualisation for the same is presented in Figure 1. In the right panel, I also depict the time variability of the IID samples generated using the `rt` function in R and the samples that we get from the MH algorithm.

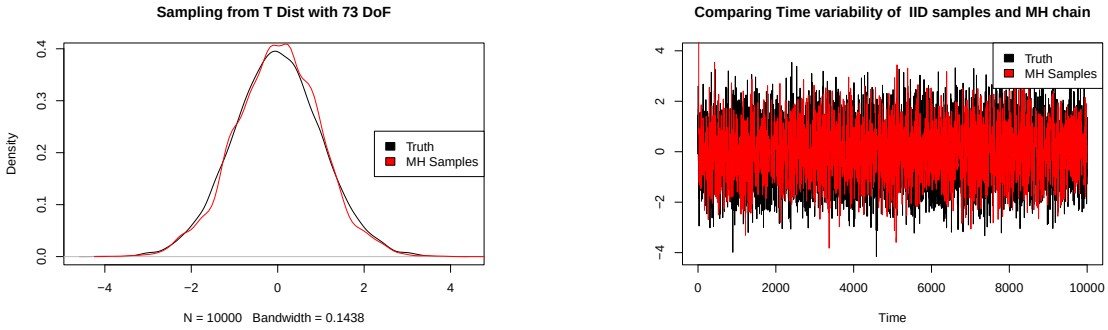

Figure 1: MH Sampling for T_{73}

3 Deterministic Proposals

3.1 Motivation and Problem Setup

In this new tool I wish to explore, we want to take a completely deterministic jump from a value we already know is from π. So for this problem, let's just humour the possibility that we can propose from $X \sim \pi$, but we want our proposal to be $Y = F(X)$ where F is some mapping from the state-space of $X \to$ state-space of X.

Problem Setup:

- Propose $z \sim \pi$ and independently $u \sim U(0, 1)$
- Set $y = F(z)$ and we can say that $y \sim Y$ for some Y
- Finally, we get an appropriate probability function $\alpha(x)$ so that we can use the accept-reject system that we have in place for the MH algorithm and still have π invariant Markov Chain for the proposal.

The accept reject step would look like this:

if $u < \min(1, \alpha(x))$ then, $x_{n+1} \leftarrow y$,

else, $x_{n+1} \leftarrow x$.

One quick observation that we can make here is since F is an involution, $(y = F(x)) \Rightarrow (x = F(y))$. Starting from x in π, we will transition between x and $F(x)$. We must ensure that the Markov kernel described above is π symmetric, hence π invariant.

3.2 The Acceptance Ratio

Drawing motivation from the Metropolis Ratio, let's see a logical guess for the above problem. In the following parallel, consider $y = F(x)$, which is what justifies us denoting the ratio as $\alpha(x)$ rather than $\alpha(x, y)$. The ratio is:

$$\alpha(x, y) = \frac{\pi(y)}{\pi(x)} \times \frac{q(y, x)}{q(x, y)}$$
$$\alpha(x, F(x)) = \alpha(x) = \frac{\pi(F(x))}{\pi(x)} \times \frac{q(F(x), x)}{q(x, F(x))}.$$

However, note that $q(y, x) = q(F(x), x)$ is simply the probability of us proposing x given that the current state is $F(x)$. According to our proposal setup, this is precisely equal to $\alpha(F(x), x) = \alpha(F(x))$. Similarly, the $q(x, y) = q(x, F(x)) = \alpha(x, F(x)) = \alpha(x)$.

Hence,

$$\alpha(x) = \frac{\pi(F(x))}{\pi(x)} \times \frac{\alpha(F(x))}{\alpha(x)}.$$

Since F is an involution, we know that $|\nabla F(x)| \times |\nabla F(y)| = 1$. Hence, one solution that works for this would be:

$$\alpha(x) = \frac{\pi(F(x))}{\pi(x)} \times |\nabla F(x)|,$$

and,

$$\alpha(F(x)) = \frac{\pi(x)}{\pi(F(x))} \times |\nabla F(F(x))|.$$

3.3 The Markov Kernel

The way we find the Markov kernel P expression in this case is the same as that in 2.2. We know:

$$\begin{aligned} P(x, dy) &= Pr(X_{n+1} = y | X_n = x) \\ &= Pr(y = F(x), u < \alpha(x) | X_n = x) \\ &= \min(1, \alpha(x)) I_{(y=F(x))} \end{aligned}$$

Similarly,

$$P(x, dx) = (1 - \min(1, \alpha(x)))$$

combining this, we get:

$$P(x, A) = \delta_{F(x)}(A) \min(1, \alpha(x)) + \delta_x(A)(1 - \min(1, \alpha(x)))$$

3.4 Target Invariance

The final thing we need to show is that this new proposal mechanism we've devised is indeed π invariant.

Theorem 3. *The Markov Kernel, defined by the deterministic proposal system, is π invariant.*

Proof. We need to show:

$$\pi(x)P(x, dy) = \pi(y)P(y, dx)$$

Note that this is clearly true if $x = y$.

and, if $y \neq F(x)$, then $P(x, dy) = P(y, dx) = 0$

finally, $y = F(x)$, then,

$$\begin{aligned}
\pi(x)P(x, dy) &= \pi(x) \min(1, \alpha(x)) I_{(y=F(x))} \\
&= \pi(x) \min(1, \alpha(x)) \\
&= \min(\pi(x), \pi(y) \times |\nabla F(x)|) \\
&= \pi(y)|\nabla F(x)| \min(\frac{\pi(x)}{\pi(y) \times |\nabla F(x)|}, 1) \\
&= \pi(y)|\nabla F(x)| \min(\alpha(y), 1) \\
&= \pi(dy)P(y, dx)
\end{aligned}$$

□

Note: With this, we have all the tools to proceed with the proof of the Hamiltonian Monte Carlo Algorithm.

4 Hamiltonian Monte Carlo

4.1 Hamiltonian Dynamics

I take this section almost directly from Vats (2023) and build a bit on a few equations. This elementary introduction to Hamiltonian dynamics will support understanding the algorithm. To grossly oversimplify the problem at hand, here are the relevant equations for the algorithm (From Chapter 5 Neal (2012b)):

Assume an imaginary particle a moving on a frictionless surface. This particle has both potential and kinetic energy. Now, assume that the state of this particle, a, is represented using two values: its position in the space q and its momentum p. Then, we can represent the potential and kinetic energy of the particle by:

$$\begin{aligned} U(q) &= -\log(\pi(q)) \\ K(p) &= \frac{p^2}{2m}. \end{aligned}$$

and, the Hamiltonian is:

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

and the Hamiltonian equations are:

$$\begin{aligned} \frac{dq}{dt} &= \frac{\partial H(p,q)}{\partial p} \\ \frac{dp}{dt} &= -\frac{\partial H(p,q)}{\partial q}. \end{aligned}$$

Now, for the example, consider $\pi \sim N(0, 1)$ and the mass of the body to be 1kg, which gives:

$$\begin{aligned} U(q) &= \frac{q^2}{2} \\ K(p) &= \frac{p^2}{2} \end{aligned}$$

and consequently,

$$H(p, q) = \frac{q^2}{2} + \frac{p^2}{2}$$

Finally, solving for the Hamiltonian Equations:

$$\frac{dq}{dt} = \frac{\partial H(p,q)}{\partial p} = p$$
$$\frac{dp}{dt} = -\frac{\partial H(p,q)}{\partial q} = -q$$

We get,

$$q(t) = r\cos(a+t)$$
$$p(t) = -r\sin(a+t)$$

4.2 The concept of "State"

With the understanding of Hamiltonian dynamics and the proposal systems that we discussed earlier. Let's define a new state space that will be the basis of the new algorithm; consider the augmented state-space (p, q) such that the tuple represents the current state of the "particle" and p is its momentum, and q is its position.

From our previous knowledge (and sticking to the specific example of $\pi \sim N(0,1)$), p and q are related by the following equation:

$$H(p,q) = \frac{q^2}{2} + \frac{p^2}{2} = c,$$

where c is some constant. The algorithm aims to sample the "momentum" p randomly, deterministically "jump" to a forward time-point, and use this relation to calculate q.

From this, we can observe a precise and valuable mapping T_s defined as follows:

$$T_s(p_t, q_t) := (p_{t+s}, q_{t+s}) = (p(t+s), q(t+s))$$

It is trivial from our definition of p_t and q_t that, $T_s(T_{-s})(p_t, q_t) = (p_t, q_t)$. This can be seen

through the following steps:

$$\begin{aligned} T_s(p_t, q_t) &= (p_{t+s}, q_{t+s}) \\ &= (-r\sin(a+t+s), r\cos(a+t+s)) \\ T_{-s}(p_{t+s}, q_{t+s}) &= (p_{t+s-s}, q_{t+s-s}) \\ &= (-r\sin(a+t+s-s), r\cos(a+t+s-s)) \\ &= (-r\sin(a+t), r\cos(a+t)). \end{aligned}$$

Hence,

$$T_s(T_{-s})(p_t, q_t) = (p_t, q_t),$$

which also reads as:

$$T_s{}^{-1} = T_{-s}.$$

This then motivates us to define an involution using T_s.

Consider the augmented space: $x = (p, q, \epsilon)$

$$P_s : (p_t, q_t, \epsilon = 1) \to (T_s(p_t, q_t), \epsilon = -1)$$

$$P_s : (p_t, q_t, \epsilon = -1) \to (T_{-s}(p_t, q_t), \epsilon = -1)$$

and with this, we have an involution that works!

4.3 Theory

We aim to sample from some density π. When we are at some state (p_k, q_k) where we start from $(p_0, q_0) = (0, 0)$ (or some other values in the augmented state space (p, q)), we sample independently from P. Do note that the fact that (p, q) in any state is distributed as $P \times \pi$ and the proposal is entirely deterministic, we can ensure that for this proposal system, our Markov Kernel is π invariant. The reason for this is the fact that the posterior P is independent of the target Q. Hence,

$$\begin{aligned} \Pr(p, q) &= \Pr(p \mid q)\Pr(q) \\ &= \Pr(p)\Pr(q) \end{aligned}$$

This allows us to independently draw from the posterior and still "be in" (P, Q). Hence, any deterministic mapping from the proposed state is also in (P, Q). Then, we ignore p to get samples from Q.

4.4 Algorithm

Algorithm 4 Hamiltonian Monte Carlo

1: Propose $p \sim P$ and use the Hamiltonian equation to arrive at the state (p, q)
2: Note: $(p, q) \sim (P \times \pi)$, since p and q are independent
3: Deterministically, transform the current state to a $+s$ time state using the previously used involution $P_s(p, q)$.

$$(p^*, q^*) = P_s(p, q)$$

4: Go ahead with the standard accept-reject step with the new acceptance ratio being:

$$\alpha(p, q) = \frac{\pi(q^*) f_P(p^*)}{\pi(q) f_P(p)} \times |\nabla P_s(p, q)|$$

Note: This comes directly from the deterministic proposal section and how we define $\alpha(x)$
5: **if** runif$(0, 1) < \min(1, \alpha(p, q)$ **then**
6: $\quad q_{k+1} \leftarrow q$
7: **else**
8: $\quad q_{k+1} \leftarrow q_k$
9: **end if**

An example of this algorithm is visually illustrated for different values of s, giving us a clear intuition of a good choice for s. Refer to Figure 2 for the same. Further Figure 4 illustrates how different values of s end up affecting the time variability of the HMC samples with the ACF plots (Figure 3) showing how a "large enough" value of s gives us uncorrelated samples and a small value of s gives us highly correlated samples. Lastly, Figure 5 illustrates the cyclic nature of HMC with respect to the variable s. We observe how the effective sample size shoots us as s approaches π and 3π. The reason for this is for these values of s; the proposed value $q*$ is the farthest away from q, and hence, we're making the "best" jumps. Whereas for values like 0 and 2π, we've essentially come back to the same location and hence the distance between q and $q*$ is minimum.

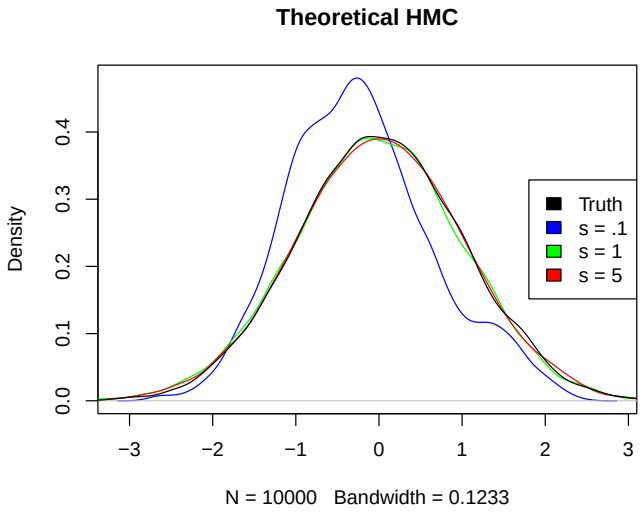

Figure 2: HMC Sampling for $s = 0.1, 1$ and 5

5 The Leap Frog Integrator

5.1 Motivation and Problem Setup

One challenge we might face in the HMC paradigm is solving the differential equations that arise from the Hamiltonian dynamics setup.

Recall that the Hamiltonian equations are:

$$\frac{dq}{dt} = \frac{\partial H(p,q)}{\partial p} \ , \ \frac{dp}{dt} = -\frac{\partial H(p,q)}{\partial q}.$$

Now, if we fix our p to come from a standard normal distribution, we can approximate the time update for p and q with the following Euler approximations:

$$p(t+\epsilon) = p(t) + \epsilon \frac{dp}{dt} = p(t) - \epsilon \frac{\partial H(p,q)}{\partial q}.$$

This means:

$$p(t+\epsilon) = p(t) - \epsilon \frac{\partial \log f_q}{\partial q}.$$

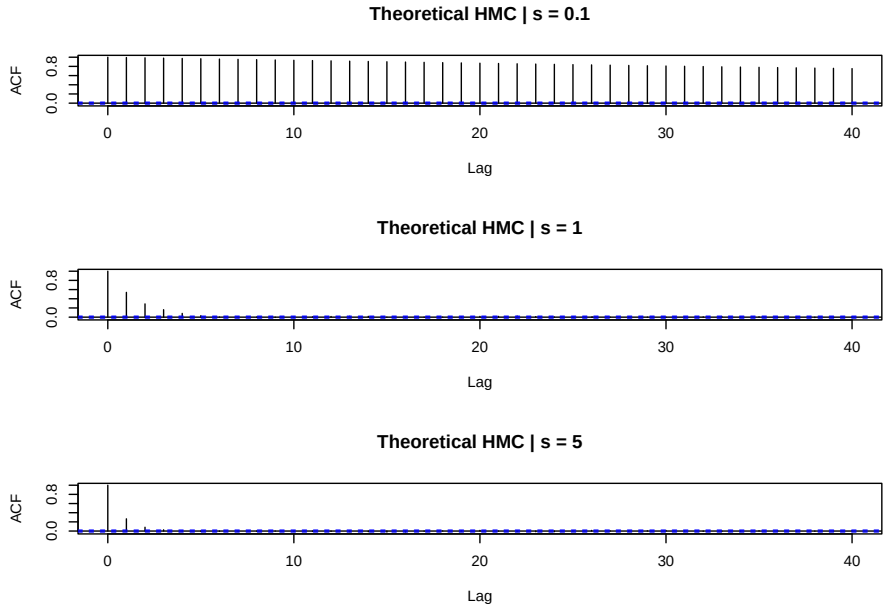

Figure 3: Comparision of ACF plots of samples with different values of s

Similarly:

$$q(t+\epsilon) = q(t) + \epsilon \frac{dq}{dt} = p(t) + \epsilon \frac{\partial H(p,q)}{\partial p}.$$

This means:

$$q(t+\epsilon) = q(t) + \epsilon \frac{p}{m}.$$

Note: In the multivariate case

$$q(t+\epsilon) = q(t) + \epsilon p M^{-1}.$$

However, these approximations show great variance and are unstable since the update of time $(t+\epsilon)$ depend on both q and p simultaneously. An excellent illustration for this has been shown in Vats (2023), in the section on the Leap Frog integrator.

To solve this problem, we scatter the update sequence so that information on p is $\frac{\epsilon}{2}$ time units ahead of information on q.

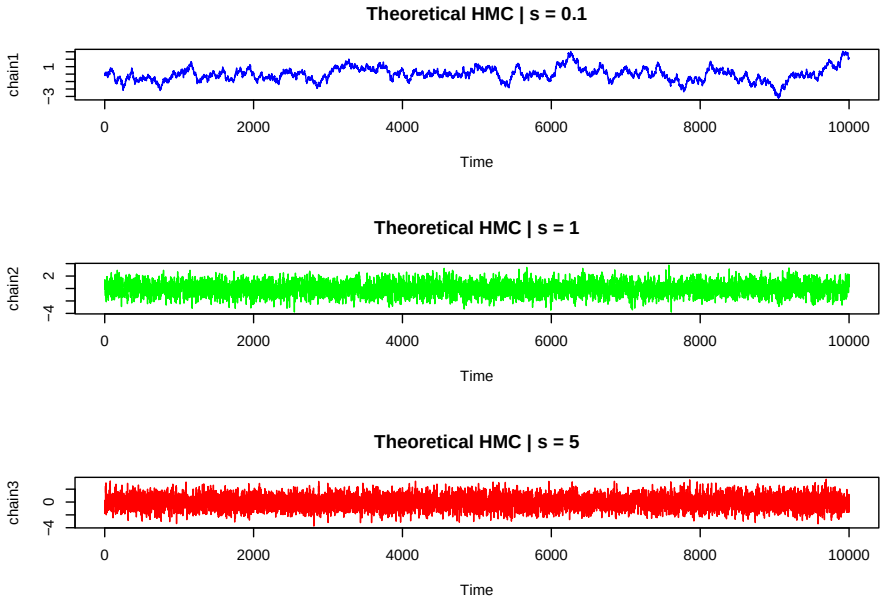

Figure 4: Comparision of the time variation of samples with different values of s

The "jumps" look like:

$$p(t+\epsilon/2) = p(t) - \epsilon/2 \times \frac{\partial \log f_q(q(t))}{\partial q}$$
$$q(t+\epsilon) = q(t) + \epsilon \frac{p(t+\epsilon/2)}{m}$$
$$p(t+\epsilon) = p(t+\epsilon/2) - \epsilon/2 \times \frac{\partial \log f_q(q(t+\epsilon))}{\partial q}$$

Note: this simulates the time jump of length ϵ.

Lastly, we want a "large" jump of size s such that our final "position" ends up being "far enough away" from our starting position in the circle. We repeat this step L many times to get a desired length of $s = L\epsilon$.

5.2 HMC using Leapfrog

Repeating the steps mentioned in the above Section 5.1 L many times, we get an approximation of T_s as discussed in the Hamiltonian setup. We denote this "leap" with $\tilde{T}_{L,\epsilon}$. Finally, we can define an involution similar to the one in Section 4.2, $\tilde{P}_{L,\epsilon}$ to get an

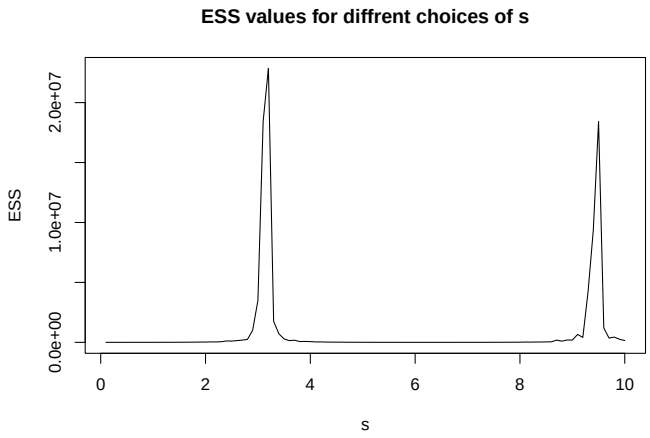

Figure 5: Visualising the cyclic nature of s in the HMC Algorithm

approximation for a volume-preserving involution for the HMC algorithm.

Important Note: Here, the analytical solutions to the equations are unavailable; hence, the acceptance ratio is not 1. With this setup, the final algorithm would be Algorithm 5:

Algorithm 5 HMC using Leapfrog

Draw $p \sim N(0, m)$
Calculate $(p^*, q^*) = \tilde{P}_{L,\epsilon}(p, q_k)$
Calculate the acceptance ratio:

$$r(q_k, q^*) = \exp(H(q^*, p^*) - H(q_k, p))$$

if runif$(0, 1) < \min(1, r(q_k, q^*))$ **then**
    $q_{k+1} = q^*$
else
    $q_{k+1} = q_k$
end if

A visual representation of Leap Frog on a target of $N(0, 1)$ is illustrated in Figure 6. We use $s = 1$ with different values of L and ϵ to understand the intuition behind the problem. Additionally, Figure 9, Figure 10 and 11 illustrate how values of ϵ and L are related with values of Effective Sample Size for $s = L\epsilon \in [0.01, 3.75]$. As expected, for large values of ϵ, we end up treading into more of a random-walk trajectory and hence cannot get a high acceptance rate. We notice two things: for large values of ϵ, we cannot find a good value

of L to get a good chain. And two, the choice of good L and ϵ seems to follow the same trend agnostic of the type of distribution we sample from.

Note: The densities that I sample from are:

$$\pi_G(x) = \prod_{i=1}^{d} \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left(-\frac{x_i^2}{2\sigma_i^2}\right),$$
$$\pi_L(x) = \prod_{i=1}^{d} \frac{1}{\sigma_i} \frac{\exp\left(x_i/\sigma_i\right)}{\left\{1+\exp\left(x_i/\sigma_i\right)\right\}^2},$$
$$\pi_{SG}(x) = \prod_{i=1}^{d} \frac{2}{\sigma_i\sqrt{2\pi}} \exp\left(-\frac{x_i^2}{2\sigma_i^2}\right) \Phi\left(\frac{\alpha x_i}{\sigma_i}\right),$$

where, $\sigma_i = 1\ \forall\ i$, $\alpha_i = 1$ and $\Phi(\theta)$ is the distribution function of the standard normal density.

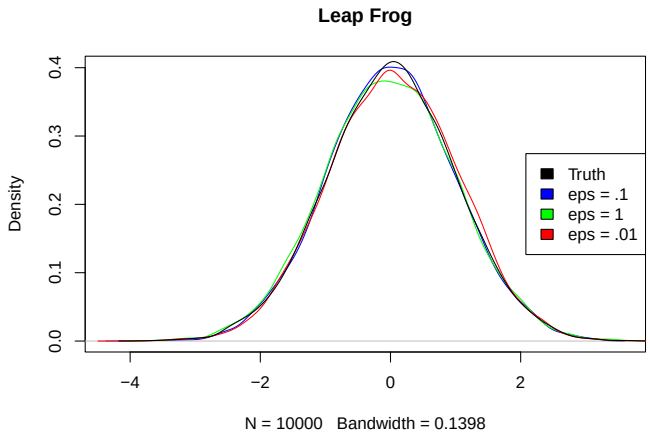

Figure 6: Leap Frog Sampling with $s = 1$ with different choices of L & ϵ

Note: It is common in most literature that I've come across while reading for this project to refer to this version of HMC as the actual algorithm. These approximations tend to have a very high acceptance rate because they are trying to simulate a setting where the acceptance ratio is 1, which makes them extremely popular. However, we must appreciate the core HMC algorithm as a separate entity and Leap Frog as a mere extension of the same.

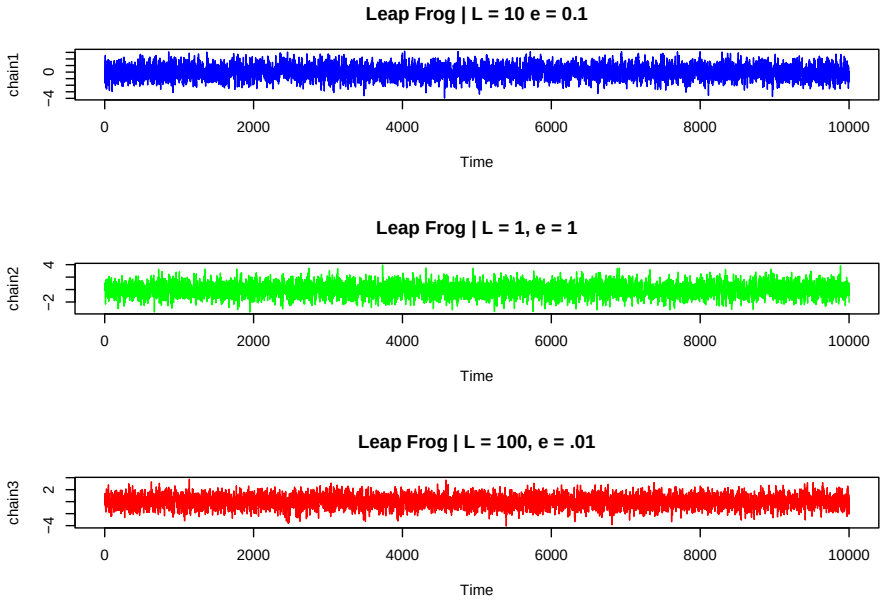

Figure 7: Time variability of samples with $s = 1$ with different choices of L & ϵ

5.3 Problems with Leap Frog

The common problem we face when working with LeapFrog is tuning the parameters L and ϵ. The value of L determines the extent of "exploration" our imaginary particle makes along the circle. If this value is "too small" likely, we do not take large enough leaps to get a decent move on the circle, resulting in undesirable random walk behaviour and slow mixing. If this value is "too large," the algorithm will generate trajectories that retrace their path and loop back. The determination of this value mainly relies on some heuristics and is not problem-agnostic.

For ϵ, if the value is "too small", the algorithm wastes a lot of time to "move ahead". If the value is too large, the jumps are not precise and vary too far from the actual circle. This means that our acceptance rate is minimal.

Tuning the variance matrix M is also related to a better acceptance rate, which we do not discuss in this project.

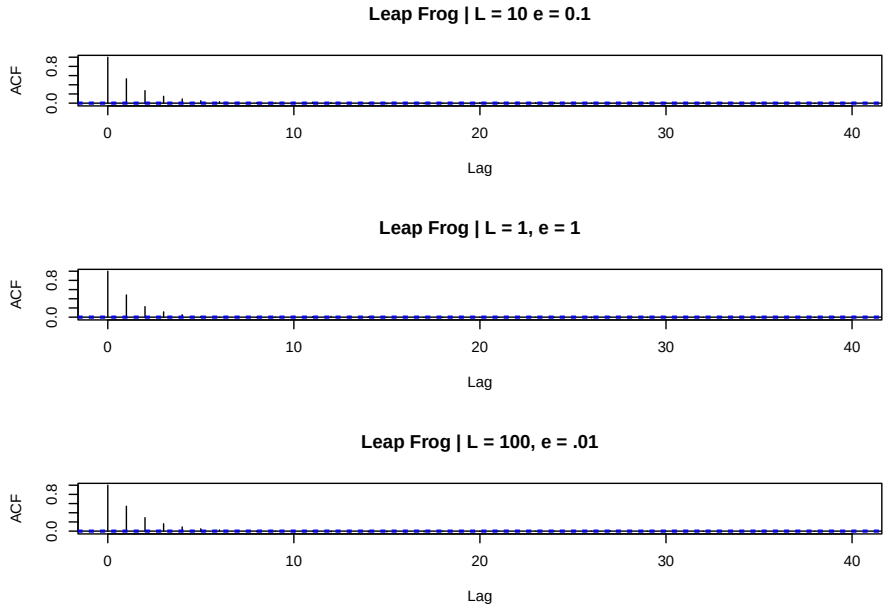

Figure 8: Comparing the ACF plots of samples with $s = 1$ with different choices of L & ϵ

6 The NUTS Algorithm

The No-U-Turn Sampler, Hoffman et al. (2014), aims to solve the two problems mentioned in the previous section. For my description of their algorithm, I stick to the vocabulary we've been using so far, i.e. p for the current momentum and q for the current position. We have $(p, q) = (r, \theta)$ to draw a clear parallel from what they present in their paper.

6.1 No U-Turns Motivation

The first challenge that we take up in the NUTS algorithm is to suppress the random-walk behaviour without setting the value for L. We need some criterion to tell us when we have simulated the dynamics for "long enough," that is, simulating more steps would no longer increase the distance between the proposal $\tilde{q}$ and the initial value of q.

6.1.1 Problem

To build intuition on the problem, I wish to discuss a quiz problem that I encountered in my basic algorithms course at my institute:

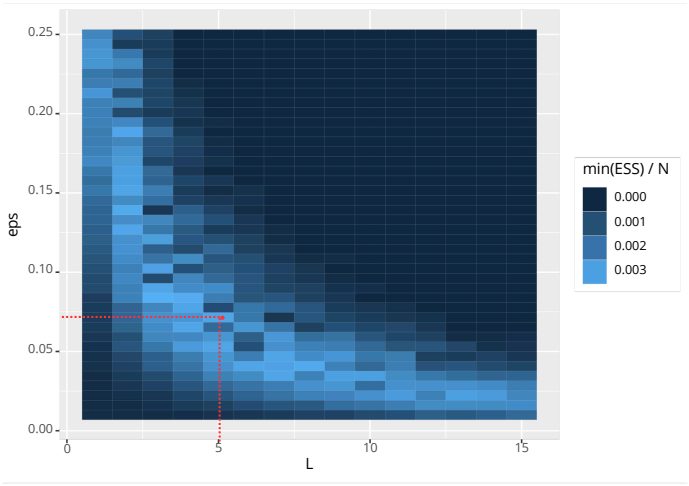

Figure 9: Heatmap of ESS values for different choices of L and ϵ for a π_G target with $d = 40$

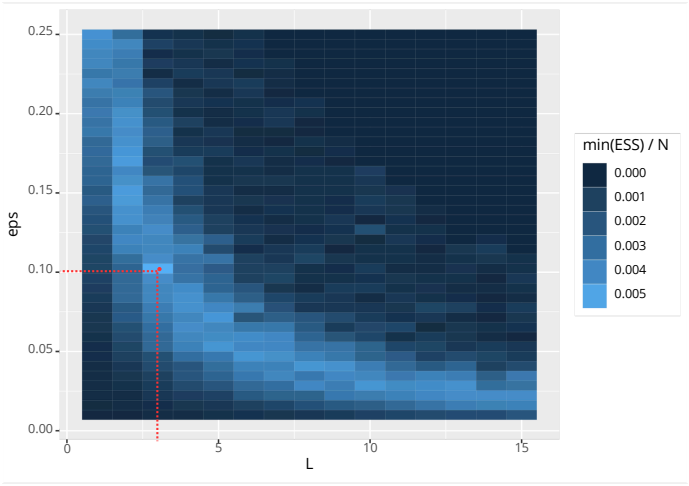

Figure 10: Heatmap of ESS values for different choices of L and ϵ for a π_{SG} target with $d = 40$

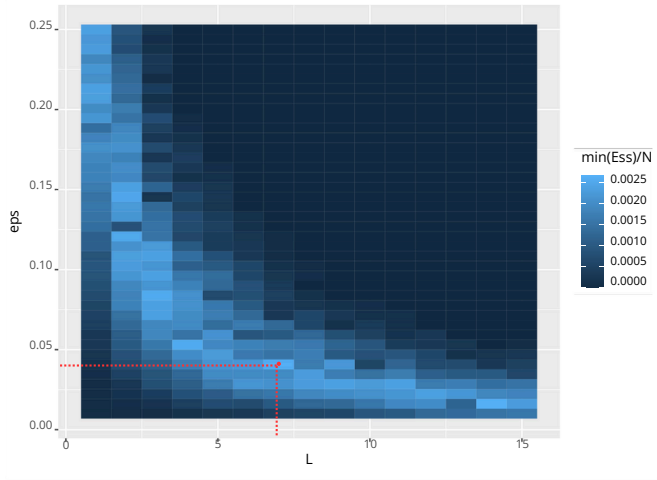

Figure 11: Heatmap of ESS values for different choices of L and ϵ for a π_L target with $d = 40$

Consider a situation where you stay in a city with two roads moving away in diametrically opposite directions. Your friend, who's not intelligent, went out on one of these roads and got a flat tyre. The poor chap does not know which road he took and calls you for help. He also does not know how far he's travelled on the road. Your task is to figure out where this poor guy is and help him return to the city.

The optimum solution to this problem gives an excellent insight into how they approach the solution to finding an appropriate value of L.

6.1.2 Solution:

Pick any road and "walk" 1km in that direction. Now, flip a coin to decide if you wish to "flip your direction". In this new direction, now walk 2km. Repeat this process of randomly switching directions and doubling the distance you walk until you "reach your friend".

Note: It can be shown that this is the optimum way of finding your friend as this is a $O(n)$ solution to the problem.

6.1.3 Insights

If we replace:

- To "walk" $n$km with taking n leapfrog steps.
- To "flip your direction" by changing the sign of time increments for the leapfrog step.
- To "reach your friend" with some notion of being the farthest away from the initial value of q. We use a convenient criterion based on the dot product between $\tilde{p}$ and $\tilde{q} - q$, which is the derivative wrt. time of $\frac{(\tilde{q}-q)^2}{2}$. This means that we should iterate till $((\tilde{q} - q).p) < 0$.

The intuition behind the algorithm becomes clear. Lastly, this algorithm does not guarantee time reversibility; hence, it is not guaranteed to converge to the correct distribution. NUTS fixes this problem by introducing a slice variable Neal (2003) u with the conditional distribution:

$$u \sim U(0, \exp\{\mathscr{L}(q) - \frac{p^2}{2}\})$$

This renders the conditional of $p|u$ and $q|u$ as:

$$p \mid u \sim U(p \mid \exp\left\{\mathscr{L}(q) - \frac{p^2}{2}\right\} \geq u)$$

$$q \mid u \sim U(q \mid \exp\left\{\mathscr{L}(q) - \frac{p^2}{2}\right\} \geq u)$$

6.2 Constructing the Algorithm

After augmenting the slice variable u to the current state vector (p, q) we get the joint probability of (p, q, u) as:

$$P(p, q, u) \propto I\left[u \in \left[0, \exp\left\{L(q) - \frac{p^2}{2}\right\}\right]\right]$$

We also add a finite set $\mathscr{C}$ of candidate position momentum states and another finite set $\mathscr{B} \supseteq \mathscr{C}$. $\mathscr{B}$ will be the set of states that the leapfrog integrator traces out during a given NUTS iteration, and $\mathscr{C}$ will be the set of those values to which we can transition without violating the detailed balance.$\mathscr{B}$ will be built up by randomly taking forward and backwards leapfrog steps, and $\mathscr{C}$ will be selected deterministically from $\mathscr{B}$.

The random procedure for building $\mathscr{B}$ and $\mathscr{C}$ given q, p, u, and ϵ will define a conditional distribution $p(\mathscr{B}, \mathscr{C}|q, p, u, \epsilon)$, upon which we place the following conditions:

- All elements of $\mathscr{C}$ must be chosen to preserve volume. That is any deterministic transformations of q, p used to add a state $q\prime$, $p\prime$ to $\mathscr{C}$ must have a Jacobian with unit determinant.
- $P((q,p) \in \mathscr{C}|q,p,u,\epsilon) = 1$
- $P(u \leq \exp\left\{L(q) - \frac{p^2}{2}\right\}|(p,q) \in \mathscr{C}) = 1$
- If $(p,q) \in \mathscr{C}$ and $(p\prime,q\prime) \in \mathscr{C}$ then for any $\mathscr{B}$, $P(\mathscr{B},\mathscr{C}|q,p,u,\epsilon) = P(\mathscr{B},\mathscr{C}|q\prime,p\prime,u,\epsilon)$

Additionally, it can be shown that the following procedure leaves the joint distribution $P(q,p,u,\mathscr{B},\mathscr{C}|\epsilon)$ invariant:

- sample $p \sim N(0,1)$
- sample $u \sim U(0, \exp\left\{L(q) - \frac{p^2}{2}\right\})$
- sample, $\mathscr{B},\mathscr{C}$
- sample $q_{t+1}, p \sim T(q_t, p, \mathscr{C})$

Where T is a transition kernel that leaves the uniform distribution over $\mathscr{C}$ invariant, that is, T must satisfy:

$$\frac{1}{|C|}\sum_{(q,p)\in C} T(q\prime,p\prime|q,p,C) = \frac{I(q\prime,p\prime \in C)}{|C|}$$

Lastly, we can shift our focus to the specific form of $P(\mathscr{B},\mathscr{C}|q,p,u,\epsilon)$ used by NUTS. Conceptually, the generative process for building B proceeds by repeatedly doubling the size of a binary tree whose leaves correspond to position-momentum states. These states will constitute the elements of B. The initial tree has a single node corresponding to the initial state. Doubling proceeds by choosing a random direction $v_j \sim U\{-1,1\}$ and taking 2^j leapfrog steps of size v_j (i.e., forwards in fictional time if $v_j = 1$ and backwards in fictional time if $v_j = -1$), where j is the current height of the tree. (The initial single-node tree is defined to have a height of 0.) For example, if $v_j = 1$, the left half of the new tree is the old tree, and the right half of the new tree is a balanced binary tree of height j whose leaf nodes correspond to the 2^j position-momentum states visited by the new leapfrog trajectory. Given the initial state q,p and the step size ϵ, there are 2^j possible trees of height j that can be built according to this procedure, each of which is equally likely. Conversely, the probability of reconstructing a particular tree of height j starting

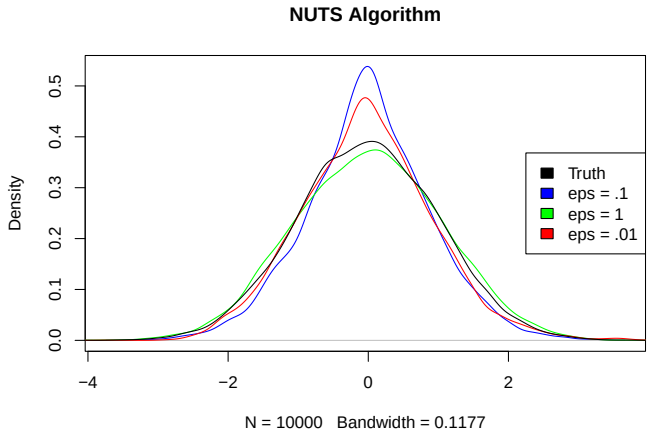

Figure 12: Sample density for NUTS Algorithm with different values of ϵ

from any leaf node of that tree is 2^j regardless of which leaf node we start from.
We cannot keep expanding the tree forever, of course. We want to continue expanding B until one end of the trajectory we are simulating makes a "U-turn" and begins to loop back towards another position on the trajectory. Continuing the simulation will likely be wasteful at that point since the trajectory will retrace its steps and visit locations in parameter space close to those we have already visited. We also want to stop expanding B if the error in the simulation becomes extremely large, indicating that any states discovered by continuing the simulation longer are likely to have astronomically low probability.

The second rule is easy to formalise — if the tree includes a leaf node whose state q, p satisfies:

$$\left\{ L(q) - \frac{p^2}{2} - \log(u) \right\} < \Delta_{max}$$

for some nonnegative Δ_{max}. We recommend setting Δ_{max} to a large value like 1000 so that it does not interfere with the algorithm as long as the simulation is moderately accurate.

6.3 The "Naive" NUTS Sampler

A visual example of the NUTS algorithm for a $N(0, 1)$ target is present in Figure 12. Figure 14 also illustrates the correlation across the sample chain for various values of ϵ. This gives us an intuition into the workings of the NUTS algorithm and how choosing the smallest value for ϵ need not provide us with the "best" results. We can also observe the

Algorithm 6 The Naive NUTS algorithm

Given $\theta^0, \epsilon, \mathcal{L}, M$:
for $m = 1$ to M **do**
  Resample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}\left(\left[0, \exp\left\{\mathcal{L}\left(\theta^{m-1}\right) - \frac{1}{2} r^0 \cdot r^0\right\}\right]\right.$
  Initialize $\theta^- = \theta^{m-1}, \theta^+ = \theta^{m-1}, r^- = r^0, r^+ = r^0, j = 0, \mathcal{C} = \left\{\left(\theta^{m-1}, r^0\right)\right\}, s = 1$
  while $s = 1$ **do**
    Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$
    if $v_j = -1$ **then**
      $\theta^-, r^-, -, -, \mathcal{C}', s' \leftarrow \text{BuildTree}\ (\theta^-, r^-, u, v_j, j, \epsilon)$
    else
      $-, -, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}\ (\theta^+, r^+, u, v_j, j, \epsilon)$
    end if
    if $s' = 1$ **then**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
    end if
    $s \leftarrow s' \gtrless (\theta^+ - \theta^-) \cdot r^- \geq 0] \, \mathbb{I}\left[(\theta^+ - \theta^-) \cdot r^+ \geq 0\right].$
    $j \leftarrow j + 1$
  end while
  Sample θ^m, r uniformly at random from $\mathcal{C}$
end for
function BUILDTREE$(\theta, r, u, v, j, \epsilon)$
  if $j = 0$ **then**
    Base case-take one leapfrog step in the direction v
    $\theta', r' \leftarrow \text{Leapfrog}\ (\theta, r, v\epsilon)$
    $\mathcal{C}' \leftarrow \begin{cases} \{(\theta', r')\} & \text{if } u \leq \exp\left\{\mathcal{L}\left(\theta'\right) - \frac{1}{2} r' \cdot r'\right\} \\ \emptyset & \text{else} \end{cases}$
    $s' \leftarrow \mathbb{I}\left[\mathcal{L}\left(\theta'\right) - \frac{1}{2} r' \cdot r' > \log u - \Delta_{\max}\right]$ **return** $\theta', r', \theta', r', \mathcal{C}', s'$
  else
    Recursion-build the left and right subtrees
    $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}\ (\theta, r, u, v, j - 1, \epsilon)$
    if $v = -1$ **then**
      $\theta^-, r^-, -, -, \mathcal{C}'', s'' \leftarrow \text{BuildTree}\left(\theta^-, r^-, u, v, j - 1, \epsilon\right)$
    else$-, -, \theta^+, r^+, \mathcal{C}'', s'' \leftarrow \text{BuildTree}\left(\theta^+, r^+, u, v, j - 1, \epsilon\right)$
    end if
    $s' \leftarrow s' s'' \mathbb{I}\left[(\theta^+ - \theta^-) \cdot r^- \geq 0\right] \mathbb{I}\left[(\theta^+ - \theta^-) \cdot r^+ \geq 0\right]$
    $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}''$ **return** $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s'$
  end if
end function

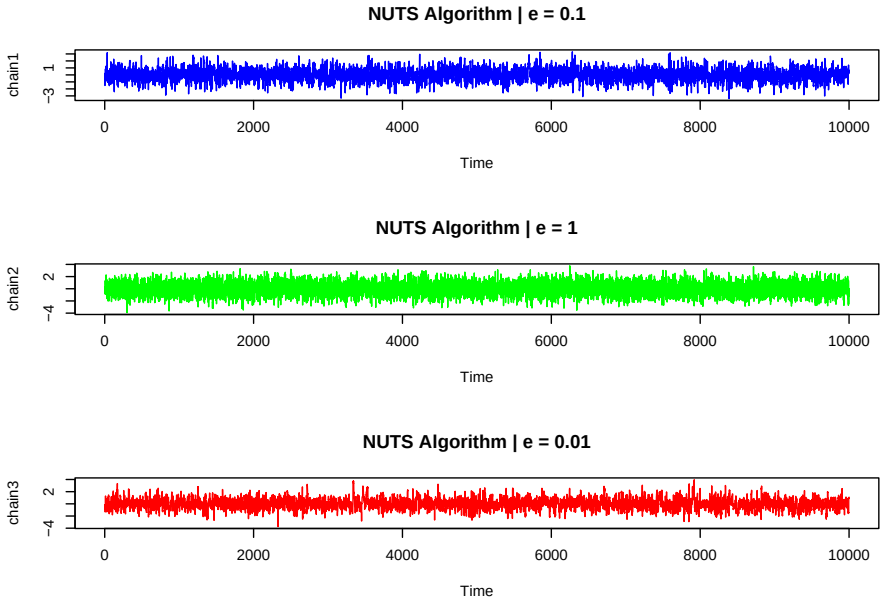

Figure 13: Time variability for NUTS Algorithm with different values of ϵ

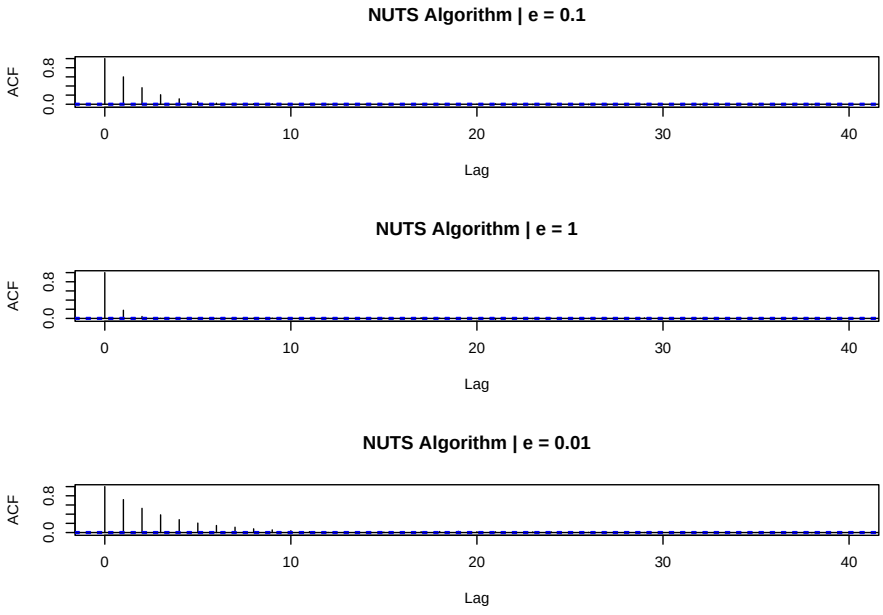

Figure 14: ACF plots for NUTS Algorithm with different values of ϵ

values of `depth` parameter for each of these ϵ's in Table 1 and also compare it with the optimum value for L from our Leap Frog Sampler in 6. Table 2 and Table 3 perform the same comparisons for a $N_{40}(0, I)$ target and a $d = 40, a = 1$ skewed Normal target. We can see that the expected pattern holds good in all the cases.

ϵ	Mean `depth`	$2^{\lfloor \text{depth} \rfloor}$	L_{optim}
0.1	4.12	16	15
1	1.53	2	3
0.01	7.42	128	103

Table 1: Comparing empirical value of depth against ϵ

6.4 Comparision with the Stan Implementation

In this section, I compare the implementation of the NUTS algorithm with the implementation described in the Stan manual. The visual difference in both implementations is the intuition behind the "Build Tree" and the "Travel Road" functions. The overarching concept that either intuition fundamentally depends upon is the exploration of the contour described by the Hamiltonian. This, in the univariate case, can be expressed as finding the appropriate value of `depth` such that:

$$2^{\text{depth}-1} - 1 < S < 2^{\text{depth}} - 1$$

Where S is the maximum distance we can achieve from our starting position based on the Euclidean metric on the Hamiltonian contour (example Table1). The core idea of either algorithm is to keep increasing the value of the `depth` parameter till the required condition is fulfilled.

The Stan implementation further increases efficiency in computation time. Algorithm 6 requires 2^{j-1} evaluations of $L(\theta)$ and its gradient (where j is the number of times `BuildTree()` is called), and O(2j) additional operations to determine when to stop doubling. In practice, for all but the smallest problems, the cost of computing $\mathcal{L}$ and its gradient still dominates the overhead costs, so the computational cost of Algorithm 6 per leapfrog step is comparable to that of a standard HMC algorithm. However, Algorithm 6 also requires 2^j position and momentum vectors, which may require an unacceptably large amount of memory. Furthermore, alternative transition kernels satisfy detailed concerns about the uniform distribution on $\mathcal{C}$ that produces larger jumps on average than simple

uniform sampling. Finasupposey, if a stopping criterion is satisfied before the final doubling iter. In that case, there is no point in wasting computation to build up a set $\mathcal{C}'$ that will never be used.

The third issue is easily addressed—if we break out of the recursion as soon as we encounter a zero value for the stop indicator s, then the correctness of the algorithm is unaffected, and we save some computation. The second issue is addressed using a more sophisticated transition kernel to move from one state in $\mathcal{C}$ to another. Particularly:

$$T\left(w' \mid w, \mathcal{C}\right) = \begin{cases} \frac{\mathbb{I}w' \in \mathcal{C}^{\text{new}}\,]}{|\mathcal{C}^{\text{new}}\mid} & \text{if } |\mathcal{C}^{\text{new}}\mid > \left|\mathcal{C}^{\text{old}}\right|, \\ \frac{|\mathcal{C}^{\text{enw}}}{\left|\mathcal{C}^{\text{old}}\right|}\frac{[w' \in \mathcal{C}^{\text{new}}\,]}{|\mathcal{C}^{\text{new}}\mid} + \left(1 - \frac{|\mathcal{C}^{\text{new}}\mid}{\left|\mathcal{C}^{\text{old}}\right|}\right)\mathbb{I}\left[w' = w\right] & \text{if } |\mathcal{C}^{\text{new}}\mid \leq \left|\mathcal{C}^{\text{old}}\right| \end{cases}$$

which admits a memory-efficient implementation that only requires storing $O(j)$ position and momentum vectors rather than $O(2^j)$. Here w and w' are shorthands for (q, p), $\mathcal{C}^{new}$ and $\mathcal{C}^{old}$ are disjoint subsets of $\mathcal{C}$ such that $\mathcal{C}^{new} \cup \mathcal{C}^{old} = \mathcal{C}$, and $w \in \mathcal{C}^{old}$.

The final optimisation that they make is based on the observation that $\Pr(q, p \mid \mathcal{C}')$ is the product of the probability of choosing some node from the subtree multiplied by the probability of selecting (q, p) uniformly at random from $\mathcal{C}_{subtree}$. For each of these smaller subtrees, we sample a (q, p) pair from $\Pr(p, q \mid (p, r) \in \mathcal{C}_{subtree})$ to represent that subtree. This procedure only requires that we store $O(j)$ position and momentum vectors in memory rather than $O(2^j)$ (a cost that, again, is usually very small compared with the $2^j - 1$ gradient computations needed to run the leapfrog algorithm).

7 Future Work

The NUTS algorithm, currently allows us to set a value of ϵ in the HMC algorithm agnostic of the target density and autotune the value of L for the sampler. An extension to this algorithm uses the concept of "dual averaging" to perform a similar auto-tuning of ϵ, allowing for the algorithm to sample from virtually any target distribution function without having to set the values of L and ϵ explicitly. I aim to understand this extension of HMC and implement it using my own understanding of the algorithm.

Lastly, I also wish to explore the idea of auto-tuning the parameters using another intuition. Assume that we fix that we wish to obtain a sample of length $1e5$ in 1 second. From various data, such as the speed of my computer processor and other hardware specs of my device, I figured out that I could compute a maximum of n gradients of my density

function. This gives us an upper bound over L. With our L fixed, we simply need to figure out what the minimum value of s that we need to achieve to get "good" effective sample size. With this value of s set, we have a value of ϵ, and we can run the algorithm. For future exploration, I wish to dive deep into this idea and try to benchmark this new algorithm with NUTS on several parameters and see how each of them performs in sampling from various distributions.

References

Geyer, C. J. (2003). The metropolis-hastings-green algorithm.

Green, P. J. (1995). Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732.

Hoffman, M. D., Gelman, A., et al. (2014). The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623.

Holbrook, A. (2021). Dr. andrew holbrook's lecture on hamiltonian monte carlo (hmc). In *Dr. Andrew Holbrook's lecture on Hamiltonian Monte Carlo (HMC)*.

Neal, R. M. (2003). Slice sampling. *The Annals of Statistics*, 31(3):705 – 767.

Neal, R. M. (2012a). *Handbook of Markov Chain Monte Carlo, Chapter 1*. arXiv preprint arXiv:1206.1901.

Neal, R. M. (2012b). *Handbook of Markov Chain Monte Carlo, Chapter 5*. arXiv preprint arXiv:1206.1901.

Sherlock, C., Urbas, S., and Ludkin, M. (2022). The apogee to apogee path sampler.

Thomas, S. and Tu, W. (2021). Learning hamiltonian monte carlo in r. *The American Statistician*, 75(4):403–413. PMID: 37465458.

Vats, D. (2023). Hamiltonian monte carlo for (physics) dummies.

	ϵ	L_{optim}	min(`depth`)	mean(`depth`)	max(`depth`)
1	0.01	15	1	1.8503	5
2	0.0161538461538462	15	1	1.5599	4
3	0.0223076923076923	13	1	1.3813	4
4	0.0284615384615385	9	1	1.2615	4
5	0.0346153846153846	7	1	1.1897	3
6	0.0407692307692308	8	1	1.143	3
7	0.0469230769230769	7	1	1.1104	3
8	0.0530769230769231	5	1	1.0793	3
9	0.0592307692307692	7	1	1.0596	3
10	0.0653846153846154	4	1	1.0431	3
11	0.0715384615384615	5	1	1.033	2
12	0.0776923076923077	4	1	1.0244	2
13	0.0838461538461538	3	1	1.015	2
14	0.09	3	1	1.0143	2
15	0.0961538461538461	2	1	1.011	2
16	0.102307692307692	3	1	1.0084	2
17	0.108461538461538	2	1	1.0054	2
18	0.114615384615385	2	1	1.0043	2
19	0.120769230769231	3	1	1.0037	2
20	0.126923076923077	2	1	1.0031	2
21	0.133076923076923	2	1	1.0025	2
22	0.139230769230769	2	1	1.0017	2
23	0.145384615384615	2	1	1.0015	2
24	0.151538461538462	2	1	1.0007	2
25	0.157692307692308	2	1	1.0009	2
26	0.163846153846154	2	1	1.0005	2
27	0.17	2	1	1.0004	2
28	0.176153846153846	2	1	1.0002	2
29	0.182307692307692	2	1	1.0006	2
30	0.188461538461538	2	1	1.0003	2
31	0.194615384615385	2	1	1.0005	2
32	0.200769230769231	1	1	1	1
33	0.206923076923077	1	1	1.0002	2
34	0.213076923076923	1	1	1	1
35	0.219230769230769	2	1	1.0002	2
36	0.225384615384615	1	1	1.0001	2

Table 2: Comparing empirical value of depth against ϵ for $N_{40}(0, I)$

	ϵ	L_{optim}	min(`depth`)	mean(`depth`)	max(`depth`)
1	0.01	15	1	1.8616	5
2	0.0161538461538462	15	1	1.5474	4
3	0.0223076923076923	15	1	1.3775	4
4	0.0284615384615385	10	1	1.2645	4
5	0.0346153846153846	9	1	1.1976	3
6	0.0407692307692308	8	1	1.1473	3
7	0.0469230769230769	6	1	1.107	3
8	0.0530769230769231	5	1	1.0791	3
9	0.0592307692307692	6	1	1.0601	2
10	0.0653846153846154	4	1	1.0423	3
11	0.0715384615384615	5	1	1.0338	2
12	0.0776923076923077	4	1	1.0239	2
13	0.0838461538461538	4	1	1.0169	2
14	0.09	4	1	1.0157	2
15	0.0961538461538461	3	1	1.01	2
16	0.102307692307692	3	1	1.0075	2
17	0.108461538461538	3	1	1.0061	2
18	0.114615384615385	3	1	1.0054	2
19	0.120769230769231	2	1	1.0037	2
20	0.126923076923077	3	1	1.0019	2
21	0.133076923076923	2	1	1.0017	2
22	0.139230769230769	2	1	1.0009	2
23	0.145384615384615	2	1	1.0017	2
24	0.151538461538462	2	1	1.0007	2
25	0.157692307692308	2	1	1.0007	2
26	0.163846153846154	2	1	1.0007	2
27	0.17	2	1	1.0005	2
28	0.176153846153846	2	1	1.0003	2
29	0.182307692307692	2	1	1.0004	2
30	0.188461538461538	2	1	1.0003	2
31	0.194615384615385	2	1	1.0002	2
32	0.200769230769231	2	1	1.0002	2
33	0.206923076923077	2	1	1	1
34	0.213076923076923	2	1	1.0001	2
35	0.219230769230769	1	1	1	1
36	0.225384615384615	2	1	1	1

Table 3: Comparing empirical value of depth against ϵ for $d = 40, \alpha = 1$ Skewed Normal target