4th Symposium on Advances in Approximate Bayesian Inference, 2022 1–31

Metropolis Augmented Hamiltonian Monte Carlo

Anonymous Authors
Anonymous Institution

Abstract

Hamiltonian Monte Carlo (HMC) is a powerful Markov Chain Monte Carlo (MCMC) method for sampling from complex high-dimensional continuous distributions. However, in many situations it is necessary or desirable to combine HMC with other Metropolis-Hastings (MH) samplers. The common HMC-within-Gibbs strategy implies a trade-off between long HMC trajectories and more frequent other MH updates. Addressing this trade-off has been the focus of several recent works. In this paper we propose Metropolis Augmented Hamiltonian Monte Carlo (MAHMC), an HMC variant that allows MH updates within HMC and eliminates this trade-off. Experiments on two representative examples demonstrate MAHMC's efficiency and ease of use when compared with within-Gibbs alternatives.

1. Introduction

Hamiltonian Monte Carlo (HMC) is a popular Markov Chain Monte Carlo (MCMC) method. It samples from complex high-dimensional distributions $\pi(q) \propto \exp(-U(q))$ on continuous variables $q \in \mathcal{R}^n$, where $U(q) : \mathcal{R}^n \to \mathcal{R}$ is commonly referred to as the potential energy. HMC has enjoyed remarkable empirical success, due to the use of powerful symplectic integrators (Leimkuhler and Reich, 2004) (e.g. the leapfrog integrator) to maintain high acceptance probabilities for long-range gradients-guided proposals (Neal, 2012b; Betancourt, 2017). Common HMC implementations involve simulating the Hamiltonian dynamics for multiple leapfrog steps, followed by a Metropolis Hastings (MH) correction. Chen et al. (2020) recently rigorously establishes the importance of using multiple leapfrog steps/long trajectories for HMC's efficiency, especially when compared with the Metropolis Adjusted Langevin Algorithm (MALA) (Dwivedi et al., 2019), a special case of HMC using only one leapfrog step and a widely used algorithm in Bayesian statistics and machine learning.

However, we often face distributions of the form $\pi(q^H, q^O) \propto \exp(-U(q^H, q^O))$, where we can only use HMC for the continuous variables $q^H \in \mathcal{R}^n$, and it is necessary or desirable to use some other MH samplers for the variables q^O (Sec. 4.3 of Neal (2012b)). Some common situations include (1) when q^O are discrete, (2) when we have specialized MH samplers for q^O that are efficient/easy to use, or (3) when q^O are continuous but $\nabla_{q^O} U(q^H, q^O)$ is expensive or impossible to compute. In such cases, we typically adopt an HMC-within-Gibbs strategy (Neal, 2012a; Dang et al., 2019; Kelly et al., 2021), where we alternate between HMC updates and other MH updates. However, this implies a trade-off between long HMC trajectories and more frequent other MH updates. Longer HMC trajectories can help suppress random walk behavior, but might hurt overall sampling since other MH updates can only be done infrequently (between HMC updates). Shorter HMC trajectories allow more frequent other MH updates, but lead to increased random walk behavior.

Algorithm 1: Basic components for sampling from $\pi(q) \propto \exp(-U(q)), q \in \mathcal{R}^n$

def `leapfrog` $(q, p, \epsilon|U)$
  $p \leftarrow p - \frac{\epsilon}{2}\nabla U(q)$; $q \leftarrow q + \epsilon p$; $p \leftarrow p - \frac{\epsilon}{2}\nabla U(q)$;
  return q, p;
end
def `MH_correction` $(q_0, p_0, q, p, v|U, K)$
  $E_0 \leftarrow U(q_0) + K(p_0)$; $E \leftarrow U(q) + K(q)$;
  if $|v| \leq \exp(-E + E_0)$ **then**
    $v \leftarrow v \exp(-E_0 + E)$
  else $q, p \leftarrow q_0, p_0$;
  return q, p, v;
end
def `multiple_leapfrogs` $(q, p, \epsilon, L|U)$
  for 1 **to** n **do** $q, p \leftarrow$ `leapfrog` $(q, p, \epsilon|U)$;
  $p \leftarrow -p$;
  return q, p;
end

Several recent works have focused on the above trade-off. Neal (2020) combines MALA with partial momentum refreshment (Horowitz, 1991) and non-reversible Metropolis accept/reject decisions to allow more frequent other MH updates while suppressing random walk behavior. Zhou (2020, 2021) proposes mixed HMC (M-HMC) for distributions with mixed discrete and continuous variables to allow making discrete MH updates as part of an HMC trajectory. In this paper, we propose Metropolis Augmented Hamiltonian Monte Carlo (MAHMC) to completely eliminate this trade-off. MAHMC interprets HMC as a deterministic MH proposal, and augments HMC by introducing MH updates as part of the HMC trajectory. MAHMC is generally applicable, can be trivially implemented on top of HMC with no additional overhead, and includes M-HMC with Laplace momentum as a special case. We demonstrate MAHMC's advantage over within-Gibbs alternatives (Neal, 2020) and ease of use on two representative examples in Sec. 4.

2. Background

2.1. HMC

HMC introduces auxiliary momentum $p \in \mathcal{R}^n$ associated with kinetic energy $K(p) = \sum_{i=1}^{n} p_i^2/2$, and simulates the Hamiltonian dynamics for L steps of size ϵ using the leapfrog integrator (*leapfrog* in Algo. 1), before making the final MH correction (*HMC* in Algo. 2). We observe that we can in fact interpret the multiple leapfrog steps (*multiple_leapfrogs* in Algo. 1) as a deterministic MH proposal, and derive the final MH correction as the usual MH acceptance probability for the deterministic proposal (Sec. 5.2 of Betancourt (2017)).

2.2. MALA and MALA variants

Using $L = 1$ leapfrog step in HMC results in the widely used special case MALA. MALA uses gradients information, and allows more frequent other MH updates, but suffers from random

Algorithm 2: Algorithms for sampling from $\pi(q) \propto \exp(-U(q)), q \in \mathcal{R}^n$

def `HMC` $(q_0, \epsilon, L|U, K)$ `// Sec. 2.1`
  $p_0 \sim N(0, I_n)$; $q, p \leftarrow q_0, p_0$;
  $q, p \leftarrow$ `multiple_leapfrogs` $(q, p, \epsilon, L|U)$; `// Defined in Algo. 1`
  $q, p, v \leftarrow$ `MH_correction` $(q_0, p_0, q, p, Uniform(0,1)|U, K)$; `// Defined in Algo. 1`
  return q;
end
def `MALA-P` $(q_0, p_0, \epsilon, \alpha|U, K)$ `// Sec. 2.2`
  $n \sim N(0,_n)$; $p_0 \leftarrow \alpha p_0 + \sqrt{1-\alpha^2} n$; $q, p \leftarrow q_0, p_0$;
  $q, p \leftarrow$ `multiple_leapfrogs` $(q, p, \epsilon, 1|U)$; `// Defined in Algo. 1`
  $q, p, v \leftarrow$ `MH_correction` $(q_0, p_0, q, p, Uniform(0,1)|U, K)$; `// Defined in Algo. 1`
  return $q, -p$;
end
def `MALA-PN` $(q_0, p_0, v, \epsilon, \alpha, \delta|U, K)$ `// Sec. 2.2`
  $n \sim N(0, I_n)$; $p_0 \leftarrow \alpha p_0 + \sqrt{1-\alpha^2} n$; $q, p \leftarrow q_0, p_0$;
  $q, p \leftarrow$ `multiple_leapfrogs` $(q, p, \epsilon, 1|U)$; `// Defined in Algo. 1`
  $q, p, v \leftarrow$ `MH_correction` $(q_0, p_0, q, p, v|U, K)$; `// Defined in Algo. 1`
  $v \leftarrow (v + 1 + \delta) \bmod 2 - 1$;
  return $q, -p, v$;
end

walk behavior due to the use of short trajectories with frequent momentum refreshments. Partial momentum refreshment (Horowitz, 1991) (MALA-P in Algo. 2) was proposed as a possible remedy. However, as Neal (2020) explains, a rejection would lead MALA-P to almost double back on itself, making it less efficient than HMC with long trajectories.

Neal (2020) proposes a non-reversible scheme for Metropolis accept/reject decisions (MALA-PN in Algo. 2) to maintain the ability to make frequent MH updates while further suppressing the random walk behavior that comes from MALA-P doubling back on itself due to rejections. MALA-PN produces long rejection-free runs by encouraging rejections to cluster together, and demonstrates improved performance on multiple problems.

2.3. Within-Gibbs sampling from $\pi(q^H, q^O) \propto \exp(-U(q^H, q^O)), q^H \in \mathcal{R}^n$

For distributions $\pi(q^H, q^O) \propto \exp(-U(q^H, q^O))$, $q^H \in \mathcal{R}^n$ where we want to use MH updates for q^O, we refer to the common strategy of alternating between updates of $q^H \in \mathcal{R}^n$ (using a sampler for continuous distributions) and MH updates of the other variables q^O as within-Gibbs sampling. In our experiments, we use four types of within-Gibbs samplers as baselines: (1) MALA within Gibbs (MALAwG), (2) HMC within Gibbs (HwG), (3) MALA-P within Gibbs (MALA-PwG), and (4) MALA-PN within Gibbs (MALA-PNwG).

2.4. M-HMC for distributions with mixed discrete and continuous variables

Zhou (2020, 2021) proposes M-HMC, an HMC variant that evolves the discrete and continuous variables in tandem for distributions with mixed support. M-HMC naturally supports frequent MH updates within long HMC trajectories, and demonstrates improved perfor-

Algorithm 3: MAHMC. Blue highlights changes on top of naive MH within HMC.

def `MAHMC` $(q_0^H, p_0^H, q_0^O, \epsilon, L | U, K, \mathbb{Q}_i, i = 1, \ldots, N^O, \mathbb{P}^{\mathcal{D}})$ `// Sec. 3.1`
  $q^H, p^H, q^O \leftarrow q_0^H, p_0^H, q_0^O$; $D \sim \mathbb{P}^{\mathcal{D}}(\cdot)$; $\Delta E \leftarrow 0$;
  for $j \leftarrow 1$ **to** L **do**
    if $D_j = 0$ **then**
      $q^H, p^H \leftarrow$ `leapfrog`$(q^H, p^H, \epsilon | U(\cdot, q^O))$; `// Defined in Algo. 1`
    else
      $\tilde{q}^O \sim \mathbb{Q}_{D_j}(\cdot | q^H, q^O)$;
      if $Uniform(0,1) \leq \frac{\exp(-U(q^H, \tilde{q}^O))\mathbb{Q}_{D_j}(q^O | q^H, \tilde{q}^O)}{\exp(-U(q^H, q^O))\mathbb{Q}_{D_j}(\tilde{q}^O | q^H, q^O)}$ **then**
        $q^O \leftarrow \tilde{q}^O$; $\Delta E \leftarrow \Delta E + U(q^H, \tilde{q}^O) - U(q^H, q^O)$;
  end
  $E \leftarrow U(q^H, q^O) + K(p^H)$; $E_0 \leftarrow U(q_0^H, q_0^O) + K(p^H)$;
  if $Uniform(0,1) \leq \frac{\exp(-E)}{\exp(-E_0)} \frac{\mathbb{P}^{\mathcal{D}}(D^{-1})\exp(\Delta E)}{\mathbb{P}^{\mathcal{D}}(D)}$ **then** $p^H \leftarrow -p^H$;
  else $q^H, p^H, q^O \leftarrow q_0^H, p_0^H, q_0^O$;
  return q^H, p^H, q^O
end

mance over strong baselines. In Sec. 3.2, we show that the practically useful M-HMC implementation (with Laplace momentum) can be seen as a special case of MAHMC.

3. Metropolis Augmented Hamiltonian Monte Carlo (MAHMC)

3.1. Augmenting HMC with MH updates

Motivated by the interpretation of HMC as a deterministic MH proposal (Sec. 2.1), for a given distribution $\pi(q^H, q^O) \propto \exp(-U(q^H, q^O))$, MAHMC allows more frequent MH updates for q^O by combining leapfrog steps for q^H and MH updates for q^O (including the MH correction) into a single MH proposal, followed by an additional final MH correction.

Formally, for some given step size ϵ and number of steps L, an MAHMC iteration makes use of the leapfrog integrator $leapfrog(q, p, \epsilon | U(\cdot, q^O))$ (Algo. 1) and N^O MH proposals $\mathbb{Q}_i(\tilde{q}^O | q^H, q^O), i = 1, \ldots, N^O$ to construct an MH proposal making L total updates of q^H, q^O. For a sequence $D \in \{0, 1, \ldots, N^O\}^L$ of L integers, define $D^{-1} = (D_L, D_{L-1}, \ldots, D_1)$. Starting from q_0^H, q_0^O, MAHMC first resamples the momentum p_0^H from $N(0, I_n)$, then samples a sequence of L variable updates represented as a sequence D of L integers from some distribution $\mathbb{P}^{\mathcal{D}}(D)$, applies the variable updates one at a time, before making a final MH correction. See Algo. 3 for a detailed description of an MAHMC iteration.

Critically, the use of MH correction in each MH update serves as a mechanism to prevent MAHMC from deviating too much into low-probability regions. As we empirically verify in Sec. 4, even with the additional MH updates as part of the trajectory, MAHMC can maintain high acceptance probabilities for long-range proposals, similar to HMC. This eliminates the need to balance long HMC trajectories and more frequent other MH updates, and contributes to MAHMC's improved performance over other within-Gibbs alternatives.

3.2. Connections to M-HMC

Zhou (2020, 2021) derives M-HMC using auxiliary Hamiltonian dynamics for the discrete variables. We note that the practically useful M-HMC implementation using Laplace momentum is in fact a variant of a special case of MAHMC, where $\mathbb{P}^{\mathcal{D}}$ is implicitly defined using the auxiliary Hamiltonian dynamics and the proposals $\mathbb{Q}_i$ are for single discrete variables (given all other discrete and continuous variables). However, M-HMC differs from MAHMC in its use of a persistent kinetic energy $k^{\mathcal{D}}$ for the MH corrections in all MH updates.

3.3. MAHMC in Algo. 3 satisfies detailed balance with respect to $\pi(q^H, q^O)$

Proof Sketch To establish detailed balance, we make use of the concept of *probabilistic paths*, similar to Sec. 2.3 in Zhou (2020, 2021). Starting from $s = (q_0^H, p_0^H, q_0^O)$, a *probabilistic path* $\boldsymbol{t}$ contains information about all randomness in an MAHMC iteration (Algo. 3):

1. The sequence D of L integers specifying which updates to use at each of the L steps.
2. The actual states $q_j^H, p_j^H, q_j^O, j = 1, \ldots, L$ (before final MH correction) at each step.
3. The sequence of proposed states $\tilde{q}_j^H, \tilde{p}_j^H, \tilde{q}_j^O, j = 1, \ldots, L$ at each step.
   - If $D_j = 0$, $\tilde{q}_j^H, \tilde{p}_j^H = \textit{leapfrog}(q_j^H, p_j^H, \epsilon | U(\cdot, q_j^O)), \tilde{q}_j^O = q_{j-1}^O$.
   - If $D_j > 0$, $\tilde{q}_j^H, \tilde{p}_j^H = q_j^H, p_j^H$, $\tilde{q}_j^O$ is a sample from $\mathbb{Q}_{D_j}(\cdot | q_j^H, q_j^O)$.
4. The sequence of accept/reject decisions $a_j \in \{\texttt{True}, \texttt{False}\}, j = 1, \ldots, L$ at each step. Note that we always accept ($a_j = \texttt{True}$) for leapfrog updates ($D_j = 0$).

We can think of an MAHMC iteration as first sampling a probabilistic path $\boldsymbol{t}$ which brings s_0 to s_L, where $s_j = (q_j^H, p_j^H, q_j^O), j = 1, \ldots, L$, before making an MH correction to either accept s_L or reject and return to s_0. We can *reverse* a probabilistic path $\boldsymbol{t}$ to get probabilistic path $\boldsymbol{t}^{-1}$ which brings s_L back to s_0. $\boldsymbol{t}^{-1}$ uses the sequence of L updates specified by D^{-1}, and reverses the sequence of actual and proposed states (with proper momentum negation) as well as the sequence of accept/rejection decisions. Denote by $\mathbb{P}(\boldsymbol{t}|s_0)$ the probability of sampling $\boldsymbol{t}$ starting from s_0, and $\mathbb{P}(s|\boldsymbol{t}), s \in \{s_0, s_L\}$ the MH correction step in MAHMC, we can derive the transition probability of MAHMC as $\mathbb{P}(s'|s) = \sum_{\boldsymbol{t}:s\to s'} \mathbb{P}(s'|\boldsymbol{t})\mathbb{P}(\boldsymbol{t}|s)$. Define $E(s) = U(q^H, q^O) + K(p^H)$. We can establish the desired detailed balance if we can prove $\exp(-E(s))\mathbb{P}(s'|s) = \exp(-E(s'))\mathbb{P}(s|s')$.

We show that $\mathbb{P}(s_L|\boldsymbol{t}) = \min\left\{1, \frac{\exp(-E(s_L))\mathbb{P}(\boldsymbol{t}^{-1}|s_L)}{\exp(-E(s_0))\mathbb{P}(\boldsymbol{t}|s_0)}\right\}$ is our desired MH acceptance probability. Note that $\exp(-E(s))\mathbb{P}(s|s) = \exp(-E(s))\mathbb{P}(s|s)$ is trivially true. For $s' \neq s$,

$$
\begin{aligned}
\exp(-E(s))\mathbb{P}(s'|s) &= \exp(-E(s)) \sum_{\boldsymbol{t}:s\to s'} \mathbb{P}(s'|\boldsymbol{t})\mathbb{P}(\boldsymbol{t}|s) \\
&= \exp(-E(s)) \sum_{\boldsymbol{t}:s\to s'} \min\left\{1, \frac{\exp(-E(s'))\mathbb{P}(\boldsymbol{t}^{-1}|s')}{\exp(-E(s))\mathbb{P}(\boldsymbol{t}|s)}\right\} \mathbb{P}(\boldsymbol{t}|s) \\
&= \sum_{\boldsymbol{t}:s\to s'} \min\left\{\exp(-E(s))\mathbb{P}(\boldsymbol{t}|s), \exp(-E(s'))\mathbb{P}(\boldsymbol{t}^{-1}|s')\right\} \\
&= \sum_{\boldsymbol{t}:s'\to s} \min\left\{\exp(-E(s))\mathbb{P}(\boldsymbol{t}^{-1}|s), \exp(-E(s'))\mathbb{P}(\boldsymbol{t}|s')\right\} \\
&= \exp(-E(s'))\mathbb{P}(s|s')
\end{aligned}
$$

For $\frac{\mathbb{P}(\boldsymbol{t}^{-1}|s_L)}{\mathbb{P}(\boldsymbol{t}|s_0)}$, sampling D contributes $\frac{\mathbb{P}^{\mathcal{D}}(D^{-1})}{\mathbb{P}^{\mathcal{D}}(D)}$. Leapfrog updates have no randomness and contribute nothing. For $D_j > 0$, define $p_j = \frac{\exp(-U(\tilde{q}_j^H, \tilde{q}_j^O))\mathbb{Q}_{D_j}(q_j^O|q_j^H, \tilde{q}_j^O)}{\exp(-U(q_j^H, q_j^O))\mathbb{Q}_{D_j}(\tilde{q}_j^O|q_j^H, q_j^O)}$. If $a_j = \texttt{True}$, the MH update contributes $\frac{\mathbb{Q}_{D_j}(q_j^O|q_j^H, \tilde{q}_j^O)\min\{1, 1/p_j\}}{\mathbb{Q}_{D_j}(\tilde{q}_j^O|q_j^H, q_j^O)\min\{1, p_j\}} = \frac{\exp(-U(q_j^H, q_j^O))}{\exp(-U(q_j^H, \tilde{q}_j^O))}$, which is captured in the ΔE updates in Algo. 3. If $a_j = \texttt{False}$, the MH update again contributes nothing (as in both $\boldsymbol{t}$ and $\boldsymbol{t}^{-1}$ we make the same proposal followed by a rejection). This proves Algo. 3 uses the correct MH acceptance probability, and establishes the desired detailed balance. ■

Table 1: Results on MDC (Sec. 4.1) and BLR (Sec. 4.2). We show ESS per sample per gradients evaluation of u for MDC, and of the potential energy of τ, β for BLR.

	MALAwG	HMCwG	MALA-PwG	MALA-PNwG	MAHMCwG
MDC	1.0×10^{-4}	4.62×10^{-3}	1.82×10^{-3}	7.38×10^{-3}	$\mathbf{1.78 \times 10^{-2}}$
BLR	1.73×10^{-3}	$7,94 \times 10^{-3}$	6.66×10^{-3}	8.86×10^{-3}	$\mathbf{9.02 \times 10^{-3}}$

4. Experiments

We use the 4 within-Gibbs samplers in Sec. 2.3 as baselines. Although MAHMC evolves q^H, q^O in tandem, to make the comparison informative, we consider a similarly structured MAHMC within Gibbs (MAHMCwG) sampler. Compared with HwG, MAHMCwG makes more frequent MH updates. Compared with MALA-based samplers, MAHMCwG uses longer HMC trajectories while maintaining frequent MH updates. We assume gradients evaluation dominates the computation (Neal, 2020), and evaluate performance using effective sample size (ESS) (calculated with Kumar et al. (2019)) per sample per gradients evaluation. We report the results in Tab. 1, and include detailed experiments setups and further analysis in Appendix A, as well as code reproducing the results in Appendix B.

4.1. A distribution with mixed discrete and continuous variables (MDC)

We consider the distribution in Sec. 5 of Neal (2020), where $q^H = (u, v), q^O = (w_1, \ldots, w_{20})$:

$$u \sim N(0, 1), v|u \sim N(u, 0.04^2), w_i|u \sim Bernoulli(1/(1 + e^u)), i = 1, \ldots, 20$$

As Tab. 1 shows, by combining longer HMC trajectories with more frequent Gibbs updates, MAHMCwG is **3.85x** more efficient than HwG, and **2.4x** more efficient than MALA-PNwG.

4.2. Bayesian Logistic Regression (BLR) with conjugate prior

We apply Bayesian Logistic Regression (BLR) to the breast cancer wisconsin dataset[1], consisting of 569 pairs of 30 dimensional features and targets $y_i \in \{0, 1\}$. We standarize the features and append 1 at the end to get $x_i \in \mathbb{R}^{31}$, and specify BLR with conjugate prior as

$$\tau \sim Gamma(1.0, scale = 100), \beta \sim N(0, \frac{1}{\tau} I_{31}), y_i \sim Bernoulli(sigmoid(x_i^T \beta)), i = 1, \ldots, 569$$

We consider sampling from the posterior $\mathbb{P}(\tau, \beta|X, y)$, where $q^H = \beta$ and $q^O = \tau$, and we make Gibbs updates for τ using $\mathbb{P}(\tau|\beta, X, y)$. MAHMCwG slightly outperforms MALA-PNwG, but is much easier to tune. See Appendix A for additional analysis.

1. Available on the UCI Machine Learning Repository. Accessed through sklearn.

References

Michael Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Yuansi Chen, Raaz Dwivedi, Martin J Wainwright, and Bin Yu. Fast mixing of Metropolized Hamiltonian Monte Carlo: Benefits of multi-step gradients. *J. Mach. Learn. Res.*, 21: 92–1, 2020.

Khue-Dung Dang, Matias Quiroz, Robert Kohn, Tran Minh-Ngoc, and Mattias Villani. Hamiltonian Monte Carlo with energy conserving subsampling. *Journal of machine learning research*, 20, 2019.

Raaz Dwivedi, Yuansi Chen, Martin J Wainwright, and Bin Yu. Log-concave sampling: Metropolis-Hastings algorithms are fast. *Journal of Machine Learning Research*, 20:1–42, 2019.

Alan M Horowitz. A generalized guided Monte Carlo algorithm. *Physics Letters B*, 268(2): 247–252, 1991.

Jacob Kelly, Richard Zemel, and Will Grathwohl. Directly training joint energy-based models for conditional synthesis and calibrated prediction of multi-attribute data. *arXiv preprint arXiv:2108.04227*, 2021.

Ravin Kumar, Carroll Colin, Ari Hartikainen, and Osvaldo A. Martin. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *The Journal of Open Source Software*, 2019. doi: 10.21105/joss.01143.

Benedict Leimkuhler and Sebastian Reich. *Simulating Hamiltonian dynamics*. Number 14. Cambridge university press, 2004.

Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012a.

Radford M Neal. MCMC using Hamiltonian dynamics. *arXiv preprint arXiv:1206.1901*, 2012b.

Radford M Neal. Non-reversibly updating a uniform [0, 1] value for Metropolis accept/reject decisions. *arXiv preprint arXiv:2001.11950*, 2020.

Guangyao Zhou. Mixed Hamiltonian Monte Marlo for Mixed Discrete and Continuous Variables. In *Advances in Neural Information Processing Systems*, volume 33, pages 17094–17104. Curran Associates, Inc., 2020.

Guangyao Zhou. Erratum to "Mixed Hamiltonian Monte Carlo for Mixed Discrete and Continuous Variables". https://stanniszhou.github.io/papers/mixed_hmc_erratum.pdf, 2021.

Appendix A. Details for the experiments

A.1. Experiments setups

We use the 4 within-Gibbs samplers in Sec. 2.3 as baselines, and follow the setups of Neal (2020) when possible. For MALA-based samplers, we alternate between N^L leapfrog updates for q^H and 1 Gibbs update for q^O. For HwG, we alternate between HMC iterations and Gibbs updates. Although MAHMC can evolve q^H, q^O in tandem, to make the comparison most informative, we consider an MAHMC within Gibbs (MAHMCwG) sampler, where we make $N^U - 1$ Gibbs updates and $N^U N^L$ leapfrog updates within each MAHMC iteration, followed by a Gibbs update. The MAHMC iteration schedules the Gibbs and leapfrog updates similarly to MALA-based samplers, alternating between N^L leapfrog updates and 1 Gibbs update. This leads to a deterministic $\mathbb{P}^{\mathcal{D}}$ that always proposes the same (symmetric) sequence of updates. We assume gradients evaluation dominates the computation (Neal, 2020), and evaluate performance using effective sample size (ESS) (calculated with Kumar et al. (2019)) per sample per gradients evaluation.

A.2. A distribution with mixed discrete and continuous variables (MDC)

Correctness For the distribution in Sec. 4.1, using the fact that the marginal distribution for u is $N(0, 1)$, we compare the histogram of u samples obtained with the 5 samplers, and empirically verify that they match the expected probability density function of $N(0, 1)$ and the samplers are sampling from the right distribution.

Efficiency For HwG and MALA-PNwG, we use the optimal hyperparameters reported in Sec. 5 of Neal (2020) ($L = 40$ steps and step size $\epsilon = 0.035$ for HwG, and $N^L = 10, \epsilon = 0.03, \alpha = 0.995, \delta = 0.01$ for MALA-PNwG). Our results roughly reproduce the results in Neal (2020) (MALA-PNwG is 1.6x more efficient than HwG, as opposed to 1.83x reporeted in Neal (2020)). The small discrepancy can be due to the use of a different metric (ESS of u instead of ESS of $\mathbb{I}(u \in (-0.5, 1.5))$ as in Neal (2020), where $\mathbb{I}$ represents the indicator function) and a different way to compute ESS (using Kumar et al. (2019)).

To make the comparison informative, we use $N^L = 10$ and $\epsilon = 0.03$ for MALAwG and MALA-PwG, and $\alpha = 0.995$ for MALA-PwG. We observe that partial momentum refreshment and non-reversible Metropolis accept/reject decisions are indeed beneficial in improving the performance of MALAwG/MALA-PwG. However, only MALA-PNwG outperforms HwG.

The optimal performance of MAHMCwG is achieved with $N^U = 10$ and $\epsilon = 0.04$, i.e. in each MAHMC we make $N^U N^L = 100$ leapfrog updates and $N^{U-1} = 9$ Gibbs updates (uniformly spreaded). This is far larger than the optimal number of steps $L = 40$ for HwG, and demonstrates MAHMC's ability to maintain high acceptance probabilities for long-range proposals that includes MH updates. We additionally test MAHMCwG with $N^U = 4$ and $\epsilon = 0.035$, i.e. making $N^U - 1 = 3$ additional Gibbs updates on top of HwG, and observe that the normalized ESS increases to 6.08×10^{-3}. This demonstrates the benefits of more frequent MH updates when we use the same number of leapfrog updates.

A.3. Bayesian Logistic Regression (BLR) with conjugate priors

Correctness To verify the correctness of the samplers, we first apply the samplers to only the prior distribution

$$\tau \sim Gamma(1.0, scale = 100), \beta \sim N(0, \frac{1}{\tau} I_{31})$$

and similarly verify the marginal distribution of τ is indeed $Gamma(1.0, scale = 100)$ by looking at the histogram of τ samples. We additionally verify that all samplers give rough the same posterior means for β, and we can classify the 569 training data points to an accuracy of 98.77% using posterior samples from the 5 different samplers.

Efficiency We fix $N^L = 5$, and conduct a grid search in

$$N^U \in \{2, 3, 4, 5\}, \epsilon \in \{0.07, 0.08, 0.09, 0.10, 0.11\}, \alpha \in \{0.9, 0.95, 0.98, 0.99\}$$

and $\delta \in \{0.005, 0.01, 0.015, 0.03\}$, and $L \in \{10, 15, 20, 25\}$ for HwG.

Optimal performance is achieved with $\epsilon = 0.11$ for MALAwG, $L = 10, \epsilon = 0.09$ for HwG, $\epsilon = 0.09, \alpha = 0.9$ for MALA-PwG, $\epsilon = 0.1, \alpha = 0.9, \delta = 0.015$ for MALA-PNwG, and $N^U = 2, \epsilon = 0.1$ for MAHMCwG.

For this example, MAHMCwG only slightly outperforms MALA-PNwG. However, we comment that MAHMCwG is easier to tune, as we essentially only need to tune the step size and number of steps, same as in HwG, and using the same setups from HwG usually already gives good performance. However, for MALA-PwG, we need to tune α and δ, which can have significant impacts on performance. For example, for the distribution in 4.1, grid search is done for $\alpha \in \{0.98, 0.99, 0.995, 0.9975, 0.9985, 0.999\}$ in Neal (2020). However, in our experiments we empirically observe that all these values give poor performance, and we have to reduce α to 0.9 to get performance comparable with MAHMCwG, suggesting the potential challenges in tuning α.

A.4. Additional verification of correctness

Since we only used Gibbs updates (which always accept) in our experiments, we additionally verify the correctness of MAHMC by modifying the script used in Zhou (2020, 2021) to make random walk MH updates within HMC for the 1D GMM example using MAHMC. See Appendix B for the updated script.

Appendix B. Code to reproduce the results

We implement all our samplers using *JAX* (Bradbury et al., 2018).

B.1. Code for samplers

```
import functools

import arviz
import jax
```

```
import jax.numpy as jnp
import joblib
import numpy as np
from tqdm import tqdm


def get_min_ess(samples, method='bulk'):
    """get_min_ess
    Parameters
    ----------
    samples : np.array
        (n_chains, n_samples, n_dim)
    Returns
    -------
    """
    if samples.ndim == 1:
        samples = samples[None, :, None]
    elif samples.ndim == 2:
        samples = samples[None, ...]

    n_chains, n_samples, n_dim = samples.shape
    ess_list = joblib.Parallel(n_jobs=joblib.cpu_count(), prefer='threads')(
        joblib.delayed(arviz.ess)(samples[..., ii], relative=True, method=method)
        for ii in tqdm(range(n_dim))
    )
    ess_list = np.array(ess_list)
    return np.min(ess_list)


def make_samplers(joint_energy, sample_q_other, get_step_size=None):
    if get_step_size is None:

        @jax.jit
        def get_step_size(q_other, epsilon):
            return epsilon

    @jax.jit
    def take_leapfrog_step(q_hmc, p, q_other, epsilon):
        step_size = get_step_size(q_other, epsilon)
        p = p - 0.5 * step_size * jax.grad(joint_energy, argnums=0)(q_hmc, q_other)
        q_hmc = q_hmc + step_size * p
        p = p - 0.5 * step_size * jax.grad(joint_energy, argnums=0)(q_hmc, q_other)
        return q_hmc, p

    @jax.jit
```

```
def mh_correction(q_hmc0, p0, q_other0, q_hmc, p, q_other, delta_U, v):
    U0 = joint_energy(q_hmc0, q_other0) + 0.5 * jnp.sum(p0 ** 2)
    U = joint_energy(q_hmc, q_other) + 0.5 * jnp.sum(p ** 2)
    accept = jnp.abs(v) <= jnp.exp(-(U - U0 - delta_U))
    q_hmc, p, q_other, v = jax.lax.cond(
        accept,
        lambda _: (q_hmc, p, q_other, v * jnp.exp(U - U0 - delta_U)),
        lambda _: (q_hmc0, p0, q_other0, v),
        None,
    )
    return q_hmc, p, q_other, v, accept

@functools.partial(jax.jit, static_argnames="L")
def take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, L):
    def scan_f(carry, ii):
        q_hmc, p = carry
        q_hmc, p = take_leapfrog_step(q_hmc, p, q_other, epsilon)
        return (q_hmc, p), None

    (q_hmc, p), _ = jax.lax.scan(scan_f, (q_hmc, p), jnp.arange(L))
    return q_hmc, p

# MALA within Gibbs
@functools.partial(jax.jit, static_argnames="L")
def take_multiple_mala_steps(q_hmc, q_other, key, epsilon, L):
    def scan_f(carry, ii):
        q_hmc, q_other, key = carry
        key, subkey = jax.random.split(key)
        p = jax.random.normal(subkey, shape=q_hmc.shape)
        q_hmc0, p0 = q_hmc, p
        q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, 1)
        key, subkey = jax.random.split(key)
        q_hmc, p, q_other, _, accept = mh_correction(
            q_hmc0, p0, q_other, q_hmc, p, q_other, 0.0, jax.random.uniform(subkey)
        )
        return (q_hmc, q_other, key), accept

    (q_hmc, q_other, key), accept_list = jax.lax.scan(
        scan_f, (q_hmc, q_other, key), jnp.arange(L)
    )
    return q_hmc, key, accept_list

@functools.partial(jax.jit, static_argnames="L")
def mala_within_gibbs(q_hmc, q_other, key, epsilon, L):
    q_hmc, key, accept_list = take_multiple_mala_steps(
```

```
        q_hmc, q_other, key, epsilon, L
    )
    q_other, key = sample_q_other(q_hmc, key)
    return q_hmc, q_other, key, accept_list

# HMC-within-Gibbs
@functools.partial(jax.jit, static_argnames="L")
def take_hmc_step(q_hmc, q_other, key, epsilon, L):
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=q_hmc.shape)
    q_hmc0, p0 = q_hmc, p
    q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, L)
    key, subkey = jax.random.split(key)
    q_hmc, p, q_other, _, accept = mh_correction(
        q_hmc0, p0, q_other, q_hmc, p, q_other, 0.0, jax.random.uniform(subkey)
    )
    return q_hmc, key, accept

@functools.partial(jax.jit, static_argnames="L")
def hmc_within_gibbs(q_hmc, q_other, key, epsilon, L):
    q_hmc, key, accept = take_hmc_step(q_hmc, q_other, key, epsilon, L)
    q_other, key = sample_q_other(q_hmc, key)
    return q_hmc, q_other, key, accept

# MAHMC
@functools.partial(jax.jit, static_argnames=("L", "N"))
def take_mahmc_step(q_hmc, q_other, key, epsilon, L, N):
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=q_hmc.shape)
    q_hmc0, q_other0, p0 = q_hmc, q_other, p

    def scan_f(carry, ii):
        q_hmc, p, q_other, delta_U, key = carry
        q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, L)
        q_other0 = q_other
        q_other, key = sample_q_other(q_hmc, key)
        delta_U = (
            delta_U + joint_energy(q_hmc, q_other) - joint_energy(q_hmc, q_other0)
        )
        return (q_hmc, p, q_other, delta_U, key), None

    (q_hmc, p, q_other, delta_U, key), _ = jax.lax.scan(
        scan_f, (q_hmc, p, q_other, 0.0, key), jnp.arange(N - 1)
    )
    q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, L)
```

```
    key, subkey = jax.random.split(key)
    q_hmc, p, q_other, _, accept = mh_correction(
        q_hmc0, p0, q_other0, q_hmc, p, q_other, delta_U, jax.random.uniform(subkey)
    )
    return q_hmc, q_other, key, accept

@functools.partial(jax.jit, static_argnames=("L", "N"))
def mahmc_within_gibbs(q_hmc, q_other, key, epsilon, L, N):
    q_hmc, q_other, key, accept = take_mahmc_step(
        q_hmc, q_other, key, epsilon, L, N
    )
    q_other, key = sample_q_other(q_hmc, key)
    return q_hmc, q_other, key, accept

# MALA with persistent momentum
@functools.partial(jax.jit, static_argnames="L")
def take_multiple_mala_persistent_steps(q_hmc, p, q_other, key, epsilon, L, alpha):
    def scan_f(carry, ii):
        q_hmc, p, q_other, key = carry
        key, subkey = jax.random.split(key)
        n = jax.random.normal(subkey, shape=q_hmc.shape)
        p = alpha * p + jnp.sqrt(1 - alpha ** 2) * n
        q_hmc0, p0 = q_hmc, p
        q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, 1)
        key, subkey = jax.random.split(key)
        q_hmc, p, q_other, _, accept = mh_correction(
            q_hmc0, -p0, q_other, q_hmc, p, q_other, 0.0, jax.random.uniform(subkey)
        )
        return (q_hmc, p, q_other, key), accept

    (q_hmc, p, q_other, key), accept_list = jax.lax.scan(
        scan_f, (q_hmc, p, q_other, key), jnp.arange(L)
    )
    return q_hmc, p, key, accept_list

@functools.partial(jax.jit, static_argnames="L")
def mala_persistent_within_gibbs(q_hmc, p, q_other, key, epsilon, L, alpha):
    q_hmc, p, key, accept_list = take_multiple_mala_persistent_steps(
        q_hmc, p, q_other, key, epsilon, L, alpha
    )
    q_other, key = sample_q_other(q_hmc, key)
    return q_hmc, p, q_other, key, accept_list

# MALA with persistent momentum and non-reversible Metropolis accept/reject
@functools.partial(jax.jit, static_argnames="L")
```

```
def take_multiple_mala_persistent_nonreversible_steps(
    q_hmc, p, q_other, v, key, epsilon, L, alpha, delta
):
    def scan_f(carry, ii):
        q_hmc, p, q_other, v, key = carry
        key, subkey = jax.random.split(key)
        n = jax.random.normal(subkey, shape=q_hmc.shape)
        p = alpha * p + jnp.sqrt(1 - alpha ** 2) * n
        q_hmc0, p0 = q_hmc, p
        q_hmc, p = take_multiple_leapfrog_steps(q_hmc, p, q_other, epsilon, 1)
        q_hmc, p, q_other, v, accept = mh_correction(
            q_hmc0, -p0, q_other, q_hmc, p, q_other, 0.0, v
        )
        v = (v + 1 + delta) % 2 - 1
        return (q_hmc, p, q_other, v, key), accept

    (q_hmc, p, q_other, v, key), accept_list = jax.lax.scan(
        scan_f, (q_hmc, p, q_other, v, key), jnp.arange(L)
    )
    return q_hmc, p, v, key, accept_list

@functools.partial(jax.jit, static_argnames="L")
def mala_persistent_nonreversible_within_gibbs(
    q_hmc, p, q_other, v, key, epsilon, L, alpha, delta
):
    (
        q_hmc,
        p,
        v,
        key,
        accept_list,
    ) = take_multiple_mala_persistent_nonreversible_steps(
        q_hmc, p, q_other, v, key, epsilon, L, alpha, delta
    )
    q_other, key = sample_q_other(q_hmc, key)
    return q_hmc, p, q_other, v, key, accept_list

return (
    mala_within_gibbs,
    hmc_within_gibbs,
    mahmc_within_gibbs,
    mala_persistent_within_gibbs,
    mala_persistent_nonreversible_within_gibbs,
)
```

B.2. Code for MDC

```
import functools

import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
from jax.scipy.special import expit
from jax.scipy.stats import bernoulli, norm

from samplers import make_samplers, get_min_ess


# %%
@jax.jit
def joint_energy(q_hmc, q_other):
    return (
        -norm.logpdf(q_hmc[0])
        - norm.logpdf(q_hmc[1], loc=q_hmc[0], scale=0.04)
        - jnp.sum(bernoulli.logpmf(q_other, p=expit(-q_hmc[0])))
    )


@jax.jit
def sample_q_other(q_hmc, key):
    key, subkey = jax.random.split(key)
    q_other = jax.random.bernoulli(subkey, p=expit(-q_hmc[0]), shape=(20,))
    return q_other, key


(
    mala_within_gibbs,
    hmc_within_gibbs,
    mahmc_within_gibbs,
    mala_persistent_within_gibbs,
    mala_persistent_nonreversible_within_gibbs,
) = make_samplers(joint_energy, sample_q_other)


# %%
n_samples, n_warm_up_samples = int(1e6), int(1e5)
L = 10
n_chains = 16
```

```
# %% [markdown]
# # MALA within Gibbs

# %%
epsilon = 0.03


@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None), out_axes=0)
def get_mala_within_gibbs_samples(key, epsilon):
    def scan_f(carry, ii):
        uv, w, key = carry
        uv, w, key, accept = mala_within_gibbs(uv, w, key, epsilon, L)
        return (uv, w, key), (uv, w, accept)

    key, subkey = jax.random.split(key)
    uv = jax.random.normal(subkey, shape=(2,))
    uv = uv.at[1].set((uv[1] + uv[0]) * 0.04)
    key, subkey = jax.random.split(key)
    w = jax.random.bernoulli(subkey, shape=(20,))
    _, samples = jax.lax.scan(scan_f, (uv, w, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_within_gibbs_samples(keys, epsilon)
print(
    f"""
### MALA-within-Gibbs, epsilon: {epsilon}

u_ess: {get_min_ess(samples[0].copy()[..., [0]])}
v_ess: {get_min_ess(samples[0].copy()[..., [1]])}
acceptance: {np.mean(samples[2])}

"""
)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.hist(samples[0][0, ..., 0].copy(), bins=200, density=True)
x = np.linspace(-5, 5, int(1e5))
ax.plot(x, norm.pdf(x))
```

```
# %% [markdown]
# # HMC-within-Gibbs

# %%
N = 4
epsilon = 0.035


@functools.partial(jax.jit, static_argnames="N")
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_hmc_within_gibbs_samples(key, N, epsilon):
    def scan_f(carry, ii):
        uv, w, key = carry
        uv, w, key, accept = hmc_within_gibbs(uv, w, key, epsilon, L * N)
        return (uv, w, key), (uv, w, accept)

    key, subkey = jax.random.split(key)
    uv = jax.random.normal(subkey, shape=(2,))
    uv = uv.at[1].set((uv[1] + uv[0]) * 0.04)
    key, subkey = jax.random.split(key)
    w = jax.random.bernoulli(subkey, shape=(20,))
    _, samples = jax.lax.scan(scan_f, (uv, w, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_hmc_within_gibbs_samples(keys, N, epsilon)
print(
    f"""
### HMC-within-Gibbs, N: {N}, epsilon: {epsilon}

u_ess: {get_min_ess(samples[0].copy()[..., [0]]) / N}
v_ess: {get_min_ess(samples[0].copy()[..., [1]]) / N}
acceptance: {np.mean(samples[2])}

"""
)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.hist(samples[0][0, ..., 0].copy(), bins=200, density=True)
x = np.linspace(-5, 5, int(1e5))
ax.plot(x, norm.pdf(x))
```

```
# %% [markdown]
# # MAHMC

# %%
N = 10
epsilon = 0.04


@functools.partial(jax.jit, static_argnames="N")
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_mahmc_within_gibbs_samples(key, N, epsilon):
    def scan_f(carry, ii):
        uv, w, key = carry
        uv, w, key, accept = mahmc_within_gibbs(uv, w, key, epsilon, L, N)
        return (uv, w, key), (uv, w, accept)

    key, subkey = jax.random.split(key)
    uv = jax.random.normal(subkey, shape=(2,))
    uv = uv.at[1].set((uv[1] + uv[0]) * 0.04)
    key, subkey = jax.random.split(key)
    w = jax.random.bernoulli(subkey, shape=(20,))
    _, samples = jax.lax.scan(scan_f, (uv, w, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mahmc_within_gibbs_samples(keys, N, epsilon)
print(
    f"""
### MAHMC-within-Gibbs, N: {N}, epsilon: {epsilon}

u_ess: {get_min_ess(samples[0].copy()[..., [0]]) / N}
v_ess: {get_min_ess(samples[0].copy()[..., [1]]) / N}
acceptance: {np.mean(samples[2])}

"""
)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.hist(samples[0][0, ..., 0].copy(), bins=200, density=True)
x = np.linspace(-5, 5, int(1e5))
ax.plot(x, norm.pdf(x))
```

```
# %% [markdown]
# # MALA with persistent momentum


# %%
epsilon = 0.03
alpha = 0.995


@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_mala_persistent_within_gibbs_samples(key, epsilon, alpha):
    def scan_f(carry, ii):
        uv, p, w, key = carry
        uv, p, w, key, accept = mala_persistent_within_gibbs(
            uv, p, w, key, epsilon, L, alpha
        )
        return (uv, p, w, key), (uv, w, accept)

    key, subkey = jax.random.split(key)
    uv = jax.random.normal(subkey, shape=(2,))
    uv = uv.at[1].set((uv[1] + uv[0]) * 0.04)
    key, subkey = jax.random.split(key)
    w = jax.random.bernoulli(subkey, shape=(20,))
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=uv.shape)
    _, samples = jax.lax.scan(scan_f, (uv, p, w, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_persistent_within_gibbs_samples(keys, epsilon, alpha)
print(
    f"""
### MALA-P-within-Gibbs, epsilon: {epsilon}, alpha: {alpha}

u_ess: {get_min_ess(samples[0].copy()[..., [0]])}
v_ess: {get_min_ess(samples[0].copy()[..., [1]])}
acceptance: {np.mean(samples[2])}

"""
)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
```

```
ax.hist(samples[0][0, ..., 0].copy(), bins=200, density=True)
x = np.linspace(-5, 5, int(1e5))
ax.plot(x, norm.pdf(x))


# %% [markdown]
# # MALA with persistent momentum and non-reversible Metropolis accept/reject


# %%
epsilon = 0.03
alpha = 0.995
delta = 0.01


@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None, None, None), out_axes=0)
def get_mala_persistent_nonreversible_within_gibbs_samples(key, epsilon, alpha, delta):
    def scan_f(carry, ii):
        uv, p, w, v, key = carry
        uv, p, w, v, key, accept = mala_persistent_nonreversible_within_gibbs(
            uv, p, w, v, key, epsilon, L, alpha, delta
        )
        return (uv, p, w, v, key), (uv, w, accept)

    key, subkey = jax.random.split(key)
    uv = jax.random.normal(subkey, shape=(2,))
    uv = uv.at[1].set((uv[1] + uv[0]) * 0.04)
    key, subkey = jax.random.split(key)
    w = jax.random.bernoulli(subkey, shape=(20,))
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=uv.shape)
    key, subkey = jax.random.split(key)
    v = jax.random.uniform(subkey) * 2 - 1
    _, samples = jax.lax.scan(scan_f, (uv, p, w, v, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_persistent_nonreversible_within_gibbs_samples(
    keys, epsilon, alpha, delta
)
print(
```

```
    f"""
### MALA-P-N-within-Gibbs, epsilon: {epsilon}, alpha: {alpha}, delta: {delta}

u_ess: {get_min_ess(samples[0].copy()[..., [0]])}
v_ess: {get_min_ess(samples[0].copy()[..., [1]])}
acceptance: {np.mean(samples[2])}

"""
)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.hist(samples[0][0, ..., 0].copy(), bins=200, density=True)
x = np.linspace(-5, 5, int(1e5))
```

B.3. Code for BLR

```
import functools

import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
from jax.scipy.special import expit
from jax.scipy.stats import bernoulli, gamma, norm
from sklearn.datasets import load_breast_cancer

from samplers import make_samplers, get_min_ess

# %% [markdown]
# # Basic setup

# %%
data = load_breast_cancer()
X = (data.data - np.mean(data.data, axis=0, keepdims=True)) / np.std(
    data.data, axis=0, keepdims=True
)
X = np.concatenate([X, np.ones((X.shape[0], 1))], axis=1)
y = data.target

# %%
a = 1.0
beta = 0.01
ndim = X.shape[1]
x = np.linspace(0.01, 1000, int(1e5))
plt.plot(x, gamma.pdf(x, a, scale=1 / beta))
```

```
@jax.jit
def joint_energy(q_hmc, q_other):
    return (
        -gamma.logpdf(q_other, a, scale=1 / beta)
        - jnp.sum(norm.logpdf(q_hmc, scale=1 / jnp.sqrt(q_other)))
        - jnp.sum(bernoulli.logpmf(y, p=expit(jnp.dot(X, q_hmc))))
    )


@jax.jit
def sample_q_other(q_hmc, key):
    key, subkey = jax.random.split(key)
    q_other = jax.random.gamma(subkey, a + 0.5 * q_hmc.shape[0]) / (
        beta + 0.5 * jnp.sum(q_hmc ** 2)
    )
    return q_other, key


@jax.jit
def get_step_size(q_other, epsilon):
    return epsilon / jnp.sqrt(q_other)


# %%
(
    mala_within_gibbs,
    hmc_within_gibbs,
    mahmc_within_gibbs,
    mala_persistent_within_gibbs,
    mala_persistent_nonreversible_within_gibbs,
) = make_samplers(joint_energy, sample_q_other, get_step_size)


# %%
n_samples, n_warm_up_samples = int(1e5), int(1e4)
L = 5
n_chains = 1

# %% [markdown]
# # MALA within Gibbs

# %%
epsilon = 0.11
```

```
@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None), out_axes=0)
def get_mala_within_gibbs_samples(key, epsilon):
    def scan_f(carry, ii):
        beta, tau, key = carry
        beta, tau, key, accept = mala_within_gibbs(beta, tau, key, epsilon, L)
        energy = joint_energy(beta, tau)
        return (beta, tau, key), (beta, tau, energy, accept)

    tau = 150.0
    key, subkey = jax.random.split(key)
    beta = jax.random.normal(subkey, shape=(ndim,)) / jnp.sqrt(tau)
    _, samples = jax.lax.scan(scan_f, (beta, tau, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_within_gibbs_samples(keys, epsilon)
print(
    f"""
### MALA-within-Gibbs, epsilon: {epsilon}

beta_min_ess: {get_min_ess(samples[0].copy())}
tau_ess: {get_min_ess(samples[1][..., None].copy())}
energies_ess: {get_min_ess(samples[2][..., None].copy())}
acceptance: {np.mean(samples[3])}
accuracy: {
    np.sum(
        y == (
            np.mean(
                expit(jnp.dot(samples[0][0], X.T)) >= 0.5,
                axis=0
            ) >= 0.5
        ).astype(int)
    ) / y.shape[0]
}

"""
)

# %% [markdown]
```

```
# # HMC-within-Gibbs

# %%
N = 2
epsilon = 0.09


@functools.partial(jax.jit, static_argnames="N")
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_hmc_within_gibbs_samples(key, N, epsilon):
    def scan_f(carry, ii):
        beta, tau, key = carry
        beta, tau, key, accept = hmc_within_gibbs(beta, tau, key, epsilon, L * N)
        energy = joint_energy(beta, tau)
        return (beta, tau, key), (beta, tau, energy, accept)

    tau = 150.0
    key, subkey = jax.random.split(key)
    beta = jax.random.normal(subkey, shape=(ndim,)) / jnp.sqrt(tau)
    _, samples = jax.lax.scan(scan_f, (beta, tau, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_hmc_within_gibbs_samples(keys, N, epsilon)
print(
    f"""
### HMC-within-Gibbs, N: {N}, epsilon: {epsilon}

beta_min_ess: {get_min_ess(samples[0].copy()) / N}
tau_ess: {get_min_ess(samples[1][..., None].copy()) / N}
energies_ess: {get_min_ess(samples[2][..., None].copy()) / N}
acceptance: {np.mean(samples[3])}
accuracy: {
    np.sum(
        y == (
            np.mean(
                expit(jnp.dot(samples[0][0], X.T)) >= 0.5,
                axis=0
            ) >= 0.5
        ).astype(int)
    ) / y.shape[0]
}
```

```
"""
)


# %% [markdown]
# # MAHMC

# %%
N = 2
epsilon = 0.10


@functools.partial(jax.jit, static_argnames="N")
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_mahmc_within_gibbs_samples(key, N, epsilon):
    def scan_f(carry, ii):
        beta, tau, key = carry
        beta, tau, key, accept = mahmc_within_gibbs(beta, tau, key, epsilon, L, N)
        energy = joint_energy(beta, tau)
        return (beta, tau, key), (beta, tau, energy, accept)

    tau = 150.0
    key, subkey = jax.random.split(key)
    beta = jax.random.normal(subkey, shape=(ndim,)) / jnp.sqrt(tau)
    _, samples = jax.lax.scan(scan_f, (beta, tau, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mahmc_within_gibbs_samples(keys, N, epsilon)
accuracy = (
    np.sum(
        y
        == (np.mean(expit(jnp.dot(samples[0][0], X.T)) >= 0.5, axis=0) >= 0.5).astype(
            int
        )
    )
    / y.shape[0]
)
print(
    f"""
### MAHMC-within-Gibbs, N: {N}, epsilon: {epsilon}
```

```
beta_min_ess: {get_min_ess(samples[0].copy()) / N}
tau_ess: {get_min_ess(samples[1][..., None].copy()) / N}
energies_ess: {get_min_ess(samples[2][..., None].copy()) / N}
acceptance: {np.mean(samples[3])}
accuracy: {
    np.sum(
        y == (
            np.mean(
                expit(jnp.dot(samples[0][0], X.T)) >= 0.5,
                axis=0
            ) >= 0.5
        ).astype(int)
    ) / y.shape[0]
}

"""
)


# %% [markdown]
# # MALA with persistent momentum


# %%
epsilon = 0.09
alpha = 0.9


@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None, None), out_axes=0)
def get_mala_persistent_within_gibbs_samples(key, epsilon, alpha):
    def scan_f(carry, ii):
        beta, p, tau, key = carry
        beta, p, tau, key, accept = mala_persistent_within_gibbs(
            beta, p, tau, key, epsilon, L, alpha
        )
        energy = joint_energy(beta, tau)
        return (beta, p, tau, key), (beta, tau, energy, accept)

    tau = 150.0
    key, subkey = jax.random.split(key)
    beta = jax.random.normal(subkey, shape=(ndim,)) / jnp.sqrt(tau)
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=beta.shape)
```

```
    _, samples = jax.lax.scan(scan_f, (beta, p, tau, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_persistent_within_gibbs_samples(keys, epsilon, alpha)
print(
    f"""
### MALA-P-within-Gibbs, epsilon: {epsilon}, alpha: {alpha}

beta_min_ess: {get_min_ess(samples[0].copy())}
tau_ess: {get_min_ess(samples[1][..., None].copy())}
energies_ess: {get_min_ess(samples[2][..., None].copy())}
acceptance: {np.mean(samples[3])}
accuracy: {
    np.sum(
        y == (
            np.mean(
                expit(jnp.dot(samples[0][0], X.T)) >= 0.5,
                axis=0
            ) >= 0.5
        ).astype(int)
    ) / y.shape[0]
}

"""
)


# %% [markdown]
# # MALA with persistent momentum and non-reversible Metropolis accept/reject


# %%
epsilon = 0.1
alpha = 0.9
delta = 0.015


@jax.jit
@functools.partial(jax.vmap, in_axes=(0, None, None, None), out_axes=0)
def get_mala_persistent_nonreversible_within_gibbs_samples(key, epsilon, alpha, delta):
    def scan_f(carry, ii):
```

```
        beta, p, tau, v, key = carry
        beta, p, tau, v, key, accept = mala_persistent_nonreversible_within_gibbs(
            beta, p, tau, v, key, epsilon, L, alpha, delta
        )
        energy = joint_energy(beta, tau)
        return (beta, p, tau, v, key), (beta, tau, energy, accept)

    tau = 150.0
    key, subkey = jax.random.split(key)
    beta = jax.random.normal(subkey, shape=(ndim,)) / jnp.sqrt(tau)
    key, subkey = jax.random.split(key)
    p = jax.random.normal(subkey, shape=beta.shape)
    key, subkey = jax.random.split(key)
    v = jax.random.uniform(subkey) * 2 - 1
    _, samples = jax.lax.scan(scan_f, (beta, p, tau, v, key), jnp.arange(n_samples))
    samples = jax.tree_util.tree_map(lambda x: x[n_warm_up_samples:], samples)
    return samples


key = jax.random.PRNGKey(np.random.randint(int(1e5)))
keys = jax.random.split(key, n_chains)
samples = get_mala_persistent_nonreversible_within_gibbs_samples(
    keys, epsilon, alpha, delta
)
print(
    f"""
### MALA-P-N-within-Gibbs, epsilon: {epsilon}, alpha: {alpha}, delta: {delta}

beta_min_ess: {get_min_ess(samples[0].copy())}
tau_ess: {get_min_ess(samples[1][..., None].copy())}
energies_ess: {get_min_ess(samples[2][..., None].copy())}
acceptance: {np.mean(samples[3])}
accuracy: {
    np.sum(
        y == (
            np.mean(
                expit(jnp.dot(samples[0][0], X.T)) >= 0.5,
                axis=0
            ) >= 0.5
        ).astype(int)
    ) / y.shape[0]
}

"""
)
```

B.4. Updated script to verify correctness

Script is adapted from https://github.com/StannisZhou/mixed_hmc/blob/master/scripts/simple_gmm/test_naive_mixed_hmc.py.

```
import matplotlib.pyplot as plt
import numba
import numpy as np
from momentum.potential.simple_gmm import get_mixture_density
from tqdm import tqdm


def mahmc(x0, q0, n_samples, epsilon, L, pi, mu_list, sigma_list):
    @numba.jit(nopython=True)
    def potential(x, q):
        potential = (
            -np.log(pi[x])
            + 0.5 * np.log(2 * np.pi * sigma_list[x] ** 2)
            + 0.5 * (q - mu_list[x]) ** 2 / sigma_list[x] ** 2
        )
        return potential

    @numba.jit(nopython=True)
    def grad_potential(x, q):
        grad_potential = (q - mu_list[x]) / sigma_list[x] ** 2
        return grad_potential

    @numba.jit(nopython=True)
    def take_mahmc_step(x0, q0, epsilon, L, n_components):
        # Resample momentum
        p0 = np.random.randn()
        # Initialize q, delta_U
        x = x0
        q = q0
        p = p0
        delta_U = 0.0
        # Take L steps
        for ii in range(L):
            q, p = leapfrog_step(x=x, q=q, p=p, epsilon=epsilon)
            x, delta_U = update_discrete(
                x0=x,
                q=q,
                delta_U=delta_U,
                n_components=n_components,
```

```
        )

    # Accept or reject
    current_E = potential(x0, q0) + 0.5 * p0 ** 2
    proposed_E = potential(x, q) + 0.5 * p ** 2
    accept = np.random.rand() < np.exp(current_E + delta_U - proposed_E)
    if not accept:
        x, q = x0, q0

    return x, q, accept

@numba.jit(nopython=True)
def leapfrog_step(x, q, p, epsilon):
    p -= 0.5 * epsilon * grad_potential(x, q)
    q += epsilon * p
    p -= 0.5 * epsilon * grad_potential(x, q)
    return q, p

@numba.jit(nopython=True)
def update_discrete(x0, q, delta_U, n_components):
    x = x0
    distribution = np.ones(n_components)
    distribution[x] = 0
    distribution /= np.sum(distribution)
    proposal_for_ind = np.argmax(np.random.multinomial(1, distribution))
    x = proposal_for_ind
    delta_E = potential(x, q) - potential(x0, q)
    # Decide whether to accept or reject
    accept = np.random.exponential() > delta_E
    if accept:
        delta_U += potential(x, q) - potential(x0, q)
    else:
        x = x0

    return x, delta_U

x, q = x0, q0
x_samples, q_samples, accept_list = [], [], []
for _ in tqdm(range(n_samples)):
    x, q, accept = take_mahmc_step(
        x0=x, q0=q, epsilon=epsilon, L=L, n_components=pi.shape[0]
    )
    x_samples.append(x)
    q_samples.append(q)
    accept_list.append(accept)
```

```
    x_samples = np.array(x_samples)
    q_samples = np.array(q_samples)
    accept_list = np.array(accept_list)
    return x_samples, q_samples, accept_list


pi = np.array([0.15, 0.3, 0.3, 0.25])
mu_list = np.array([-2, 0, 2, 4])
sigma_list = np.sqrt(0.1) * np.ones(pi.shape[0])


x0 = np.random.randint(4)
q0 = np.random.randn()
n_warm_up_samples = int(1e6)
n_samples = int(4e6)
epsilon = 0.3
L = 15


x_samples, q_samples, accept_list = mahmc(
    x0,
    q0,
    n_warm_up_samples + n_samples,
    epsilon,
    L,
    pi,
    mu_list,
    sigma_list,
)

print(np.mean(accept_list))

x = np.linspace(-10, 10, int(1e4))
mixture_density = get_mixture_density(x, pi, mu_list, sigma_list)
fig, ax = plt.subplots(1, 1)
ax.hist(q_samples[n_warm_up_samples:], density=True, bins=500)
ax.plot(x, mixture_density)
plt.show()
```