An Adaptive MCMC Scheme for Setting Trajectory Lengths in Hamiltonian Monte Carlo

Matthew D. Hoffman
Google Research

Alexey Radul
Google Research

Pavel Sountsov
Google Research

Abstract

Hamiltonian Monte Carlo (HMC) is a powerful MCMC algorithm based on simulating Hamiltonian dynamics. Its performance depends strongly on choosing appropriate values for two parameters: the step size used in the simulation, and how long the simulation runs for. The step-size parameter can be tuned using standard adaptive-MCMC strategies, but it is less obvious how to tune the simulation-length parameter. The no-U-turn sampler (NUTS) eliminates this problematic simulation-length parameter, but NUTS's relatively complex control flow makes it difficult to efficiently run many parallel chains on accelerators such as GPUs. NUTS also spends some extra gradient evaluations relative to HMC in order to decide how long to run each iteration without violating detailed balance. We propose ChEES-HMC, a simple adaptive-MCMC scheme for automatically tuning HMC's simulation-length parameter, which minimizes a proxy for the autocorrelation of the state's second moments. We evaluate ChEES-HMC and NUTS on many tasks, and find that ChEES-HMC typically yields larger effective sample sizes per gradient evaluation than NUTS does. When running many chains on a GPU, ChEES-HMC can also run significantly more gradient evaluations per second than NUTS, allowing it to quickly provide accurate estimates of posterior expectations.

Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130. Copyright 2021 by the author(s).

1 Introduction and background

Hamiltonian Monte Carlo (HMC; Neal, 2011) is one of the most successful families of MCMC algorithms currently in use. Its ability to suppress the random-walk behavior that plagues most MCMC algorithms, and its relatively "black-box" nature (requiring only an unnormalized log-density and its gradient function, which can be provided using automatic differentiation) make it a common default choice for probabilistic-programming languages such as Stan (Carpenter et al., 2017). In particular, the no-U-turn sampler (NUTS; Hoffman & Gelman, 2014) variant of HMC can be run with very minimal tuning by the user, which has led to its wide adoption.

Since NUTS was introduced, the balance of floating-point computational power has moved from CPUs to special-purpose single-instruction multiple-data (SIMD) processors such as GPUs. These devices can perform huge numbers of floating-point operations per second. HMC is well positioned to exploit these resources by running many chains in parallel (Lao et al., 2020). But NUTS is a control-flow-heavy recursive algorithm, and designing implementations that take full advantage of GPU resources has been challenging (Radul et al., 2020; Lao & Dillon, 2019; Phan et al., 2019). In this paper, we propose an alternate strategy for developing a tuning-free HMC algorithm based on adaptive MCMC (Andrieu & Thoms, 2008). This algorithm can outcompete NUTS both in performance per gradient evaluation and in gradient evaluations per second on modern hardware and software platforms.

Our goal is to sample from a distribution $p(\theta) \propto e^{\mathcal{L}(\theta)}$ over a vector $\theta \in \mathbb{R}^D$. We assume we can only compute p up to a normalizing constant. We augment this model with a set of "momentum" variables $r \in \mathbb{R}^D$ that are given independent standard normal[1] distributions, so that the augmented model's joint density

[1]HMC admits a non-identity mass matrix, but an equivalent effect can be achieved by a linear change of variables (Neal, 2011). For ease of exposition, we assume this change of variables is absorbed into the definition of $\mathcal{L}$ and θ.

is $p(\theta, r) \propto \exp\{\mathcal{L}(\theta) - \frac{1}{2}||r||^2\}$. We then interpret the negative log-joint $-\log p(\theta, r)$ as a Hamiltonian, with $-\mathcal{L}(\theta)$ being a potential energy function and $\frac{1}{2}||r||^2$ being a kinetic energy function. We can update the joint state by simulating Hamilton's equations $\frac{\partial\theta}{\partial t} = r$; $\frac{\partial r}{\partial t} = \nabla\mathcal{L}(\theta)$. In practice, we cannot usually solve these equations exactly, so we use the "leapfrog" integrator, each iteration of which proceeds by the updates

$$r \leftarrow r + \tfrac{\epsilon}{2}\nabla\mathcal{L}(\theta); \quad \theta \leftarrow \theta + \epsilon r; \quad r \leftarrow r + \tfrac{\epsilon}{2}\nabla\mathcal{L}(\theta). \quad (1)$$

We will denote the application of L leapfrog updates with step size ϵ as $\text{leapfrog}_{\epsilon,L}(\theta, r)$. The leapfrog updates are reversible and volume preserving (Neal, 2011), so if we compute $\theta', -r' = \text{leapfrog}_{\epsilon,L}(\theta, r)$ then we can treat the new state θ', r' as a Metropolis proposal and accept it with probability $\alpha = \frac{p(\theta', r')}{p(\theta, r)}$. Since Hamiltonian dynamics conserve energy, if the leapfrog integrator accurately approximates the dynamics, the change in log-density from θ, r to θ', r' will be small and the probability of acceptance will be large, even if θ' is far from θ. After each of these Metropolis proposals, we resample the momenta $r \sim \mathcal{N}(0, I)$, which is a valid Gibbs-sampling move.

As with any MCMC algorithm, one can run multiple parallel HMC chains to get a larger sample size and reduce the variance of one's MCMC estimator. Implementing parallel-chain HMC on top of a vector-oriented library such as TensorFlow, JAX, or PyTorch (Abadi et al., 2015; Bradbury et al., 2017–2019; Paszke et al., 2019) that supports automatic differentiation is relatively easy as long as the chains run in lockstep; a batch of gradients for the chains can be computed in parallel quickly on SIMD accelerators such as GPUs or TPUs (Lao et al., 2020). As we will see in section 3, a GPU can run tens of chains in parallel nearly as fast as a CPU core can run a single chain.

HMC has two key parameters that must be tuned carefully to achieve good results: the step size ϵ and the number of leapfrog steps L. HMC ensures high Metropolis acceptance rates by controlling the change in energy of the Hamiltonian system. When the leapfrog integrator is run over long trajectories, this error is $O(\epsilon^2)$, so ϵ must be kept small enough to ensure a reasonable acceptance rate. On the other hand, smaller step sizes lead to either less progress (if the number of leapfrog steps L is held fixed) or more work (if the trajectory length $T \triangleq \epsilon L$ is held fixed) per iteration. Happily, the symplecticness of the leapfrog integrator ensures that the energy change does not depend strongly on L (Hairer et al., 2006), so ϵ can be independently tuned to achieve a high-but-not-too-high acceptance rate using standard heuristics from the adaptive-MCMC literature (Andrieu & Thoms, 2008).

Tuning the number-of-steps parameter L is trickier. Pasarica & Gelman (2010) observe that maximizing expected squared jumped distance (ESJD) $\mathbb{E}[||\theta' - \theta||^2]$ minimizes the variance-weighted sum over dimensions of the chain's first-order autocorrelations. This idea motivated Hoffman & Gelman (2014) to find a way to run the leapfrog integrator until the simulation makes a "U-turn"—that is, until increasing L would *decrease* the distance from the proposal to the initial state. The resulting "no-U-turn sampler" (NUTS) is widely used as a turnkey sampling algorithm in software packages such as Stan (Carpenter et al., 2017), PYMC3 (Salvatier et al., 2016), Tensorflow Probability (The TFP Team, 2018–2019; Dillon et al., 2017), Pyro (Bingham et al., 2018), and Turing (Ge et al., 2018).

We will see in section 3 that, although NUTS is reliable, robust, and easy to use, it often requires more leapfrog steps to converge and mix than an optimally tuned HMC algorithm. This is in part because it must waste about half of its leapfrog steps to satisfy detailed balance (Wu et al., 2018). Furthermore, adapting NUTS to take full advantage of SIMD hardware is challenging because it requires many control-flow operations, and each chain may use a different number of leapfrog steps per iteration (Radul et al., 2020; Lao & Dillon, 2019; Phan & Pradhan, 2019). A well-tuned SIMD-friendly implementation of a simpler HMC algorithm run on a GPU can therefore produce estimates of posterior expectations with low variance and bias in much less wallclock time than NUTS, even on the same hardware.

But this advantage only exists for "well-tuned" HMC; practitioners need some way to set HMC's parameters, and in particular the number-of-steps parameter L. In this paper, we propose an adaptive-MCMC approach to setting L, based on the heuristic of adjusting the trajectory length ϵL to increase the expected Change in the Estimator of the Expected Square (ChEES) of the parameters θ. Empirically, we find that the resulting ChEES-HMC algorithm consistently outperforms NUTS and often finds trajectory lengths that are competitive with those found by grid search. An implementation of ChEES-HMC is available as part of TensorFlow Probability[2].

Figure 1 illustrates ChEES-HMC's advantage over NUTS on a Pascal Titan X GPU. The parallel compute capacity of the GPU lets it run 100 NUTS chains in less than twice the time it takes to run 4 NUTS chains, so 100-chain NUTS gets reasonable estimates much faster than 4-chain NUTS. But NUTS takes about four times as long as ChEES-HMC to run 1000 iterations with

[2]`https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/experimental/mcmc/gradient_based_trajectory_length_adaptation.py`

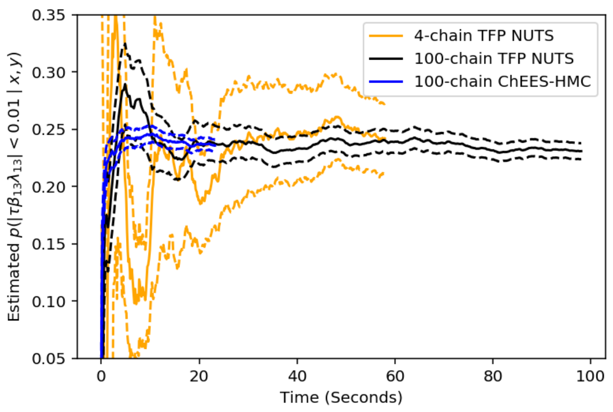

Figure 1: Estimates of the probability that the magnitude of a centered regression coefficient is less than 0.01. $\tau\beta_{13}\lambda_{13}$ is the centered parameterization of the 13th regression coefficient from the German credit sparse logistic regression model defined in section 3. Similar results for other coefficients are provided in the supplement. Estimates are obtained by discarding the second half of the chain after different numbers of iterations. ChEES-HMC is the method proposed in this work; TFP NUTS is the TensorFlow Probability (TFP) implementation of the No-U-Turn Sampler (NUTS) (Lao & Dillon, 2019; Hoffman & Gelman, 2014). All algorithms were run for 1000 iterations, with adaptation frozen after 500 iterations. Dotted lines are ± two standard errors, computed using TFP's effective sample size estimator.

100 chains, and achieves a final effective sample size about half that of ChEES-HMC; altogether ChEES-HMC gets to NUTS-quality estimates almost an order of magnitude faster.

2 Tuning HMC to maximize ChEES

In this section we develop and analyze our adaptive-HMC procedure. To develop intuition, in section 2.1 we begin by considering the behavior of HMC with an exact integrator and a multivariate-Gaussian target distribution; this analytically tractable case is important in its own right (a sampler can hardly be called "general-purpose" if it fails on multivariate Gaussians), but also serves as a proxy for more-general unimodal targets or single modes of multimodal targets.

The Gaussian example demonstrates the importance of randomizing ("jittering") HMC's trajectory length to ensure convergence in all directions. An analysis of the jittered chain's first-order autocorrelations leads us to conclude that maximizing expected squared jumped distance (ESJD; Pasarica & Gelman, 2010) leads to an average trajectory length that is as much as twice the optimal trajectory length. These too-long trajectories do extra work to generate estimates of the highest-variance parameters that are *better* than the estimates of lower-variance parameters; if we want to minimize the maximum autocorrelation across dimensions, this is a waste of effort. ESJD is also fooled by long trajectories' ability to produce *anti*-correlated samples, which are good for estimating means but bad for estimating variances. However, maximizing ESJD after centering and squaring the parameters yields a shorter, more efficient trajectory length. In section 2.2, we apply these intuitions to derive a practical adaptive-MCMC algorithm which we call ChEES-HMC.

2.1 Autocorrelation analysis of the multivariate Gaussian

We consider HMC applied to a D-dimensional multivariate Gaussian distribution. Without loss of generality,[3] we assume that this distribution has mean 0 and diagonal covariance matrix $\Sigma_{dd} = \sigma_d^2$. We further assume that we are using a small enough step size ϵ that we can approximate the action of the leapfrog$_{\epsilon,L}$ integrator with the exact Hamiltonian dynamics, that is, that for a trajectory length $t = \epsilon L$ the new state θ', r' satisfies

$$\begin{aligned} a_d^2 &= \tfrac{\theta_d^2}{\sigma_d^2} + r_d^2; \quad & \phi_d &= \tan^{-1}\left(\tfrac{\theta_d}{\sigma_d r_d}\right); \\ r_d &= a_d\cos(\phi_d); \quad & \theta_d &= a_d\sigma_d\sin(\phi_d); \\ r'_d &= a_d\cos(\phi_d + t/\sigma_d); \quad & \theta'_d &= a_d\sigma_d\sin(\phi_d + t/\sigma_d), \end{aligned} \tag{2}$$

where (a, ϕ) is the magnitude-angle representation of the phase-space state (θ, r). One can readily verify that equation 2 solves Hamilton's equations in this case.

Now, we consider the first-order autocorrelation in dimension d. At equilibrium, $a_d^2 \sim$ Exponential(2), $\phi_d \sim$ Uniform$(0, 2\pi)$, $a_d \perp\!\!\!\perp \phi_d$. The first-order autocorrelation is therefore

$$\begin{aligned} \sigma_d^{-2}\mathbb{E}[\theta'_d\theta_d] &= \mathbb{E}[a_d^2\sin(\phi_d)\sin(\phi_d + t/\sigma_d)] \\ &= \mathbb{E}[\cos(t/\sigma_d) - \cos(2\phi_d + t/\sigma_d)] \\ &= \cos(t/\sigma_d), \end{aligned} \tag{3}$$

since $\mathbb{E}[a_d^2] = 2$, $\sin(\phi)\sin(\phi+\delta) = \frac{1}{2}(\cos(\delta) - \cos(2\phi+\delta))$, and $\mathbb{E}[\cos(2\phi_d + t/\sigma_d)] = 0$. If the number of dimensions D is large, then for any t it is likely that there is some d such that $t/\sigma_d \mod 2\pi \approx 0$; that is, there may be no setting of t that leads to low autocorrelation in every dimension. This phenomenon is discussed by Neal (2011, section 3.2), who observes that it can

[3]HMC with an isotropic kinetic-energy function is invariant to shifts and rotations; kinetic-energy functions $\frac{1}{2}r^\top M^{-1}r$ with mass matrix M can be adapted to a unitary transformation $r' \equiv Ur$ by choosing $M' \equiv UMU^\top$.

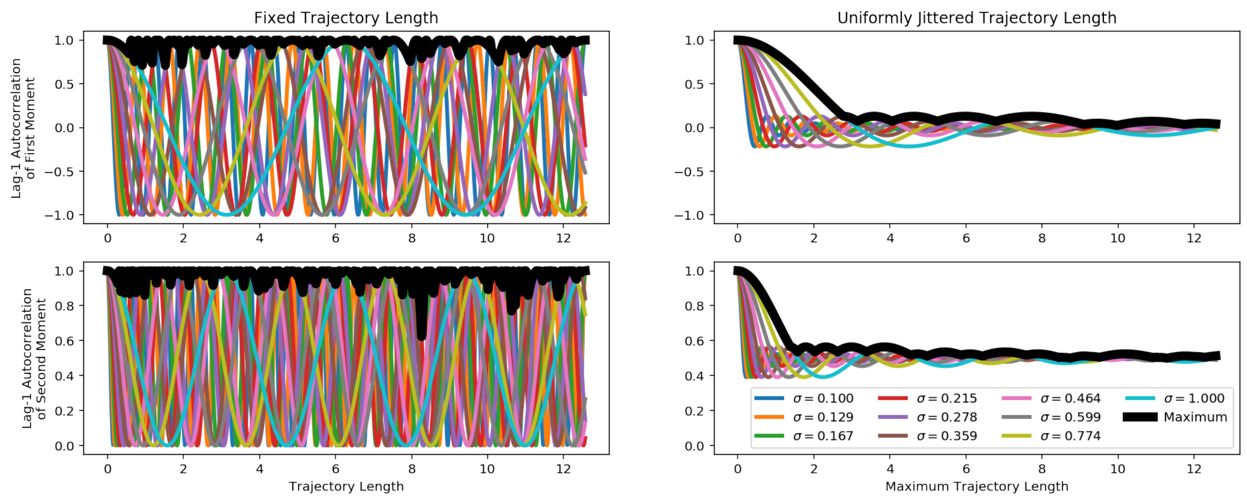

Figure 2: First-order autocorrelations for Hamiltonian Monte Carlo (HMC) targeting Gaussian distributions with 10 logarithmically spaced scales σ, as a function of HMC's trajectory-length parameter $t = \epsilon L$ (left plots) or the maximum-trajectory-length parameter T for jittered HMC (i.e., HMC where each iteration we randomly choose t uniformly between 0 and T). We assume the step size ϵ is small enough that we can ignore integration error. Heavy black lines show the worst (highest) autocorrelation across scales; if the different squared scales σ^2 were the eigenvalues of the covariance matrix of a multivariate normal distribution that we were trying to sample from using HMC, the black lines would show the autocorrelation in the slowest-mixing direction. When using a fixed trajectory length, the autocorrelations as a function of trajectory length are cosines with different periods; no matter what trajectory length we choose, there will be some direction in which we have high autocorrelation. If instead we randomly jitter the trajectory length, the autocorrelations behave like sincs (which converge to a constant) as a function of maximum trajectory length. Considering the second moment (bottom plots) instead of the first moment (top plots) makes the autocorrelation vary twice as fast, and raises the minimum achievable autocorrelation from -1 to 0 (HMC with fixed t) or from about -0.22 to about 0.39 (jittered HMC).

interfere with HMC's ergodicity unless one randomizes the trajectory length. Indeed, if we uniformly jitter the trajectory length between 0 and T, we get an autocorrelation of

$$\frac{1}{T}\int_0^T \cos(t/\sigma_d)dt = \frac{\sin(T/\sigma_d)}{T/\sigma_d}, \qquad (4)$$

which goes to 0 as T/σ_d gets large. In particular, it takes a minimum of about -0.22 at $T \approx 4.5\sigma_d$, and if $T = 4.5\sigma_{\max}$ then $\max_d \frac{\sin(T/\sigma_d)}{T/\sigma_d} \leq 0.129$.

$T = 4.5\sigma_{\max}$ produces an *anti*autocorrelated chain, which is good for estimating first moments. But if we care about estimating the expectation of a nonlinear function of our variables, we must consider the autocorrelation under that change of variables. For example, if we want to estimate posterior variances, we need to worry about the autocorrelation of θ_d^2 (whose mean and variance at equilibrium are σ^2 and $2\sigma^4$):

$$\frac{1}{2\sigma^4}\mathbb{E}[((\theta_d')^2 - \sigma^2)(\theta_d^2 - \sigma^2)] = \tfrac{1}{2}(1 + \cos(2t/\sigma_d)). \qquad (5)$$

See the supplement for a full derivation. The autocorrelation under uniform jitter is

$$\tfrac{1}{2} + \tfrac{1}{2T}\int_0^T \cos(2t/\sigma_d)dt = \tfrac{1}{2} + \tfrac{1}{2}\frac{\sin(2T/\sigma_d)}{2T/\sigma_d}. \qquad (6)$$

This differs from equation 4 in two ways: as T gets large, this no longer goes to 0, but to $\frac{1}{2}$; and it decays twice as fast, taking a minimum at $T \approx 2.25\sigma_d$. Using $T := 2.25\sigma_{\max}$ ensures that the maximum across dimensions of the first-moment autocorrelation from equation 4 is no more than 0.346, and the second-moment autocorrelation from equation 6 is no more than 0.564.

Figure 2 summarizes these results. If we want low-variance estimators of many expectations with different posterior scales, we must jitter trajectory lengths and use a maximum trajectory length of the same order as the largest posterior scale. If we jitter trajectory lengths and we want to estimate $\mathbb{E}[(\theta_d - \mathbb{E}[\theta_d])^2]$ for some d with relatively small posterior scale (so that $\sigma_d \ll \sigma_{\max}$, $\sigma_d \ll T$, and therefore $\frac{1}{2} + \frac{1}{2}\sin(T/\sigma_d)/(T/\sigma_d) \approx \frac{1}{2}$), then the maximum first-order autocorrelation over dimensions and expectations will be at least 0.5; therefore, we may as well use the shortest (and therefore cheapest) maximum trajectory length that approximately achieves this autocorrelation in all dimensions for both the first and second moments of θ, which $T := 2.25\sigma_{\max}$ does.

While this analysis may not generalize in detail to arbitrary target distributions, the Bayesian central limit theorem ensures that approximately Gaussian posterior distributions are common enough that general-purpose samplers must deal with them gracefully. That is, working well in the Gaussian case is a necessary-but-not-sufficient condition for being a good sampler. In the following section, we use the analysis above to derive a jittered adaptive-HMC algorithm based on the lessons from this section.

2.2 The ChEES criterion

Above, we argued for choosing a trajectory length ϵL in jittered HMC by trying to minimize the first-order autocorrelation of the statistic $((\theta - \mathbb{E}[\theta])^\top v)^2$, where v is a unit vector pointing in the highest-variance direction of the posterior. In this section we will propose a heuristic for doing so.

We want our heuristic to be (i) invariant to shifts and rotations, (ii) relatively insensitive to directions with low posterior variance (since under uniform jitter the long trajectories needed to explore high-variance directions will always explore low-variance directions as well), and (iii) focused on second-moment estimation (since the trajectories that are optimal for estimating first moments are longer than those that are optimal for estimating second moments, and would therefore waste computation).

We propose tuning T to maximize the following "Change in the Estimator of the Expected Square" (ChEES) criterion (expectations are with respect to the chain at equilibrium, with $\theta \sim p$, $r \sim \mathcal{N}(0, I)$, and $\theta', r' = \text{leapfrog}_{\epsilon,L}(\theta, r)$):

$$\text{ChEES} \triangleq \tfrac{1}{4}\mathbb{E}[(||\theta' - \mathbb{E}[\theta]||^2 - ||\theta - \mathbb{E}[\theta]||^2)^2]. \tag{7}$$

This criterion is invariant to shifts (because of the centering by $\mathbb{E}[\theta]$) and rotations (because it only depends on the squared norm of the centered θ). Below we will show that it satisfies the other two criteria. At equilibrium, assuming without loss of generality (to reduce clutter) that $\mathbb{E}[\theta] = 0$ and $\mathbb{E}[\theta_i\theta_j] = 0$ for $i \neq j$, and letting $\sigma_\cdot^2 \triangleq \sum_d \sigma_d^2$ be the trace of the covariance matrix of θ,

$$\begin{aligned}\text{ChEES} &\triangleq \tfrac{1}{4}\mathbb{E}[((||\theta'||^2 - \sigma_\cdot^2) - (||\theta||^2 - \sigma_\cdot^2))^2] \\ &= \tfrac{1}{2}\mathbb{E}[(||\theta||^2 - \sigma_\cdot^2)^2] \\ &\quad - \tfrac{1}{2}\mathbb{E}[(||\theta'||^2 - \sigma_\cdot^2)(||\theta||^2 - \sigma_\cdot^2)].\end{aligned} \tag{8}$$

The first term is constant with respect to T. The second includes a weighted sum of the first-order autocorrelations of θ_d^2:

$$\begin{aligned}\mathbb{E}[(||\theta'||^2 - \sigma_\cdot^2)(||\theta||^2 - \sigma_\cdot^2)] &= \textstyle\sum_{i,j} \mathbb{E}[((\theta_i')^2 - \sigma_i^2)(\theta_j^2 - \sigma_j^2)] \\ &= \textstyle\sum_i (\mathbb{E}[\theta_i^4] - \sigma_i^4)\frac{\mathbb{E}[((\theta_i')^2 - \sigma_i^2)(\theta_i^2 - \sigma_i^2)]}{\mathbb{E}[\theta_i^4] - \sigma_i^4} \\ &\quad + \textstyle\sum_{i,j\neq i} \mathbb{E}[((\theta_i')^2 - \sigma_i^2)(\theta_j^2 - \sigma_j^2)].\end{aligned} \tag{9}$$

The second sum will be small if each dimension is approximately independent under the posterior once θ has been rotated so that $\mathbb{E}[\theta_i\theta_j] = 0$ for $i \neq j$. The first sum is the sum of the autocorrelations of each θ_d^2, weighted by the variance of θ_d^2, which will generally be proportional to the posterior scale σ_d^4. This scheme therefore aggressively weights high-variance directions, as desired.

We can compute the derivative of the expected ChEES with respect to T from Hamilton's equations:

$$\begin{aligned}&\tfrac{d}{dT}\textstyle\int_{u=0}^{u=1} \mathbb{E}_r[\tfrac{1}{4}(||\theta'(uT, \theta, r) - \mathbb{E}[\theta]||^2 - ||\theta - \mathbb{E}[\theta]||^2)^2]du \\ &= \textstyle\int_{u=0}^{u=1} u\mathbb{E}_r[(||\theta'(uT, \theta, r) - \mathbb{E}[\theta]||^2 - ||\theta - \mathbb{E}[\theta]||^2) \\ &\qquad \times (\theta'(uT, \theta, r) - \mathbb{E}[\theta])^\top r']du.\end{aligned} \tag{10}$$

We can follow this gradient signal to maximize ChEES with respect to the maximum trajectory length T. The procedure is summarized in algorithm 1. We highlight in blue the new computations introduced by ChEES-HMC relative to length-jittered and step-size-adapted HMC with a fixed, user-supplied maximum trajectory length T. The rationales for some of the details follow below.

Since the scale of $\hat{g}$ is highly problem-dependent, we use Adam (Kingma & Ba, 2015) to adjust the scale of the gradient steps. We use step size $\alpha = 0.025$ (the algorithm was generally not very sensitive to this parameter; see the supplement for further analysis), no momentum ($\beta_1 = 0$), and relatively fast second-moment adaptation ($\beta_2 = 0.95$). We average $\hat{g}$ across all chains, and use the same sampled t for each of them to ensure that each chain does the same amount of work per iteration (this results in faster and simpler code). We weight the contribution of each chain to $\hat{g}$ by its acceptance probability to avoid being influenced by obviously invalid states. We estimate $\mathbb{E}[\theta]$ from the initial and final states of the integrator across chains.

When using the same step size for all chains, one of the chains may get "left behind"—that is, while warming up a chain may get stuck in a region that it cannot escape from without using a relatively small step size, while the other chains may have already escaped from this region. Tuning average acceptance probability may not solve this problem when we run many chains, since a single stuck chain contributes relatively little to the average.

Algorithm 1 ChEES-HMC; new elements in blue

```
Input: unnormalized target distribution $p(\theta) \propto e^{\mathcal{L}(\theta)}$, initial states $\theta_0^{(m)}$ for each chain $m \in \{1, \dots, M\}$, initial step size $\epsilon_0$, desired number of samples $N$, number of adaptation steps $N_{\text{warmup}}$, Halton sequence $h_{1:N}$.
Set initial trajectory length $T_0 = \epsilon_0$.
Initialize moving averages $\bar{T} = 0$, $\bar{\epsilon} = 0$.
for $n = 1$ to $N$ do
  Sample momenta $r^{(m)} \sim \mathcal{N}(0, I)$.
  Select jittered trajectory length $t_n = h_n T_{n-1}$.
  Compute HMC proposal
  $\tilde{\theta}^{(m)}, \tilde{r}^{(m)} = \text{leapfrog}_{\epsilon_{n-1}, \lceil t_n/\epsilon_{n-1} \rceil}(\theta_{n-1}^{(m)}, r^{(m)})$.
  Compute accept probabilities
  $\alpha_n^{(m)} = \min\left\{1, \exp\{\mathcal{L}(\tilde{\theta}^{(m)}) - \mathcal{L}(\theta_{n-1}^{(m)})\} \frac{\mathcal{N}(\tilde{r}^{(m)}; 0, I)}{\mathcal{N}(r^{(m)}; 0, I)}\right\}$.
  With probability $\alpha^{(m)}$, set next state $\theta_n^{(m)} := \tilde{\theta}^{(m)}$, otherwise set $\theta_n^{(m)} := \theta_{n-1}^{(m)}$.
  if $n < N_{\text{warmup}}$ then
    Compute harmonic-mean acceptance probability
    $\bar{\alpha} = ((1/M) \sum_m 1/\alpha^{(m)})^{-1}$.
    Update step size $\epsilon_n$ using dual averaging targeting $\bar{\alpha} = 0.651$.
    Estimate the means of the states
    $\hat{\tilde{\theta}} := \frac{1}{M} \sum_m \tilde{\theta}^{(m)}$, $\hat{\theta} := \frac{1}{M} \sum_m \theta_{n-1}^{(m)}$.
    Compute trajectory gradient estimates
    $\hat{g}^{(m)} := t_n(||\tilde{\theta}^{(m)} - \hat{\tilde{\theta}}||^2 - ||\theta_{n-1}^{(m)} - \hat{\theta}||^2)(\tilde{\theta}^{(m)} - \hat{\tilde{\theta}})^\top \tilde{r}^{(m)}$.
    Update log-trajectory length $\log T_n$ with Adam with weighted gradient
    $\hat{g} = \sum_m \alpha^{(m)} \hat{g}^{(m)} / (\sum_m \alpha^{(m)})$.
    Set $\bar{\epsilon} \leftarrow 0.9\bar{\epsilon} + 0.1\epsilon_n$, $\bar{T} \leftarrow 0.9\bar{T} + 0.1T_n$.
  end if
  if $n = N_{\text{warmup}}$ then
    Set $\epsilon_{n:N} := \bar{\epsilon}$, $T_{n:N} := \bar{T}$.
  end if
end for
```

(See the supplement for an example.) To ensure that all chains progress, we tune the average *harmonic* mean of the acceptance probabilities across chains. If any chains have a very low acceptance probability, this reduces the step size for all the chains, giving "stragglers" a chance to catch up.

When sampling using the leapfrog integrator we round the jittered trajectory length t up to the nearest integer multiple of the step size ϵ. We only adapt trajectory length during the initial warmup phase to avoid bias due to letting the hyperparameters depend on the chains' recent states (although this is less of a concern when averaging adaptation signals across many chains).

A possible issue with jittering trajectory lengths is that it adds another source of variance to the procedure, which may slow adaptation or lead to suboptimal mixing. To alleviate this, rather than use uniform random noise to jitter the trajectory lengths, we use a quasi-random Halton sequence (Owen, 2017), which ensures a more even distribution of trajectory lengths. Since this sequence is determined a priori, it does not affect the correctness or stationary distribution of the algorithm.

3 Experiments

In this section we empirically study the speed with which various flavors of HMC can generate large numbers of effectively independent samples, measuring compute cost in terms of both number of gradient evaluations (which is independent of hardware and software substrates) and wallclock time on a V100 GPU running in the cloud.

We studied HMC and NUTS on seven target distributions, which we describe below. Two are synthetic distributions, two are models run on data sampled from the prior, and three are models run on real data.

Ill-conditioned Gaussian ($D = 100$): A 100-dimensional multivariate-normal distribution with mean 0 and covariance Σ. The eigenvalues of Σ are drawn from a gamma distribution with shape 0.5 and scale 1, and the eigenvectors are chosen to be a random orthogonal basis. The condition number of Σ is about 1.3×10^5.

Banana ($D = 2$): This is a two-dimensional distribution with

$$\theta_1 \sim \mathcal{N}(0, 10); \quad \theta_2 \sim \mathcal{N}(0.03(\theta_1^2 - 100), 1). \tag{11}$$

It is a simple transformation of a 2-dimensional normal, but it has highly non-convex level sets.

Logistic regression ($D = 25$): This is a logistic-regression model on the numerical version of the German credit dataset (Dua & Graff, 2017). There are 24 features and an additional intercept parameter. Letting $\sigma(x) \triangleq \frac{1}{1+e^{-x}}$, the model is

$$\theta \sim \mathcal{N}(0, I); \quad y_n \sim \text{Bernoulli}(\sigma(\theta^\top x_n)). \tag{12}$$

Probit regression ($D = 25$): This is a probit-regression model on the same German credit dataset as in the logistic regression. Letting $\Phi(x) \triangleq \int_{-\infty}^x \mathcal{N}(x; 0, 1)$, the model is

$$\theta \sim \mathcal{N}(0, I); \quad y_n \sim \text{Bernoulli}(\Phi(\theta^\top x_n)). \tag{13}$$

Table 1: Effective sample size (ESS) per gradient evaluation for jittered HMC with fixed average trajectory length chosen by grid search, NUTS, EHMC, and jittered HMC tuned by gradient ascent on ESJD or ChEES. ESS depends on the statistic being estimated; we compute it for the mean and variance of each parameter and report the minimum across statistics and dimensions of the median ESS across chains, averaged across 10 runs for NUTS, EHMC, ESJD-HMC, and ChEES-HMC. We also report ± three standard errors. "HMC Grid" is the best result across the range of trajectory lengths considered. Bold denotes the best result outside of the HMC grid search. Results were generally consistent across runs; see the supplement for more detailed results.

Target	HMC Grid	NUTS	EHMC	ESJD-HMC	ChEES-HMC
Banana	1.16e-2	4.62e-3 ±1.6e-4	5.92e-3 ±2.0e-4	4.22e-3 ±1.8e-4	**9.04e-3 ±3.1e-4**
Gaussian	4.50e-4	1.60e-4 ±4.2e-6	1.70e-4 ±5.7e-6	3.41e-4 ±7.5e-6	**4.80e-4 ±1.2e-5**
Item-Response	2.71e-3	1.19e-3 ±2.8e-5	4.89e-4 ±9.7e-5	**2.46e-3 ±7.4e-5**	1.77e-3 ±9.1e-4
Logistic Regr.	4.80e-2	2.45e-2 ±3.5e-4	1.67e-2 ±4.5e-4	3.44e-2 ±6.7e-4	**5.23e-2 ±1.6e-3**
Probit Regr.	4.95e-2	2.50e-2 ±5.7e-4	1.58e-2 ±5.2e-4	3.64e-2 ±1.3e-3	**5.19e-2 ±1.5e-3**
Sparse Logistic	9.49e-4	4.07e-4 ±2.1e-5	**7.00e-4 ±2.1e-5**	6.72e-4 ±9.7e-5	5.36e-4 ±2.7e-5
Stochastic Vol.	1.71e-3	6.95e-4 ±3.8e-6	8.08e-5 ±6.4e-6	1.16e-3 ±1.6e-5	**1.94e-3 ±5.8e-5**

Sparse logistic regression ($D = 51$) This is a sparse, hierarchical version of the logistic-regression model above (Hoffman et al., 2019). The model is

$$\tau \sim \mathrm{Gamma}(0.5, 0.5); \quad \lambda_d \sim \mathrm{Gamma}(0.5, 0.5); \qquad (14)$$
$$\beta_d \sim \mathcal{N}(0, I); \quad y_n \sim \mathrm{Bernoulli}(\sigma((\tau\lambda \circ \beta)^\top x_n)).$$

We unconstrain the non-negative parameters τ and λ by sampling them in log-space.

Item-response theory ($D = 501$): This is a 1-parameter logistic item-response theory model from the Stan example models repository[4]. The model is

$$\begin{aligned}&\delta \sim \mathcal{N}(3/4, 1); \quad \beta_k \sim \mathcal{N}(0, 1); \quad \alpha_j \sim \mathcal{N}(0, 1);\\ &y_n \sim \mathrm{Bernoulli}(\sigma(\alpha_{j_n} - \beta_{k_n} + \delta)).\end{aligned} \qquad (15)$$

α_j is the ability of student j, β_k is the difficulty of question k, and δ is the average student ability. There are $J = 100$ students, $K = 400$ questions, and $N = 30105$ responses, for a total of 501 parameters. We condition on data drawn from the prior.

Stochastic volatility ($D = 3003$): This is a stochastic-volatility model due to Kim et al. (1998). For a time series of $N = 3000$ points, the model is

$$\begin{aligned}&\sigma \sim \mathrm{HalfCauchy}(0, 2); \quad \mu \sim \mathrm{Exponential}(1);\\ &\tfrac{\phi+1}{2} \sim \mathrm{Beta}(20, 1.5); \quad z_n \sim \mathcal{N}(0, 1);\\ &h_1 = \mu + \tfrac{\sigma z_1}{\sqrt{1-\phi^2}}; \quad h_{n>1} = \mu + \sigma z_n + \phi(h_{n-1} - \mu);\\ &y_n \sim \mathcal{N}(0, e^{h_n/2}).\end{aligned} \qquad (16)$$

There are three top-level parameters and 3000 per-time-step latent variables, for a total of 3003 dimensions. We condition on data y drawn from the prior.

[4]https://github.com/stan-dev/example-models/blob/master/misc/irt/irt.stan

3.1 Effective samples per gradient

For each target distribution and algorithm, we ran 100 parallel chains with step size tuned by dual averaging (Nesterov, 2009; Hoffman & Gelman, 2014) to achieve a harmonic-mean acceptance rate of 0.651. Initial step sizes were chosen by repeatedly halving the step size (starting from a consistently too-large value of 1.0) until an HMC proposal with a single leapfrog step achieved a harmonic-mean acceptance probability of at least 0.5.

For each of a logarithmically spaced series of fixed maximum trajectory lengths T, we ran HMC for 2000 iterations, discarding the first 1000 as warmup. We also ran 10 runs of ChEES-HMC, ESJD-HMC (identical to ChEES-HMC but maximizing ESJD instead of ChEES), empirical HMC[5] (EHMC; Wu et al., 2018), and TensorFlow Probability's NUTS implementation (Lao & Dillon, 2019), again with 100 parallel chains, 2000 iterations, and 1000 discarded warmup iterations. For each chain, we used Tensorflow Probability (The TFP Team, 2018–2019) to estimate effective sample size (ESS) for each dimension θ_d of the parameter vector θ as well as for the square of each dimension θ_d^2 (which is more relevant for estimating posterior variances). We report ESS per gradient evaluation (including warmup) as a hardware-independent measure of efficiency. NUTS and EHMC use a different number of leapfrog steps per chain; we use batched, padded implementations that wait until each chain has finished (Lao & Dillon, 2019), but only count useful gradient evaluations.

[5]EHMC has an important number-of-leapfrog-steps-during-warmup hyperparameter L_0. Unfortunately, Wu et al. (2018) give no guidance on how to set L_0, so we follow their code (https://github.com/jstoehr/eHMC) and set L_0 for EHMC to the average number of leapfrog steps taken by NUTS on each target (although this would not be available in real applications). We do not count the cost of this NUTS pilot run against EHMC.

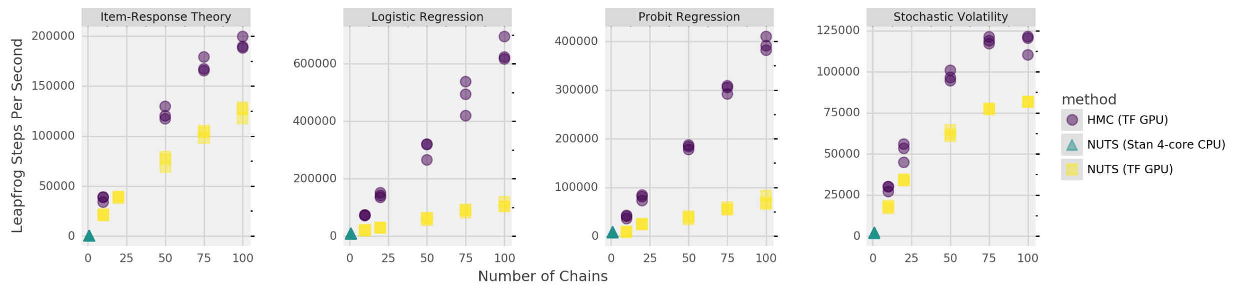

Figure 3: Total number of gradients evaluated per second summed across chains, versus number of parallel chains for HMC and NUTS run on a V100 GPU and NUTS run on a 4-core CPU. HMC incurs much less overhead than NUTS on GPU, and GPU-HMC can run dozens of chains with little increase in wallclock time. We do not distinguish ChEES-HMC here because the adaptation scheme's overhead is negligible

Table 1 summarizes the results. Plots with more detailed results are available in the supplement. ChEES-HMC consistently performs well, outperforming NUTS and achieving the top ESS/gradient among adaptive algorithms on 5 of 7 targets, and in the other two cases scoring within a factor of two of the best HMC procedure found by grid search. ESJD-HMC twice slightly outperforms ChEES-HMC, but when it underperforms the gap is more dramatic. Despite having access to information from a NUTS pilot run, EHMC does not consistently outperform NUTS. We believe the superior results reported by Wu et al. (2018) are because they only considered ESS in terms of mean estimates—for example, on the logistic regression target, EHMC's ESS for means is more than twice its ESS for variances.

These results suggest that ChEES-HMC can be at least as efficient as NUTS in terms of ESS per gradient evaluation, without requiring the user to manually tune trajectory lengths. ChEES-HMC picks up a still-greater speed advantage over NUTS when running multiple chains on a GPU.

3.2 Wallclock time per gradient

How quickly HMC generates useful samples is a function not just of the algorithm, but of its implementation and especially of the hardware and software platforms it runs on. In this section, we examine the ability of HMC and NUTS to scale to many chains on V100 GPUs running on cloud servers, and compare the results to a state-of-the-art single-threaded CPU implementation of NUTS run on a 4-core desktop CPU. The results will show how much of an advantage practitioners can get from using modern hardware accelerators.

On GPU, we ran TensorFlow Probability's implementations of HMC and NUTS with 10, 20, 50, 75, and 100 parallel chains, targeting the logistic regression, probit regression, item-response theory, and stochastic-volatility distributions. We omit the Gaussian and banana distributions (since they are synthetic), as well as the sparse logistic regression (since the cost of its gradient evaluations is essentially identical to that of the basic logistic regression). Each target/number-of-chains pair was run three times. We also ran Stan targeting the same distributions with 4 parallel chains 60 times. In each experiment, we measured the total number of gradient evaluations (summing across chains) per second to evaluate the implementations' throughput on these platforms.

Figure 3 displays the results. On GPU, HMC's throughput increases linearly with the number of parallel chains up to at least 50 chains for all targets considered, and only starts to roofline at around 75–100 chains for the higher-dimensional item-response-theory and stochastic-volatility models. That is, the cost in wallclock time of running 50 HMC chains on the GPU is not much more than that of runing 10 chains. NUTS on GPU scales up to many chains well, but it has significantly higher overhead than HMC due to its more elaborate control flow and padding scheme. Stan's CPU-based NUTS implementation yields reasonable throughput per chain, but on a 4-core CPU it cannot compete with the GPU implementations in terms of total throughput.

In summary, GPUs can cheaply run many parallel HMC chains (automatically yielding dramatic increases in ESS), but GPU implementations of NUTS incur significant control-flow overhead relative to simpler HMC schemes. Taken with the results of section 3.1, this implies that ChEES-HMC run on GPU can yield ESS-per-second rates orders of magnitude higher than NUTS run on a 4-core CPU.

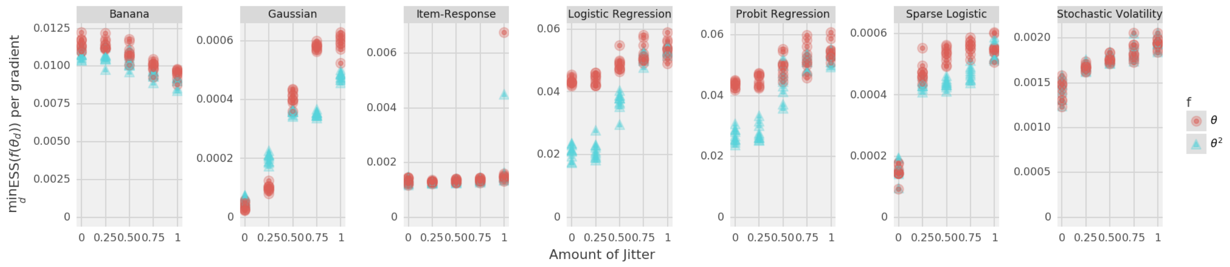

Figure 4: Effective sample size (ESS) per gradient evaluation for ChEES-HMC with trajectory length jittered by scaling the Halton sequence between $(1-\alpha)T$ and T for $\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$. Red circles and blue triangles denote ESS for estimating first and second moments, respectively.

3.3 Jitter ablations

We studied the effects of applying different amounts of trajectory-length jitter to ChEES-HMC, and of replacing the Halton sequence used to jitter trajectory lengths with standard uniform pseudo-random numbers. The results are summarized in figure 4 and in the supplement.

Uniformly jittering trajectory length between 0 and T is always close to optimal, and lower amounts of jitter are often worse. As predicted in section 2.1, using too little jitter in the Gaussian case is disastrous; inadequate jitter also leads to poor results for the logistic, probit, and sparse logistic regression models. In other models jitter has little effect on effective sample size per gradient. We conclude that jittering the number of leapfrog steps uniformly between 1 and T/ϵ offers robustness against resonances with little downside.

Using Halton sequences instead of uniform random numbers consistently reduces the variability of the trajectory length chosen by ChEES-HMC. The step-size tuning procedure exhibits relatively little variation under either kind of jitter.

4 Discussion

We have proposed ChEES-HMC, an adaptive Hamiltonian Monte Carlo variant that can automatically tune its step size and trajectory-length parameters. ChEES-HMC is competitive with the widely used NUTS algorithm in terms of effective samples per gradient evaluation, but it really shines when running many parallel chains on a GPU, since its lack of control-flow overhead lets it fully exploit these accelerators' massive floating-point computational resources.

We found that, when run on a GPU, ChEES-HMC can run tens of chains at a wallclock-time cost similar to that of running a few chains on a CPU. If, as is common in Bayesian inference problems, we need only an ESS of a few hundred before our Monte Carlo error is small relative to the posterior uncertainty, we may be able to get a sufficient number of independent samples by running, say, 50 chains for a few iterations past warmup (Hoffman & Ma, 2020). This is far less than the hundreds or thousands of post-warmup iterations practictioners often run on CPU-based workflows.

Acknowledgements

We thank the entire Tensorflow Probability team at Google for many helpful discussions, especially Rif A. Saurous and Sharad Vikram for their helpful comments on early versions of the paper.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

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

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX, 2017–2019. URL `https://github.com/google/jax`. Specifically the vmap functionality.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., and Saurous, R. A. TensorFlow Distributions, 2017. URL `https://arxiv.org/abs/1711.10604`.

Dua, D. and Graff, C. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Ge, H., Xu, K., and Ghahramani, Z. Turing: A language for flexible probabilistic inference. 2018.

Hairer, E., Lubich, C., and Wanner, G. *Geometric numerical integration: Structure-preserving algorithms for ordinary differential equations*, volume 31. Springer Science & Business Media, 2006.

Hoffman, M. and Ma, Y.-A. Black-Box variational inference as distilled Langevin dynamics. In *ICML 2020*, 2020.

Hoffman, M., Sountsov, P., Dillon, J. V., Langmore, I., Tran, D., and Vasudevan, S. NeuTra-lizing bad geometry in Hamiltonian Monte Carlo using neural transport. March 2019.

Hoffman, M. D. and Gelman, A. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *The Journal of Machine Learning Research*, 2014.

Kim, S., Shephard, N., and Chib, S. Stochastic volatility: likelihood inference and comparison with arch models. *The review of economic studies*, 65(3):361–393, 1998.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Lao, J. and Dillon, J. V. Unrolled implementation of no-U-turn sampler, August 2019. URL `https://github.com/tensorflow/probability/blob/master/discussion/technical_note_on_unrolled_nuts.md`. Software contributed to TensorFlow Probability as `https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/mcmc/nuts.py`.

Lao, J., Suter, C., Langmore, I., Chimisov, C., Saxena, A., Sountsov, P., Moore, D., Saurous, R. A., Hoffman, M. D., and Dillon, J. V. tfp.mcmc: Modern Markov chain Monte Carlo tools built for modern hardware. *arXiv preprint arXiv:2002.01184*, 2020.

Neal, R. M. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*. CRC Press New York, NY, 2011.

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

Owen, A. B. A randomized Halton algorithm in R. *arXiv preprint arXiv:1706.02808*, 2017.

Pasarica, C. and Gelman, A. Adaptively scaling the Metropolis algorithm using expected squared jumped distance. *Statistica Sinica*, pp. 343–364, 2010.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, 2019.

Phan, D. and Pradhan, N. Iterative NUTS, May 2019. URL `https://github.com/pyro-ppl/numpyro/wiki/Iterative-NUTS`.

Phan, D., Pradhan, N., and Jankowiak, M. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

Radul, A., Patton, B., Maclaurin, D., Hoffman, M. D., and Saurous, R. A. Automatically batching control-intensive programs for modern accelerators. In *Machine Learning Systems*, 2020.

Salvatier, J., Wiecki, T. V., and Fonnesbeck, C. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2016(4):1–24, 2016.

The TFP Team. TensorFlow Probability, 2018–2019. URL `https://github.com/tensorflow/probability`.

Wu, C., Stoehr, J., and Robert, C. P. Faster hamiltonian monte carlo by learning leapfrog scale. October 2018.