MS Thesis for Masters in Statistics

— The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo

Jun Jie Choo

Advisor(s): Dr. Daniel Sanz-Alonso

{Approved} ~ (Signatures of the advisors)

{Date} ~ (Date of the signatures)

Winter Quarter, 2021

Abstract

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) method that is able to effectively sample high dimensional target distributions by using Hamilton's equations to design a proposal mechanism that exploits the target geometry. However, HMC's performance is highly sensitive to tuning parameters. The No-U-Turn Sampler (NUTS) eliminates the need to hand-tune parameters, whilst showing a competitive performance when compared with a well tuned vanilla HMC. This thesis will provide a background introduction to HMC and describe in detail the NUTS algorithm. Several numerical simulations will illustrate the flexibility of the NUTS algorithm and its practical advantage over vanilla HMC.

Contents

Introduction

Before the introduction of Hamiltonian Monte Carlo (HMC) (Duane et al., 1987), Random Walk Metropolis-Hastings (Hastings, 1970; Metropolis et al., 1953) (RWMH) was a popular and simple MCMC (Brooks et al., 2011; Robert & Casella, 2013) algorithm to sample a given target distribution. However, RWMH has an important drawback: its performance scales poorly to high dimensions. A characteristic of high dimensional distributions is that there is much more volume outside any given neighborhood. Hence, most of the RWMH proposed moves lie in regions of low probability and the acceptance probability of RWMH becomes very small, wasting computation. To counteract the low acceptance probability, one can tune the RWMH proposal to have smaller standard deviation. But this results in proposals that make extremely small jumps and a Markov chain that mixes slowly. Because of the reasons above, regardless of how the RWMH is tuned, the Markov chain will explore high dimensional parameter space very slowly. Additionally, the random walk nature of RWMH causes it to be inefficient as the sampler blindly samples around the area from the current point and subsequent proposals are rarely in the same direction. This is significant because the ability of a Markov chain to produce samples efficiently depends on how fast the Markov chain can traverse the direction that is most constrained. And since RWMH is rarely able to explore the same direction consecutively, much more samples and hence computation is needed (Neal, 1993). Although these problems can in theory be alleviated by suitably transforming the parameter space, finding such transformation is often computationally expensive, especially in high dimensionsal settings. Moreover, in most cases where the target distribution is not Gaussian, a transformation that produces uniform level sets may not be found.

This leads us naturally to HMC as it is able to explore parameter space in the same direction consecutively, avoiding the random walk behavior of RWMH, and also maintain a high acceptance probability even when the proposed moves are far from the current state. It does so by numerically simulating Hamiltonian dynamics to get the proposed moves, and then performing corrections via a Metropolis-Hastings acceptance rule. However, this comes at the cost of introducing new tuning parameters: stepsize ϵ and number of

numerical integrator steps L. The performance of HMC is sensitive to the tuning parameters used. Choosing the wrong tuning parameters can result in poor performance as the numerical trajectory of Hamiltonian dynamics may diverge to infinity or the proposed values may end up close to the current state. Additionally, the optimal tuning parameters may vary depending on the location of the current state, which can result in the problems mentioned above if the tuning parameters are not tuned conservatively enough. This provides the motivation for introducing new methods for identifying the optimal tuning parameters dynamically. The No-U-Turn Sampler (NUTS) is able to identify L dynamically by using a recursive algorithm that builds a set of proposal points, stopping automatically when the Hamiltonian trajectory starts to return to its starting point. It also uses a method based on primal-dual averaging for automatically adapting ϵ. Thus, the main advantage of NUTS is that it offers performance similar to a well-tuned HMC without any hand-tuning.

This thesis will be organized into two main chapters. The first chapter will focus on HMC. We introduce Hamilton's equations and illustrate some of their useful properties. We then describe the HMC algorithm and provide further background on the need to tune it appropriately. The second chapter will introduce the No-U-Turn Sampler (NUTS) and will go in detail over the criteria used to adaptively choose the tuning parameters, ending with a numerical investigation of the algorithm.

Chapter 1 Hamiltonian Monte Carlo

The random walk nature of RWMH causes it to be inefficient, especially in high dimensions, as the sampler blindly samples around the area from the current point. In order to avoid the local behavior of RWMH and be able to move to unexplored regions far away from the current state, we need to exploit the geometrical information of the target distribution. One natural way to quantify and understand information about a surface is through its differential structure. In particular, the gradient. But naively following the direction of the gradient will lead us to the mode and in high dimensions, volume concentrates away from the mode, (Betancourt, 2017). And since integrals are an accumulation of the integrand (which includes the density) over a volume of parameter space, this will cause computational inefficiencies as it prevents the area around the mode from contributing much to the overall expectation. In order to avoid these regions, and focus on areas of parameter space where there is significant contribution, HMC expands the probabilistic system by introducing auxiliary momentum variables.

For every probabilistic system, there is an equivalent physical system. The intuition of the auxiliary momentum variable can be understood via a physical example. Efficient exploration of the target distribution is just like maintaining a satellite in a stable orbit around the planet, (Betancourt, 2017). We want to give it just the right amount of momentum so that the satellite will be able to maintain its orbit around a planet. Too much momentum, the satellite will fly off towards space, and too little will cause the satellite to crash. We also want to introduce our momentum variables in such a way that energy is conserved. Appealing to the physical analogy, the satellite gains momentum as it drops towards the planet, and the satellite loses momentum as it gets further from the planet. In other words, any compression or expansion in position space must have an equivalent expansion or compression in momentum space to preserve volume in position-momentum space.

We will now introduce the Hamiltonian equations, which will be used to define the HMC proposals.

Hamilton's Equations

Given a d-dimensional position vector q, and a d-dimensional momentum vector p, we define the Hamiltonian $H(q, p)$ as a function of position and momentum:

$$H(q, p) := \mathcal{L}(q) + K(p),$$

where $\mathcal{L}(q)$ is interpreted as the potential energy and $K(p)$ is interpreted as the kinetic energy. In HMC, the potential energy is chosen (up to an additive constant) as the negative log-density of our target, and q represents the variables of interest. The kinetic energy function $K(p)$ is the first degree of freedom in HMC design that we can tune, with a common choice being the negative log-density of a Gaussian distribution with zero mean and covariance M, leading to

$$K(p) = p^T M^{-1} p/2,$$

where M is a symmetric positive definite mass matrix. With these, we can write the joint distribution of the position and momentum as follows:

$$\begin{aligned} P(q, p) &= \frac{1}{Z} \exp(-H(q, p)) \\ &= \frac{1}{Z} \exp(-\mathcal{L}(q)) \exp(-K(p)). \end{aligned}$$

The equations of motion are defined in terms of the partial derivatives of the Hamiltonian and determine how the position and momentum variables change over time t:

$$\begin{aligned} \frac{dq_i}{dt} &= \frac{\partial H}{\partial p_i}, \\ \frac{dp_i}{dt} &= -\frac{\partial H}{\partial q_i}, \end{aligned}$$

where $1 \leq i \leq d$. We can also vectorize the partial derivatives as follows:

$$\frac{dz}{dt} = J \nabla H(z),$$

where $z = (q, p)$, ∇H is the gradient of H, and

$$J = \begin{bmatrix} 0_{dxd} & I_{dxd} \\ -I_{dxd} & 0_{dxd} \end{bmatrix}.$$

Properties of Hamiltonian Dynamics

Hamiltonian dynamics can be used to construct efficient MCMC methods because of the following properties:

Reversibility

Hamiltonian dynamics are reversible. Hence, the mapping $\rho_s : (q(t), p(t)) \mapsto (q(t+s), p(t+s))$ is one-to-one and has an inverse, which is obtained by negating the partial derivatives $\frac{dq_i}{dt}$ and $\frac{dp_i}{dt}$. If the kinetic function chosen is an even function, i.e. $K(p) = K(-p)$, the inverse map can also be obtained by negating p, applying ρ_s, and then negating p again.

The common choice of $K(p) = p^T M^{-1} p/2$ satisfies the even function condition which implies that the Hamiltonian dynamics with this choice of kinetic function are reversible. This property is important for constructing Markov kernels that satisfy detailed balance and thus lead to MCMC updates that leave the target distribution invariant.

Conservation of Hamiltonian

A second property of the flow map is that it keeps the Hamiltonian invariant:

$$\frac{dH}{dt} = \sum_{i=1}^{d} \left(\frac{dq_i}{dt}\frac{\partial H}{\partial q_i} + \frac{dp_i}{dt}\frac{\partial H}{\partial p_i}\right) = \sum_{i=1}^{d} \left(\frac{\partial H}{\partial p_i}\frac{\partial H}{\partial q_i} - \frac{\partial H}{\partial q_i}\frac{\partial H}{\partial p_i}\right) = 0.$$

Conservation of the Hamiltonian implies that if we start at any $z = (q, p)$ and then apply flow map ρ_s, we will move along energy level sets. In other words, $H(z) = H(\rho_s(z))$ for any $s > 0$.

Symplecticness and Volume Preservation

Hamiltonian dynamics are symplectic. Defining J as before, the symplectic condition ensures that the Jacobian matrix B_s of the flow map ρ_s satisfies

$$B_s^T J^{-1} B_s = J^{-1},$$

which implies volume conservation since $\det(B_s^T)\det(J^{-1})\det(B_s) = \det(J^{-1})$ implies that $\det(B_s)^2 = 1$. This is important as using an algorithm for simulating Hamiltonian dynamics that does not preserve volume makes the computation of the Metropolis acceptance probability complicated (Lan et al., 2012).

MCMC from Hamiltonian Dynamics

For most practical applications, we are unable to generate the Hamiltonian trajectories since Hamilton's equations cannot be solved in closed form. But the flow map ρ_s can be approximated by a family of numerical solvers called symplectic integrators (Hairer et al., 2006; Leimkuhler & Reich, 2004), that are robust to drift, able to preserve volume and have the symplectic property. However, they are unable to preserve the Hamiltonian. Thus, an accept/reject step has to be implemented to deal with the error in the Hamiltonian.

Symplectic Integrators

In this thesis we will consider a simple and reversible symplectic integrator known as the leapfrog integrator. Although there are other symplectic integrators like the implicit midpoint method (Calvo et al., 2019; Pourzanjani & Petzold, 2019), we use the leapfrog integrator for simplicity. We will also choose the mass matrix to be the identity, $M = I$. Then, given a stepsize ϵ, one iteration of the leapfrog integrator with initial state $(p(t), q(t))$ is given by:

$$
\begin{aligned}
p_i(t+\epsilon/2) &= p_i(t) - (\epsilon/2)\frac{\partial \mathcal{L}}{\partial q_i}(q(t)), \\
q_i(t+\epsilon) &= q_i(t) + \epsilon p_i(t+\epsilon/2), \\
p_i(t+\epsilon) &= p_i(t+\epsilon/2) - (\epsilon/2)\frac{\partial \mathcal{L}}{\partial q_i}(q(t+\epsilon)).
\end{aligned}
$$

In HMC, we will propose moves using L leafrog iterations from the current state, with a suitable momentum flip discussed below. As noted earlier, the trajectories generated by the leapfrog method preserve volume in (q, p) space and it is this incompressibility that limits the error in the numerical trajectory and allows us to generate accurate numerical trajectories. Even though the generated trajectories are quite accurate, it still introduces some bias that requires correction.

Correcting for Symplectic Integrators

One way to correct the bias is to implement a Metropolis Hastings style accept/reject step for the proposed state. In order to construct that acceptance probability, we have to carefully modify the Hamiltonian transition.

Due to the way our trajectories are generated, we can only propose states going forwards and not backwards, the Metropolis Hastings acceptance probability always vanishes. We need to modify the Hamiltonian transition such that it is reversible so that the acceptance probability does not vanish. One way to achieve this is to flip the sign of the momentum variable of the proposed state.

Using a reversible Hamiltonian transition, the acceptance probability is thus:

$$
\begin{aligned}
\alpha(q^k, -p^k | q^{k-1}, p^{k-1}) &= \min\left(1, \frac{\rho(q^{k-1}, p^{k-1} | q^k, -p^k)\mathcal{L}(q^k, -p^k)}{\rho(q^k, -p^k | q^{k-1}, p^{k-1})\mathcal{L}(q^{k-1}, p^{k-1})}\right) \\
&= \min\left(1, \frac{\mathcal{L}(q^k, -p^k)}{\mathcal{L}(q^{k-1}, p^{k-1})}\right) \\
&= \min\left(1, \frac{\exp(-H(q^k, -p^k))}{\exp(-H(q^{k-1}, p^{k-1}))}\right).
\end{aligned}
$$

Because we can evaluate the Hamiltonian, we can evaluate the acceptance probability and correct for the bias induced by the symplectic integrator error. And since the error is limited, we are able to propose states far away from the initial state and have a high acceptance probability.

After accepting the proposal, we can just project the joint distribution onto q-space to obtain an appropriate sample of the variable of interest.

Algorithm 1: HMC algorithm

Given $q^0, \epsilon, L, \mathcal{L}$;
for $k = 1$ *to* K **do**
  Sample $p^0 \sim \mathcal{N}(0, I)$;
  Set $q^k \leftarrow q^{k-1}$, $\tilde{q} \leftarrow q^{k-1}$, $\tilde{p} \leftarrow p^0$;
  for $l = 1$ *to* L **do**
    Set $\tilde{q}, \tilde{p} \leftarrow \text{Leapfrog}(\tilde{p}, \tilde{q}, \epsilon)$;
  end
  With probability $\alpha = \min\Big(1, \frac{\exp\{\mathcal{L}(\tilde{q}) - \frac{1}{2}\tilde{p}.\tilde{p}\}}{\exp\{\mathcal{L}(q^{k-1}) - \frac{1}{2}p^0.p^0\}}\Big)$, set
  $q^k \leftarrow \tilde{q}$, $p^k \leftarrow -\tilde{p}$;
end

function: Leapfrog(q, p, ϵ)
Set $\tilde{p} \leftarrow p + (\epsilon/2)\nabla_q \mathcal{L}(q)$;
Set $\tilde{q} \leftarrow q + \epsilon\,\tilde{p}$;
Set $\tilde{p} \leftarrow \tilde{p} + (\epsilon/2)\nabla_q \mathcal{L}(\tilde{q})$;
return $\tilde{q}, \tilde{p}$;

Output: Samples $q^1, \ldots, q^K$.

Proof of Invariance

The algorithm has two steps. In the first step, new momentum values are drawn from the Gaussian distribution. Since q isn't changed and p is drawn from the same distribution, the first step leaves the canonical joint distribution invariant.

The second step involving the leapfrog method and accept/reject step also leaves the joint distribution invariant. Suppose we partition the (q, p) space into regions A_i. Let B_i be the image of A_i with respect to the leapfrog map ρ and momentum flip.

Detailed balance holds if for all i, j, $P(A_i)\rho(B_j|A_i) = P(B_j)\rho(A_i|B_j)$.

We will now give a heuristic argument that the second step satisfies detailed balance.

Clearly, when $i \neq j$, $\rho(B_j|A_i) = \rho(A_i|B_j) = 0$. Hence the equation above holds for $i \neq j$.

For $i = j$, we note that the probability that the next state is in region B_i is the probability that the current state is in region B_i and the proposal is rejected, plus the probability that the current state is in region A_i and a move to B_i was proposed and accepted. Let $R(B_i)$ be the probability of the proposal in region B_i being rejected. The probability of the next state being in region B_i is therefore:

$$\begin{aligned} P(B_i)R(B_i) + \sum_j P(A_j)\rho(B_i|A_j) &= P(B_i)R(B_i) + \sum_j P(B_i)\rho(A_j|B_i) \\ &= P(B_i)R(B_i) + P(B_i)\sum_j \rho(A_j|B_i) \\ &= P(B_i)R(B_i) + P(B_i)(1 - R(B_i)) \\ &= P(B_i). \end{aligned}$$

Since the second step satisfies detailed balance, it leaves the canonical joint distribution invariant. Thus overall, the HMC algorithm leaves the canonical joint distribution invariant. A rigorous proof of this claim can be found in (Neal & others, 2011).

Ergodicity of HMC

Theoretical analysis of the ergodicity (Roberts et al., 2004) of HMC is difficult, and is only starting to emerge (Livingstone et al., 2019). However, we note that there are pathological cases where HMC is not ergodic.

Ergodicity can fail if the choice of $\lambda = \epsilon L$ produces an exact periodicity for some function of the state, (Neal & others, 2011). Periodicity causes the trajectory to return to the same position and hence HMC will take a long time to explore the full state space. One way to solve this problem is by choosing ϵ and L randomly over some small interval, (Neal & others, 2011).

Tuning HMC

In the simple implementation of HMC, the tuning parameters are the stepsize ϵ and number of integrator steps L, both of which when multiplied together give the trajectory length λ. These parameters are hard to tune in practice.

Stepsize

We will first focus on how to tune the stepsize. Too small a stepsize will lead to slow exploration of the target distribution by random walk if the number of integrator steps L is not large enough to compensate. Stepsizes that are too large will cause approximation errors thus lowering acceptance probabilities and it may even cause the trajectory to become unstable and diverge entirely. When a stepsize that produces unstable trajectories is used, the value of the Hamiltonian may grow exponentially with L, causing the acceptance probability to vanish.

It is for these reasons that choosing a stepsize just below the stability limit would give the optimal result, (Neal & others, 2011). But the problem lies in the changing stability limit for different regions. If the preliminary run to determine the stability limit was started in a region where the stability limit is large, HMC will get stuck due to the low acceptance probability and might not explore the target distribution fully, even if the unexplored regions have substantial probability, thus biasing the results, (Neal & others, 2011).

This problem can be solved by choosing the stepsize from a distribution. (Bou-Rabee et al., 2017) And even if the mean of the distribution is large, a suitably small stepsize will occasionally be chosen, alleviating the unstable trajectory problem and consequently the HMC will be able to leave the region, (Neal & others, 2011).

Trajectory Length

Intuitively, if we integrate for too short a time, we will not take full advantage of the ability of HMC to propose moves that are far away but can be accepted with high probability. On the other hand, because the energy level sets are topologically compact in well-behaved problems, trajectories that are too long will take a long time to compute and will reverse directions and eventually return to previously explored regions, (Betancourt, 2017).

Thus, just like the stepsize, choosing a suitable trajectory length is crucial for efficient exploration of the state space. As for the stepsize, randomly varying the length of the trajectory may be desirable as in general, the optimal integration time will vary depending on the trajectory and which region we

are in. Consequently, no static integration time will perform well everywhere as HMC will get stuck in certain regions.

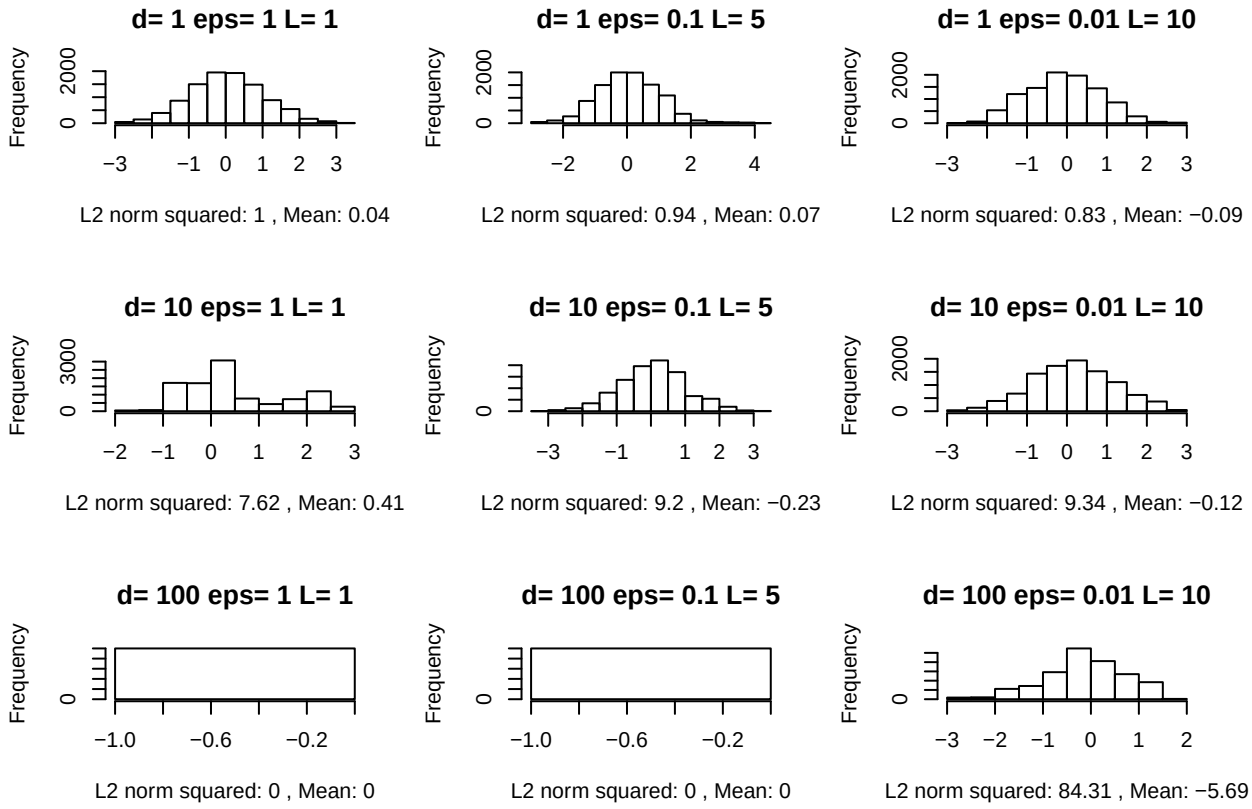
Aside from randomly choosing a trajectory length, we can attempt to identify the optimal integration time dynamically as the trajectory is being generated. Coincidentally, this is the main motivation of the No-U-Turn Sampler (NUTS), where the No-U-Turn criterion is used to determine the integration time adaptively in such a way that tuning is automatic.

HMC Sensitivity to Tuning Parameters

The results below are from a vanilla HMC that was run on a standard multivariate normal of dimension d with $K = 10,000$ iterations. The figure below is a demonstration of how the tuning parameters ϵ and L affect the performance of HMC as dimension d increases.

As the dimension d increases, we have to reduce the stepsize ϵ. Since whether the trajectory is accepted depends on the total error in the Hamiltonian due to the leapfrog discretization, which is a sum of the errors due to each (q_i, p_i) pair, in order to keep the cumulative error low, the stepsize has to be reduced as d grows. As we reduce the stepsize ϵ, we must also increase the number of integration steps L so that the exploration of the parameter space by HMC does not become a random walk. We remark that while the stepsize of HMC needs to be reduced as the dimension d increases, the rate at which it has to be reduced is slower than that of RWMH and other competitive algorithms (Neal & others, 2011).

The following diagnostics are used to quantify the performance of HMC. A histogram of the first dimension of parameter space is shown to get a sense of how well HMC has performed with respect to its parameters. An estimate of the L2 norm squared and the mean are also reported. The estimate is calculated by averaging the mean and L2 norm squared across 10,000 samples. As the target distribution is a standard multivariate normal of dimension d, we expect the L2 norm squared to be equal to dimension d, and the mean to be 0. This does not prove the convergence of the Markov chain but it is a useful diagnostic to show if HMC does not produce the appropriate samples.

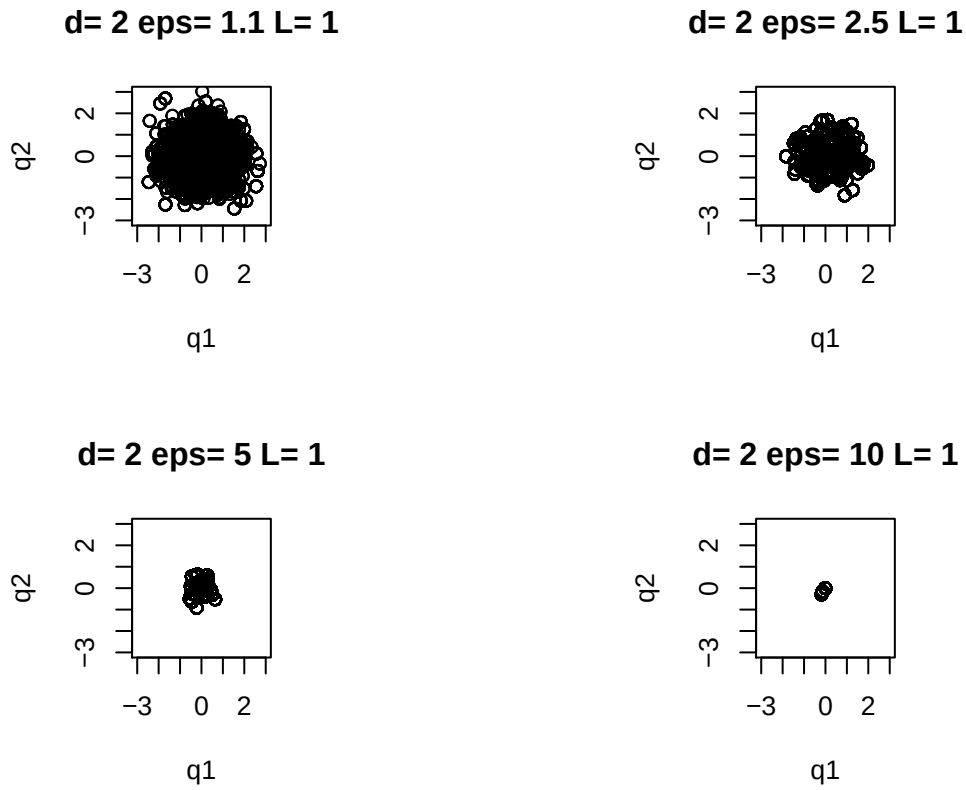


As expected, when dimensions are high, the stepsize ϵ needs to be reduced so that HMC does not go into regions of low density and volume. Indeed, the figure in the first column of the second row shows that if the stepsize ϵ is large, many proposed samples are rejected and HMC becomes inefficient. The figures in the first and second columns of the third row show the extreme case where if ϵ is too large relative to regions of significant volume and density, HMC can become completely stuck and all proposed values will be rejected.

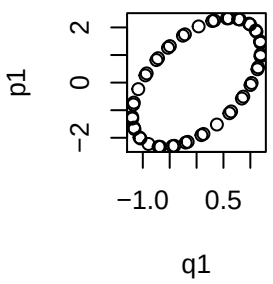
Reducing the stepsize too much can also result in deterioration in performance as HMC becomes a random walk. As shown by the second and third columns of the first row, we can see that both estimates of norms L2 squared and the mean deteriorate when the stepsize is too small.

Below is another numerical simulation that illustrates the behavior of HMC when ϵ is larger than the stability limit of 2. When ϵ is larger than the stability limit, the error of the Hamiltonian becomes large and acceptance probability deteriorates and hence efficiency is reduced. We demonstrate this by showing a 2-dimensional Gaussian target distribution with $10,000$ iterates

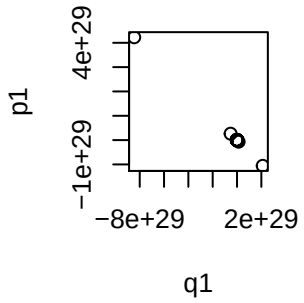
and also its leapfrog trajectory.



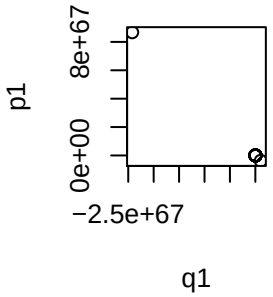
Hamiltonian Trajectory stepsize 1.1



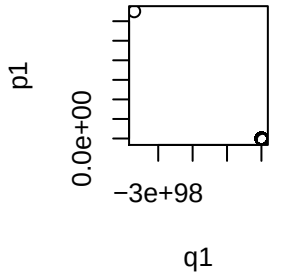
Hamiltonian Trajectory stepsize 2.5



Hamiltonian Trajectory stepsize 5



Hamiltonian Trajectory stepsize 10



Chapter 2 No-U-Turn Sampler (NUTS)

The performance of HMC is highly sensitive to two user-specified parameters: a stepsize ϵ and a desired number of numerical integrations of Hamiltonian equations L. If ϵ is too large, then the simulation will be inaccurate and yield low acceptance rates. If ϵ is too small, then computation will be wasted taking many small steps. If L is too small, then successive samples will be close to one another, resulting in undesirable random walk behavior and slow mixing. If L is too large, then HMC will generate trajectories that loop back and retrace their steps.

Selecting L

As noted before, we need an L that is not too small or large. To determine this, NUTS uses a criterion based on the dot product of the current momentum $\tilde{p}$ and distance between proposed position $\tilde{q}$ and current position q. We call this the No U-Turn criterion, (Hoffman & Gelman, 2014). To see why this would help, we note that, due to chain rule:

$$\begin{aligned}(\tilde{q}-q).\tilde{p} &= (\tilde{q}-q).\frac{d}{dt}(\tilde{q}-q) \\ &= \frac{d}{dt}\frac{(\tilde{q}-q).(\tilde{q}-q)}{2},\end{aligned}$$

From this equation, we can see that the dot product $(\tilde{q}-q).\tilde{p}$ is equal to the derivative with respect to time of the half squared distance of the initial position q and current position $\tilde{q}$. With this, we will be able to know when the trajectory starts to make a U-turn as the squared distance will start to decrease then, causing the derivative of the half squared distance to become negative. Thus a possible algorithm to determine L is to increase the number of leapfrog steps L until this quantity becomes negative, (Hoffman & Gelman, 2014). We detail the simplified algorithm and its derivation below.

Algorithm Overview

NUTS begins with a momentum refresh step and a slice sampling step by augmenting the model $P(q,p) \propto \exp(\mathcal{L}(q) - \frac{1}{2}p.p)$ with a slice variable u with

conditional distribution $P(u|q,p) = \mathcal{U}(u; [0, \exp(\mathcal{L}(q) - \frac{1}{2}p.p])$. This slice sampling step also renders the conditional distribution $P(q,p|u) = \mathcal{U}(q,p; \{q',p' : \exp(\mathcal{L}(q') - \frac{1}{2}p.p) \geq u\})$.

After sampling $u|q,p$, NUTS selects L by randomly simulating Hamiltonian dynamics forward or backwards in time using the leapfrog method, starting with $L = 1$ and doubling the number of leapfrog steps L each time. This doubling process builds a balanced binary tree whose leaf nodes correspond to position-momentum states and stops when the No U-Turn criterion is satisfied by the leftmost and rightmost nodes of any balanced subtree of the overall binary tree. We define $\mathcal{B}$ as the set of all position-momentum states that this random doubling process traces out. A position-momentum pair is then uniformly sampled from $\mathcal{B}$ in a way that leads to a certain detailed balance condition being satisfied. And we define this subset of candidate position-momentum states that satisfy detailed balance as $\mathcal{C}$. Finally, we note that the process for building $\mathcal{B}$ and $\mathcal{C}$ given u, q, p is random and defines a conditional distribution $P(\mathcal{B}, \mathcal{C}|q,p,u)$.

To ensure that the candidate position-momentum states satisfy detailed balance, we must put in place the following conditions:

C.1: Elements of $\mathcal{C}$ must be chosen in a way that preserves volume.
C.2: $P((q,p) \in \mathcal{C}|q,p,u) = 1$.
C.3: $P(u \leq \exp\{\mathcal{L}(q) - \frac{1}{2}p.p\}|(q,p) \in \mathcal{C}) = 1$.
C.4: If $(q,p) \in \mathcal{C}$ and $(q',p') \in \mathcal{C}$, then for any $\mathcal{B}$, $P(\mathcal{B}, C|q,p,u) = P(\mathcal{B}, C|q',p',u)$.

In summary, NUTS consists of four steps:

1. Sample $p \sim \mathcal{N}(0, I)$.
2. Sample $u \sim \mathcal{U}([0, \exp(L(q^k) - \frac{1}{2}p.p)])$.
3. Sample $\mathcal{B}, \mathcal{C}$ from the conditional distribution $P(\mathcal{B}, \mathcal{C}|q^k, p, u)$.
4. Sample $q^{k+1}, p^{k+1} \sim Q(q^k, p, \mathcal{C})$, where $Q(q^k, p, \mathcal{C})$ is a transitional kernel that leaves the uniform distribution over $\mathcal{C}$ invariant.

Proof of Invariance for NUTS

To see why this works, we first note that steps 1,2,3 are Gibbs sampling updates since they are sampling full conditionals. We now show below that $p(q, p|u, \mathcal{B}, \mathcal{C})$ is uniform.

$$\begin{aligned} P(q, p|u, \mathcal{B}, \mathcal{C}) &\propto P(\mathcal{B}, \mathcal{C}|q, p, u)P(q, p|u) \\ &\propto P(\mathcal{B}, \mathcal{C}|q, p, u)\mathbb{I}[u \leq \exp(L(q) - \frac{1}{2}p.\, p] \\ &\propto \mathbb{I}[(q, p) \in \mathcal{C}]. \end{aligned}$$

As noted before, the conditional distribution $P(q, p|u)$ is uniform as a result of the slice sampling step. Condition C.1 also allows us to treat the unnormalized conditional density $P(q, p|u)$ as an unnormalized conditional mass function. In the final step of the above equation, condition C.3 which ensures $(q, p) \in \mathcal{C} \implies u \leq \exp(L(q) - \frac{1}{2}p.p)$, makes the $P(q, p|u)$ term equal to one. And condition C.2 and C.4 ensure that $P(\mathcal{B}, \mathcal{C}, q, p, u) \propto \mathbb{I}[(q, p) \in \mathcal{C}]$ because condition C.2 ensures that $(q, p) \in \mathcal{C}$ and condition C.4 ensures that, for any $(q, p) \in \mathcal{C}$, $P(\mathcal{B}, \mathcal{C}, q, p, u)$ will be constant as a function of (q, p).

We have shown that $P(q, p|u, \mathcal{B}, \mathcal{C})$ is uniform and hence we will need the transition kernel in step 4 $Q(q, p, \mathcal{C})$ to leave the uniform distribution invariant. One simple way to ensure invariance is to sample a position-momentum state from $\mathcal{C}$ uniformly. We can easily see that this transition kernel satisfies general balance:

$$\begin{aligned} \sum_{(q,p)\in\mathcal{C}} P(q, p)Q(q', p'|q, p, \mathcal{C}) &= \sum_{(q,p)\in\mathcal{C}} \frac{1}{|\mathcal{C}|} Q(q', p'|q, p, \mathcal{C}) \\ &= \sum_{(q,p)\in\mathcal{C}} \frac{1}{|\mathcal{C}|} \frac{|\mathcal{C}|}{|\mathcal{C}|} \\ &= \frac{1}{|\mathcal{C}|}. \end{aligned}$$

And the transition kernel quantity is zero if $(q', p') \notin \mathcal{C}$ by definition of $\mathcal{C}$. Since steps 1 to 3 are Gibbs updates that leave the distribution invariant and step 4 is also invariant as shown above, we have just proved that NUTS leaves the target invariant.

Construction of Valid Position-Momentum states

We have so far assumed that we are able to sample from $P(\mathcal{B}, \mathcal{C}|q, p, u)$, but let us see now how it is done by NUTS.

Starting from a single initial node, NUTS builds $\mathcal{B}$ by doubling the number of nodes repeatedly by simulating Hamiltonian dynamics for 2^j leapfrog steps L, where j is the number of previous doublings. The doubling can happen either forward or backwards in time and the direction $v_j \sim \mathcal{U}(\{-1, 1\})$ is chosen uniformly.

We continue to double the number of leapfrog steps and thus expand $\mathcal{B}$ until any of the balanced subtree satisfies the No U-Turn criterion. i.e.: if any of q^+, p^+, q^-, p^- associated with the leftmost and rightmost nodes satisfies:

$$(q^+ - q^-).\, p^- < 0 \;\text{ or }\; (q^+ - q^-).\, p^+ < 0\,,$$

as the No-U-Turn criterion suggests that simulating Hamiltonian dynamics any further will cause the trajectory to start make a loop back to the initial node, thus wasting computation. We also stop doubling when any of the states visited satisfy the following inequality:

$$\mathcal{L}(q) - \frac{1}{2} p.p - \log\, u < -\Delta_{max}$$

for some non-negative Δ_{max}. In other words, we stop doubling when the error of the Hamiltonian becomes too large. This results from the leapfrog integrator's inability to conserve the Hamiltonian and choosing a stepsize ϵ that is too large, causing the trajectory to diverge.

After we build $\mathcal{B}$, $\mathcal{C}$ is chosen in a deterministic way from $\mathcal{B}$ such that it satisfies the conditions C.1-C.4 from above.

C.1 is automatically satisfied as leapfrog integrators preserve volume and the elements of $\mathcal{C}$ are traced out using leapfrog integrators. C.2 is satisfied if the initial state is included in $\mathcal{C}$. C.3 is easily satisfied by excluding states that do not satisfy $\exp(\mathcal{L}(q) - \frac{1}{2} p.p) < u$.

For C.4, we note that for any state $(q', p') \in \mathcal{B}$, there is only one series of directions $\{v_0, \ldots, v_j\}$ for which the doubling produces $\mathcal{B}$. And because we choose

$\mathcal{C}$ deterministically from $\mathcal{B}$, we have $P(\mathcal{B}, \mathcal{C}|q, p, u) = 2^{-j} = P(\mathcal{B}, \mathcal{C}|q', p', u)$ if it is able to fully reconstruct $\mathcal{B}$ with the series of directions $\{v_0, \ldots, v_j\}$, or $P(\mathcal{B}, \mathcal{C}|q', p', u) = 0$ when it is unable to. Thus we have to exclude the initial states that are unable to fully construct the set $\mathcal{B}$. Such states are created when the doubling process was stopped due to the new half-tree satisfying one of the two stopping criterions. For condition C.4 to be satisfied, we just have to remove this half-tree. Otherwise, if $\mathcal{B}$ can be constructed fully, then it implies that the doubling process was stopped because the No U-Turn criterion was satisfied for the full tree. And as noted earlier, the probability for constructing $\mathcal{B}$ from these states is 2^{-j} for a $\mathcal{B}$ that contains 2^j states. Thus, C.4 is satisfied.

Thus we have shown the validity of steps 1-4 of NUTS and also shown specifically how NUTS performs steps 3 and 4 of the algorithm. Below, the full NUTS algorithm is detailed.

Algorithm 2: NUTS algorithm

Given $q^0, \mathcal{L}, \epsilon$, L;
for *k=1 to K* **do**
  Resample $p^0 \sim \mathcal{N}(0, I)$;
  Resample $u \sim \mathcal{U}([0, \exp\{\mathcal{L}(q^{k-1} - \frac{1}{2}p^0.p^0\}])$;
  Initialise $q^- = q^{k-1}$, $q^+ = q^{k-1}$, $p^- = p^0$, $p^+ = p^0$, $j = 0, \mathcal{C} = \{(q^{k-1}, p^0)\}, s = 1$;
  while *s=1* **do**
    Choose a direction $v_j \sim \mathcal{U}(\{-1, 1\})$;
    if $v_j = -1$ **then**
      $q^-, p^-, \mathcal{C}', s' \leftarrow$ BuildTree$(q^-, p^- u, v_j, j, \epsilon)$;
    else
      $q^+, p^+, \mathcal{C}', s' \leftarrow$ BuildTree$(q^+, p^+ u, v_j, j, \epsilon)$;
    end
    if *s'=1* **then**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$;
    end
    $s \leftarrow s' \mathbb{I}[(q^+ - q^-).p^- \geq 0] \mathbb{I}[(q^+ - q^-).p^+ \geq 0]$;
    $j \leftarrow j + 1$;
  end
  Sample q^k, p^k, uniformly at random from $\mathcal{C}$;
end

function: BuildTree(q, p, u, j, ϵ)
if $j = 0$ **then**
  Base case - take one leapfrog step in the direction v;
  $q', p' \leftarrow$ Leapfrog$(q, p, v\epsilon)$;
  if $u \leq \exp(\mathcal{L}(q') - \frac{1}{2}p'.p')$ **then**
    $\mathcal{C}' \leftarrow (q', p')$;
  else
    $\mathcal{C}' \leftarrow \emptyset$;
    $s' \leftarrow \mathbb{I}[\mathcal{L}(q') - \frac{1}{2}p'.p' > \log u - \Delta_{max}]$;
    return $q', p', \mathcal{C}', s'$;
  end
else
  Recursion - Build the left and right subtrees;
  $q^-, p^-, q^+, p^+, \mathcal{C}', s' \leftarrow$ BuildTree$(q, p, u, v, j - 1, \epsilon)$;
  if $v = -1$ **then**
    $q^-, p^-, \mathcal{C}'', s'' \leftarrow$ BuildTree$(q^-, p^-, u, v, j - 1, \epsilon)$;
  else
    $q^+, p^+, \mathcal{C}'', s'' \leftarrow$ BuildTree$(q^+, p^+, u, v, j - 1, \epsilon)$;
  end
  $s' \leftarrow s' s'' \mathbb{I}[(q^+ - q^-).p^- \geq 0] \mathbb{I}[(q^+ - q^-).p^+ \geq 0]$;
  $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}''$;
  return $q^-, p^-, q^+, p^+, \mathcal{C}', s'$;
end

Tuning the Stepsize ϵ

NUTS uses a stochastic approximation technique with vanishing adaptations to optimize the stepsize ϵ, where vanishing adaptation is used to ensure the algorithm produces samples that are asymptotically distributed as the target (Andrieu & Thoms, 2008). Specifically, it is a modification of the primal-dual algorithm of (Nesterov, 2009), where the modifications added are used to improve the stability of the algorithm.

Robbins-Monro's Subgradient Method

The motivation for the use of the primal-dual algorithm comes from the subgradient method algorithm of (Robbins & Monro, 1951). Suppose we have a criteria H_k that acts as a proxy for a MCMC tuning parameter x. We then define its expectation $h(x)$ as:

$$h(x) \equiv \mathbb{E}_k[H_k|x] \equiv \lim_{K\to\infty} \frac{1}{K} \sum_{k=0}^{K} \mathbb{E}[H_k|x].$$

A useful proxy H_k that we can use is $H_k = \delta - \alpha_k$, where δ is the desired average acceptance probability. Assuming h is a nondecreasing function of x, defining optimality as $h = 0$, we can then optimize h using the following update scheme for x:

$$x_{k+1} \leftarrow x_k - \eta_k H_k \,,$$

provided that $||H_k||_2 \leq C$, for some constant C, and that the time-step η_k satisfies the divergent-series rule:

$$\sum_k \eta_k = \infty; \quad \sum_k \eta_k^2 < \infty.$$

Since for any $x \in \mathbb{R}^d$ we get:

$$\begin{aligned} ||x - x_{k+1}||_2^2 &= ||x - x_k||_2^2 + 2\eta_k \langle H_k \,, x - x_k\rangle + \eta_k^2 ||H_k||_2^2 \\ &\leq ||x - x_k||_2^2 + 2\eta_k \langle H_k \,, x - x_k\rangle + \eta_k^2 C^2. \end{aligned}$$

Hence, for any convex function f and $x \in \mathbb{R}^d$ with $\frac{||x-x_{k+1}||_2^2+||x-x_0||_2^2}{2} \leq D$, for some constant D:

$$\begin{aligned} f(x) &\geq l_K(x) \\ &:= \frac{\sum_{k=0}^K \eta_k (f(x_k) + \langle H_k \, , x - x_k \rangle)}{\sum_{k=0}^K \eta_k} \\ &\geq \frac{\sum_{k=0}^K \eta_k f(x_K) - \frac{||x-x_{K+1}||_2^2+||x-x_0||_2^2}{2} - \frac{1}{2} C^2 \sum_{k=0}^K \eta_k^2}{\sum_{k=0}^K \eta_k} \\ &\geq \frac{\sum_{k=0}^K \eta_k f(x_k) - D - \frac{1}{2} C^2 \sum_{k=0}^K \eta_k^2}{\sum_{k=0}^K \eta_k}, \end{aligned} \tag{1}$$

where $l_K(x)$ is the lower linear model of the objective function f. The divergent series rule is important because, for $\bar{f}_K = \frac{\sum_{k=0}^K \eta_k f(x_k)}{\sum_{k=0}^K \eta_k}$, $f^* = \min_x (f(x) : \frac{||x-x_{k+1}||_2^2+||x-x_0||_2^2}{2} \leq D)$, $x^* = \operatorname{argmin}_x (f(x) : \frac{||x-x_{k+1}||_2^2+||x-x_0||_2^2}{2} \leq D)$ and a convex function f, we have:

$$\bar{f}_K - f^* \leq \frac{2\,||x^* - x_{K+1}||_2^2 - 2\,||x^* - x_0||_2^2 + C^2 \sum_{k=1}^K \eta_k^2}{\sum_{k=1}^K \eta_k} \to 0.$$

In other words, if we want $\bar{f}_K$ to converge to the optimal value f^*, we need the sum of square terms to not diverge and the denominator to diverge. The divergent series rule is satisfied by $\eta_k = k^{-\kappa}$ for $\kappa \in (0.5, 1]$, since the series $\sum_{k=1}^\infty k^{-\kappa}$ diverges for $\kappa \leq 1$ and converges for $\kappa > 1$. We have therefore proved that the update scheme of Robbins-Monro minimises the convex function f. And if we assume $h(x)$ is convex, the update scheme of Robbins-Monro will be able to optimize the function h.

However, we can notice that from inequalities (1), newer H_k's have smaller weights than the older ones. Because of this, even though asymptotically the update scheme causes $\bar{f}_K$ to converge to the optimal solution, it does so at a slower rate than needed.

Dual Averaging

(Nesterov, 2009) then proposes a different update scheme for the iterates x in his dual averaging paper:

$$x_{K+1} = \underset{x}{\operatorname{argmin}} \frac{1}{K} \sum_{k=1}^{K} [f(x_k) + g_k.(x - x_k)] + \mu_K \frac{||x||^2}{2K}$$
$$= \underset{x}{\operatorname{argmin}} \hat{g}_K.x + \mu_K \frac{||x||^2}{2K},$$

where $\mu_K \in O(\frac{1}{\sqrt{K}}) \to 0$ is a stepsize at iterate K and $\hat{g}_K = \frac{1}{K} \sum_{k=1}^{K} g_k$. Note that the second equation is due to $f(x_k)$ and x_k not being dependent on x. If we differentiate with respect to x and solve for zero, we get the update scheme:

$$x_K = -K \frac{\hat{g}_K}{\mu_K}.$$

(Nesterov, 2009) then shows that for $\hat{x}_K = \frac{1}{K} \sum_{k=1}^{K} x_k$, f convex, this update scheme leads to:

$$f(\hat{x}_k) - f(x^*) \leq \frac{0.5 + \sqrt{2K}}{K} ||x^* - x_0||^2 + \frac{C^2}{2}.$$

And it is clear that $f(\hat{x}_K) - f(x^*) \in O(\frac{1}{\sqrt{K}})$. In addition to giving new iterates x_k more weight, it also converges faster than the Robbins-Monro subgradient method, where it is shown by (Waugh, 2012) to converge with rate $O(\frac{\log K}{\sqrt{K}})$.

Applying Dual Averaging to NUTS

Since solving an unconstrained convex optimization problem is equivalent to finding a zero of a nondecreasing function (the (sub)gradient of the cost function) (Hoffman & Gelman, 2014), we can replace the stochastic gradients g_k with H_k. Assuming we want to find a x such that a non-decreasing function $h(x) \equiv \mathbb{E}_k[H_k|x]$ is zero, NUTS modifies the Nesterov dual averaging algorithm and apply the updates:

$$x_{k+1} \leftarrow \mu - \frac{\sqrt{k}}{\gamma} \frac{1}{k + k_0} \sum_{i=0}^{k} H_i \,; \quad \bar{x}_{k+1} \leftarrow \eta_k x_{k+1} + (1 - \eta_k) \bar{x}_k \,,$$

where μ is a freely chosen point that the iterates x_k are shrunk towards, $\gamma > 0$ is a free parameter that controls the shrinkage to μ, $k_0 \geq 0$ is a parameter

that stabilises the iterations of the algorithm, and $\eta_k \equiv k^{-\kappa}$ is a time-step satisfying the divergent series rule from Robbins-Monro. Note that setting $\kappa < 1$ will give larger weights to recent iterates x_k.

Given that H_k is bounded, we have $x_{k+1} - x_k \in O(-H_k k^{-0.5}) \to 0$. Hence, $\bar{x}_k$ will converge to a value such that $h(\bar{x}_k) = 0$.

Setting $\mu = 0$, $k_0 = 0$, $\gamma = \frac{1}{K}$, we can recover the update scheme from Nesterov's dual averaging update scheme. We also have that for any $\kappa \in (0.5, 1]$, $\bar{x}_k \to \frac{1}{k}\sum_{i=1}^{k} x_k$. Therefore, we can see that the NUTS update scheme is a generalisation of the dual averaging update scheme in (Nesterov, 2009). We follow a common practice and adapt the tunable parameter ϵ in a warm-up phase and then fix it for the rest of the run (Gelman et al., 2013).

Setting a Good Initial Value for ϵ and μ

We want to set a μ that is not too small to avoid long computation times of NUTS. The NUTS paper suggests an algorithm that doubles or halves an initial ϵ until it crosses 0.5 and then setting $\mu = \log(10\epsilon)$. That way, the iterates x_k are shrunk towards larger values. The algorithm for finding the initial value of ϵ and hence μ is as follows:

Algorithm 3: FindReasonableEpsilon Algorithm

function: FindReasonableEpsilon(q)
Initialise $\epsilon = 1, p \sim \mathcal{N}(0, I)$;
Set $q', p' \leftarrow \text{Leapfrog}(q, p, \epsilon)$;
$a \leftarrow 2\mathbb{I}[\frac{\mathcal{L}(q',p')}{\mathcal{L}(q,p)} > 0.5] - 1$;
while $(\frac{\mathcal{L}(q',p')}{\mathcal{L}(q,p)})^a > 2^{-1}$ **do**
  $\epsilon \leftarrow 2^a \epsilon$;
  Set $q', p' \leftarrow \text{Leapfrog}(q, p, \epsilon)$;
end
return ϵ;

Choosing Statistics H_t

Finally, we need to choose a statistics H_k to optimize. The NUTS paper motivates its choice of H_k with an analogy from HMC. One possible but standard approach is to tune ϵ such that the average Metropolis-Hastings acceptance probability is equal to some optimal value δ. It has been shown in (Beskos et al., 2013; Neal & others, 2011) that the optimal value for δ is 0.65. Hence, we can define $h^{HMC}(\epsilon)$ as follows:

$$H_k^{HMC} \equiv \min\Big(1, \frac{\mathcal{L}(q^k, p^k)}{\mathcal{L}(q^{k-1}, p^{k,0})}\Big)\,; \quad h^{HMC}(\epsilon) \equiv \mathbb{E}_k[\delta - H_k^{HMC}\,|\epsilon],$$

where $p^{k,0}$ is the resampled momentum of the kth iteration. Setting $x \equiv \log \epsilon$ and assuming h^{HMC} is a non-increasing function of ϵ, we can apply the NUTS update scheme to optimize h^{HMC} for any $\delta \in (0, 1)$.

The statistics H_k^{NUTS} for NUTS is similar. Unlike HMC which deterministically picks the last position-momentum pair of the trajectory, NUTS has $|\mathcal{B}_k^{final}|$ many position-momentum pairs and picks the proposal uniformly at random. Hence, an intuitive way to define the statistic H_k^{NUTS} is to weigh all the "acceptance probabilities" of the final doubling iteration equally:

$$H_k^{NUTS} \equiv \frac{1}{\mathcal{B}_k^{final}} \sum_{q,p \in \mathcal{B}_k^{final}} \min\Big(1, \frac{\mathcal{L}(q, p)}{\mathcal{L}(q^{k-1}, p^{k,0})}\Big)\,; \quad h_k^{NUTS}(\epsilon) \equiv \mathbb{E}_k[\delta - H_k^{NUTS}|\epsilon].$$

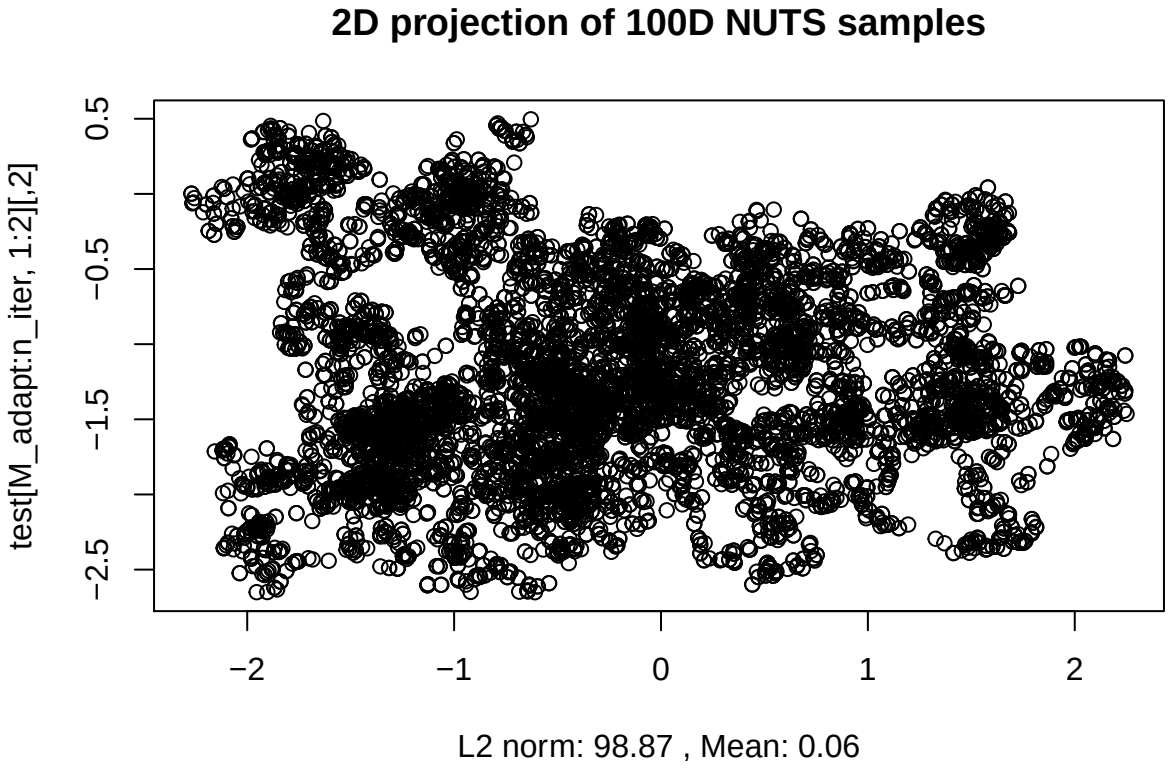
And just like in the HMC case, we can set $x \equiv \log \epsilon$ and apply the dual averaging update scheme to optimize h_k^{NUTS}. Hence, the final algorithm for NUTS is as follows.

Algorithm 4: NUTS Algorithm with Dual Averaging

Given $q^0, \delta, \mathcal{L}$, K, K^{adapt}:;
Set $\epsilon_0 = \text{FindReasonableEpsilon}(q), \mu = \log(10\epsilon_0), \bar{H}_0 = 0, \gamma = 0.05, k_0 = 10, \kappa = 0.75$;
for $k = 1$ *to* K **do**
  Resample $p^0 \sim \mathcal{N}0, I)$;
  Resample $u \sim \mathcal{U}([0, \exp\{\mathcal{L}(q^{k-1} - \frac{1}{2}p^0.p^0\}])$;
  Initialise $q^- = q_{k-1}$, $q^+ = q_{k-1}$, $p^- = p^0$, $p^+ = p^0$, $j = 0, \mathcal{C} = \{(q_{k-1}, p^0)\}, s = 1$;
  while $s = 1$ **do**
    Choose a direction $v_j \sim \mathcal{U}(\{-1, 1\})$;
    if $v_j = -1$ **then**
      $q^-, p^-, \mathcal{C}', s' \leftarrow \text{BuildTree}(q^-, p^- u, v_j, j, \epsilon)$;
    else
      $q^+, p^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(q^+, p^+ u, v_j, j, \epsilon)$;
    end
    if $s' = 1$ **then**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$;
    end
    $s \leftarrow s' \mathbb{I}[(q^+ - q^-).p^- \geq 0] \mathbb{I}[(q^+ - q^-).p^+ \geq 0]$;
    $j \leftarrow j + 1$;
    if $s = 1$ **then**
      $\mathcal{B} \leftarrow \mathcal{B}'$
    end
  end
  Sample q^k, p, uniformly at random from $\mathcal{C}$;
  With probability $\alpha = \frac{1}{|\mathcal{B}|} \sum_{q,p \in \mathcal{B}} \min\Big(1, \frac{p(q,p)}{p(q_{k-1}, p_{k,0})}\Big)$;
  if $k \leq K^{adapt}$ **then**
    Set $\bar{H}_k = (1 - \frac{1}{k+k_0})\bar{H}_{k-1} + \frac{1}{k+k_0}(\delta - \alpha)$;
    Set $\log \epsilon_k = \mu - \frac{k}{\gamma}\bar{H}_k$, $\log \bar{\epsilon}_k = k^{-\kappa} \log \epsilon_k + (1 - k^{-\kappa}) \log \bar{\epsilon}_{k-1}$;
  else
    Set $\epsilon_k = \bar{\epsilon}_{K^{adapt}}$;
  end
end

NUTS Implementation

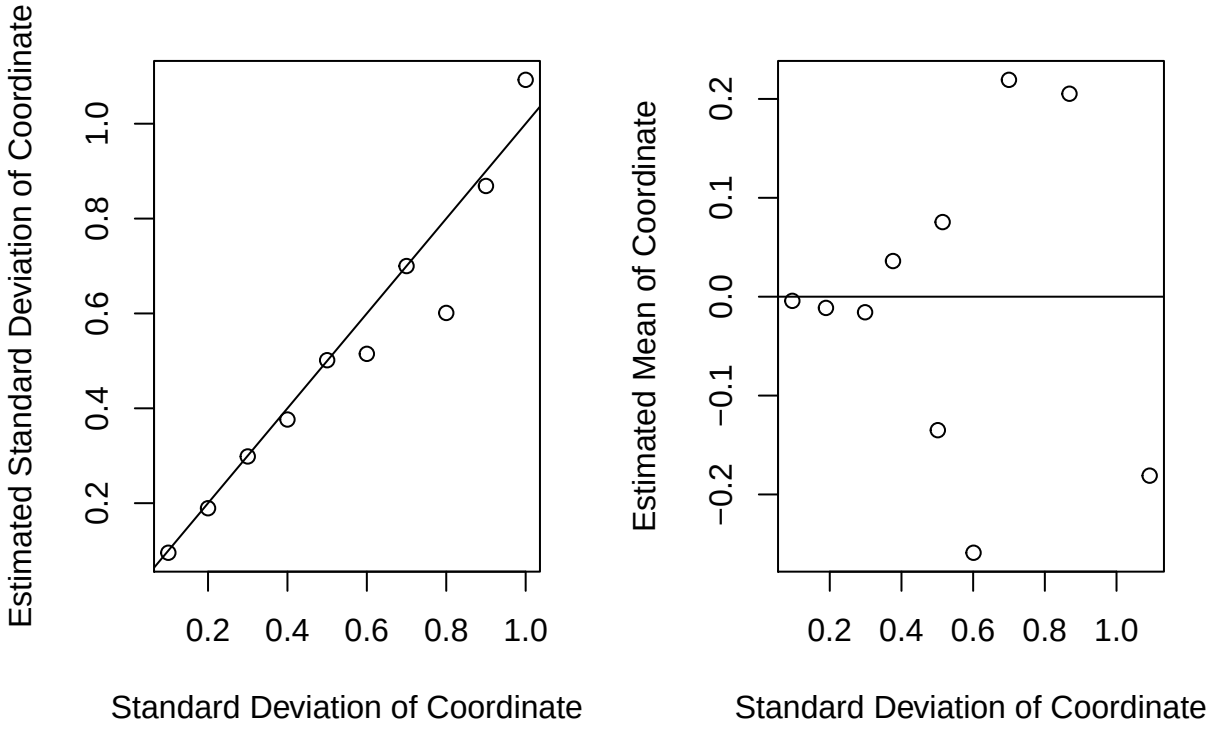
Below is a figure of the 2D projection of the first 2 dimensions of NUTS samples with target distribution of 100-dimensional standard multivariate normal. We use the same diagnostics of L2 norm squared and the mean.



The advantages of NUTS are clear as performance is good and no tuning is required.

Multiscale Independent Gaussian

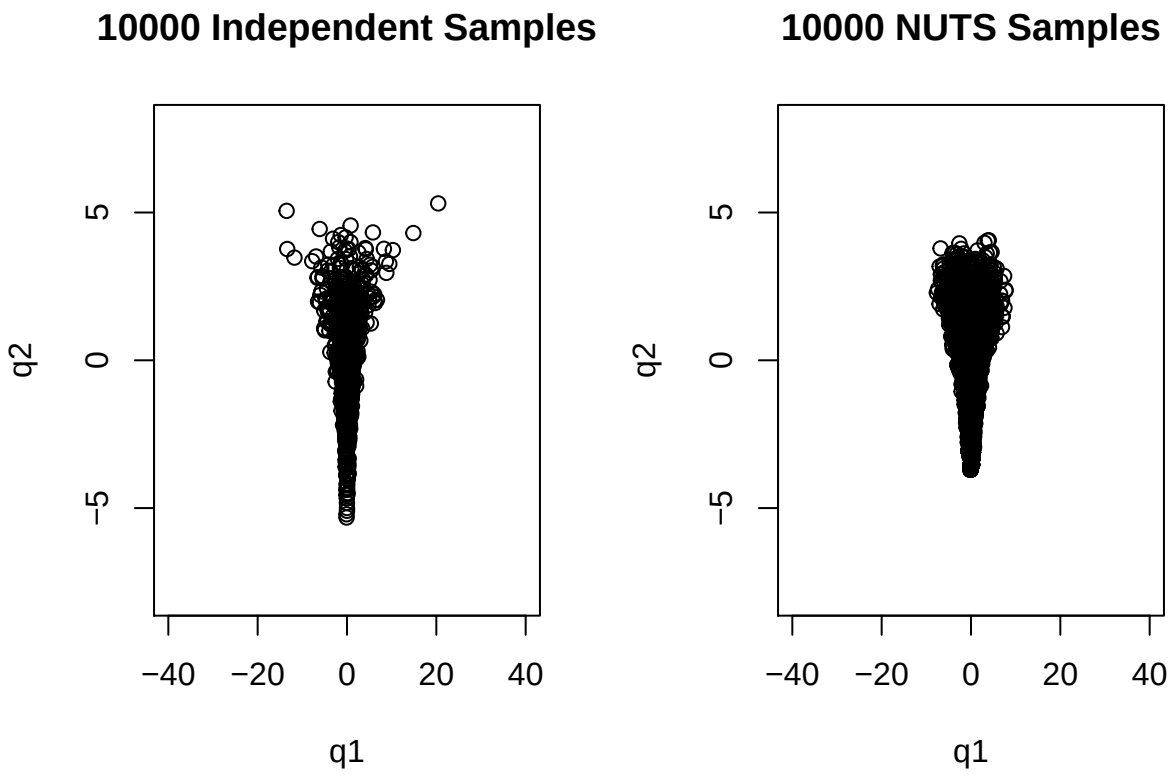
Here, a multiscale Gaussian example is showcased. MHRW is known to have trouble exploring these distributions due to the surface being narrow, but NUTS is able to perform well. The distribution used here is a 10-dimensional multivariate Gaussian distribution with independent variables, means zero, and standard deviations $0.1, 0.2, \ldots, 0.9, 1.0$. An estimate of the mean and standard deviation are compared to the original parameters to illustrate the performance of NUTS. The estimates are calculated from 10000 samples.



Neal's Funnel

Next, NUTS is demonstrated on Neal's funnel which is a distribution that captures the characteristics of typical Bayesian hierarchical models that are known to give many samplers difficulty. Neal's funnel is highly correlated and non-Gaussian. A 2-D plot of the first two dimensions of NUTS is compared with an independent simulation to showcase NUTS' performance. The Neal's funnel distribution over $q \in \mathbb{R}^5$ used for the simulation is as follows:

$$
\begin{aligned}
q_1 &\sim \mathcal{N}(0, 3) \\
q_i &\sim \mathcal{N}(0, \exp^{q_1}),\ 2 \leq i \leq 5.
\end{aligned}
$$



Conclusion

In this thesis, we have chosen the mass matrix M to be the identity matrix I for simplicity. But NUTS can be extended by varying the mass matrix M. When M approximates the covariance of $\mathcal{L}(q)$, the parameter space is transformed (Neal & others, 2011), and the effects of strong correlations and multiple scaling can be reduced. However, care must be taken to ensure that the no-U-turn condition is computed on the transformed parameter space and not the original.

Instead of fixing the mass matrix M, we can also let it depend on the position q. (Girolami & Calderhead, 2011) introduced the concept of Riemannian Manifold Hamiltonian Monte Carlo (RMHWMC) that allows this by simulating Hamiltonian dynamics in Riemannian rather than Euclidean spaces. By adapting the mass matrix based on the current position, NUTS can potentially be made more efficient.

In summary, NUTS is a variant of HMC that does not require the hand tun-

ing of parameters yet retains HMC's ability to generate samples efficiently, allowing its users to focus on building better models and not worry about hand tuning parameters. More challenging computational problems can potentially be solved by incorporating more sophisticated geometric techniques as the one suggested above, which is a topic for further discussion.

Appendix

HMC Code

```
joint_log_density = function(theta, r, f,
    M_diag) {
    f(theta) - 0.5 * sum(r^2/M_diag)
}

leapfrog = function(theta, r, eps, grad_f) {
    r_tilde <- r + 0.5 * eps * grad_f(theta)
    theta_tilde <- theta + eps * r_tilde/M_diag
    r_tilde <- r_tilde + 0.5 * eps * grad_f(theta_tilde)
    list(theta = theta_tilde, r = r_tilde)
}

HMC_function = function(theta_init, eps = 1,
    L = 1, f, grad_f, M_diag = NULL, n_iter = 100) {

    theta_m = theta_init
    theta_tilde = theta_init
    theta_trace = matrix(0, nrow = n_iter,
        ncol = length(theta_init))

    M_diag <- rep(1, length(theta_init))

    for (iter in 1:n_iter) {
        r0 = rnorm(length(theta_m))
        r_tilde = r0
        theta_tilde = theta_m

        # leapfrog
        for (i in 1:L) {
            results = leapfrog(theta_tilde,
                r_tilde, eps, grad_f)
```

```
            theta_tilde = results$theta
            r_tilde = results$r
        }

        log_ratio = joint_log_density(t(theta_tilde),
            r_tilde, f, M_diag) - joint_log_density(t(theta_m),
            r0, f, M_diag)
        if (runif(1) < exp(log_ratio)) {
            theta_m = theta_tilde
        }

        theta_trace[iter, ] = theta_m


        # if(iter%%1000==0){print(iter)}
    }
    theta_trace
}
```

HMC Figure code

```
# 2d plot code

library(mvtnorm)
set.seed(1)

L_list = c(1, 5, 10)
eps_list = c(1, 0.1, 0.01)
d_list = c(1, 10, 100)

par(mfrow = c(3, 3))
for (i in 1:3) {
    for (j in 1:3) {
        d = d_list[i]
        eps = eps_list[j]
```

```
        L = L_list[j]
        theta_init = rep(0, times = d)
        n_iter = 10000
        res = HMC_function(theta_init, eps = eps,
            L = L, f, grad_f, n_iter = n_iter)
        hist(res[1:n_iter, 1], main = paste("d=",
            d, "eps=", eps, "L=", L), xlab = paste("L2 norm squared:",
            round(sum(res[1:n_iter, 1:d]^2)/n_iter,
                digits = 2), ", Mean:", round(sum(res)/n_iter,
                digits = 2)))

    }
}

# trajectory plot code
d = 2
M_diag = 1
f = function(theta) {
    return(dmvnorm(theta, log = TRUE))
}
grad_f = function(theta) {
    return(diag(d) %*% theta)
}

leapfrog = function(theta, r, eps, grad_f) {
    r_tilde <- r - 0.5 * eps * grad_f(theta)
    theta_tilde <- theta + eps * r_tilde
    r_tilde <- r_tilde - 0.5 * eps * grad_f(theta_tilde)
    list(theta = theta_tilde, r = r_tilde)
}

par(mfrow = c(2, 2), pty = "s")
eps_list = c(1.1, 2.5, 5, 10)
d = 2
L = 50
```

```
for (i in 1:4) {
    eps = eps_list[i]
    theta_trace = c()
    theta_trace2 = c()
    theta_tilde = c(0, 2)
    r_tilde = c(1, 1)
    for (i in 1:L) {
        results = leapfrog(theta_tilde, r_tilde,
            eps, grad_f)
        theta_tilde = results$theta
        r_tilde = results$r
        theta_trace = c(theta_trace, theta_tilde[1])
        theta_trace2 = c(theta_trace2, theta_tilde[2])
    }
    plot(theta_trace, theta_trace2, main = paste("Hamiltonian Trajectory
        eps), xlab = "q1", ylab = "p1", cex.main = 1)


}
```

Multiscale Gaussian Code

```
d = 10
M_adapt = 1000
n_iter = 10000

# setting up the parameters
theta = rep(0, times = d)
mu = rep(0, d)
diag_vector = seq(from = 0.1, to = 1, by = 0.1)
theta = rep(0, d)
sigma = diag(diag_vector^2)
sigma_inverse = chol2inv(chol(sigma))
```

```r
grad_f = function(theta) {
    return(sigma_inverse %*% (theta))
}
f = function(theta) {
    return(dmvnorm(c(theta), sigma = sigma,
        log = TRUE))
}
M_diag <- rep(1, length(theta))
eps = 1
verbose = TRUE

test = NUTS(theta, f, grad_f, n_iter = n_iter,
    M_adapt = M_adapt)
theta_trace = test$theta_trace

estimated_diag_vector = c()
for (i in 1:10) {
    estimated_diag_vector = c(estimated_diag_vector,
        sqrt(var(theta_trace[1:n_iter, i])))
}

estimated_mean = c()
for (i in 1:10) {
    estimated_mean = c(estimated_mean, mean(theta_trace[1:n_iter,
        i]))
}

par(mfrow = c(1, 2))  # 3 rows and 2 columns

plot(diag_vector, estimated_diag_vector,
    xlab = "Standard Deviation of Coordinate",
    ylab = "Estimated Standard Deviation of Coordinate")
abline(0, 1)

plot(estimated_diag_vector, estimated_mean,
```

```
    xlab = "Standard Deviation of Coordinate",
    ylab = "Estimated Mean of Coordinate")
abline(0, 0)
```

Neal's Funnel code

```
opts_chunk$set(tidy.opts = list(width.cutoff = 40),
    tidy = TRUE)
library(adr)
library(mvtnorm)

f = function(theta) {

    return(log(dnorm(theta[2], mean = 0,
        sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[3], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[4], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[5], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[6], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[7], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[8], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[9], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[10], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[11], mean = 0, sd = sqrt((exp(theta[1]/2))^2)) *
        dnorm(theta[1], mean = 0, sd = sqrt(3))))

}

grad_f = function(theta) {

    sink("file")
    temp = adrDiffFor(f, list(theta))
    sink()

    # temp = quiet(adrDiffFor(f,
```

```
    # list(theta)))

    return(c(temp$J))
}

set.seed(1)
v = rnorm(1, sd = sqrt(3))
theta = rnorm(5, sd = sqrt(exp(v/2)^2))
theta = c(v, theta)

# independent simulation
theta_plot = matrix(0, 1000, 11)
set.seed(1)
for (i in 1:1000) {
    v = rnorm(1, sd = sqrt(3))
    theta = rnorm(10, sd = sqrt((exp(v/2)))^2)
    theta = c(v, theta)
    theta_plot[i, ] = theta
}

# NUTS
n = 10
delta = 0.5
max_treedepth = 5
M_adapt = 1000
n_iter = 10000

v_test = rnorm(1, sd = sqrt(3))
theta = rnorm(n, sd = sqrt((exp(v_test/2)))^2)
theta = c(v_test, theta)

set.seed(1)
test = NUTS(theta, f, grad_f, n_iter, M_adapt,
    max_treedepth)
```

```r
par(mfrow = c(1, 2))  # 3 rows and 2 columns
plot(theta_plot[, 2:1], main = "1000 Independent Samples")
plot(test$theta_trace[1:n_iter, 2:1], main = "1000 NUTS Samples")
```

NUTS Code

```r
leapfrog_step = function(theta, r, eps, grad_f,
    M_diag) {
    r_tilde <- r + 0.5 * eps * grad_f(theta)
    theta_tilde <- theta + eps * r_tilde/M_diag
    r_tilde <- r_tilde + 0.5 * eps * grad_f(theta_tilde)
    list(theta = theta_tilde, r = r_tilde)
}

joint_log_density = function(theta, r, f,
    M_diag) {
    f(theta) - 0.5 * sum(r^2/M_diag)
}

find_reasonable_epsilon = function(theta,
    f, grad_f, M_diag, eps = 1, verbose = TRUE) {
    r <- rnorm(length(theta), 0, sqrt(M_diag))
    proposed <- leapfrog_step(theta, r, eps,
        grad_f, M_diag)
    new_theta = t(proposed$theta)
    new_r = t(proposed$r)

    log_ratio <- joint_log_density(new_theta,
        new_r, f, M_diag) - joint_log_density(theta,
        r, f, M_diag)
    alpha <- ifelse(exp(log_ratio) > 0.5,
        1, -1)
    if (is.nan(alpha))
        alpha <- -1
    count <- 1
```

```
    while (is.nan(log_ratio) || alpha * log_ratio >
        (-alpha) * log(2)) {
        eps <- 2^alpha * eps
        proposed <- leapfrog_step(theta,
            r, eps, grad_f, M_diag)
        new_theta = t(proposed$theta)
        new_r = t(proposed$r)
        log_ratio <- joint_log_density(new_theta,
            new_r, f, M_diag) - joint_log_density(theta,
            r, f, M_diag)
        count <- count + 1
        # if(count > 100) { stop('Could not find
        # reasonable epsilon in 100 iterations!')
        # }
    }
    # if(verbose) message('Reasonable epsilon
    # = ', eps, ' found after ', count, '
    # steps')
    eps
}

check_NUTS = function(s, theta_plus, theta_minus,
    r_plus, r_minus) {
    if (is.na(s))
        return(0)
    condition1 <- crossprod(theta_plus -
        theta_minus, r_minus) >= 0
    condition2 <- crossprod(theta_plus -
        theta_minus, r_plus) >= 0
    s && condition1 && condition2
}

build_tree = function(theta, r, u, v, j,
    eps, theta0, r0, f, grad_f, M_diag, Delta_max = 1000) {
    if (j == 0) {
```

```
        proposed <- leapfrog_step(theta,
            r, v * eps, grad_f, M_diag)
        theta <- t(proposed$theta)
        r <- t(proposed$r)
        log_prob <- joint_log_density(theta,
            r, f, M_diag)
        log_prob0 <- joint_log_density(theta0,
            r0, f, M_diag)
        n <- (log(u) <= log_prob)
        s <- (log(u) < Delta_max + log_prob)
        alpha <- min(1, exp(log_prob - log_prob0))
        if (is.nan(alpha))
            stop()
        if (is.na(s) || is.nan(s)) {
            s <- 0
        }
        if (is.na(n) || is.nan(n)) {
            n <- 0
        }
        return(list(theta_minus = theta,
            theta_plus = theta, theta = theta,
            r_minus = r, r_plus = r, s = s,
            n = n, alpha = alpha, n_alpha = 1))
    } else {
        obj0 <- build_tree(c(theta), r, u,
            v, j - 1, eps, c(theta0), r0,
            f, grad_f, M_diag)
        theta_minus <- (obj0$theta_minus)
        r_minus <- (obj0$r_minus)
        theta_plus <- (obj0$theta_plus)
        r_plus <- (obj0$r_plus)
        theta <- (obj0$theta)

        if (obj0$s == 1) {
            if (v == -1) {
```

```
            obj1 <- build_tree(c(obj0$theta_minus),
              c(obj0$r_minus), u, v,
              j - 1, eps, c(theta0),
              c(r0), f, grad_f, M_diag)
            theta_minus <- (obj1$theta_minus)
            r_minus <- (obj1$r_minus)
        } else {
            obj1 <- build_tree(c(obj0$theta_plus),
              c(obj0$r_plus), u, v, j -
                1, eps, c(theta0), c(r0),
              f, grad_f, M_diag)
            theta_plus <- (obj1$theta_plus)
            r_plus <- (obj1$r_plus)
        }
        n <- obj0$n + obj1$n
        if (n != 0) {
            prob <- obj1$n/n
            if (runif(1) < prob) {
              theta <- t(obj1$theta)
            }
        }
        s <- check_NUTS(obj1$s, theta_plus,
            theta_minus, r_plus, r_minus)
        alpha <- obj0$alpha + obj1$alpha
        n_alpha <- obj0$n_alpha + obj1$n_alpha

    } else {
        n <- obj0$n
        s <- obj0$s
        alpha <- obj0$alpha
        n_alpha <- obj0$n_alpha
    }
    if (is.na(s) || is.nan(s)) {
        s <- 0
    }
```

```
        if (is.na(n) || is.nan(n)) {
            n <- 0
        }
        return(list(theta_minus = theta_minus,
            theta_plus = theta_plus, theta = theta,
            r_minus = r_minus, r_plus = r_plus,
            s = s, n = n, alpha = alpha,
            n_alpha = n_alpha))
    }
}

NUTS_one_step <- function(theta, iter, f,
    grad_f, par_list, delta = 0.5, max_treedepth = 10,
    eps = 1, verbose = TRUE) {
    kappa <- 0.75  #same parameter values in paper
    t0 <- 10
    gamma <- 0.05
    M_adapt <- par_list$M_adapt
    if (is.null(par_list$M_diag)) {
        M_diag <- rep(1, length(theta))  #default 1 diagonals for M matr
    } else {
        M_diag <- par_list$M_diag
    }

    if (iter == 1) {
        eps <- find_reasonable_epsilon(theta,
            f, grad_f, M_diag, eps = eps,
            verbose = verbose)
        mu <- log(10 * eps)
        H <- 0
        eps_bar <- 1
    } else {
        eps <- par_list$eps
        eps_bar <- par_list$eps_bar
        H <- par_list$H
```

```
    mu <- par_list$mu
}

r0 <- rnorm(length(theta), 0, sqrt(M_diag))
u <- runif(1, 0, exp(f(theta) - 0.5 *
    sum(r0^2/M_diag)))
if (is.nan(u)) {
    warning("NUTS: sampled slice u is NaN")
    u <- runif(1, 0, 1e+05)
}
theta_minus <- theta
theta_plus <- theta
r_minus <- r0
r_plus <- r0
j = 0
n = 1
s = 1
if (iter > M_adapt) {
    eps <- runif(1, 0.9 * eps_bar, 1.1 *
        eps_bar)  #random jitter
}
while (s == 1) {
    # choose direction {-1, 1}
    direction <- sample(c(-1, 1), 1)
    if (direction == -1) {
        temp <- build_tree(c(theta_minus),
            c(r_minus), u, direction,
            j, eps, c(theta), r0, f,
            grad_f, M_diag)
        theta_minus <- t(t(temp$theta_minus))
        r_minus <- t(t(temp$r_minus))
    } else {
        temp <- build_tree(c(theta_plus),
            c(r_plus), u, direction,
            j, eps, c(theta), r0, f,
```

```
            grad_f, M_diag)
        theta_plus <- t(t(temp$theta_plus))
        r_plus <- t(t(temp$r_plus))
    }
    if (is.nan(temp$s))
        temp$s <- 0
    if (temp$s == 1) {
        if (runif(1) < temp$n/n) {
            theta <- t(t(temp$theta))
        }
    }
    n <- n + temp$n
    s <- check_NUTS(temp$s, c(theta_plus),
        c(theta_minus), c(r_plus), c(r_minus))
    j <- j + 1
    if (j > max_treedepth) {
        warning("NUTS: Reached max tree depth")
        break
    }
}  #endwhile

if (iter <= M_adapt) {
    H <- (1 - 1/(iter + t0)) * H + 1/(iter +
        t0) * (delta - temp$alpha/temp$n_alpha)
    log_eps <- mu - sqrt(iter)/gamma *
        H
    eps_bar <- exp(iter^(-kappa) * log_eps +
        (1 - iter^(-kappa)) * log(eps_bar))
    eps <- exp(log_eps)
} else {
    eps <- eps_bar
}

# print(eps) if(iter%%100 ==
# 0){print(iter)}
```

```
    return(list(theta = theta, pars = list(eps = eps,
        eps_bar = eps_bar, H = H, mu = mu,
        M_adapt = M_adapt, M_diag = M_diag)))
}


NUTS <- function(theta, f, grad_f, n_iter,
    M_diag = NULL, M_adapt = 50, delta = 0.5,
    max_treedepth = 10, eps = 1, verbose = TRUE) {
    theta_trace <- matrix(0, n_iter, length(theta))
    par_list <- list(M_adapt = M_adapt)
    for (iter in 1:n_iter) {
        nuts <- NUTS_one_step(theta, iter,
            f, grad_f, par_list, delta = delta,
            max_treedepth = max_treedepth,
            eps = eps, verbose = verbose)
        theta <- t(nuts$theta)
        par_list <- nuts$pars
        theta_trace[iter, ] <- theta
    }
    theta_trace
}
```

References

Andrieu, C., & Thoms, J. (2008). A tutorial on adaptive MCMC. *Statistics and Computing*, *18*(4), 343–373.

Beskos, A., Pillai, N., Roberts, G., Sanz-Serna, J.-M., Stuart, A., & others. (2013). Optimal tuning of the hybrid Monte Carlo algorithm. *Bernoulli*, *19*(5A), 1501–1534.

Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. *arXiv Preprint arXiv:1701.02434*.

Bou-Rabee, N., Sanz-Serna, J. M., & others. (2017). Randomized Hamiltonian Monte Carlo. *The Annals of Applied Probability*, *27*(4), 2159–2194.

Brooks, S., Gelman, A., Jones, G., & Meng, X.-L. (2011). *Handbook of Markov chain Monte Carlo*. CRC press.

Calvo, M. P., Sanz-Alonso, D., & Sanz-Serna, J. (2019). HMC: avoiding rejections by not using leapfrog and some results on the acceptance rate. *arXiv Preprint arXiv:1912.03253*.

Duane, S., Kennedy, A. D., Pendleton, B. J., & Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, *195*(2), 216–222.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian data analysis*. CRC press.

Girolami, M., & Calderhead, B. (2011). Riemann Manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, *73*(2), 123–214.

Hairer, E., Lubich, C., & Wanner, G. (2006). *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations* (Vol. 31). Springer Science & Business Media.

Hastings, W. K. (1970). *Monte Carlo sampling methods using Markov chains and their applications*.

Hoffman, M. D., & Gelman, A. (2014). The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.*, *15*(1), 1593–1623.

Lan, S., Stathopoulos, V., Shahbaba, B., & Girolami, M. (2012). Lagrangian Dynamical Monte Carlo. *arXiv Preprint arXiv:1211.3759*.

Leimkuhler, B., & Reich, S. (2004). *Simulating Hamiltonian dynamics* (Vol. 14). Cambridge university press.

Livingstone, S., Betancourt, M., Byrne, S., Girolami, M., & others. (2019). On the geometric ergodicity of Hamiltonian Monte Carlo. *Bernoulli*, *25*(4A), 3109–3138.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, *21*(6), 1087–1092.

Neal, R. M. (1993). *Probabilistic inference using Markov chain Monte Carlo methods*. Department of Computer Science, University of Toronto Toronto, Ontario, Canada.

Neal, R. M., & others. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, *2*(11), 2.

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems. *Mathematical Programming*, *120*(1), 221–259.

Pourzanjani, A. A., & Petzold, L. R. (2019). Implicit Hamiltonian Monte Carlo for sampling multiscale distributions. *arXiv Preprint arXiv:1911.05754*.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 400–407.

Robert, C., & Casella, G. (2013). *Monte Carlo statistical methods*. Springer Science & Business Media.

Roberts, G. O., Rosenthal, J. S., & others. (2004). General state space Markov chains and MCMC algorithms. *Probability Surveys*, *1*, 20–71.

Waugh, K. (2012). *Advanced Topic: Dual Averaging*.