Journal of Machine Learning Research 15 (2014) 1593-1623 Submitted 11/11; Revised 10/13; Published 4/14

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

Matthew D. Hoffman MATHOFFM@ADOBE.COM
Adobe Research
601 Townsend St.
San Francisco, CA 94110, USA

Andrew Gelman GELMAN@STAT.COLUMBIA.EDU
Departments of Statistics and Political Science
Columbia University
New York, NY 10027, USA

Editor: Anthanasios Kottas

Abstract

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) algorithm that avoids the random walk behavior and sensitivity to correlated parameters that plague many MCMC methods by taking a series of steps informed by first-order gradient information. These features allow it to converge to high-dimensional target distributions much more quickly than simpler methods such as random walk Metropolis or Gibbs sampling. However, HMC's performance is highly sensitive to two user-specified parameters: a step size ϵ and a desired number of steps L. In particular, if L is too small then the algorithm exhibits undesirable random walk behavior, while if L is too large the algorithm wastes computation. We introduce the No-U-Turn Sampler (NUTS), an extension to HMC that eliminates the need to set a number of steps L. NUTS uses a recursive algorithm to build a set of likely candidate points that spans a wide swath of the target distribution, stopping automatically when it starts to double back and retrace its steps. Empirically, NUTS performs at least as efficiently as (and sometimes more efficiently than) a well tuned standard HMC method, without requiring user intervention or costly tuning runs. We also derive a method for adapting the step size parameter ϵ on the fly based on primal-dual averaging. NUTS can thus be used with no hand-tuning at all, making it suitable for applications such as BUGS-style automatic inference engines that require efficient "turnkey" samplers.

Keywords: Markov chain Monte Carlo, Hamiltonian Monte Carlo, Bayesian inference, adaptive Monte Carlo, dual averaging

1. Introduction

Hierarchical Bayesian models are a mainstay of the machine learning and statistics communities. Exact posterior inference in such models is rarely tractable, however, and so researchers and practitioners must usually resort to approximate statistical inference methods. Deterministic approximate inference algorithms (for example, those reviewed by Wainwright and Jordan 2008) can be efficient, but introduce bias and can be difficult to apply to some models. Rather than computing a deterministic approximation to a target posterior (or other) distribution, Markov chain Monte Carlo (MCMC) methods offer schemes for drawing a series of correlated samples that will converge in distribution to the target distribution (Neal, 1993). MCMC methods are sometimes less efficient than their deterministic counterparts, but are more generally applicable and are asymptotically unbiased.

Not all MCMC algorithms are created equal. For complicated models with many parameters, simple methods such as random-walk Metropolis (Metropolis et al., 1953) and Gibbs sampling (Geman and Geman, 1984) may require an unacceptably long time to converge to the target distribution. This is in large part due to the tendency of these methods to explore parameter space via inefficient random walks (Neal, 1993). When model parameters are continuous rather than discrete, Hamiltonian Monte Carlo (HMC), also known as hybrid Monte Carlo, is able to suppress such random walk behavior by means of a clever auxiliary variable scheme that transforms the problem of sampling from a target distribution into the problem of simulating Hamiltonian dynamics (Neal, 2011). The cost of HMC per independent sample from a target distribution of dimension D is roughly $O(D^{5/4})$, which stands in sharp contrast with the $O(D^2)$ cost of random-walk Metropolis (Creutz, 1988).

HMC's increased efficiency comes at a price. First, HMC requires the gradient of the log-posterior. Computing the gradient for a complex model is at best tedious and at worst impossible, but this requirement can be made less onerous by using automatic differentiation (Griewank and Walther, 2008). Second, HMC requires that the user specify at least two parameters: a step size ϵ and a number of steps L for which to run a simulated Hamiltonian system. A poor choice of either of these parameters will result in a dramatic drop in HMC's efficiency. Methods from the adaptive MCMC literature (see Andrieu and Thoms 2008 for a review) can be used to tune ϵ on the fly, but setting L typically requires one or more costly tuning runs, as well as the expertise to interpret the results of those tuning runs. This hurdle limits the more widespread use of HMC, and makes it challenging to incorporate HMC into a general-purpose inference engine such as BUGS (Gilks and Spiegelhalter, 1992), JAGS (`http://mcmc-jags.sourceforge.net`), Infer.NET (Minka et al.), HBC (Daume III, 2007), or PyMC (Patil et al., 2010).

The main contribution of this paper is the No-U-Turn Sampler (NUTS), an MCMC algorithm that closely resembles HMC, but eliminates the need to choose the problematic number-of-steps parameter L. We also provide a new dual averaging (Nesterov, 2009) scheme for automatically tuning the step size parameter ϵ in both HMC and NUTS, making it possible to run NUTS with no hand-tuning at all. We will show that the tuning-free version of NUTS samples as efficiently as (and sometimes more efficiently than) HMC, even ignoring the cost of finding optimal tuning parameters for HMC. Thus, NUTS brings the efficiency of HMC to users (and generic inference systems) that are unable or disinclined to spend time tweaking an MCMC algorithm.

Our algorithm has been implemented in C++ as part of the new open-source Bayesian inference package, Stan (Stan Development Team, 2013). Matlab code implementing the algorithms, along with Stan code for models used in our simulation study, are also available at `http://www.cs.princeton.edu/~mdhoffma/`.

2. Hamiltonian Monte Carlo

In Hamiltonian Monte Carlo (HMC) (Neal, 2011, 1993; Duane et al., 1987), we introduce an auxiliary momentum variable r_d for each model variable θ_d. In the usual implementation,

Algorithm 1 Hamiltonian Monte Carlo

```
Given $\theta^0$, $\epsilon$, $L$, $\mathcal{L}, M$:
for $m = 1$ to $M$ do
    Sample $r^0 \sim \mathcal{N}(0, I)$.
    Set $\theta^m \leftarrow \theta^{m-1}, \tilde{\theta} \leftarrow \theta^{m-1}, \tilde{r} \leftarrow r^0$.
    for $i = 1$ to $L$ do
        Set $\tilde{\theta}, \tilde{r} \leftarrow$ Leapfrog$(\tilde{\theta}, \tilde{r}, \epsilon)$.
    end for
    With probability $\alpha = \min\left\{1, \frac{\exp\{\mathcal{L}(\tilde{\theta}) - \frac{1}{2}\tilde{r}\cdot\tilde{r}\}}{\exp\{\mathcal{L}(\theta^{m-1}) - \frac{1}{2}r^0\cdot r^0\}}\right\}$, set $\theta^m \leftarrow \tilde{\theta}$, $r^m \leftarrow -\tilde{r}$.
end for

function Leapfrog$(\theta, r, \epsilon)$
Set $\tilde{r} \leftarrow r + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$.
Set $\tilde{\theta} \leftarrow \theta + \epsilon \tilde{r}$.
Set $\tilde{r} \leftarrow \tilde{r} + (\epsilon/2)\nabla_\theta \mathcal{L}(\tilde{\theta})$.
return $\tilde{\theta}, \tilde{r}$.
```

these momentum variables are drawn independently from the standard normal distribution, yielding the (unnormalized) joint density

$$p(\theta, r) \propto \exp\{\mathcal{L}(\theta) - \tfrac{1}{2} r \cdot r\},$$

where $\mathcal{L}$ is the logarithm of the joint density of the variables of interest θ (up to a normalizing constant) and $x \cdot y$ denotes the inner product of the vectors x and y. We can interpret this augmented model in physical terms as a fictitious Hamiltonian system where θ denotes a particle's position in D-dimensional space, r_d denotes the momentum of that particle in the dth dimension, $\mathcal{L}$ is a position-dependent negative potential energy function, $\frac{1}{2} r \cdot r$ is the kinetic energy of the particle, and $\log p(\theta, r)$ is the negative energy of the particle. We can simulate the evolution over time of the Hamiltonian dynamics of this system via the Störmer-Verlet ("leapfrog") integrator, which proceeds according to the updates

$$r^{t+\epsilon/2} = r^t + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta^t); \quad \theta^{t+\epsilon} = \theta^t + \epsilon r^{t+\epsilon/2}; \quad r^{t+\epsilon} = r^{t+\epsilon/2} + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta^{t+\epsilon}),$$

where r^t and θ^t denote the values of the momentum and position variables r and θ at time t and ∇_θ denotes the gradient with respect to θ. Since the update for each coordinate depends only on the other coordinates, the leapfrog updates are volume-preserving—that is, the volume of a region remains unchanged after mapping each point in that region to a new point via the leapfrog integrator.

A standard procedure for drawing M samples via Hamiltonian Monte Carlo is described in Algorithm 1. I denotes the identity matrix and $\mathcal{N}(\mu, \Sigma)$ denotes a multivariate normal distribution with mean μ and covariance matrix Σ. For each sample m, we first resample the momentum variables from a standard multivariate normal, which can be interpreted as a Gibbs sampling update. We then apply L leapfrog updates to the position and momentum variables θ and r, generating a proposal position-momentum pair $\tilde{\theta}, \tilde{r}$. We propose setting $\theta^m = \tilde{\theta}$ and $r^m = -\tilde{r}$, and accept or reject this proposal according to the Metropolis algorithm (Metropolis et al., 1953). This is a valid Metropolis proposal because it is time-reversible and the leapfrog integrator is volume-preserving; using an algorithm for simulating Hamiltonian dynamics that did not preserve volume complicates the computation of the Metropolis acceptance probability (Lan et al., 2012). The negation of $\tilde{r}$ in the proposal is theoretically necessary to produce time-reversibility, but can be omitted in practice if one is only interested in sampling from $p(\theta)$.

The term $\log \frac{p(\tilde{\theta},\tilde{r})}{p(\theta,r)}$, on which the acceptance probability α depends, is the negative change in energy of the simulated Hamiltonian system from time 0 to time ϵL. If we could simulate the Hamiltonian dynamics exactly, then α would always be 1, since energy is conserved in Hamiltonian systems. The error introduced by using a discrete-time simulation depends on the step size parameter ϵ—specifically, the change in energy $|\log \frac{p(\tilde{\theta},\tilde{r})}{p(\theta,r)}|$ is proportional to ϵ^2 for large L, or ϵ^3 if $L = 1$ (Leimkuhler and Reich, 2004). In principle the error can grow without bound as a function of L, but it typically does not due to the symplecticness of the leapfrog discretization. This allows us to run HMC with many leapfrog steps, generating proposals for θ that have high probability of acceptance even though they are distant from the previous sample.

The performance of HMC depends strongly on choosing suitable values for ϵ and 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. This is doubly wasteful, since work is being done to bring the proposal $\tilde{\theta}$ *closer* to the initial position θ^{m-1}. Worse, if L is chosen so that the parameters jump from one side of the space to the other each iteration, then the Markov chain may not even be ergodic (Neal, 2011). More realistically, an unfortunate choice of L may result in a chain that is ergodic but slow to move between regions of low and high density.

3. Eliminating the Need to Hand-Tune HMC

HMC is a powerful algorithm, but its usefulness is limited by the need to tune the step size parameter ϵ and number of steps L. Tuning these parameters for any particular problem requires some expertise, and usually one or more preliminary runs. Selecting L is particularly problematic; it is difficult to find a simple metric for when a trajectory is too short, too long, or "just right," and so practitioners commonly rely on heuristics based on autocorrelation statistics from preliminary runs (Neal, 2011).

Below, we present the No-U-Turn Sampler (NUTS), an extension of HMC that eliminates the need to specify a fixed value of L. In Section 3.2 we present schemes for setting ϵ based on the dual averaging algorithm of Nesterov (2009).

3.1 No-U-Turn Hamiltonian Monte Carlo

Our first goal is to devise an MCMC sampler that retains HMC's ability to suppress random walk behavior without the need to set the number L of leapfrog steps that the algorithm takes to generate a proposal. We need some criterion to tell us when we have simulated the dynamics for "long enough," that is, when running the simulation for more steps would

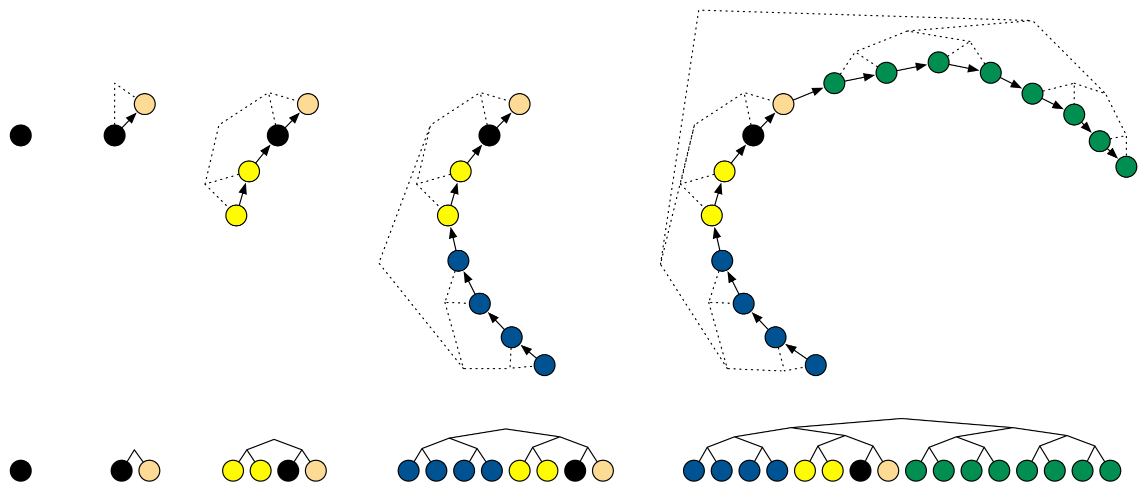

Figure 1: Example of building a binary tree via repeated doubling. Each doubling proceeds by choosing a direction (forwards or backwards in time) uniformly at random, then simulating Hamiltonian dynamics for 2^j leapfrog steps in that direction, where j is the number of previous doublings (and the height of the binary tree). The figures at top show a trajectory in two dimensions (with corresponding binary tree in dashed lines) as it evolves over four doublings, and the figures below show the evolution of the binary tree. In this example, the directions chosen were forward (light orange node), backward (yellow nodes), backward (blue nodes), and forward (green nodes).

no longer increase the distance between the proposal $\tilde{\theta}$ and the initial value of θ. We use a convenient criterion based on the dot product between $\tilde{r}$ (the current momentum) and $\tilde{\theta}-\theta$ (the vector from our initial position to our current position), which is the derivative with respect to time (in the Hamiltonian system) of half the squared distance between the initial position θ and the current position $\tilde{\theta}$:

$$\frac{d}{dt}\frac{(\tilde{\theta}-\theta)\cdot(\tilde{\theta}-\theta)}{2} = (\tilde{\theta}-\theta)\cdot\frac{d}{dt}(\tilde{\theta}-\theta) = (\tilde{\theta}-\theta)\cdot\tilde{r}. \tag{1}$$

In other words, if we were to run the simulation for an infinitesimal amount of additional time, then this quantity is proportional to the progress we would make away from our starting point θ.

This suggests an algorithm in which one runs leapfrog steps until the quantity in Equation 1 becomes less than 0; such an approach would simulate the system's dynamics until the proposal location $\tilde{\theta}$ started to move back towards θ. Unfortunately this algorithm does not guarantee time reversibility, and is therefore not guaranteed to converge to the correct distribution. NUTS overcomes this issue by means of a recursive algorithm that preserves reversibility by running the Hamiltonian simulation both forward and backward in time.

NUTS begins by introducing a slice variable u with conditional distribution $p(u|\theta,r) = \text{Uniform}(u;[0,\exp\{\mathcal{L}(\theta)-\frac{1}{2}r\cdot r\}])$, which renders the conditional distribution $p(\theta,r|u) =$

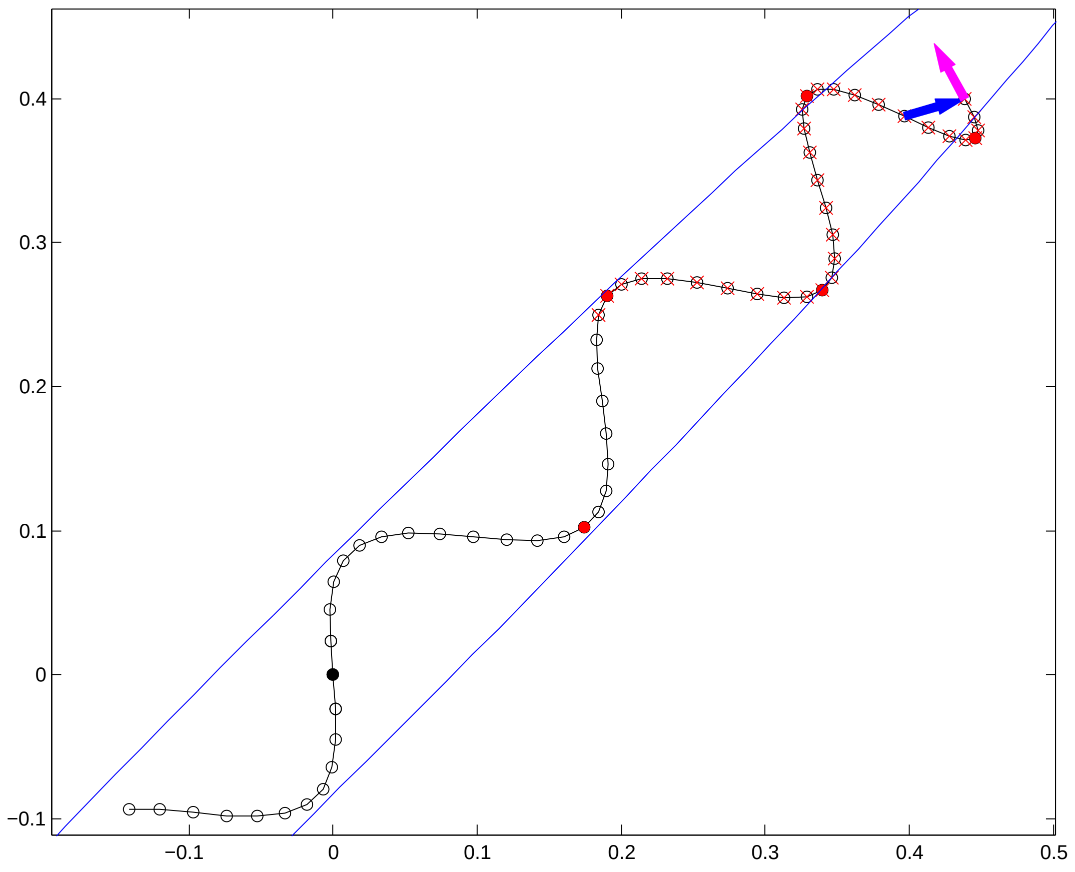

Figure 2: Example of a trajectory generated during one iteration of NUTS. The blue ellipse is a contour of the target distribution, the black open circles are the positions θ traced out by the leapfrog integrator and associated with elements of the set of visited states $\mathcal{B}$, the black solid circle is the starting position, the red solid circles are positions associated with states that must be excluded from the set $\mathcal{C}$ of possible next samples because their joint probability is below the slice variable u, and the positions with a red "x" through them correspond to states that must be excluded from $\mathcal{C}$ to satisfy detailed balance. The blue arrow is the vector from the positions associated with the leftmost to the rightmost leaf nodes in the rightmost height-3 subtree, and the magenta arrow is the (normalized) momentum vector at the final state in the trajectory. The doubling process stops here, since the blue and magenta arrows make an angle of more than 90 degrees. The crossed-out nodes with a red "x" are in the right half-tree, and must be ignored when choosing the next sample.

$\text{Uniform}(\theta, r; \{\theta', r' | \exp\{\mathcal{L}(\theta) - \frac{1}{2} r \cdot r\} \geq u\})$. This slice sampling step is not strictly necessary, but it simplifies both the derivation and the implementation of NUTS.

At a high level, after resampling $u|\theta, r$, NUTS uses the leapfrog integrator to trace out a path forwards and backwards in fictitious time, first running forwards or backwards 1 step, then forwards or backwards 2 steps, then forwards or backwards 4 steps, etc. This doubling process implicitly builds a balanced binary tree whose leaf nodes correspond to position-momentum states, as illustrated in Figure 1. The doubling is halted when the subtrajectory from the leftmost to the rightmost nodes of any balanced subtree of the overall binary tree starts to double back on itself (i.e., the fictional particle starts to make a "U-turn"). At this point NUTS stops the simulation and samples from among the set of points computed during the simulation, taking care to preserve detailed balance. Figure 2 illustrates an example of a trajectory computed during an iteration of NUTS.

Pseudocode implementing a efficient version of NUTS is provided in Algorithm 3. A detailed derivation follows below, along with a simplified version of the algorithm that motivates and builds intuition about Algorithm 3 (but uses much more memory and makes smaller jumps).

3.1.1 Derivation of Simplified NUTS Algorithm

NUTS further augments the model $p(\theta, r) \propto \exp\{\mathcal{L}(\theta) - \frac{1}{2} r \cdot r\}$ with a slice variable u (Neal, 2003). The joint probability of θ, r, and u is

$$p(\theta, r, u) \propto \mathbb{I}[u \in [0, \exp\{\mathcal{L}(\theta) - \tfrac{1}{2} r \cdot r\}]],$$

where $\mathbb{I}[\cdot]$ is 1 if the expression in brackets is true and 0 if it is false. The (unnormalized) marginal probability of θ and r (integrating over u) is

$$p(\theta, r) \propto \exp\{\mathcal{L}(\theta) - \tfrac{1}{2} r \cdot r\},$$

as in standard HMC. The conditional probabilities $p(u|\theta, r)$ and $p(\theta, r|u)$ are each uniform, so long as the condition $u \leq \exp\{\mathcal{L}(\theta) - \frac{1}{2} r \cdot r\}$ is satisfied.

We also add a finite set $\mathcal{C}$ of candidate position-momentum states and another finite set $\mathcal{B} \supseteq \mathcal{C}$ to the model. $\mathcal{B}$ will be the set of all position-momentum states that the leapfrog integrator traces out during a given NUTS iteration, and $\mathcal{C}$ will be the subset of those states to which we can transition without violating detailed balance. $\mathcal{B}$ will be built up by randomly taking forward and backward leapfrog steps, and $\mathcal{C}$ will selected deterministically from $\mathcal{B}$. The random procedure for building $\mathcal{B}$ and $\mathcal{C}$ given θ, r, u, and ϵ will define a conditional distribution $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$, upon which we place the following conditions:

C.1: All elements of $\mathcal{C}$ must be chosen in a way that preserves volume. That is, any deterministic transformations of θ, r used to add a state θ', r' to $\mathcal{C}$ must have a Jacobian with unit determinant.

C.2: $p((\theta, r) \in \mathcal{C}|\theta, r, u, \epsilon) = 1$.

C.3: $p(u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2} r' \cdot r'\}|(\theta', r') \in \mathcal{C}) = 1$.

C.4: If $(\theta, r) \in \mathcal{C}$ and $(\theta', r') \in \mathcal{C}$ then for any $\mathcal{B}$, $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|\theta', r', u, \epsilon)$.

C.1 ensures that $p(\theta, r|(\theta, r) \in \mathcal{C}) \propto p(\theta, r)$, that is, if we restrict our attention to the elements of $\mathcal{C}$ then we can treat the unnormalized probability density of a particular element of $\mathcal{C}$ as an unnormalized probability mass. C.2 says that the current state θ, r must be included in $\mathcal{C}$. C.3 requires that any state in $\mathcal{C}$ be in the slice defined by u, that is, that any state $(\theta', r') \in \mathcal{C}$ must have equal (and positive) conditional probability density $p(\theta', r'|u)$. C.4 states that $\mathcal{B}$ and $\mathcal{C}$ must have equal probability of being selected regardless of the current state θ, r as long as $(\theta, r) \in \mathcal{C}$ (which it must be by C.2).

Deferring for the moment the question of how to construct and sample from a distribution $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$ that satisfies these conditions, we will now show that the the following procedure leaves the joint distribution $p(\theta, r, u, \mathcal{B}, \mathcal{C}|\epsilon)$ invariant:

1. sample $r \sim \mathcal{N}(0, I)$,
2. sample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^t) - \frac{1}{2} r \cdot r\}])$,
3. sample $\mathcal{B}, \mathcal{C}$ from their conditional distribution $p(\mathcal{B}, \mathcal{C}|\theta^t, r, u, \epsilon)$,
4. sample $\theta^{t+1}, r \sim T(\theta^t, r, \mathcal{C})$,

where $T(\theta', r'|\theta, r, \mathcal{C})$ is a transition kernel that leaves the uniform distribution over $\mathcal{C}$ invariant, that is, T must satisfy

$$\frac{1}{|\mathcal{C}|} \sum_{(\theta, r) \in \mathcal{C}} T(\theta', r'|\theta, r, \mathcal{C}) = \frac{\mathbb{I}[(\theta', r') \in \mathcal{C}]}{|\mathcal{C}|}$$

for any θ', r'. The notation $\theta^{t+1}, r \sim T(\theta^t, r, \mathcal{C})$ denotes that we are resampling r in a way that depends on its current value.

Steps 1, 2, and 3 resample r, u, $\mathcal{B}$, and $\mathcal{C}$ from their conditional joint distribution given θ^t, and therefore together constitute a valid Gibbs sampling update. Step 4 is valid because the joint distribution of θ and r given $u, \mathcal{B}, \mathcal{C}$, and ϵ is uniform on the elements of $\mathcal{C}$:

$$\begin{aligned} p(\theta, r|u, \mathcal{B}, \mathcal{C}, \epsilon) &\propto p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon) p(\theta, r|u) \\ &\propto p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon) \mathbb{I}[u \leq \exp\{\mathcal{L}(\theta) - \tfrac{1}{2} r \cdot r\}] \\ &\propto \mathbb{I}[(\theta, r) \in \mathcal{C}]. \end{aligned} \tag{2}$$

Condition C.1 allows us to treat the unnormalized conditional density $p(\theta, r|u) \propto \mathbb{I}[u \leq \exp\{\mathcal{L}(\theta) - \frac{1}{2} r \cdot r\}]$ as an unnormalized conditional probability mass function. Conditions C.2 and C.4 ensure that $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon) \propto \mathbb{I}[(\theta, r) \in \mathcal{C}]$ because by C.2 (θ, r) must be in $\mathcal{C}$, and by C.4 for any $\mathcal{B}, \mathcal{C}$ pair $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$ is constant as a function of θ and r as long as $(\theta, r) \in \mathcal{C}$. Condition C.3 ensures that $(\theta, r) \in \mathcal{C} \Rightarrow u \leq \exp\{\mathcal{L}(\theta) - \frac{1}{2} r \cdot r\}$ (so the $p(\theta, r|u, \epsilon)$ term is redundant). Thus, Equation 2 implies that the joint distribution of θ and r given u and $\mathcal{C}$ is uniform on the elements of $\mathcal{C}$, and we are free to choose a new θ^{t+1}, r^{t+1} from any transition kernel that leaves this uniform distribution on $\mathcal{C}$ invariant.

We now turn our attention to the specific form for $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$ used by NUTS. Conceptually, the generative process for building $\mathcal{B}$ proceeds by repeatedly doubling the size of a binary tree whose leaves correspond to position-momentum states. These states will constitute the elements of $\mathcal{B}$. The initial tree has a single node corresponding to the initial state. Doubling proceeds by choosing a random direction $v_j \sim \text{Uniform}(\{-1, 1\})$ and taking 2^j leapfrog steps of size $v_j \epsilon$ (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 height 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. This doubling process is illustrated in Figure 1. Given the initial state θ, r 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 from any leaf node of that tree is 2^{-j} regardless of which leaf node we start from.[1]

We cannot keep expanding the tree forever, of course. We want to continue expanding $\mathcal{B}$ until one end of the trajectory we are simulating makes a "U-turn" and begins to loop back towards another position on the trajectory. At that point continuing the simulation is likely to be wasteful, 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 $\mathcal{B}$ 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. (This may happen if we use a step size ϵ that is too large, or if the target distribution includes hard constraints that make the log-density $\mathcal{L}$ go to $-\infty$ in some regions.)

The second rule is easy to formalize—we simply stop doubling if the tree includes a leaf node whose state θ, r satisfies

$$\mathcal{L}(\theta) - \frac{1}{2} r \cdot r - \log u < -\Delta_{\max} \tag{3}$$

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

We must be careful when defining the first rule so that we can build a sampler that neither violates detailed balance nor introduces excessive computational overhead. To determine whether to stop doubling the tree at height j, NUTS considers the $2^j - 1$ balanced binary subtrees of the height-j tree that have height greater than 0. NUTS stops the doubling process when for one of these subtrees the states θ^-, r^- and θ^+, r^+ associated with the leftmost and rightmost leaves of that subtree satisfies

$$(\theta^+ - \theta^-) \cdot r^- < 0 \quad \text{or} \quad (\theta^+ - \theta^-) \cdot r^+ < 0. \tag{4}$$

That is, we stop if continuing the simulation an infinitesimal amount either forward or backward in time would reduce the distance between the position vectors θ^- and θ^+. Evaluating the condition in Equation 4 for each balanced subtree of a tree of height j requires $2^{j+1} - 2$ inner products, which is comparable to the number of inner products required by the $2^j - 1$ leapfrog steps needed to compute the trajectory. Except for very simple models with very little data, the cost of these inner products should be negligible compared to the cost of computing gradients.

1. This procedure resembles the doubling procedure devised by Neal (2003) to update scalar variables in a way that leaves their conditional distribution invariant. The doubling procedure finds a set of candidate points by repeatedly doubling the size of a segment of the real line containing the initial point. NUTS, by contrast, repeatedly doubles the size of a finite candidate set of vectors that contains the initial state.

This doubling process defines a distribution $p(\mathcal{B}|\theta, r, u, \epsilon)$. We now define a deterministic process for deciding which elements of $\mathcal{B}$ go in the candidate set $\mathcal{C}$, taking care to satisfy conditions C.1–C.4 on $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$ laid out above. C.1 is automatically satisfied, since leapfrog steps are volume preserving and any element of $\mathcal{C}$ must be within some number of leapfrog steps of every other element of $\mathcal{C}$. C.2 is satisfied as long as we include the initial state θ, r in $\mathcal{C}$, and C.3 is satisfied if we exclude any element θ', r' of $\mathcal{B}$ for which $\exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\} < u$. To satisfy condition C.4, we must ensure that $p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|\theta', r', u, \epsilon)$ for any $(\theta', r') \in \mathcal{C}$. For any start state $(\theta', r') \in \mathcal{B}$, there is at most one series of directions $\{v_0, \ldots, v_j\}$ for which the doubling process will reproduce $\mathcal{B}$, so as long as we choose $\mathcal{C}$ deterministically given $\mathcal{B}$ either $p(\mathcal{B}, \mathcal{C}|\theta', r', u, \epsilon) = 2^{-j} = p(\mathcal{B}, \mathcal{C}|\theta, r, u, \epsilon)$ or $p(\mathcal{B}, \mathcal{C}|\theta', r', u, \epsilon) = 0$. Thus, condition C.4 will be satisfied as long as we exclude from $\mathcal{C}$ any state θ', r' that could not have generated $\mathcal{B}$. The only way such a state can arise is if starting from θ', r' results in the stopping conditions in Equations 3 or 4 being satisfied before the entire tree has been built, causing the doubling process to stop too early. There are two cases to consider:

1. The doubling procedure was stopped because either equation 3 or Equation 4 was satisfied by a state or subtree added during the final doubling iteration. In this case we must exclude from $\mathcal{C}$ any element of $\mathcal{B}$ that was added during this final doubling iteration, since starting the doubling process from one of these would lead to a stopping condition being satisfied before the full tree corresponding to $\mathcal{B}$ has been built.

2. The doubling procedure was stopped because equation 4 was satisfied for the leftmost and rightmost leaves of the full tree corresponding to $\mathcal{B}$. In this case no stopping condition was met by any state or subtree until $\mathcal{B}$ had been completed, and condition C.4 is automatically satisfied.

Algorithm 2 shows how to construct $\mathcal{C}$ incrementally while building $\mathcal{B}$. After resampling the initial momentum and slice variables, it uses a recursive procedure resembling a depth-first search that eliminates the need to explicitly store the tree used by the doubling procedure. The BuildTree() function takes as input an initial position θ and momentum r, a slice variable u, a direction $v \in \{-1, 1\}$, a depth j, and a step size ϵ. It takes 2^j leapfrog steps of size $v\epsilon$ (i.e., forwards in time if $v = 1$ and backwards in time if $v = -1$), and returns

1. the backwardmost and forwardmost position-momentum states θ^-, r^- and θ^+, r^+ among the 2^j new states visited;

2. a set $\mathcal{C}'$ of position-momentum states containing each newly visited state θ', r' for which $\exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\} > u$; and

3. an indicator variable s; $s = 0$ indicates that a stopping criterion was met by some state or subtree of the subtree corresponding to the 2^j new states visited by BuildTree().

At the top level, NUTS repeatedly calls BuildTree() to double the number of points that have been considered until either BuildTree() returns $s = 0$ (in which case doubling stops and the new set $\mathcal{C}'$ that was just returned must be ignored) or Equation 4 is satisfied for the new backwardmost and forwardmost position-momentum states θ^-, r^- and θ^+, r^+ yet considered (in which case doubling stops but we can use the new set $\mathcal{C}'$). Finally, we select

Algorithm 2 Naive No-U-Turn Sampler

```
Given $\theta^0$, $\epsilon$, $\mathcal{L}$, $M$:
for $m = 1$ to $M$ do
  Resample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2}r^0 \cdot r^0\}])$
  Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0$, $r^+ = r^0$, $j = 0$, $\mathcal{C} = \{(\theta^{m-1}, r^0)\}, s = 1$.
  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
    $s \leftarrow s'\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
    $j \leftarrow j + 1$.
  end while
  Sample $\theta^m, r$ 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 $v$.
  $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
  $\mathcal{C}' \leftarrow \begin{cases} \{(\theta', r')\} & \text{if } u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\} \\ \emptyset & \text{else} \end{cases}$
  $s' \leftarrow \mathbb{I}[\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r' > \log u - \Delta_{\max}]$.
  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
    $\theta^-, r^-, -, -, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon)$.
  else
    $-, -, \theta^+, r^+, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon)$.
  end if
  $s' \leftarrow s's''\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
  $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}''$.
  return $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s'$.
end if
```

the next position and momentum θ^m, r uniformly at random from $\mathcal{C}$, the union of all of the valid sets $\mathcal{C}'$ that have been returned, which clearly leaves the uniform distribution over $\mathcal{C}$ invariant.

To summarize, Algorithm 2 defines a transition kernel that leaves $p(\theta, r, u, \mathcal{B}, \mathcal{C}|\epsilon)$ invariant, and therefore leaves the target distribution $p(\theta) \propto \exp\{\mathcal{L}(\theta)\}$ invariant. It does so by resampling the momentum and slice variables r and u, simulating a Hamiltonian trajectory forwards and backwards in time until that trajectory either begins retracing its steps or encounters a state with very low probability, carefully selecting a subset $\mathcal{C}$ of the states encountered on that trajectory that lie within the slice defined by the slice variable u, and finally choosing the next position and momentum variables θ^m and r uniformly at random from $\mathcal{C}$. Figure 2 shows an example of a trajectory generated by an iteration of NUTS where Equation 4 is satisfied by the height-3 subtree at the end of the trajectory. Below, we will introduce some improvements to algorithm 2 that boost the algorithm's memory efficiency and allow it to make larger jumps on average.

3.1.2 Efficient NUTS

Algorithm 2 requires $2^j - 1$ evaluations of $\mathcal{L}(\theta)$ and its gradient (where j is the number of times BuildTree() is called), and $O(2^j)$ 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 2 per leapfrog step is comparable to that of a standard HMC algorithm. However, Algorithm 2 also requires that we store 2^j position and momentum vectors, which may require an unacceptably large amount of memory. Furthermore, there are alternative transition kernels that satisfy detailed balance with respect to the uniform distribution on $\mathcal{C}$ that produce larger jumps on average than simple uniform sampling. Finally, if a stopping criterion is satisfied in the middle of the final doubling iteration then 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. We can address the second issue by using a more sophisticated transition kernel to move from one state $(\theta, r) \in \mathcal{C}$ to another state $(\theta', r') \in \mathcal{C}$ while leaving the uniform distribution over $\mathcal{C}$ invariant. This kernel admits a memory-efficient implementation that only requires that we store $O(j)$ position and momentum vectors, rather than $O(2^j)$.

Consider the transition kernel

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

where w and w' are shorthands for position-momentum states (θ, r), $\mathcal{C}^{\text{new}}$ and $\mathcal{C}^{\text{old}}$ are disjoint subsets of $\mathcal{C}$ such that $\mathcal{C}^{\text{new}} \cup \mathcal{C}^{\text{old}} = \mathcal{C}$, and $w \in \mathcal{C}^{\text{old}}$. In English, T proposes a move from $\mathcal{C}^{\text{old}}$ to a random state in $\mathcal{C}^{\text{new}}$ and accepts the move with probability $\frac{|\mathcal{C}^{\text{new}}|}{|\mathcal{C}^{\text{old}}|}$. This is equivalent to a Metropolis-Hastings kernel with proposal distribution $q(w', \mathcal{C}^{\text{old}\prime}, \mathcal{C}^{\text{new}\prime} | w, \mathcal{C}^{\text{old}}, \mathcal{C}^{\text{new}}) \propto \mathbb{I}[w' \in \mathcal{C}^{\text{new}}]\mathbb{I}[\mathcal{C}^{\text{old}\prime} = \mathcal{C}^{\text{new}}]\mathbb{I}[\mathcal{C}^{\text{new}\prime} = \mathcal{C}^{\text{old}}]$, and it is straightforward to show that it satisfies detailed balance with respect to the uniform distribution on $\mathcal{C}$, that is,

$$p(w|\mathcal{C})T(w'|w, \mathcal{C}) = p(w'|\mathcal{C})T(w|w', \mathcal{C}),$$

and that T therefore leaves the uniform distribution over $\mathcal{C}$ invariant. If we let $\mathcal{C}^{\text{new}}$ be the (possibly empty) set of elements added to $\mathcal{C}$ during the final iteration of the doubling (i.e., those returned by the final call to BuildTree() and $\mathcal{C}^{\text{old}}$ be the older elements of $\mathcal{C}$, then we can replace the uniform sampling of $\mathcal{C}$ at the end of Algorithm 2 with a draw from $T(\theta^t, r^t, \mathcal{C})$ and leave the uniform distribution on $\mathcal{C}$ invariant. In fact, we can apply T after *every* doubling, proposing a move to each new half-tree in turn. Doing so leaves the uniform distribution on each partially built $\mathcal{C}$ invariant, and therefore does no harm to the invariance of the uniform distribution on the fully built set $\mathcal{C}$. Repeatedly applying T in this way increases the probability that we will jump to a state θ^{t+1} far from the initial state θ^t; considering the process in reverse, it is as though we first tried to jump to the other side of $\mathcal{C}$, then if that failed tried to make a more modest jump, and so on. This transition kernel is thus akin to delayed-rejection MCMC methods (Tierney and Mira, 1999), but in this setting we can avoid the usual costs associated with evaluating new proposals.

The transition kernel above still requires that we be able to sample uniformly from the set $\mathcal{C}'$ returned by BuildTree(), which may contain as many as 2^{j-1} elements. In fact, we can sample from $\mathcal{C}'$ without maintaining the full set $\mathcal{C}'$ in memory by exploiting the binary tree structure in Figure 1. Consider a subtree of the tree explored in a call to BuildTree(), and let $\mathcal{C}_{\text{subtree}}$ denote the set of its leaf states that are in $\mathcal{C}'$: we can factorize the probability that a state $(\theta, r) \in \mathcal{C}_{\text{subtree}}$ will be chosen uniformly at random from $\mathcal{C}'$ as

$$\begin{aligned} p(\theta, r|\mathcal{C}') = \frac{1}{|\mathcal{C}'|} = \frac{|\mathcal{C}_{\text{subtree}}|}{|\mathcal{C}'|}\frac{1}{|\mathcal{C}_{\text{subtree}}|} \\ = p((\theta, r) \in \mathcal{C}_{\text{subtree}}|\mathcal{C})p(\theta, r|(\theta, r) \in \mathcal{C}_{\text{subtree}}, \mathcal{C}). \end{aligned}$$

That is, $p(\theta, r|\mathcal{C}')$ is the product of the probability of choosing some node from the subtree multiplied by the probability of choosing θ, r uniformly at random from $\mathcal{C}_{\text{subtree}}$. We use this observation to sample from $\mathcal{C}'$ incrementally as we build up the tree. Each subtree above the bottom layer is built of two smaller subtrees. For each of these smaller subtrees, we sample a θ, r pair from $p(\theta, r|(\theta, r) \in \mathcal{C}_{\text{subtree}})$ to represent that subtree. We then choose between these two pairs, giving the pair representing each subtree weight proportional to how many elements of $\mathcal{C}'$ are in that subtree. This continues until we have completed the subtree associated with $\mathcal{C}'$ and we have returned a sample θ' from $\mathcal{C}'$ and an integer weight n' encoding the size of $\mathcal{C}'$, which is all we need to apply T. This procedure only requires that we store $O(j)$ position and momentum vectors in memory, rather than $O(2^j)$, and requires that we generate $O(2^j)$ extra random numbers (a cost that again is usually very small compared with the $2^j - 1$ gradient computations needed to run the leapfrog algorithm).

Algorithm 3 implements all of the above improvements in pseudocode.

3.2 Adaptively Tuning ϵ

Having addressed the issue of how to choose the number of steps L, we now turn our attention to the step size parameter ϵ. To set ϵ for both NUTS and HMC, we propose using stochastic optimization with vanishing adaptation (Andrieu and Thoms, 2008), specifically an adaptation of the primal-dual algorithm of Nesterov (2009).

Perhaps the most commonly used vanishing adaptation algorithm in MCMC is the stochastic approximation method of Robbins and Monro (1951). Suppose we have a statistic H_t that describes some aspect of the behavior of an MCMC algorithm at iteration $t \geq 1$,

Algorithm 3 Efficient No-U-Turn Sampler

```
Given $\theta^0$, $\epsilon$, $\mathcal{L}$, $M$:
for $m = 1$ to $M$ do
  Resample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2}r^0 \cdot r^0\}])$
  Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0, r^+ = r^0, j = 0, \theta^m = \theta^{m-1}, n = 1, s = 1$.
  while $s = 1$ do
    Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$.
    if $v_j = -1$ then
      $\theta^-, r^-, -, -, \theta', n', s' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v_j, j, \epsilon)$.
    else
      $-, -, \theta^+, r^+, \theta', n', s' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v_j, j, \epsilon)$.
    end if
    if $s' = 1$ then
      With probability $\min\{1, \frac{n'}{n}\}$, set $\theta^m \leftarrow \theta'$.
    end if
    $n \leftarrow n + n'$.
    $s \leftarrow s'\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
    $j \leftarrow j + 1$.
  end while
end for

function BuildTree($\theta, r, u, v, j, \epsilon$)
if $j = 0$ then
  Base case—take one leapfrog step in the direction $v$.
  $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
  $n' \leftarrow \mathbb{I}[u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\}]$.
  $s' \leftarrow \mathbb{I}[\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r' > \log u - \Delta_{\max}]$
  return $\theta', r', \theta', r', \theta', n', s'$.
else
  Recursion—implicitly build the left and right subtrees.
  $\theta^-, r^-, \theta^+, r^+, \theta', n', s' \leftarrow \text{BuildTree}(\theta, r, u, v, j - 1, \epsilon)$.
  if $s' = 1$ then
    if $v = -1$ then
      $\theta^-, r^-, -, -, \theta'', n'', s'' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon)$.
    else
      $-, -, \theta^+, r^+, \theta'', n'', s'' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon)$.
    end if
    With probability $\frac{n''}{n'+n''}$, set $\theta' \leftarrow \theta''$.
    $s' \leftarrow s''\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$
    $n' \leftarrow n' + n''$
  end if
  return $\theta^-, r^-, \theta^+, r^+, \theta', n', s'$.
end if
```

and define its expectation $h(x)$ as

$$h(x) \equiv \mathbb{E}_t[H_t|x] \equiv \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[H_t|x],$$

where $x \in \mathbb{R}$ is a tunable parameter to the MCMC algorithm. For example, if α_t is the Metropolis acceptance probability for iteration t, we might define $H_t = \delta - \alpha_t$, where δ is the desired average acceptance probability. If h is a nondecreasing function of x and a few other conditions such as boundedness of the iterates x_t are met (see Andrieu and Thoms 2008 for details), the update

$$x_{t+1} \leftarrow x_t - \eta_t H_t$$

is guaranteed to cause $h(x_t)$ to converge to 0 as long as the step size schedule defined by η_t satisfies the conditions

$$\sum_t \eta_t = \infty; \quad \sum_t \eta_t^2 < \infty. \tag{5}$$

These conditions are satisfied by schedules of the form $\eta_t \equiv t^{-\kappa}$ for $\kappa \in (0.5, 1]$. As long as the per-iteration impact of the adaptation goes to 0 (as it will if $\eta_t \equiv t^{-\kappa}$ and $\kappa > 0$) the asymptotic behavior of the sampler is unchanged. That said, in practice x often gets "close enough" to an optimal value well before the step size η has gotten close enough to 0 to avoid disturbing the Markov chain's stationary distribution. A common practice, which we follow here, is to adapt any tunable MCMC parameters during the warmup phase, and freeze the tunable parameters afterwards (e.g., Gelman et al., 2004). For the present paper, the step size ϵ is the only tuning parameter x in the algorithm. More advanced implementations could have more options, though, so we consider the tuning problem more generally.

3.2.1 Dual Averaging

The optimal values of the parameters to an MCMC algorithm during the warmup phase and the stationary phase are often quite different. Ideally those parameters would therefore adapt quickly as we shift from the sampler's initial, transient regime to its stationary regime. However, the diminishing step sizes of Robbins-Monro give disproportionate weight to the *early* iterations, which is the opposite of what we want.

Similar issues motivate the dual averaging scheme of Nesterov (2009), an algorithm for nonsmooth and stochastic convex optimization. Since solving an unconstrained convex optimization problem is equivalent to finding a zero of a nondecreasing function (the (sub)gradient of the cost function), it is straightforward to adapt dual averaging to the problem of MCMC adaptation by replacing stochastic gradients with the statistics H_t. Again assuming that we want to find a setting of a parameter $x \in \mathbb{R}$ such that $h(x) \equiv \mathbb{E}_t[H_t|x] = 0$, we can apply the updates

$$x_{t+1} \leftarrow \mu - \frac{\sqrt{t}}{\gamma}\frac{1}{t+t_0}\sum_{i=1}^{t} H_i; \quad \bar{x}_{t+1} \leftarrow \eta_t x_{t+1} + (1-\eta_t)\bar{x}_t, \tag{6}$$

where μ is a freely chosen point that the iterates x_t are shrunk towards, $\gamma > 0$ is a free parameter that controls the amount of shrinkage towards μ, $t_0 \geq 0$ is a free parameter that stabilizes the initial iterations of the algorithm, $\eta_t \equiv t^{-\kappa}$ is a step size schedule obeying the conditions in Equation 5, and we define $\bar{x}_1 = x_1$. As in Robbins-Monro, the per-iteration impact of these updates on x goes to 0 as t goes to infinity. Specifically, for large t we have

$$x_{t+1} - x_t = O(-H_t t^{-0.5}),$$

which clearly goes to 0 as long as the statistic H_t is bounded. The sequence of averaged iterates $\bar{x}_t$ is guaranteed to converge to a value such that $h(\bar{x}_t)$ converges to 0.

The update scheme in Equation 6 is slightly more elaborate than the update scheme of Nesterov (2009), which implicitly has $t_0 \equiv 0$ and $\kappa \equiv 1$. Introducing these parameters addresses issues that are more important in MCMC adaptation than in more conventional stochastic convex optimization settings. Setting $t_0 > 0$ improves the stability of the algorithm in early iterations, which prevents us from wasting computation by trying out extreme values. This is particularly important for NUTS, and for HMC when simulation lengths are specified in terms of the overall simulation length ϵL instead of a fixed number of steps L. In both of these cases, lower values of ϵ result in more work being done per sample, so we want to avoid casually trying out extremely low values of ϵ. Setting the parameter $\kappa < 1$ allows us to give higher weight to more recent iterates and more quickly forget the iterates produced during the early warmup stages. The benefits of introducing these parameters are less apparent in the settings originally considered by Nesterov, where the cost of a stochastic gradient computation is assumed to be constant and the stochastic gradients are assumed to be drawn i.i.d. given the parameter x.

Allowing $t_0 > 0$ and $\kappa \in (0.5, 1]$ does not affect the asymptotic convergence of the dual averaging algorithm. For any $\kappa \in (0.5, 1]$, $\bar{x}_t$ will eventually converge to the same value $\frac{1}{t}\sum_{i=1}^{t} x_t$. We can rewrite the term $\frac{\sqrt{t}}{\gamma}\frac{1}{t+t_0}$ as $\frac{t\sqrt{t}}{\gamma(t+t_0)}\frac{1}{t}$; $\frac{t\sqrt{t}}{\gamma(t+t_0)}$ is still $O(\sqrt{t})$, which is the only feature needed to guarantee convergence.

We used the values $\gamma = 0.05, t_0 = 10$, and $\kappa = 0.75$ for all our experiments. We arrived at these values by trying a few settings for each parameter by hand with NUTS and HMC (with simulation lengths specified in terms of ϵL) on the stochastic volatility model described below and choosing a value for each parameter that seemed to produce reasonable behavior. Better results might be obtained with further tweaking, but these default parameters seem to work consistently well for both NUTS and HMC for all of the models that we tested. It is entirely possible that these parameter settings may not work as well for other sampling algorithms or for H statistics other than the ones described below.

3.2.2 Finding a Good Initial Value of ϵ

The dual averaging scheme outlined above should work for any initial value ϵ_1 and any setting of the shrinkage target μ. However, convergence will be faster if we start from a reasonable setting of these parameters. We recommend choosing an initial value ϵ_1 according to the simple heuristic described in Algorithm 4. In English, this heuristic repeatedly doubles or halves the value of ϵ_1 until the acceptance probability of the Langevin proposal with step size ϵ_1 crosses 0.5. The resulting value of ϵ_1 will typically be small enough to produce reasonably accurate simulations but large enough to avoid wasting large amounts of computation. We recommend setting $\mu = \log(10\epsilon_1)$, since this gives the dual averaging algorithm a preference for testing values of ϵ that are larger than the initial value ϵ_1. Large values of ϵ cost less to evaluate than small values of ϵ, and so erring on the side of trying large values can save computation.

3.2.3 Setting ϵ in HMC

In HMC we want to find a value for the step size ϵ that is neither too small (which would waste computation by taking needlessly tiny steps) nor too large (which would waste computation by causing high rejection rates). A standard approach is to tune ϵ so that HMC's average Metropolis acceptance probability is equal to some value δ. Indeed, it has been shown that (under fairly strong assumptions) the optimal value of ϵ for a given simulation length ϵL is the one that produces an average Metropolis acceptance probability of approximately 0.65 (Beskos et al., 2010; Neal, 2011). For HMC, we define a criterion $h^{\text{HMC}}(\epsilon)$ so that

$$H_t^{\text{HMC}} \equiv \min\left\{1, \frac{p(\tilde{\theta}^t, \tilde{r}^t)}{p(\theta^{t-1}, r^{t,0})}\right\}; \quad h^{\text{HMC}}(\epsilon) \equiv \mathbb{E}_t[H_t^{\text{HMC}}|\epsilon],$$

where $\tilde{\theta}^t$ and $\tilde{r}^t$ are the proposed position and momentum at the tth iteration of the Markov chain, θ^{t-1} and $r^{t,0}$ are the initial position and (resampled) momentum for the tth iteration of the Markov chain, H_t^{HMC} is the acceptance probability of this tth HMC proposal and h^{HMC} is the expected average acceptance probability of the chain in equilibrium for a fixed ϵ. Assuming that h^{HMC} is nonincreasing as a function of ϵ, we can apply the updates in Equation 6 with $H_t \equiv \delta - H_t^{\text{HMC}}$ and $x \equiv \log \epsilon$ to coerce $h^{\text{HMC}} = \delta$ for any $\delta \in (0, 1)$.

3.2.4 Setting ϵ in NUTS

Since there is no single accept/reject step in NUTS we must define an alternative statistic to Metropolis acceptance probability. For each iteration we define the statistic H_t^{NUTS} and its expectation when the chain has reached equilibrium as

$$H_t^{\text{NUTS}} \equiv \frac{1}{|\mathcal{B}_t^{\text{final}}|} \sum_{\theta, r \in \mathcal{B}_t^{\text{final}}} \min\left\{1, \frac{p(\theta, r)}{p(\theta^{t-1}, r^{t,0})}\right\}; \quad h^{\text{NUTS}} \equiv \mathbb{E}_t[H_t^{\text{NUTS}}],$$

where $\mathcal{B}_t^{\text{final}}$ is the set of all states explored during the final doubling of iteration t of the Markov chain and θ^{t-1} and $r^{t,0}$ are the initial position and (resampled) momentum for the tth iteration of the Markov chain. H^{NUTS} can be understood as the average acceptance probability that HMC would give to the position-momentum states explored during the final doubling iteration. As above, assuming that H^{NUTS} is nonincreasing in ϵ, we can apply the updates in Equation 6 with $H_t \equiv \delta - H^{\text{NUTS}}$ and $x \equiv \log \epsilon$ to coerce $h^{\text{NUTS}} = \delta$ for any $\delta \in (0, 1)$.

Algorithms 5 and 6 show how to implement HMC (with simulation length specified in terms of ϵL rather than L) and NUTS while incorporating the dual averaging algorithm derived in this section, with the above initialization scheme. Algorithm 5 requires as input a target simulation length $\lambda \approx \epsilon L$, a target mean acceptance probability δ, and a number of iterations M^{adapt} after which to stop the adaptation. Algorithm 6 requires only a target mean acceptance probability δ and a number of iterations M^{adapt}.

Algorithm 4 Heuristic for choosing an initial value of ϵ

```
**function** FindReasonableEpsilon($\theta$)
Initialize $\epsilon = 1$, $r \sim \mathcal{N}(0, I)$. ($I$ denotes the identity matrix.)
Set $\theta', r' \leftarrow \mathrm{Leapfrog}(\theta, r, \epsilon)$.
$a \leftarrow 2\mathbb{I}\left[\frac{p(\theta',r')}{p(\theta,r)} > 0.5\right] - 1$.
**while** $\left(\frac{p(\theta',r')}{p(\theta,r)}\right)^a > 2^{-a}$ **do**
  $\epsilon \leftarrow 2^a \epsilon$.
  Set $\theta', r' \leftarrow \mathrm{Leapfrog}(\theta, r, \epsilon)$.
**end while**
**return** $\epsilon$.
```

Algorithm 5 Hamiltonian Monte Carlo with Dual Averaging

```
Given $\theta^0$, $\delta$, $\lambda$, $\mathcal{L}, M, M^{\mathrm{adapt}}$:
Set $\epsilon_0 = \mathrm{FindReasonableEpsilon}(\theta), \mu = \log(10\epsilon_0), \bar{\epsilon}_0 = 1, \bar{H}_0 = 0, \gamma = 0.05, t_0 = 10, \kappa = 0.75$.
**for** $m = 1$ to $M$ **do**
  Reample $r^0 \sim \mathcal{N}(0, I)$.
  Set $\theta^m \leftarrow \theta^{m-1}, \tilde{\theta} \leftarrow \theta^{m-1}, \tilde{r} \leftarrow r^0, L_m = \max\{1, \mathrm{Round}(\lambda/\epsilon_{m-1})\}$.
  **for** $i = 1$ to $L_m$ **do**
    Set $\tilde{\theta}, \tilde{r} \leftarrow \mathrm{Leapfrog}(\tilde{\theta}, \tilde{r}, \epsilon_{m-1})$.
  **end for**
  With probability $\alpha = \min\left\{1, \frac{\exp\{\mathcal{L}(\tilde{\theta}) - \frac{1}{2}\tilde{r}\cdot\tilde{r}\}}{\exp\{\mathcal{L}(\theta^{m-1}) - \frac{1}{2}r^0\cdot r^0\}}\right\}$, set $\theta^m \leftarrow \tilde{\theta}, r^m \leftarrow -\tilde{r}$.
  **if** $m \leq M^{\mathrm{adapt}}$ **then**
    Set $\bar{H}_m = \left(1 - \frac{1}{m+t_0}\right)\bar{H}_{m-1} + \frac{1}{m+t_0}(\delta - \alpha)$.
    Set $\log \epsilon_m = \mu - \frac{\sqrt{m}}{\gamma}\bar{H}_m, \log \bar{\epsilon}_m = m^{-\kappa}\log \epsilon_m + (1 - m^{-\kappa})\log \bar{\epsilon}_{m-1}$.
  **else**
    Set $\epsilon_m = \bar{\epsilon}_{M^{\mathrm{adapt}}}$.
  **end if**
**end for**
```

4. Empirical Evaluation

In this section we examine the effectiveness of the dual averaging algorithm outlined in Section 3.2, examine what values of the target δ in the dual averaging algorithm yield efficient samplers, and compare the efficiency of NUTS and HMC.

For each target distribution, we ran HMC (as implemented in algorithm 5) and NUTS (as implemented in algorithm 6) with four target distributions for 2000 iterations, allowing the step size ϵ to adapt via the dual averaging updates described in Section 3.2 for the first 1000 iterations. In all experiments the dual averaging parameters were set to $\gamma = 0.05, t_0 = 10$, and $\kappa = 0.75$. We evaluated HMC with 10 logarithmically spaced target simulation lengths λ per target distribution. For each target distribution the largest value of λ that we tested was 40 times the smallest value of λ that we tested, meaning that each successive λ is $40^{1/9} \approx 1.5$ times larger than the previous λ. We tried 15 evenly spaced values of the dual averaging target δ between 0.25 and 0.95 for NUTS and 8 evenly spaced values of the dual averaging target δ between 0.25 and 0.95 for HMC. For each sampler-simulation length-δ-target distribution combination we ran 10 iterations with different random seeds.

Algorithm 6 No-U-Turn Sampler with Dual Averaging

```
Given $\theta^0$, $\delta$, $\mathcal{L}, M, M^{\text{adapt}}$:
Set $\epsilon_0 = \text{FindReasonableEpsilon}(\theta), \mu = \log(10\epsilon_0), \bar{\epsilon}_0 = 1, \bar{H}_0 = 0, \gamma = 0.05, t_0 = 10, \kappa = 0.75$.
for $m = 1$ to $M$ do
  Sample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2}r^0 \cdot r^0\}])$
  Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0, r^+ = r^0, j = 0, \theta^m = \theta^{m-1}, n = 1, s = 1$.
  while $s = 1$ do
    Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$.
    if $v_j = -1$ then
      $\theta^-, r^-, -, -, \theta', n', s', \alpha, n_\alpha \leftarrow \text{BuildTree}(\theta^-, r^-, u, v_j, j, \epsilon_{m-1}\theta^{m-1}, r^0)$.
    else
      $-, -, \theta^+, r^+, \theta', n', s', \alpha, n_\alpha \leftarrow \text{BuildTree}(\theta^+, r^+, u, v_j, j, \epsilon_{m-1}, \theta^{m-1}, r^0)$.
    end if
    if $s' = 1$ then
      With probability $\min\{1, \frac{n'}{n}\}$, set $\theta^m \leftarrow \theta'$.
    end if
    $n \leftarrow n + n'$.
    $s \leftarrow s'\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
    $j \leftarrow j + 1$.
  end while
  if $m \leq M^{\text{adapt}}$ then
    Set $\bar{H}_m = \left(1 - \frac{1}{m+t_0}\right) \bar{H}_{m-1} + \frac{1}{m+t_0}(\delta - \frac{\alpha}{n_\alpha})$.
    Set $\log \epsilon_m = \mu - \frac{\sqrt{m}}{\gamma}\bar{H}_m, \log \bar{\epsilon}_m = m^{-\kappa} \log \epsilon_m + (1 - m^{-\kappa}) \log \bar{\epsilon}_{m-1}$.
  else
    Set $\epsilon_m = \bar{\epsilon}_{M^{\text{adapt}}}$.
  end if
end for

function BuildTree($\theta, r, u, v, j, \epsilon, \theta^0, r^0$)
if $j = 0$ then
  *Base case—take one leapfrog step in the direction $v$.*
  $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
  $n' \leftarrow \mathbb{I}[u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\}]$.
  $s' \leftarrow \mathbb{I}[u < \exp\{\Delta_{\max} + \mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\}]$.
  return $\theta', r', \theta', r', \theta', n', s', \min\{1, \exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r' - \mathcal{L}(\theta^0) + \frac{1}{2}r^0 \cdot r^0\}\}, 1$.
else
  *Recursion—implicitly build the left and right subtrees.*
  $\theta^-, r^-, \theta^+, r^+, \theta', n', s', \alpha', n'_\alpha \leftarrow \text{BuildTree}(\theta, r, u, v, j - 1, \epsilon, \theta^0, r^0)$.
  if $s' = 1$ then
    if $v = -1$ then
      $\theta^-, r^-, -, -, \theta'', n'', s'', \alpha'', n''_\alpha \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon, \theta^0, r^0)$.
    else
      $-, -, \theta^+, r^+, \theta'', n'', s'', \alpha'', n''_\alpha \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon, \theta^0, r^0)$.
    end if
    With probability $\frac{n''}{n'+n''}$, set $\theta' \leftarrow \theta''$.
    Set $\alpha' \leftarrow \alpha' + \alpha''$, $n'_\alpha \leftarrow n'_\alpha + n''_\alpha$.
    $s' \leftarrow s''\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$
    $n' \leftarrow n' + n''$
  end if
  return $\theta^-, r^-, \theta^+, r^+, \theta', n', s', \alpha', n'_\alpha$.
end if
```

In total, we ran 3,200 experiments with HMC and 600 experiments with NUTS. Traditional HMC can sometimes exhibit pathological behavior when using a fixed step size and number of steps per iteration (Neal, 2011), so after warmup we jitter HMC's step size, sampling it uniformly at random each iteration from the range $[0.9\bar{\epsilon}_{M^{\text{adapt}}}, 1.1\bar{\epsilon}_{M^{\text{adapt}}}]$ so that the trajectory length may vary by $\pm 10\%$ each iteration.

We measure the efficiency of each algorithm in terms of effective sample size (ESS) normalized by the number of gradient evaluations used by each algorithm. The ESS of a set of M correlated samples $\theta^{1:M}$ with respect to some function $f(\theta)$ is the number of independent draws from the target distribution $p(\theta)$ that would give a Monte Carlo estimate of the mean under p of $f(\theta)$ with the same level of precision as the estimate given by the mean of f for the correlated samples $\theta^{1:M}$. That is, the ESS of a sample is a measure of how many independent samples a set of correlated samples is worth for the purposes of estimating the mean of some function; a more efficient sampler will give a larger ESS for less computation. We use the number of gradient evaluations performed by an algorithm as a proxy for the total amount of computation performed; in all of the models and distributions we tested the computational overhead of both HMC and NUTS is dominated by the cost of computing gradients. Details of the method we use to estimate ESS are provided in appendix A. In each experiment, we discarded the first 1000 samples as warmup when estimating ESS.

ESS is inherently a univariate statistic, but all of the distributions we test HMC and NUTS on are multivariate. Following Girolami and Calderhead (2011) we compute ESS separately for each dimension and report the minimum ESS across all dimensions, since we want our samplers to effectively explore all dimensions of the target distribution. For each dimension we compute ESS in terms of the variance of the estimator of that dimension's mean and second central moment (where the estimate of the mean used to compute the second central moment is taken from a separate long run of 50,000 iterations of NUTS with $\delta = 0.5$), reporting whichever statistic has a lower effective sample size. We include the second central moment as well as the mean in order to measure each algorithm's ability to estimate uncertainty.

4.1 Models and Data Sets

To evaluate NUTS and HMC, we used the two algorithms to sample from four target distributions, one of which was synthetic and the other three of which are posterior distributions arising from real data sets.

4.1.1 250-dimensional Multivariate Normal (MVN)

In these experiments the target distribution was a zero-mean 250-dimensional multivariate normal with known precision matrix A, that is,

$$p(\theta) \propto \exp\{-\tfrac{1}{2}\theta^T A \theta\}.$$

The matrix A was generated from a Wishart distribution with identity scale matrix and 250 degrees of freedom. This yields a target distribution with many strong correlations. The same matrix A was used in all experiments.

4.1.2 Bayesian Logistic Regression (LR)

In these experiments the target distribution is the posterior of a Bayesian logistic regression model fit to the German credit data set available from the UCI repository (Frank and Asuncion, 2010). The target distribution is

$$\begin{aligned} p(\alpha, \beta | x, y) &\propto p(y|x, \alpha, \beta) p(\alpha) p(\beta) \\ &\propto \exp\{-\textstyle\sum_i \log(1 + \exp\{-y_i(\alpha + x_i \cdot \beta\}) - \frac{1}{2\sigma^2}\alpha^2 - \frac{1}{2\sigma^2}\beta \cdot \beta\}, \end{aligned}$$

where x_i is a 24-dimensional vector of numerical predictors associated with a customer i, y_i is -1 if customer i should be denied credit and 1 if that customer should receive credit, α is an intercept term, and β is a vector of 24 regression coefficients. All predictors are normalized to have zero mean and unit variance. α and each element of β are given weak zero-mean normal priors with variance $\sigma^2 = 100$. The data set contains predictor and response data for 1000 customers.

4.1.3 Hierarchical Bayesian Logistic Regression (HLR)

In these experiments the target distribution is again the posterior of a Bayesian logistic regression model fit to the German credit data set, but this time the variance parameter in the prior on α and β is given an exponential prior and estimated as well. Also, we expand the predictor vectors by including two-way interactions, resulting in $\binom{24}{2}+24 = 300$-dimensional vectors of predictors x and a 300-dimensional vector of coefficients β. These elaborations on the model make for a more challenging problem; the posterior is in higher dimensions, and the variance term σ^2 interacts strongly with the remaining 301 variables. The target distribution for this problem is

$$\begin{aligned} p(\alpha, \beta, \sigma^2 | x, y) &\propto p(y|x, \alpha, \beta) p(\beta|\sigma^2) p(\alpha|\sigma^2) p(\sigma^2) \\ &\propto \exp\{-\textstyle\sum_i \log(1 + \exp\{-y_i x_i \cdot \beta\}) - \frac{1}{2\sigma^2}\alpha^2 - \frac{1}{2\sigma^2}\beta \cdot \beta - \frac{N}{2}\log\sigma^2 - \lambda\sigma^2\}, \end{aligned}$$

where $N = 1000$ is the number of customers and λ is the rate parameter to the prior on σ^2. We set $\lambda = 0.01$, yielding a weak exponential prior distribution on σ^2 whose mean and standard deviation are 100.

4.1.4 Stochastic Volatility (SV)

In the final set of experiments the target distribution is the posterior of a relatively simple stochastic volatility model fit to 3000 days of returns from the S&P 500 index. The model assumes that the observed values of the index are generated by the following generative process:

$$\tau \sim \text{Exponential}(100); \quad \nu \sim \text{Exponential}(100); \quad s_1 \sim \text{Exponential}(100);$$
$$\log s_{i>1} \sim \text{Normal}(\log s_{i-1}, \tau^{-1}); \quad \tfrac{\log y_i - \log y_{i-1}}{s_i} \sim \text{t}_\nu,$$

where $s_{i>1}$ refers to a scale parameter s_i where $i > 1$. We integrate out the precision parameter τ to speed mixing, leading to the 3001-dimensional target distribution

$$\begin{aligned} p(s, \nu | y) \propto e^{-0.01\nu} e^{-0.01 s_1} (\textstyle\prod_{i=1}^{3000} \text{t}_\nu(s_i^{-1}(\log y_i - \log y_{i-1}))) \times \\ (0.01 + 0.5 \textstyle\sum_{i=2}^{3000} (\log s_i - \log s_{i-1})^2)^{-\frac{3001}{2}}. \end{aligned}$$

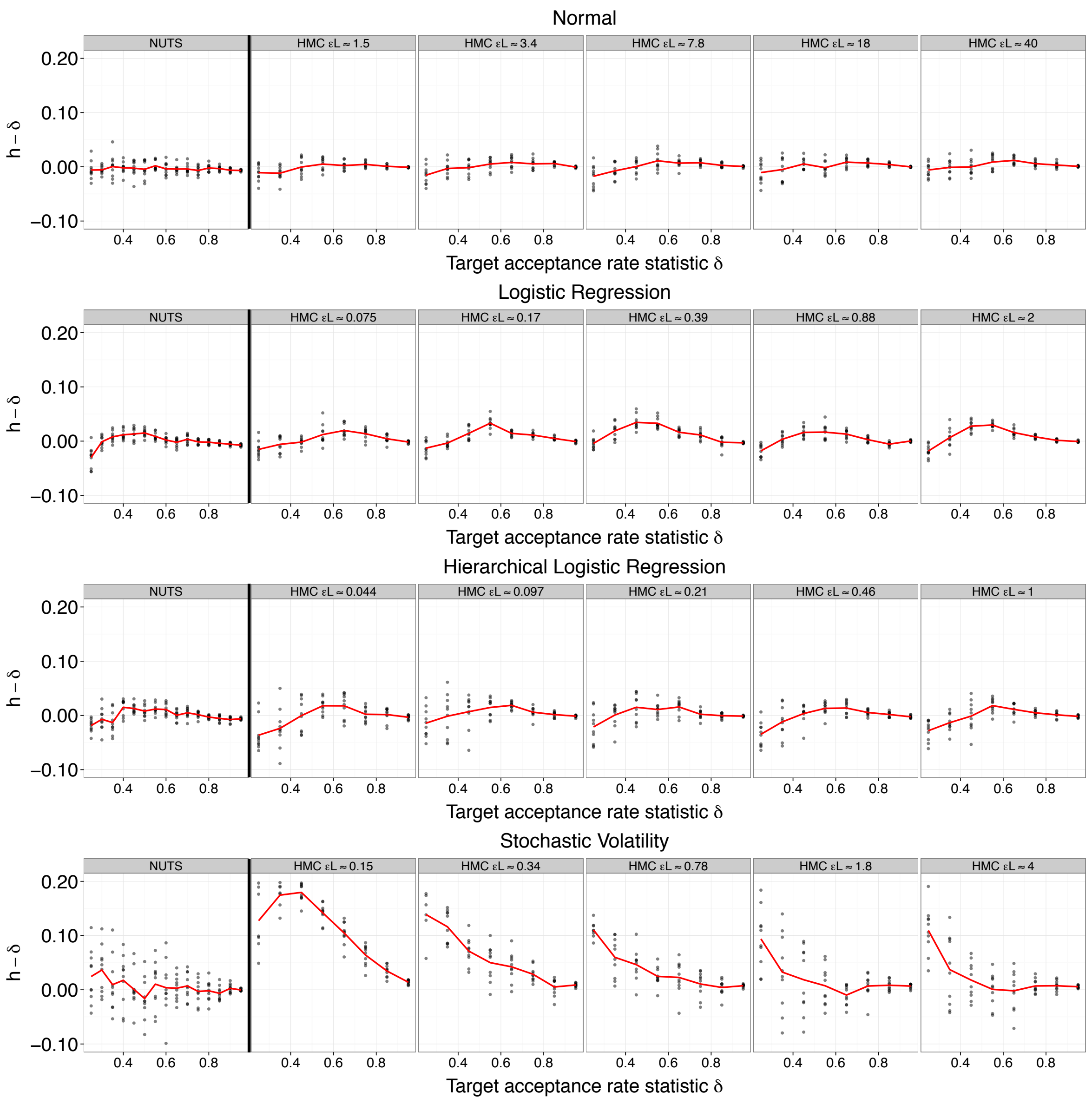

Figure 3: Discrepancies between the realized average acceptance probability statistic h and its target δ for the multivariate normal, logistic regression, hierarchical logistic regression, and stochastic volatility models. Each point's distance from the x-axis shows how effectively the dual averaging algorithm tuned the step size ϵ for a single experiment. Leftmost plots show experiments run with NUTS, other plots show experiments run with HMC with various settings of ϵL.

4.2 Convergence of Dual Averaging

Figure 3 plots the realized versus target values of the statistics h^{HMC} and h^{NUTS}. The h statistics were computed from the 1000 post-warmup samples. The dual averaging algorithm of Section 3.2 usually does a good job of coercing the statistic h to its desired value δ. It performs somewhat worse for the stochastic volatility model, which we attribute to the

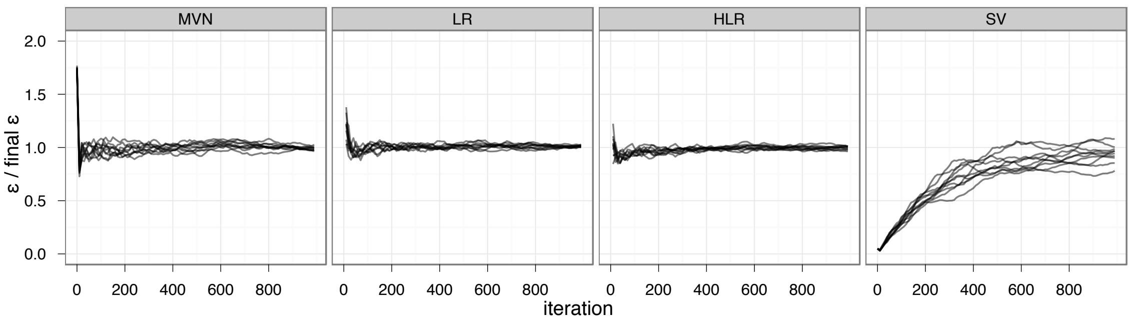

Figure 4: Plots of the convergence of $\bar{\epsilon}$ as a function of the number of iterations of NUTS with dual averaging with $\delta = 0.65$ applied to the multivariate normal (MVN), logistic regression (LR), hierarchical logistic regression (HLR), and stochastic volatility (SV) models. Each trace is from an independent run. The y-axis shows the value of $\bar{\epsilon}$, divided by one of the final values of $\bar{\epsilon}$ so that the scale of the traces for each problem can be readily compared.

longer warmup period needed for this model; since it takes more samples to reach the stationary regime for the stochastic volatility model, the adaptation algorithm has less time to tune ϵ to be appropriate for the stationary distribution. This is particularly true for HMC with small values of δ, since the overly high rejection rates caused by setting δ too small lead to slower convergence.

Figure 4 plots the convergence of the averaged iterates $\bar{\epsilon}_m$ as a function of the number of dual averaging updates for NUTS with $\delta = 0.65$. Except for the stochastic volatility model, which requires longer to warm up, $\bar{\epsilon}$ roughly converges within a few hundred iterations.

4.3 NUTS Trajectory Lengths

Figure 5 shows histograms of the trajectory lengths generated by NUTS. Most of the trajectory lengths are integer powers of two, indicating that the U-turn criterion in Equation 4 is usually satisfied only after a doubling is complete and not by one of the intermediate subtrees generated during the doubling process. This behavior is desirable insofar as it means that we only occasionally have to throw out entire half-trajectories to satisfy detailed balance.

The trajectory length (measured in number of states visited) grows as the acceptance rate target δ grows, which is to be expected since a higher δ will lead to a smaller step size ϵ, which in turn will mean that more leapfrog steps are necessary before the trajectory doubles back on itself and satisfies Equation 4.

4.4 Comparing the Efficiency of HMC and NUTS

Figure 6 compares the efficiency of HMC (with various simulation lengths $\lambda \approx \epsilon L$) and NUTS (which chooses simulation lengths automatically). The x-axis in each plot is the target δ used by the dual averaging algorithm from Section 3.2 to automatically tune the step size ϵ. The y-axis is the effective sample size (ESS) generated by each sampler, normalized by the number of gradient evaluations used in generating the samples. HMC's best performance seems to occur around $\delta = 0.65$, suggesting that this is indeed a reasonable default value

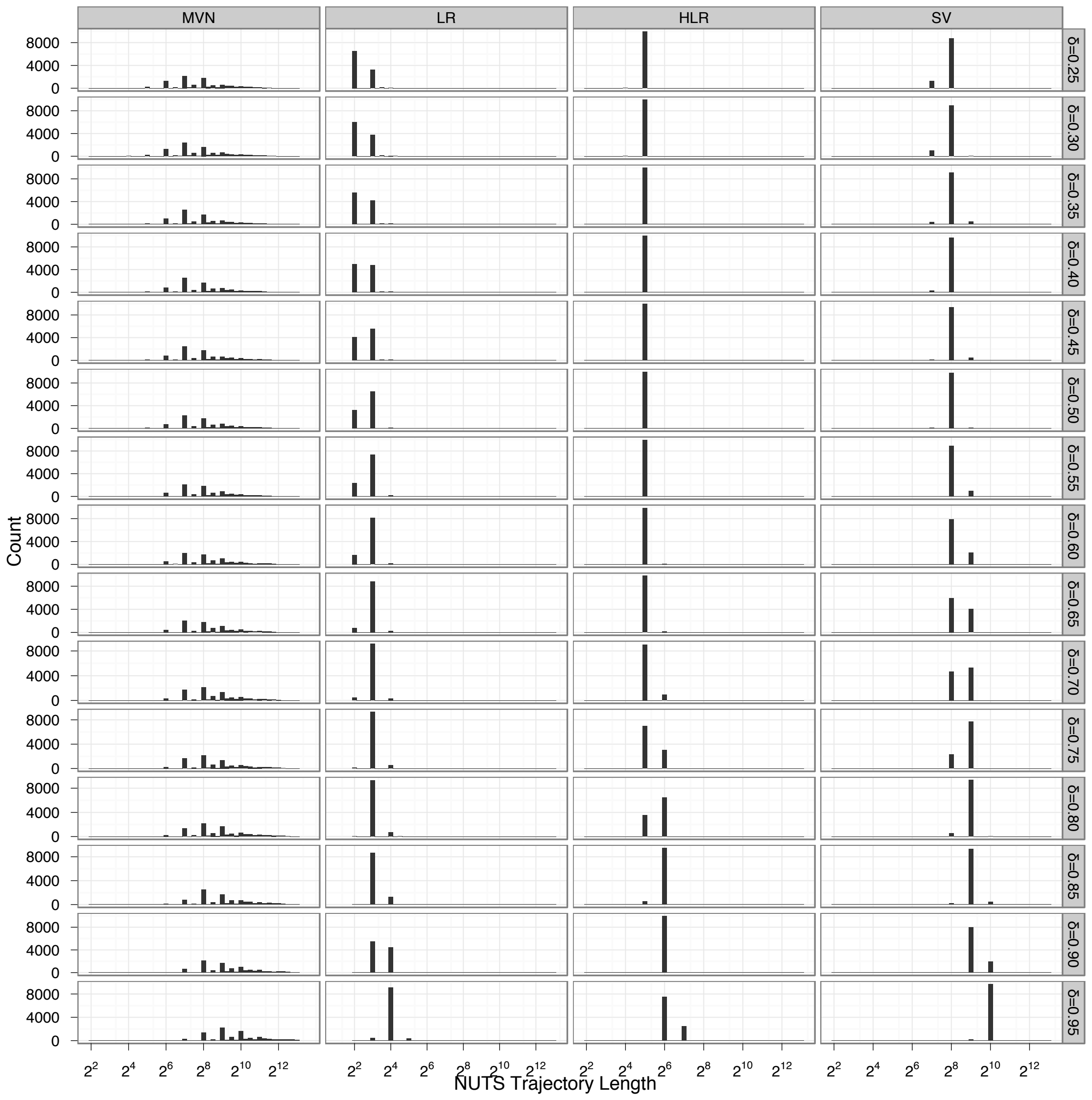

Figure 5: Histograms of the trajectory lengths generated by NUTS with various acceptance rate targets δ for the multivariate normal (MVN), logistic regression (LR), hierarchical logistic regression (HLR), and stochastic volatility (SV) models.

for a variety of problems. NUTS's best performance seems to occur around $\delta = 0.6$, but does not seem to depend strongly on δ within the range $\delta \in [0.45, 0.65]$. $\delta = 0.6$ therefore seems like a reasonable default value for NUTS.

On the two logistic regression problems NUTS is able to produce effectively independent samples about as efficiently as HMC can. On the multivariate normal and stochastic volatility problems, NUTS with $\delta = 0.6$ outperforms HMC's best ESS by about a factor of 2 and 1.5, respectively.

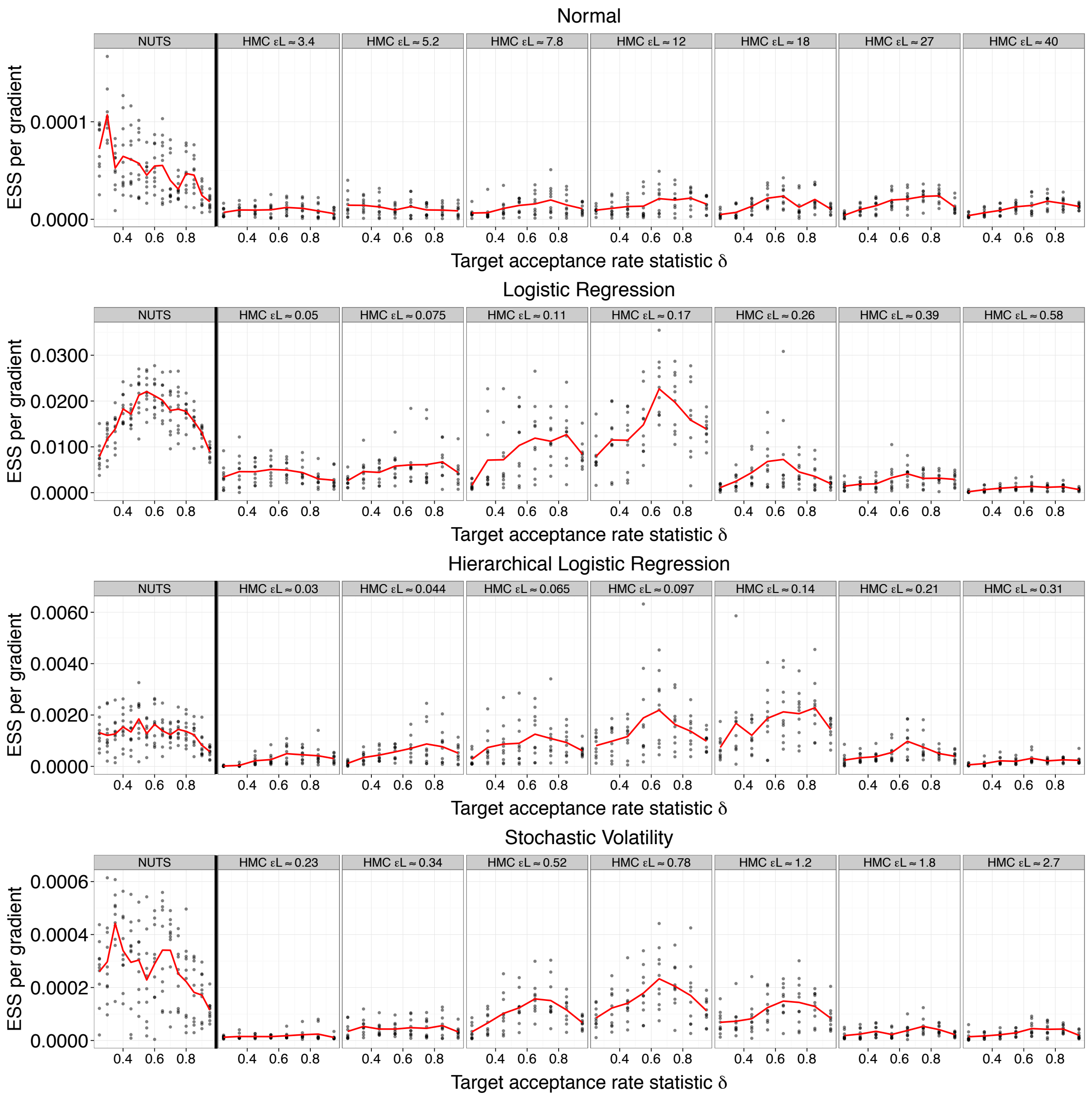

Figure 6: Effective sample size (ESS) as a function of δ and (for HMC) simulation length ϵL for the multivariate normal, logistic regression, hierarchical logistic regression, and stochastic volatility models. Each point shows the ESS divided by the number of gradient evaluations for a separate experiment; lines denote the average of the points' y-values for a particular δ. Leftmost plots are NUTS's performance, other plots shows HMC's performance for various settings of ϵL.

As expected, HMC's performance degrades if an inappropriate simulation length is chosen. Across the four target distributions we tested, the best simulation lengths λ for HMC varied by about a factor of 100, with the longest optimal λ being about 18 (for the multivariate normal) and the shortest optimal λ being about 0.14 (for the hierarchical logistic regression). In practice, finding a good simulation length for HMC will usually require

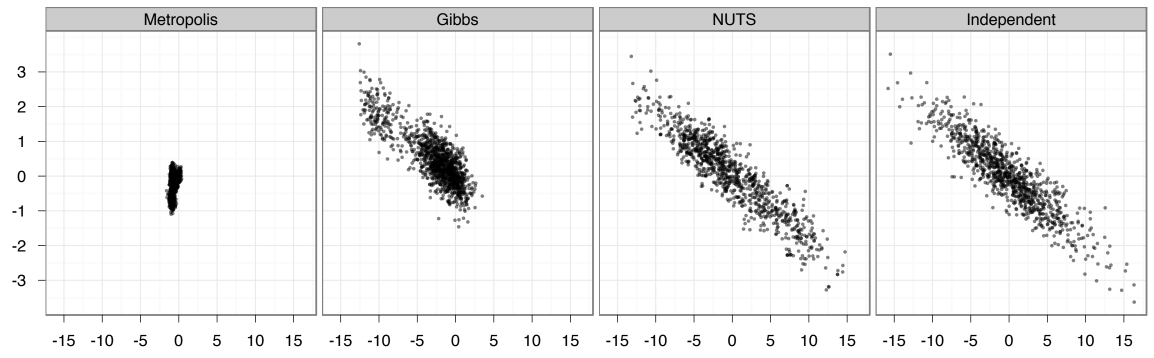

Figure 7: Samples generated by random-walk Metropolis, Gibbs sampling, and NUTS. The plots compare 1,000 independent draws from a highly correlated 250-dimensional distribution (right) with 1,000,000 samples (thinned to 1,000 samples for display) generated by random-walk Metropolis (left), 1,000,000 samples (thinned to 1,000 samples for display) generated by Gibbs sampling (second from left), and 1,000 samples generated by NUTS (second from right). Only the first two dimensions are shown here.

some number of preliminary runs. The results in Figure 6 suggest that NUTS can generate samples at least as efficiently as HMC, even discounting the cost of any preliminary runs needed to tune HMC's simulation length.

4.5 Qualitative Comparison of NUTS, Random-Walk Metropolis, and Gibbs

In Section 4.4, we compared the efficiency of NUTS and HMC. In this section, we informally demonstrate the advantages of NUTS over the popular random-walk Metropolis (RWM) and Gibbs sampling algorithms. We ran NUTS, RWM, and Gibbs sampling on the 250-dimensional multivariate normal distribution described in Section 4.1. NUTS was run with $\delta = 0.5$ for 2,000 iterations, with the first 1,000 iterations being used as warmup and to adapt ϵ. This required about 1,000,000 gradient and likelihood evaluations in total. We ran RWM for 1,000,000 iterations with an isotropic normal proposal distribution whose variance was selected beforehand to produce the theoretically optimal acceptance rate of 0.234 (Gelman et al., 1996). The cost per iteration of RWM is effectively identical to the cost per gradient evaluation of NUTS, and the two algorithms ran for about the same amount of time. We ran Gibbs sampling for 1,000,000 sweeps over the 250 parameters. This took longer to run than NUTS and RWM, since for the multivariate normal each Gibbs sweep costs more than a single gradient evaluation; we chose to nonetheless run the same number of Gibbs sweeps as RWM iterations, since for some other models Gibbs sweeps can be done more efficiently.

Figure 7 visually compares independent samples (projected onto the first two dimensions) from the target distribution with samples generated by the three MCMC algorithms. RWM has barely begun to explore the space. Gibbs does better, but still has left parts of the space unexplored. NUTS, on the other hand, is able to generate many effectively independent samples.

We use this simple example to visualize the relative performance of NUTS, Gibbs, and RWM on a moderately high-dimensional distribution exhibiting strong correlations. For the multivariate normal, Gibbs or RWM would of course work much better after an appropriate rotation of the parameter space. But finding and applying an appropriate rotation can be expensive when the number of parameters D gets large, and RWM and Gibbs both require $O(D^2)$ operations per effectively independent sample even under the highly optimistic assumption that a transformation can be found that renders all parameters i.i.d. and can be applied cheaply (e.g., in $O(D)$ rather than the usual $O(D^2)$ cost of matrix-vector multiplication and the $O(D^3)$ cost of matrix inversion). This is shown for RWM by Creutz (1988), and for Gibbs is the result of needing to apply a transformation requiring $O(D)$ operations D times per Gibbs sweep. For complicated models, even more expensive transformations often cannot render the parameters sufficiently independent to make RWM and Gibbs run efficiently. NUTS, on the other hand, is able to efficiently sample from high-dimensional target distributions without needing to be tuned to the shape of those distributions.

5. Discussion

We have presented the No-U-Turn Sampler (NUTS), a variant of the powerful Hamiltonian Monte Carlo (HMC) Markov chain Monte Carlo (MCMC) algorithm that eliminates HMC's dependence on a number-of-steps parameter L but retains (and in some cases improves upon) HMC's ability to generate effectively independent samples efficiently. We also developed a method for automatically adapting the step size parameter ϵ shared by NUTS and HMC via an adaptation of the dual averaging algorithm of Nesterov (2009), making it possible to run NUTS with no hand tuning at all. The dual averaging approach we developed in this paper could also be applied to other MCMC algorithms in place of more traditional adaptive MCMC approaches based on the Robbins-Monro stochastic approximation algorithm (Andrieu and Thoms, 2008; Robbins and Monro, 1951).

In this paper we have only compared NUTS with the basic HMC algorithm, and not its extensions, several of which are reviewed by Neal (2011). We only considered simple kinetic energy functions of the form $\frac{1}{2}r \cdot r$, but both NUTS and HMC can benefit from introducing a "mass" matrix M and using the kinetic energy function $\frac{1}{2}r^T M^{-1} r$. If M^{-1} approximates the covariance matrix of $p(\theta)$, then this kinetic energy function will reduce the negative impacts strong correlations and bad scaling have on the efficiency of both NUTS and HMC; indeed, experiments with hierarchical regression models with high correlations show substantial reduction in total computation time from nonidentity and nondiagonal mass matrices. Fitting an appropriate mass matrix can only be done during the warmup stage, and care must be taken to ensure the stability of the fitting procedure. Since using a mass matrix is equivalent to linearly transforming the parameter space (Neal, 2011), the no-U-turn condition should be computed on the transformed parameters instead of in the original space.

Another extension of HMC introduced by Neal (1994) considers windows of proposed states rather than simply the state at the end of the trajectory to allow for larger step sizes without sacrificing acceptance rates (at the expense of introducing a window size parameter that must be tuned). The effectiveness of the windowed HMC algorithm suggests that NUTS's lack of a single accept/reject step may be responsible for some of its performance gains over vanilla HMC.

Girolami and Calderhead (2011) recently introduced Riemannian Manifold Hamiltonian Monte Carlo (RMHMC), a variant on HMC that simulates Hamiltonian dynamics in Riemannian rather than Euclidean spaces, effectively allowing for position-dependent mass matrices. Although the worst-case $O(D^3)$ matrix inversion costs associated with this algorithm often make it expensive to apply in high dimensions, when these costs are not too onerous RMHMC's ability to adapt its kinetic energy function makes it very efficient. There are no technical barriers that stand in the way of combining NUTS's ability to adapt its trajectory lengths with RMHMC's ability to adapt its mass matrices, although naively applying the no-U-turn condition (which is tied to Euclidean geometry) to Riemannian algorithms may be suboptimal (Betancourt, 2013).

Like HMC, NUTS can only be used to resample unconstrained continuous-valued variables with respect to which the target distribution is differentiable almost everywhere. HMC and NUTS can deal with simple constraints such as nonnegativity or restriction to the simplex by an appropriate change of variable, but discrete variables must either be summed out or handled by other algorithms such as Gibbs sampling. In models with discrete variables, NUTS's ability to automatically choose a trajectory length may make it more effective than HMC when discrete variables are present, since it is not tied to a single simulation length that may be appropriate for one setting of the discrete variables but not for others.

Some models include hard constraints that are too complex to eliminate by a simple change of variables. Such models will have regions of the parameter space with zero posterior probability. When HMC encounters such a region, the best it can do is stop short and restart with a new momentum vector, wasting any work done before violating the constraints (Neal, 2011). By contrast, when NUTS encounters a zero-probability region it stops short and samples from the set of points visited up to that point, making at least some progress.

NUTS with dual averaging makes it possible for Bayesian data analysts to obtain the efficiency of HMC without spending time and effort hand-tuning HMC's parameters. This is desirable even for those practitioners who have experience using and tuning HMC, but it is especially valuable for those who lack this experience. In particular, NUTS's ability to operate efficiently without user intervention makes it well suited for use in generic inference engines in the mold of BUGS (Gilks and Spiegelhalter, 1992), which until now have largely relied on much less efficient algorithms such as Gibbs sampling. We are currently developing an automatic Bayesian inference system called Stan, which uses NUTS as its core inference algorithm for continuous-valued parameters (Stan Development Team, 2013). Stan promises to be able to generate effectively independent samples from complex models' posteriors orders of magnitude faster than previous systems such as BUGS and JAGS (Plummer, 2003).

In summary, NUTS makes it possible to efficiently perform Bayesian posterior inference on a large class of complex, high-dimensional models with minimal human intervention. It is our hope that NUTS will allow researchers and data analysts to spend more time developing and testing models and less time worrying about how to fit those models to data.

Acknowledgments

We thank Bob Carpenter, Michael Betancourt, and Radford Neal for helpful comments. This work was partially supported by Institute of Education Sciences grant ED-GRANTS-032309-005, Department of Energy grant DE-SC0002099, National Science Foundation grant ATM-0934516, and National Science Foundation grant SES-1023189.

Appendix A. Estimating Effective Sample Size

For a function $f(\theta)$, a target distribution $p(\theta)$, and a Markov chain Monte Carlo (MCMC) sampler that produces a set of M correlated samples drawn from some distribution $q(\theta^{1:M})$ such that $q(\theta^m) = p(\theta^m)$ for any $m \in \{1, \ldots, M\}$, the effective sample size (ESS) of $\theta^{1:M}$ is the number of independent samples that would be needed to obtain a Monte Carlo estimate of the mean of f with equal variance to the MCMC estimate of the mean of f:

$$\mathrm{ESS}_{q,f}(\theta^{1:M}) = M \frac{\mathbb{V}_q[\frac{1}{M}\sum_{s=1}^M f(\theta^s)]}{\frac{\mathbb{V}_p[f(\theta)]}{M}} = \frac{M}{1 + 2\sum_{s=1}^{M-1}(1-\frac{s}{M})\rho_s^f};$$
$$\rho_s^f \equiv \frac{\mathbb{E}_q[(f(\theta^t) - \mathbb{E}_p[f(\theta)])(f(\theta^{t-s}) - \mathbb{E}_p[f(\theta)])]}{\mathbb{V}_p[f(\theta)]},$$

where ρ_s^f denotes the autocorrelation under q of f at lag s and $\mathbb{V}_p[x]$ denotes the variance of a random variable x under the distribution $p(x)$.

To estimate ESS, we first compute the following estimate of the autocorrelation spectrum for the function $f(\theta)$:

$$\hat{\rho}_s^f = \frac{1}{\hat{\sigma}_f^2(M-s)} \sum_{m=s+1}^{M} (f(\theta^m) - \hat{\mu}_f)(f(\theta^{m-s}) - \hat{\mu}_f),$$

where the estimates $\hat{\mu}_f$ and $\hat{\sigma}_f^2$ of the mean and variance of the function f are computed with high precision from a separated 50,000-sample run of NUTS with $\delta = 0.5$. We do not take these estimates from the chain whose autocorrelations we are trying to estimate—doing so can lead to serious underestimates of the level of autocorrelation (and thus a serious overestimate of the number of effective samples) if the chain has not yet converged or has not yet generated a fair number of effectively independent samples.

Any estimator of ρ_s^f is necessarily noisy for large lags s, so using the naive estimator $\hat{\mathrm{ESS}}_{q,f}(\theta^{1:M}) = \frac{M}{1+2\sum_{s=1}^{M-1}(1-\frac{s}{M})\hat{\rho}_s^f}$ will yield bad results. Instead, we truncate the sum over the autocorrelations when the autocorrelations first dip below 0.05, yielding the estimator

$$\hat{\mathrm{ESS}}_{q,f}(\theta^{1:M}) = \frac{M}{1 + 2\sum_{s=1}^{M_f^{\mathrm{cutoff}}}(1-\frac{s}{M})\hat{\rho}_s^f}; \quad M_f^{\mathrm{cutoff}} \equiv \min_s s \quad \text{s.t. } \hat{\rho}_s^f < 0.05.$$

We found that this method for estimating ESS gave more reliable confidence intervals for MCMC estimators than the autoregressive approach used by CODA (Plummer et al., 2006). (The more accurate estimator comes at the expense of needing to compute a costly high-quality estimate of the true mean and variance of the target distribution.) The 0.05 cutoff is somewhat arbitrary; in our experiments we did not find the results to be very sensitive to the precise value of this cutoff.

References

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

A. Beskos, N. Pillai, G. Roberts, J. Sanz-Serna, and A. Stuart. Optimal tuning of the hybrid monte-carlo algorithm. *Arxiv preprint arXiv:1001.4460*, 2010.

M. Betancourt. Generalizing the no-U-turn sampler to Riemannian manifolds. *ArXiv preprint arXiv:1304.1920*, 2013.

M. Creutz. Global Monte Carlo algorithms for many-fermion systems. *Physical Review D*, 38(4):1228–1238, 1988.

H. Daume III. HBC: Hierarchical Bayes compiler, 2007. URL `http://hal3.name/HBC`.

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

A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

A. Gelman, G. Roberts, and W. Gilks. Efficient Metropolis jumping rules. *Bayesian Statistics*, 5:599–608, 1996.

A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. Chapman & Hall, 2004.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

W. Gilks and D. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 3:169–177, 1992.

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

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics (SIAM), 2008.

S. Lan, V. Stathopoulos, B. Shahbaba, and M. Girolami. Lagrangian Dynamical Monte Carlo. *ArXiv preprint arXiv:1211.3759*, 2012.

B. Leimkuhler and S. Reich. *Simulating Hamiltonian Dynamics*, volume 14. Cambridge University Press, 2004.

N. Metropolis, A. Rosenbluth, M. Rosenbluth, M. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.4, Microsoft Research Cambridge, 2010., utl=http://research.microsoft.com/infernet.

R. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, 1993.

R. Neal. An improved acceptance procedure for the hybrid Monte Carlo algorithm. *Journal of Computational Physics*, 111:194–203, 1994.

R. Neal. Slice sampling. *Annals of Statistics*, 31(3):705–741, 2003.

R. Neal. *Handbook of Markov Chain Monte Carlo*, chapter 5: MCMC Using Hamiltonian Dynamics. CRC Press, 2011.

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

A. Patil, D. Huard, and C. Fonnesbeck. PyMC: Bayesian stochastic modelling in python. *Journal of Statistical Software*, 35(4):1–81, 2010.

M. Plummer. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling, 2003. URL `http://sourceforge.net/projects/mcmc-jags/`.

M. Plummer, N. Best, K. Cowles, and K. Vines. CODA: Convergence diagnosis and output analysis for MCMC. *R News*, 6(1):7–11, March 2006.

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

Stan Development Team. Stan: A C++ library for probability and sampling, version 1.1, 2013. URL `http://mc-stan.org/`.

L. Tierney and A. Mira. Some adaptive Monte Carlo methods for Bayesian inference. *Statistics in Medicine*, 18:2507–2515, 1999.

M. Wainwright and M. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2):1–305, 2008.