HAMILTONIAN NEURAL NETWORK AND HAMILTONIAN MONTE CARLO: PART ONE AND PART TWO REPORT

SUBMITTED VERSION OF BOTH PART 1 AND PART 2 FOR JP.MORGAN

Chuanmiao Yan
Academy of Mathematics and Systems Science
Chinese Academy of Sciences
Beijing, 100190
yanchuanmiao22@mails.ucas.ac.cn
My LinkedIn Profile
My GitHub

April 5, 2025

Preface

As a PhD candidate at the Academy of Mathematics and Systems Science, Chinese Academy of Sciences (CAS), my academic journey began with the purity of mathematical proofs but collided with the awe-inspiring complexity of Bayesian computation. Equipped with undergraduate training in mathematics, I approached Hamiltonian Monte Carlo (HMC) and Pseudo-Marginal HMC (PM-HMC) as an outsider—struggling to reconcile physical metaphors like "momentum variables" with measure-theoretic formalism. This report embodies a mathematician's earnest, if clumsy, tribute to the art of high-dimensional sampling.

My purpose is not to replicate theoretical completeness but to answer two pragmatic questions:

1. How can non-physics-trained researchers grasp the essence of HMC and No-U-Turn Sampling (NUTS)?
2. What computational DNA do Hamiltonian Neural Networks (HNNs) and PM-HMC inject into classical Monte Carlo methods?

Constrained by local computing resources (a consumer-grade GPU), experiments focused on validating core algorithmic components: implementing leapfrog integration steps and testing the recursion depth control of the *BuildTree* function in a relative low-dimensional latent spaces via TensorFlow. These "micro-experiments," while insufficient for broad performance claims, offer a granular lens to examine gradient noise-mixing efficiency tradeoffs.

I must acknowledge two limitations:

1. To enhance accessibility for non-specialists, mathematical constructions or illustrations of L-HNNs and PM-HMC were simplified, potentially sacrificing theoretical nuance;
2. As a novice in Bayesian inference, my interpretations of certain terminologies may deviate from statistical conventions—such deviations reflect nothing but my own scholarly gaps.

If this imperfect report emboldens interdisciplinary researchers to cross methodological boundaries, it will have achieved its highest purpose. I warmly welcome corrections to its inevitable inaccuracies.

Chuanmiao Yan
Academy of Mathematics and Systems Science
Chinese Academy of Sciences
February 2025

Contents

Preface **2**

1 Part One **6**

1.1 Introduction . . . 6
1.2 Literature Review . . . 7
1.2.1 Hamiltonian Monte Carlo and No-U-Turn Sampling . . . 7
1.2.2 Extensions and Applications of Monte Carlo Methods . . . 7
1.2.3 Impact of HMC on Modern Bayesian Inference . . . 8
1.3 Motivation: Why Efficient Exploration Is Challenging . . . 8
1.3.1 Challenges of High-Dimensional Probability Distributions . . . 8
1.3.2 Limitations of Classical Monte Carlo Methods . . . 9
1.3.3 Why Hamiltonian Monte Carlo (HMC)? . . . 9
1.4 Latent Hamiltonian Neural Networks (L-HNN) . . . 11
1.4.1 Shared Loss Function . . . 11
1.4.2 Direct HNN . . . 11
1.4.3 Latent HNN . . . 11
1.5 What Is NUTS and Why Use It? . . . 12
1.5.1 Intuition Behind the U-Turn Condition . . . 12
1.5.2 Why Bidirectional Evolution . . . 13
1.6 Additional Implementation Details . . . 13
1.6.1 Leapfrog Integration . . . 13
1.6.2 Metropolis Acceptance Criterion . . . 14
1.6.3 Data Generation . . . 15
1.6.4 Online Monitoring . . . 15
1.6.5 *BuildTree* Function . . . 17
1.6.6 Workflow . . . 18
1.7 Unit test and Integration Test Implementation . . . 19
1.7.1 Manual Test . . . 19
1.7.2 Integration Test . . . 19
1.7.3 Unit Test . . . 19
1.7.4 Updating Configuration Files . . . 20

1.8 Experiment Results for Part 1 . . . 20
1.8.1 Comparison Between NUTS and Non-NUTS Methods . . . 20
1.8.2 Effectiveness of L-HNN Across Different Distributions . . . 20
1.8.3 Summary of Findings . . . 21

2 Part Two **22**
2.1 Introduction . . . 22
2.2 Literature Review . . . 23
2.2.1 Pseudo-Marginal Methods . . . 23
2.2.2 Pseudo-Marginal Hamiltonian Monte Carlo . . . 23
2.2.3 Importance Sampling . . . 23
2.2.4 Applications and Extensions . . . 24
2.3 Motivation: Why Pseudo Marginal Hamiltonian Monte Carlo Sampling? . . . 24
2.3.1 Limitations of Pure HMC Compared to Pseudo-Marginal and Marginal Variants . . . 24
2.3.2 Foundations of Pseudo-Marginal Hamiltonian Monte Carlo . . . 25
2.3.3 Estimating the Gradients . . . 27
2.3.4 Splitting Hamiltonian Components . . . 28
2.4 Perform the Pseudo-marginal Hamiltonian Neural Network . . . 29
2.4.1 Why use the Pseudo-marginal Hamiltonian Neural Network? . . . 29
2.4.2 Workflow of Using PM-HNN in PM-HMC Sampling . . . 30
2.5 Unit test and Integration Test Implementation . . . 30
2.5.1 Manual Test . . . 30
2.5.2 Integration Test . . . 33
2.5.3 Unit Test . . . 33
2.5.4 Updating Configuration Files . . . 33
2.6 Experiment Results for Part 2.e . . . 34

3 Empirical Study: Bayesian Inference for VaR and CVaR Using HMC **37**
3.1 Introduction . . . 37
3.2 Bayesian Models for VaR and CVaR Estimation . . . 38
3.2.1 Normal Distribution Model . . . 38
3.2.2 Student-*t* Distribution Model . . . 38
3.2.3 Generalized Extreme Value (GEV) Distribution Model . . . 39
3.2.4 Conclusion . . . 39
3.3 Empirical Analysis . . . 39
3.4 Performance Across Different Asset Classes and Horizons . . . 39
3.4.1 Bonds . . . 39
3.4.2 Commodities . . . 40
3.4.3 Equities . . . 40
3.4.4 REITs (Real Estate Investment Trusts) . . . 40

3.5 Final Recommendations . . . 40
3.6 Discussion: Comparison of Short-Term and Long-Term VaR Estimation . . . 41

4 Self-Assessment and Future Plans **42**
4.1 Extent of Code Conversion to TensorFlow Probability . . . 42
4.2 Challenges Encountered During the Conversion . . . 42
4.3 Short-Term Improvement Plan . . . 42
4.4 Long-Term Plan for Improving Coding Skills . . . 43
4.5 Integration of Research and Technical Development . . . 43

5 Response to the Questions of Interviews in TFP Part **44**
5.1 What are the built-in Python data structures? Could you explain how different data structures are used in the *tensorflow_probability.mcmc* package, and why these design choices are made? . . . 44
5.2 How does TensorFlow Probability (TFP) handle distribution shapes? What changes could you make to your library to align with TFP's design? . . . 46
5.3 How does TFP manage randomness? . . . 46
5.4 What are the different types of tests in Python? What are the criteria for effective tests in each category? How could you rewrite your tests to make your program more robust? . . . 47
5.5 Summary . . . 49

A Some Tables and Figures **51**
A.1 Figures . . . 51
A.2 Tables . . . 55

Chapter 1

Part One

ABSTRACT

Hamiltonian Monte Carlo (HMC) and No-U-Turn Sampling (NUTS) have become pivotal tools in Bayesian computation, enabling efficient exploration of high-dimensional posterior distributions through gradient-based methods. Recent advancements, including Latent Hamiltonian Neural Networks (L-HNNs), further extend these techniques by integrating machine learning to handle high-dimensional and latent space representations. This paper reviews the foundational literature on HMC and NUTS, identifying their limitations and emphasizing the challenges of high-dimensional Bayesian inference. To address the risk of losing focus on the core problem, the discussion highlights why traditional Monte Carlo methods often fail and how L-HNNs and NUTS provide effective solutions. Critical implementation details, including leapfrog integration, the Metropolis acceptance criterion, and the *BuildTree* function, are also examined to ensure practical applicability. A comprehensive testing strategy, including manual, integration, and unit tests, is outlined, with an emphasis on maintaining consistent configuration updates to acquire the replication results in *TensorFlow*. By bridging theoretical advancements and practical execution, this paper provides a roadmap for leveraging modern sampling methods to address computational challenges in Bayesian inference.

Keywords Hamiltonian Monte Carlo (HMC) · No-U-Turn Sampling (NUTS) · Bayesian inference · *TensorFlow* Replication

1.1 Introduction

Statistical modeling and Bayesian inference often require sampling from complex, high-dimensional probability distributions. However, designing efficient sampling algorithms for such problems remains challenging. Classical methods such as random-walk-based Monte Carlo simulations suffer from inefficiency, particularly in high-dimensional spaces where the probability mass is often concentrated in narrow regions. Hamiltonian Monte Carlo (HMC) addresses this issue by introducing auxiliary momentum variables and leveraging Hamiltonian dynamics to explore the probability space more effectively.

Latent Hamiltonian Neural Networks (LHNNs) extend this approach by learning the underlying Hamiltonian dynamics in a latent representation, enabling greater flexibility and scalability. When combined with the No-U-Turn Sampling (NUTS) algorithm, LHNNs can further enhance efficiency by adaptively determining trajectory lengths, removing the need for manual tuning.

This document provides a clear discussion of Hamiltonian Monte Carlo (HMC), No-U-Turn Sampling (NUTS), and extensions such as Latent Hamiltonian Neural Networks (L-HNNs) to address challenges in Bayesian computation and sampling efficiency. The remainder of this paper is organized as follows. Section 1.2 provides a literature review of foundational methods, including Hamiltonian Monte Carlo (HMC) and No-U-Turn Sampling (NUTS), as well as recent extensions such as Latent Hamiltonian Neural Networks (L-HNNs). This section establishes the theoretical groundwork necessary for understanding modern advancements in Bayesian computation, clarifying key concepts for readers without prior knowledge of Hamiltonian mechanics, Bayesian inference, or Monte Carlo methods. Sections 1.3, 1.4, and 1.5 aim to prevent readers from getting lost in the extensive literature or overlooking the core problem that motivates these methods. Section 1.3 highlights the challenges posed by high-dimensional probability distributions and the

limitations of classical Monte Carlo techniques. Section 1.4 introduces the structure of L-HNNs, explaining how these networks address these challenges by leveraging latent space representations. Section 1.5 explores the mechanics of NUTS, showing how specific innovations, including the U-turn condition, overcome critical inefficiencies in traditional sampling approaches. Section 1.6 complements this discussion by presenting implementation-critical elements that are not fully explored in earlier sections, such as leapfrog integration, the Metropolis acceptance criterion, and the *BuildTree* function, which ensures the practical feasibility of NUTS. Section 1.7 concludes with a comprehensive description of the testing strategy, covering manual, integration, and unit tests, while underscoring the importance of maintaining updated configuration files to ensure consistency. These sections collectively emphasize the need to focus on the problem itself and highlight how advanced methods bridge theoretical challenges and practical execution.

1.2 Literature Review

Efficient sampling methods are critical for Bayesian inference, particularly in high-dimensional and complex systems where posterior distributions are difficult to analyze. Traditional Monte Carlo sampling (MCS) methods, including Metropolis-Hastings [Metropolis et al., 1953, Hastings, 1970], have been widely used but suffer from inefficiencies in high-dimensional spaces due to their reliance on random-walk proposals, which result in slow convergence and high autocorrelation between samples. These limitations motivated the development of Hamiltonian Monte Carlo (HMC) and its extensions, which have become powerful tools in modern Bayesian inference.

1.2.1 Hamiltonian Monte Carlo and No-U-Turn Sampling

HMC, first introduced in the context of molecular dynamics by Duane et al. [1987] and later adapted for Bayesian inference by Neal [2012], combines classical mechanics with Monte Carlo methods to explore probability distributions more efficiently. By introducing auxiliary momentum variables and simulating Hamiltonian dynamics, HMC generates proposals that traverse long distances in the probability space, reducing the random-walk behavior and improving sampling efficiency. These trajectories are guided by the gradient of the log-posterior distribution, enabling HMC to adapt to the geometry of the target distribution and navigate anisotropic curvature effectively.

Despite its advantages, HMC requires careful tuning of the trajectory length and step size, which can be challenging in practice. Hoffman et al. [2014] addressed this issue with the No-U-Turn Sampler (NUTS), an extension of HMC that automatically adapts the trajectory length during sampling. NUTS employs a recursive doubling procedure and stops the trajectory when it detects a "U-turn," ensuring efficient exploration without manual tuning. NUTS has been successfully applied to a wide range of applications and is implemented in popular probabilistic programming frameworks , including Stan Carpenter et al. [2017]. Nishio and Arakawa Nishio and Arakawa [2019] further demonstrated the effectiveness of HMC and NUTS for estimating genetic parameters and breeding values, highlighting their robustness in practical scenarios.

1.2.2 Extensions and Applications of Monte Carlo Methods

The development of HMC has inspired numerous extensions and applications in diverse fields. Vishnoi [2021] provides an accessible introduction to the theoretical foundations of HMC, offering insights into its utility for sampling in high-dimensional spaces.Che et al. [2021] applied Monte Carlo methods, including Kriging and Variational Bayesian Monte Carlo, to improve predictions of doped UO_2 fission gas release, demonstrating the versatility of these techniques in nuclear energy applications. Similarly, Dhulipala et al. [2022] proposed an active learning framework with multifidelity modeling to improve rare event simulations, showcasing the integration of Monte Carlo methods with machine learning for efficient simulation.

In recent years, the combination of HMC with artificial neural networks has garnered significant attention. Levy et al. [2017] proposed generalizing HMC with neural networks, leveraging their flexibility to model complex systems and improve sampling efficiency. This approach paves the way for methods such as Latent Hamiltonian Neural Networks (LHNNs), which learn Hamiltonian dynamics in a latent space to handle high-dimensional and nonlinearly constrained systems. Applications of neural-network-based HMC for Bayesian inference of reaction kinetics models have further illustrated the potential of these hybrid techniques to address high-dimensional problems. Additionally, hierarchical Bayesian models [Yin et al., 2023] have been explored to design more sophisticated sampling methods that combine domain knowledge with computational efficiency.

1.2.3 Impact of HMC on Modern Bayesian Inference

Bayesian inference provides a robust framework for estimating parameters and quantifying uncertainty in dynamic and complex systems. Researchers have developed efficient methods to improve Bayesian computation across various domains, including time series analysis, finance, and stochastic modeling. Gerlach et al. [2000] designed Bayesian inference techniques for dynamic mixture models, enabling effective analysis of time-varying phenomena. Kandel et al. [1995] applied Bayesian methods to portfolio efficiency, demonstrating their usefulness in optimizing financial decision-making. Kastner [2017] introduced the stochvol package, delivering computationally efficient tools for analyzing stochastic volatility models in financial data. Whiteley et al. [2010] proposed discrete particle Markov chain Monte Carlo methods, which advanced inference for switching state-space models in discrete-time dynamic systems.

Advancements in Bayesian computation have addressed challenges associated with high-dimensional and complex models. Kastner et al. [2017] developed multivariate factor stochastic volatility models to facilitate inference in large-scale financial data. Han et al. [2022] improved probabilistic model updating by integrating polynomial chaos expansions with Gibbs sampling, which reduced computational costs while maintaining accuracy. These contributions highlight the growing focus on improving computational efficiency in Bayesian inference for modern applications.

Modern methods, including Hamiltonian Monte Carlo (HMC) and No-U-Turn Sampling (NUTS), have revolutionized Bayesian computation by utilizing gradient-based approaches to efficiently explore posterior distributions. Hoffman et al. [2014] proposed NUTS which are regarded as an adaptive extension of HMC, which removes the need for manual tuning of the trajectory length parameter, significantly improving the robustness and accessibility. Dhulipala et al. [2023] expanded this field by introducing Latent Hamiltonian Neural Networks (LHNNs), which integrate HMC with machine learning to sample high-dimensional and latent space representations in a more efficient manner. These advancements have enhanced the potential of Bayesian inference, enabling researchers to address problems that were previously computationally infeasible. By merging traditional methodologies with modern computational frameworks, HMC and NUTS have bridged the gap between classical Bayesian techniques and the computational demands of contemporary scientific and industrial challenges.

1.3 Motivation: Why Efficient Exploration Is Challenging

Efficient exploration of high-dimensional probability distributions is a core problem in Bayesian inference. These challenges arise due to the inherent nature of high-dimensional spaces, inefficiencies in classical methods, and the need for algorithms that can adapt to the structure of the target distribution. To avoid getting lost in the literature, this section provides an intuitive understanding of these issues and explains why Hamiltonian Monte Carlo (HMC) offers a superior solution.

1.3.1 Challenges of High-Dimensional Probability Distributions

- **Concentration of Probability Mass**: In high-dimensional spaces, the volume of the space grows exponentially. As a result, the majority of the probability mass concentrates in a narrow "shell" away from the center. For instance, consider a multivariate Gaussian distribution with mean $\mathbf{0}$ and covariance matrix $\mathbf{I}$. The squared Euclidean distance from the origin follows a χ^2 distribution with d degrees of freedom. As d increases, the density concentrates around $\sqrt{d}$, forming a thin annular region. Random-walk-based methods struggle to locate and stay in this high-probability region because steps taken in random directions often fall outside this shell.

- **Curvature of the Probability Space**: A number of distributions in Bayesian inference exhibit anisotropic curvature, meaning the shape of the distribution stretches more in some directions than others. For example, imagine a stretched-out elliptical valley. If the variance along one direction is much larger than along another, the probability space becomes difficult to traverse. Sampling methods that assume isotropy, such as random-walk proposals, fail to adapt to this curvature, leading to wasted computational effort.

- **Autocorrelation Between Samples**: In classical Monte Carlo methods, new samples are generated based on small perturbations of previous samples, creating a strong dependency between consecutive points. This results in high autocorrelation, reducing the effective sample size. For example, in a random-walk Metropolis algorithm, the next state depends on whether the current proposal lies within the target distribution, leading to slow exploration of the space.

1.3.2 Limitations of Classical Monte Carlo Methods

Efficient exploration of high-dimensional probability distributions is a core problem in Bayesian inference. These challenges arise due to the inherent nature of high-dimensional spaces, inefficiencies in classical methods, and the need for algorithms that can adapt to the structure of the target distribution. This section provides an intuitive understanding of these issues and explains why Hamiltonian Monte Carlo (HMC) offers a superior solution.

- **Inefficiency in High Dimensions**: The random-walk proposal mechanism generates small, local steps that explore the space incrementally. In high dimensions, the volume of plausible proposals shrinks, and most proposed steps are rejected. A useful metaphor is a person trying to explore a high-dimensional maze by taking random steps—most steps lead to dead ends, and progress is exceedingly slow. Mathematically, the acceptance probability decreases exponentially with the dimensionality d, as small perturbations are less likely to land in high-probability regions.
- **Poor Adaptation to Geometry**: Classical Monte Carlo methods do not account for the local geometry of the target distribution. For example, in a highly anisotropic distribution, random-walk proposals will take many steps to traverse the narrow directions, even if the wide directions are easy to cross. This is analogous to trying to walk along the length of a canyon using random steps—most effort goes into climbing the steep walls instead of advancing along the bottom.
- **High Autocorrelation**: Random-walk methods generate new samples by perturbing the current state, which leads to strong correlations between consecutive samples. This slows down convergence and reduces the effective sample size. For example, if a sample is stuck in a local mode, it can take many steps to escape, resulting in redundant samples that do not represent the full posterior distribution.

These limitations severely affect the performance of classical Monte Carlo methods, particularly when dealing with complex and high-dimensional distributions.

1.3.3 Why Hamiltonian Monte Carlo (HMC)?

Hamiltonian Monte Carlo overcomes these limitations by leveraging principles from classical mechanics to explore probability distributions more efficiently. The algorithm introduces auxiliary momentum variables $\mathbf{p}$ and simulates the dynamics of a particle moving through the probability space. The system is governed by the Hamiltonian function:

$$H(\mathbf{q}, \mathbf{p}) = U(\mathbf{q}) + K(\mathbf{p}),$$

where $\mathbf{q}$ represents the position (parameters of interest), $\mathbf{p}$ represents the momentum, $U(\mathbf{q}) = -\log P(\mathbf{q})$ is the potential energy derived from the target distribution, and $K(\mathbf{p}) = \frac{1}{2}\mathbf{p}^\top \mathbf{M}^{-1}\mathbf{p}$ is the kinetic energy with mass matrix $\mathbf{M}$. The following features explain why HMC performs better:

- **Efficient Exploration of Probability Space**: Hamiltonian Monte Carlo (HMC) simulates the trajectory of a particle under Hamiltonian dynamics, enabling it to travel long distances in the probability space without getting stuck in local modes. This advantage arises from the use of momentum variables $\mathbf{p}$ and the gradients of the log-probability density, which guide the particle to efficiently explore the target distribution.

  In classical Monte Carlo methods, such as the Metropolis-Hastings algorithm, the random-walk proposal mechanism generates small, local steps:

  $$\mathbf{q}_{t+1} = \mathbf{q}_t + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I}).$$

  This approach struggles in high-dimensional scenarios because the volume of plausible proposals shrinks exponentially with the dimensionality d. To see why, consider the acceptance probability in Metropolis-Hastings, which is defined as:

  $$\alpha(\mathbf{q}_t, \mathbf{q}_{t+1}) = \min\left(1, \frac{\pi(\mathbf{q}_{t+1})}{\pi(\mathbf{q}_t)}\right),$$

  where $\pi(\mathbf{q})$ is the target probability density function (PDF). If the target distribution is a multivariate Gaussian:

  $$\pi(\mathbf{q}) = \frac{1}{(2\pi)^{d/2}} \exp\left(-\frac{\|\mathbf{q}\|^2}{2}\right),$$

  the acceptance probability depends on the relative distance of the proposed sample $\mathbf{q}_{t+1}$ from the origin compared to the current sample $\mathbf{q}_t$. For a random-walk proposal, the squared distance of the proposed sample from the origin is:

  $$\|\mathbf{q}_{t+1}\|^2 = \|\mathbf{q}_t + \epsilon\|^2 = \|\mathbf{q}_t\|^2 + 2\mathbf{q}_t^\top \epsilon + \|\epsilon\|^2.$$

In high dimensions, the term $\|\epsilon\|^2 \sim \chi^2(d)$ grows roughly proportionally to d, while the cross-term $2\mathbf{q}_t^\top \epsilon$ averages to zero because ϵ is independent of $\mathbf{q}_t$. This means that $\|\mathbf{q}_{t+1}\|$ is likely to deviate significantly from the thin shell of high probability (concentrated around $\|\mathbf{q}\| \sim \sqrt{d}$). As a result, the ratio $\pi(\mathbf{q}_{t+1})/\pi(\mathbf{q}_t)$ becomes vanishingly small, causing the acceptance probability to decay exponentially with d. This leads to frequent rejections, slow exploration, and inefficient sampling in high dimensions.

HMC avoids this inefficiency by leveraging Hamiltonian dynamics. Instead of relying on small perturbations, HMC simulates a deterministic trajectory through the probability space, governed by the Hamiltonian:

$$\frac{d\mathbf{q}}{dt} = \nabla_{\mathbf{p}} H, \quad \frac{d\mathbf{p}}{dt} = -\nabla_{\mathbf{q}} H.$$

Here, $\mathbf{q}$ represents the position (parameters of interest), $\mathbf{p}$ is the momentum, and H is the Hamiltonian function that incorporates the target distribution through the potential energy $U(\mathbf{q}) = -\log P(\mathbf{q})$. These dynamics enable the particle to "build up speed" in regions of low curvature (wide valleys) and traverse long distances without being constrained by the dimensionality of the space.

A key reason HMC maintains high acceptance rates, even in high dimensions, is that proposals are informed by the gradients of the log-probability density, $-\nabla U(\mathbf{q})$. This gradient information ensures that the trajectory aligns with the geometry of the target distribution, allowing the sampler to move efficiently into regions of high probability. Unlike random-walk methods, HMC does not propose steps blindly; instead, it uses the structure of the probability space to make long, informed jumps. As a result, the acceptance probability remains stable as the dimensionality d increases, avoiding the exponential decay seen in classical methods.

To provide a concrete comparison, consider a high-dimensional Gaussian distribution with mean $\mathbf{0}$ and covariance matrix $\mathbf{I}$. Classical random-walk proposals must take very small step sizes σ to maintain an acceptable acceptance rate, leading to slow exploration. In contrast, HMC exploits the smoothness of the Gaussian to propose large, efficient steps along trajectories that preserve the probability density, resulting in significantly faster convergence and exploration.

This efficient traversal of the probability space eliminates the slow, incremental steps of random-walk methods and reduces the chance of revisiting the same regions repeatedly. The combination of momentum and gradient-driven proposals allows HMC to explore the space globally, achieving much higher efficiency in high-dimensional settings.

- **Geometry Awareness**: The gradients of the potential energy, $-\nabla U(\mathbf{q})$, guide the trajectory, allowing HMC to adapt to the local geometry of the distribution. For instance, in an anisotropic distribution, the particle moves faster along directions with low curvature and slower along directions with high curvature, akin to navigating a winding river by following the current.
- **Reduced Autocorrelation**: Hamiltonian Monte Carlo reduces autocorrelation between consecutive samples by leveraging deterministic trajectories based on Hamiltonian dynamics. To understand this, consider how classical Monte Carlo methods, such as the Metropolis-Hastings algorithm, generate samples. In these methods, the next state $\mathbf{q}_{t+1}$ is proposed by adding a random perturbation ϵ to the current state $\mathbf{q}_t$:

  $$\mathbf{q}_{t+1} = \mathbf{q}_t + \epsilon,$$

  where $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$. This random-walk behavior leads to small, incremental steps that are heavily dependent on the previous state. As a result, the autocorrelation between consecutive samples is high, especially in high-dimensional spaces where most proposals are rejected, causing the sampler to remain stuck in a local region for many iterations.

  In contrast, HMC simulates a trajectory in the probability space using the equations of motion derived from the Hamiltonian:

  $$\frac{d\mathbf{q}}{dt} = \nabla_{\mathbf{p}} H, \quad \frac{d\mathbf{p}}{dt} = -\nabla_{\mathbf{q}} H.$$

  These dynamics allow the sampler to "leap" across the probability space over many dimensions in a single proposal. Instead of relying on small random steps, HMC uses the gradient of the log-probability to guide the trajectory, enabling large, informed jumps. The resulting state $\mathbf{q}_{t+1}$ depends not only on the current position $\mathbf{q}_t$ but also on the auxiliary momentum $\mathbf{p}$, which introduces a second degree of freedom. This reduces the dependency between consecutive samples and decreases autocorrelation.

  A simple comparison of autocorrelation can be made using the effective sample size (ESS), which measures how many independent samples are equivalent to the correlated samples produced by a sampler. Classical Monte Carlo methods often yield small ESS because of high autocorrelation, while HMC achieves a much higher ESS because of its ability to efficiently explore the space without revisiting the same regions repeatedly.

Despite these advantages, HMC requires careful tuning of the step size and trajectory length. Small step sizes improve accuracy but reduce efficiency, while large step sizes risk numerical instability. The No-U-Turn Sampler (NUTS) automates this tuning process by dynamically adjusting the trajectory length to ensure efficient exploration, as discussed in Section 1.5.

1.4 Latent Hamiltonian Neural Networks (L-HNN)

Hamiltonian Neural Networks (HNNs) aim to learn the Hamiltonian function H_θ that governs a system's dynamics through Hamilton's equations:

$$\frac{d\mathbf{q}}{dt} = \frac{\partial H_\theta}{\partial \mathbf{p}}, \quad \frac{d\mathbf{p}}{dt} = -\frac{\partial H_\theta}{\partial \mathbf{q}}. \tag{1.1}$$

The key idea is to model the Hamiltonian H_θ using a neural network parameterized by θ and train it to predict the system's dynamics. Two approaches exist for implementing HNNs: *Direct HNN* and *Latent HNN*. Below, we explain their differences and why Latent HNN provides additional flexibility for complex systems.

1.4.1 Shared Loss Function

Both Direct HNN and Latent HNN share the same loss function, which measures the discrepancy between the predicted dynamics, derived from the Hamiltonian, and the observed dynamics. Specifically:

$$L = \left\| \frac{\partial H_\theta}{\partial \mathbf{p}} - \frac{\Delta \mathbf{q}}{\Delta t} \right\| + \left\| -\frac{\partial H_\theta}{\partial \mathbf{q}} - \frac{\Delta \mathbf{p}}{\Delta t} \right\|, \tag{1.2}$$

where $\Delta\mathbf{q}/\Delta t$ and $\Delta\mathbf{p}/\Delta t$ are the observed time derivatives of position $\mathbf{q}$ and momentum $\mathbf{p}$, respectively. This loss function ensures that the learned Hamiltonian H_θ generates dynamics that align with the observed data.

1.4.2 Direct HNN

In Direct HNN, the Hamiltonian is modeled directly in terms of the observed state variables $\mathbf{q}$ and $\mathbf{p}$:

$$H_\theta = H_\theta(\mathbf{q}, \mathbf{p}), \tag{1.3}$$

where $\mathbf{q}$ and $\mathbf{p}$ are the generalized position and momentum, respectively. This approach is straightforward and effective when $\mathbf{q}$ and $\mathbf{p}$ are fully observable and accurately represent the state of the system.

However, this direct parameterization imposes limitations on the degrees of freedom and expressivity, reducing adaptability to complex or high-dimensional systems. Consequently, integration errors are more likely to accumulate when predicting Hamiltonian dynamics, particularly during long-term simulations.

1.4.3 Latent HNN

To address the limitations of Direct HNN, Latent HNN introduces a set of latent variables $\boldsymbol{\lambda} = \{\lambda_1, \lambda_2, \ldots, \lambda_d\}$, where d is the dimension of the latent space. Instead of directly parameterizing the Hamiltonian in terms of $\mathbf{q}$ and $\mathbf{p}$, Latent HNN models the Hamiltonian as:

$$H_\theta = \sum_{i=1}^{d} \lambda_i, \tag{1.4}$$

where $\boldsymbol{\lambda}$ is the output of a neural network.

Network Architecture

The latent variables $\boldsymbol{\lambda}$ are computed through a multi-layer perceptron (MLP) as follows:

$$\mathbf{u}_p = \phi(\mathbf{W}_{p-1}\mathbf{u}_{p-1} + \mathbf{b}_{p-1}), \quad p \in \{1, \ldots, P\}, \tag{1.5}$$

$$\boldsymbol{\lambda} = \mathbf{W}_P\mathbf{u}_P + \mathbf{b}_P, \tag{1.6}$$

where $\mathbf{W}_p$ and $\mathbf{b}_p$ are the weights and biases of the p-th layer, $\phi(\cdot)$ is the activation function, and $\mathbf{u}_0$ represents the input state variables $\mathbf{z} = \{\mathbf{q}, \mathbf{p}\}$.

Advantages of Latent HNN

Through the use of latent variables, Latent HNN provides the following key advantages:

- **Increased Degrees of Freedom:** The latent variables introduce additional degrees of freedom, allowing the model to better capture complex dependencies and dynamics that may not be directly observable in the input space.
- **Enhanced Expressivity:** By leveraging a latent representation, L-HNNs can model a wider range of Hamiltonian functions, making them more flexible compared to traditional HNNs.
- **Reduced Integration Errors:** The increased expressivity and flexibility help reduce integration errors when predicting Hamiltonian dynamics, particularly for long-term simulations or high-dimensional systems.

1.5 What Is NUTS and Why Use It?

No-U-Turn Sampling (NUTS) is an adaptive version of Hamiltonian Monte Carlo (HMC) designed to eliminate the need for manually tuning the trajectory length, a critical parameter in standard HMC. In traditional HMC, the trajectory length (or equivalently, the number of leapfrog steps) must strike a delicate balance: if it is too short, the sampler behaves like a random walk and explores the space inefficiently; if it is too long, computational resources are wasted, and the sampler risks retracing its path unnecessarily, reducing efficiency.

NUTS addresses this issue by dynamically and adaptively determining the trajectory length during sampling. It uses a recursive doubling procedure to simulate the trajectory and monitors a "U-turn" condition to decide when to stop extending the trajectory. The U-turn condition is given by:

$$(\mathbf{q}_{n+1} - \mathbf{q}_n)^\top \mathbf{p}_{n+1} < 0, \tag{1.7}$$

where:

- $\mathbf{q}_n$ and $\mathbf{q}_{n+1}$ are the current and next positions in the trajectory,
- $\mathbf{p}_{n+1}$ is the momentum at the next position, and
- the dot product $(\mathbf{q}_{n+1} - \mathbf{q}_n)^\top \mathbf{p}_{n+1}$ measures whether the trajectory is starting to reverse direction.

1.5.1 Intuition Behind the U-Turn Condition

To understand the U-turn condition intuitively, recall that in HMC, the particle's motion through the probability space is guided by Hamiltonian dynamics. The momentum $\mathbf{p}$ acts like velocity, and the position $\mathbf{q}$ represents the particle's current location. As the particle moves, it explores the probability space, ideally traveling in a direction that efficiently samples the target distribution.

The U-turn condition captures the scenario where the trajectory starts to reverse back toward previously visited regions. Specifically:

- The vector $\mathbf{q}_{n+1} - \mathbf{q}_n$ represents the direction of motion between two successive positions in the trajectory.
- The momentum $\mathbf{p}_{n+1}$ indicates the current "velocity" of the particle.
- The displacement direction $(\mathbf{q}_{n+1} - \mathbf{q}_n)$ and the momentum vectors $\mathbf{p}^{n+1}$) should generally align. That is, the particle is still exploring new areas of the probability space.
- If the dot product $(\mathbf{q}_{n+1} - \mathbf{q}_n)^\top \mathbf{p}_{n+1}$ is negative, it means the particle's momentum is now oriented in the opposite direction of its recent movement. In other words, the trajectory is "turning back" on itself, like making a U-turn. At this point, further extending the trajectory would waste computational resources because the particle would start retracing its path, revisiting regions that have already been explored.

To visualize this, imagine walking through a valley:

- Initially, you walk downhill, guided by the gradients of the terrain (analogous to the gradients of the log-posterior in HMC).
- As you approach a flat or uphill region, your momentum gradually decreases due to resistance.
- If you continue walking too far, you might end up turning back and revisiting the same path you just traveled. The U-turn condition detects this reversal and stops the trajectory before you waste further effort retracing your steps.

By dynamically stopping the trajectory when the U-turn condition is met, NUTS avoids unnecessary computation while ensuring that the sampler remains efficient.

1.5.2 Why Bidirectional Evolution

The modified bidirectional evolution condition implemented by [Dhulipala et al., 2023] presents a subtle yet important variation from the traditional NUTS algorithm:

- $(\mathbf{q}^+ - \mathbf{q}^-)$ is the displacement vector between the leftmost and rightmost positions of the trajectory,
- $\mathbf{p}^-$ and $\mathbf{p}^+$ are the momenta at $\mathbf{q}^-$ and $\mathbf{q}^+$, respectively, and
- The dot products $(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^-$ and $(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^+$ check whether the displacement direction aligns with the momenta at the trajectory's extremes.

The necessity of bidirectional evolution in NUTS algorithm stems from several key considerations:

- **Enhanced Space Exploration**
  - Unidirectional evolution might miss crucial regions of the parameter space
  - Bidirectional sampling provides more comprehensive exploration
  - The algorithm can adapt to local geometric structures in both time directions
- **Time Reversibility**
  - Hamiltonian systems are inherently time-reversible
  - Bidirectional evolution preserves this fundamental physical property
  - Ensures detailed balance in the sampling process
- **Robust U-turn Detection**
  - Considers momentum states at both endpoints ($\mathbf{p}^-$ and $\mathbf{p}^+$)
  - Avoids bias from local trajectory behavior
  - Provides more reliable termination criteria through the condition:

$$s \leftarrow s \cdot \mathbf{1}[(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^- \geq 0] \cdot \mathbf{1}[(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^+ \geq 0] \tag{1.8}$$

- **Symmetric Tree Building**
  - Enables balanced exploration in phase space
  - Reduces correlation between successive samples
  - Improves mixing properties of the Markov chain

1.6 Additional Implementation Details

This section describes additional key components that are essential for understanding and implementing the discussed ideas. These points include the Metropolis acceptance criterion, leapfrog integration, and specific data generation techniques. While these concepts were not discussed in earlier sections, they are crucial for practical implementation.

1.6.1 Leapfrog Integration

Leapfrog integration is a symplectic numerical method commonly used to simulate Hamiltonian dynamics. It ensures accurate solutions to the equations of motion while preserving the energy and phase-space volume of the system over long trajectories. The kinetic energy $K(\mathbf{p})$ is assumed to be diagonal and is expressed as:

$$K(\mathbf{p}) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i},$$

where:

- m_i is the mass associated with the i-th degree of freedom (or dimension). It determines the "inertia" of the system in that dimension, i.e., how much a change in momentum p_i affects the velocity $\frac{p_i}{m_i}$.
- d is the total number of dimensions in the system.

In this study, we assume $m_i = 1$ for all dimensions i. This simplifies the equations, as the velocity becomes directly proportional to the momentum ($\frac{p_i}{m_i} = p_i$) and eliminates the need for explicit mass terms in the updates.

The synchronized leapfrog integration scheme updates the position $q_i(t)$ and momentum $p_i(t)$ in each dimension i as follows:

1. **Position Update**:

$$q_i(t + \Delta t) = q_i(t) + \Delta t\, p_i(t) - \frac{\Delta t^2}{2} \frac{\partial H}{\partial q_i(t)}. \tag{1.9}$$

   Here:

   - The term $\Delta t\, p_i(t)$ accounts for the velocity contribution to the position update (since $m_i = 1$).
   - The term $-\frac{\Delta t^2}{2} \frac{\partial H}{\partial q_i(t)}$ accounts for the acceleration due to the potential energy gradient.
   - Δt is the discretization time step that controls the resolution of the simulation.

2. **Momentum Update**: The momentum is updated in two synchronized half-steps:

$$p_i\left(t + \frac{\Delta t}{2}\right) = p_i(t) - \frac{\Delta t}{2} \frac{\partial H}{\partial q_i(t)}, \tag{1.10}$$

$$p_i(t + \Delta t) = p_i\left(t + \frac{\Delta t}{2}\right) - \frac{\Delta t}{2} \frac{\partial H}{\partial q_i(t + \Delta t)}. \tag{1.11}$$

   The momentum update accounts for the force (negative gradient of the potential energy) acting on the system.

3. **Synchronized Updates**: The leapfrog integration alternates between updating positions and momenta in a synchronized manner:

   - First, $p_i(t)$ is updated by a half-step.
   - Next, $q_i(t)$ is updated by a full step using the updated momentum.
   - Finally, $p_i(t)$ is updated by another half-step, completing the synchronization.

The synchronized leapfrog scheme, as shown in Algorithm 2, ensures that the symplectic property (volume preservation in phase space) and energy conservation are approximately maintained, even over long trajectories. These features make it particularly suitable for Hamiltonian-based methods like Hamiltonian Monte Carlo (HMC).

1.6.2 Metropolis Acceptance Criterion

In this study, we use the Metropolis Acceptance Criterion in both HNN and L-HNN Monte Carlo. The Metropolis acceptance criterion is used to decide whether to accept or reject a candidate state generated by leapfrog integration. This step ensures that theMonte Carlo sampling process produces samples that follow the target distribution. The update rule is given by:

$$\{\mathbf{q}^i, \mathbf{p}^i\} \leftarrow \{\mathbf{q}^*, \mathbf{p}^*\} \text{ with probability } \alpha,$$

where the acceptance probability α is defined as:

$$\alpha = \min\left(1, \exp\left(H(\mathbf{q}^{i-1}, \mathbf{p}^{i-1}) - H(\mathbf{q}^*, \mathbf{p}^*)\right)\right),$$

and:

- $(\mathbf{q}^*, \mathbf{p}^*)$ is the candidate state after leapfrog integration.
- $(\mathbf{q}^{i-1}, \mathbf{p}^{i-1})$ is the current state.
- $H(\mathbf{q}, \mathbf{p}) = U(\mathbf{q}) + K(\mathbf{p})$ is the Hamiltonian.

If the candidate state is rejected, the current state $(\mathbf{q}^{i-1}, \mathbf{p}^{i-1})$ is retained. This criterion ensures detailed balance and preserves the target distribution.

1.6.3 Data Generation

Generating data for training Hamiltonian Neural Networks (HNNs) and their extensions involves simulating trajectories of a Hamiltonian system. The following specific steps are used:

1. **Initialize position:** The initial position $\mathbf{q}(0)$ is set to the final position from the previous sample (zero at the beginning):
$$\mathbf{q}(0) = \mathbf{q}^{\text{previous}}.$$
2. **Sample momentum:** The initial momentum $\mathbf{p}(0)$ is drawn randomly from a standard Gaussian distribution:
$$\mathbf{p}(0) \sim \mathcal{N}(0, \mathbf{I}_d),$$
where $\mathbf{I}_d$ is the d-dimensional identity matrix.
3. **Simulate dynamics:** Using leapfrog integration, simulate trajectories by alternating updates to $\mathbf{q}(t)$ and $\mathbf{p}(t)$ over a time interval $[0, T]$ with step size Δt.
4. **Record samples:** Save the position and momentum samples $\{\mathbf{q}(t), \mathbf{p}(t)\}$ at each time step.

These steps ensure that the generated dataset accurately reflects the Hamiltonian system dynamics, yielding suitable training data for HNNs.

Algorithm 1 Hamiltonian neural network training

HNN parameters: $\boldsymbol{\theta}$; **Training data:** $\{\mathbf{q}, \mathbf{p}\}$
Evaluate gradients of the training data $\frac{\Delta \mathbf{q}}{\Delta t}$ and $\frac{\Delta \mathbf{p}}{\Delta t}$
Initialize HNN parameters $\boldsymbol{\theta}$
Compute HNN Hamiltonian $H_{\boldsymbol{\theta}}$
Compute $H_{\boldsymbol{\theta}}$ gradients $\frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{p}}$ and $-\frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{q}}$
Compute loss
$$\mathcal{L} = \left\| \frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{p}} - \frac{\Delta \mathbf{q}}{\Delta t} \right\| + \left\| -\frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{q}} - \frac{\Delta \mathbf{p}}{\Delta t} \right\|$$
Minimize $\mathcal{L}$ with respect to $\boldsymbol{\theta}$

Algorithm 2 Hamiltonian neural networks evaluation in leapfrog integration

Hamiltonian: H; **Initial conditions:** $\mathbf{z}(0) = \{\mathbf{q}(0), \mathbf{p}(0)\}$; **Dimensions:** d; **Steps:** N; **End time:** T
$\Delta t = \frac{T}{N}$
for $j = 0 : N - 1$ **do**
  $t = j\Delta t$
  Compute HNN output gradient $\frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{q}(t)}$
  for $i = 1 : d$ **do**
    $q_i(t + \Delta t) = q_i(t) + \frac{\Delta t}{m_i} p_i(t) - \frac{\Delta t^2}{2m_i} \frac{\partial H_{\boldsymbol{\theta}}}{\partial q_i(t)}$
  end for
  Compute HNN output gradient $\frac{\partial H_{\boldsymbol{\theta}}}{\partial \mathbf{q}(t+\Delta t)}$
  for $i = 1 : d$ **do**
    $p_i(t + \Delta t) = p_i(t) - \frac{\Delta t}{2} \left(\frac{\partial H_{\boldsymbol{\theta}}}{\partial q_i(t)} + \frac{\partial H_{\boldsymbol{\theta}}}{\partial q_i(t+\Delta t)} \right)$
  end for
end for

1.6.4 Online Monitoring

Online monitoring is a critical component of the proposed sampling workflow, ensuring accuracy and stability when integrating Hamiltonian dynamics. This mechanism evaluates the integration process dynamically and adapts the workflow based on a predefined error metric, reducing the impact of integration errors.

Algorithm 3 L-HNNs in Hamiltonian Monte Carlo

Hamiltonian: $H = U(\mathbf{q}) + K(\mathbf{p})$; **Samples:** M; **Starting sample:** $\{\mathbf{q}^0, \mathbf{p}^0\}$; **End time for trajectory:** T; **Steps:** N; **Dimensions:** d
for $i = 1 : M$ **do**
  $\mathbf{p}(0) \sim \mathcal{N}(0, \mathbf{I}_d)$
  $\mathbf{q}(0) = \mathbf{q}^{i-1}$
  Compute $\{\mathbf{q}^*, \mathbf{p}^*\} = \{\mathbf{q}(T), \mathbf{p}(T)\}$ with Algorithm 2
  $\alpha = \min\{1, \exp(H(\{\mathbf{q}^{i-1}, \mathbf{p}^{i-1}\}) - H(\{\mathbf{q}^*, \mathbf{p}^*\}))\}$
  With probability α, set $\{\mathbf{q}^i, \mathbf{p}^i\} \leftarrow \{\mathbf{q}^*, \mathbf{p}^*\}$
end for

Algorithm 4 L-HNNs in efficient NUTS with online error monitoring (main loop)

Hamiltonian: $H = U(\mathbf{q}) + K(\mathbf{p})$; **Samples:** M; **Starting sample:** $\{\mathbf{q}^0, \mathbf{p}^0\}$; **Step size:** Δt; **Threshold for leapfrog:** $\Delta_{\text{lf,max}}$; **Threshold for L-HNNs:** $\Delta_{\text{hnn,max}}$; **Number of leapfrog samples:** N_{lf}
Initialize $l_{\text{lf}} = 0$, $n_{\text{lf}} = 0$
for $i = 1 : M$ **do**
  $\mathbf{p}(0) \sim \mathcal{N}(0, \mathbf{I}_d)$
  $\mathbf{q}(0) = \mathbf{q}^{i-1}$
  $u \sim$ Uniform $([0, \exp\{-H(\mathbf{q}(0), \mathbf{p}(0))\}])$
  Initialize $\mathbf{q}^- = \mathbf{q}(0), \mathbf{q}^+ = \mathbf{q}(0), \mathbf{p}^- = \mathbf{p}(0), \mathbf{p}^+ = \mathbf{p}(0), j = 0, \mathbf{q}^* = \mathbf{q}^{i-1}, n = 1, s = 1$
  if $l_{\text{lf}} = 1$ **then**
    $n_{\text{lf}} \leftarrow n_{\text{lf}} + 1$
  end if
  if $n_{\text{lf}} = N_{\text{lf}}$ **then**
    $l_{\text{lf}} = 0, n_{\text{lf}} = 0$
  end if
  while $s = 1$ **do**
    Choose direction $\nu_j \sim$ Uniform$(\{-1, 1\})$
    if $j = -1$ **then**
      $\mathbf{q}^-, \mathbf{p}^-, \ldots, \mathbf{q}', \mathbf{p}', n', s', l_{\text{lf}} \leftarrow$ BuildTree$(\mathbf{q}^-, \mathbf{p}^-, u, \nu_j, j, \Delta t, l_{\text{lf}})$
    else
      $\ldots, \mathbf{q}^+, \mathbf{p}^+, \mathbf{q}', \mathbf{p}', n', s', l_{\text{lf}} \leftarrow$ BuildTree$(\mathbf{q}^+, \mathbf{p}^+, u, \nu_j, j, \Delta t, l_{\text{lf}})$
    end if
    if $s' = 1$ **then**
      With probability $\min\{1, n'/n\}$, set $\{\mathbf{q}^i, \mathbf{p}^i\} \leftarrow \{\mathbf{q}', \mathbf{p}'\}$
    end if
    $n \leftarrow n + n'$
    $s \leftarrow s'\mathbf{1}[(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^- \geq 0]\mathbf{1}[(\mathbf{q}^+ - \mathbf{q}^-) \cdot \mathbf{p}^+ \geq 0]$
    $j \leftarrow j + 1$
  end while
end for

Algorithm 5 L-HNNs in efficient NUTS with online error monitoring (build tree function)

function BUILDTREE($\mathbf{q}, \mathbf{p}, u, \nu, j, \Delta t, l_{\text{lf}}$)
  if $j = 0$ **then**
    Base case taking one leapfrog step
    $\mathbf{q}', \mathbf{p}' \leftarrow$ Algorithm 2 with initial conditions: $\mathbf{z}(0) = \{\mathbf{q}, \mathbf{p}\},$ Steps: 1; End Time: Δt
    $l_{\text{lf}} \leftarrow l_{\text{lf}} \vee \mathbf{1}[H(\mathbf{q}', \mathbf{p}') + \ln u > \Delta_{\text{hnn,max}}]$
    $s' \leftarrow \mathbf{1}[H(\mathbf{q}', \mathbf{p}') + \ln u \leq \Delta_{\text{hnn,max}}]$
    if $l_{\text{lf}} = 1$ **then**
      $\mathbf{q}', \mathbf{p}' \leftarrow$ Leapfrog integration with initial conditions: $\mathbf{z}(0) = \{\mathbf{q}, \mathbf{p}\},$ Steps: 1; End Time: Δt
      $s' \leftarrow \mathbf{1}[H(\mathbf{q}', \mathbf{p}') + \ln u \leq \Delta_{\text{lf,max}}]$
    end if
    $n' \leftarrow \mathbf{1}[u \leq \exp\{-H(\mathbf{q}', \mathbf{p}')\}]$
    return $\mathbf{q}', \mathbf{p}', \mathbf{q}', \mathbf{p}', n', s', l_{\text{lf}}$
  else
    Recursion to build left and right sub-trees (follows from Algorithm 3 in Hoffman et al. [2014], with l_{lf} additionally passed to and retrieved from every BuildTree evaluation)
    return $\mathbf{q}^-, \mathbf{p}^-, \mathbf{q}^+, \mathbf{p}^+, \mathbf{q}', \mathbf{p}', n', s', l_{\text{lf}}$
  end if
end function

Error Metric and Threshold

To monitor the accuracy of the integration process, we define an error metric ϵ as:

$$\epsilon \equiv H(q, p) + \ln u,$$

where $H(q, p)$ is the Hamiltonian (the sum of kinetic and potential energy), and u is a uniformly distributed random variable used in the Metropolis acceptance criterion. Ideally, under perfect integration, $\epsilon \leq 0$. However, in practical applications, numerical integration errors can cause $\epsilon > 0$ during iterations.

To handle these cases, we introduce a threshold $\Delta_{\text{max}}^{\text{hnn}}$ for the error metric when using L-HNNs (Hamiltonian Neural Networks). When ϵ exceeds $\Delta_{\text{max}}^{\text{hnn}}$, the L-HNN is considered unreliable, and the sampler automatically switches to standard Leapfrog integration, which uses numerical gradients of the target density.

Cooldown Period with N_{lf}

After switching to standard Leapfrog integration, the sampler undergoes a cooldown period before reintroducing the L-HNN. This cooldown period is controlled by a parameter N_{lf}, which specifies the number of Leapfrog samples to take before returning to L-HNN-based sampling. The cooldown period ensures that the sampler has moved to regions of phase space where the L-HNN is sufficiently accurate, preventing repeated switches caused by transient errors.

Practical Recommendations

In the traditional No-U-Turn Sampler (NUTS), Hoffman and Gelman recommend setting the error threshold for Leapfrog integration, $\Delta_{\text{lf,max}}$, to a large value (e.g., $\Delta_{\text{lf,max}} = 1,000$). However, for L-HNNs, a more conservative threshold is recommended. For instance, setting $\Delta_{\text{hnn,max}} \approx 10$ helps prevent sampling errors caused by the higher integration inaccuracies of L-HNNs.

1.6.5 *BuildTree* Function

The *BuildTree* function checks this criterion at each recursive step by evaluating whether the trajectory satisfies the stopping condition. As shown in Algorithm 5 if the criterion is violated, the tree-building process halts.

- The *BuildTree* function checks this criterion at each recursive step by evaluating whether the trajectory satisfies the stopping condition. If the criterion is violated, the tree-building process halts.
- *BuildTree* constructs the trajectory by recursively building left and right subtrees. This allows the sampler to balance exploration (new states) and exploitation (accepting samples) while maintaining detailed balance. The recursive nature makes the algorithm computationally efficient, as it avoids the need to store or compute the entire trajectory explicitly.

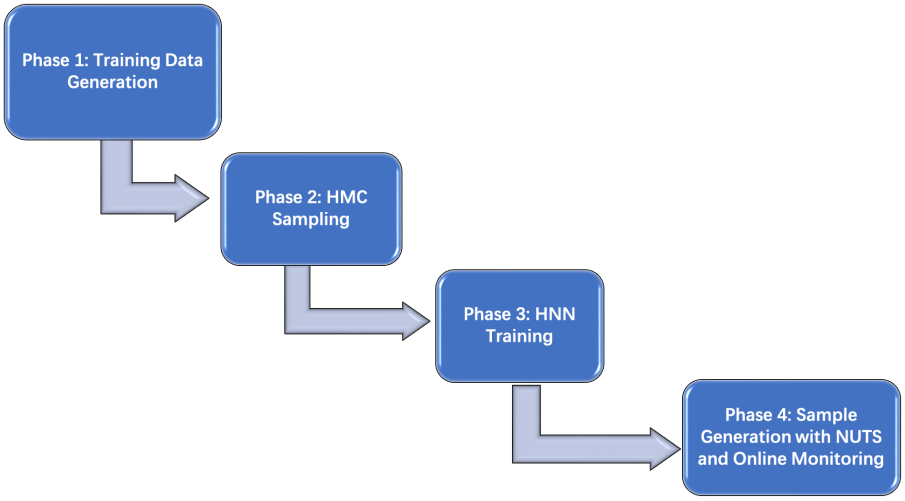


Figure 1.1: The meta workflow for L-HNN in efficient NUTS

- As the tree expands, *BuildTree* stores candidate samples along the trajectory. It probabilistically selects a sample from the trajectory based on the Hamiltonian and Metropolis acceptance criteria, ensuring that the stationary distribution is preserved.
- In the provided algorithm (Algorithm 5), *BuildTree* incorporates online error monitoring by using thresholds ($\Delta_{\text{hnn,max}}$ and $\Delta_{\text{lf,max}}$. This ensures that the leapfrog integration remains accurate and efficient while building the trajectory.

Summary for Online Monitoring

Online monitoring ensures robust and efficient sampling through the following mechanisms:

- Dynamically evaluating the error metric ϵ during sampling.
- Automatically switching to standard Leapfrog integration when the error exceeds $\Delta_{\text{hnn,max}}$.
- Implementing a cooldown period (N_{lf}) to stabilize the sampler before reintroducing L-HNN-based sampling.
- Preventing numerical instability and improving sample quality by adapting the workflow in real time.
- The *BuildTree* function recursively constructs a balanced binary tree of Hamiltonian trajectory states, ensuring efficient exploration of the parameter space while satisfying the No-U-Turn criterion and maintaining detailed balance.

These features are critical for combining the efficiency of L-HNNs with the reliability of standard Leapfrog integration, ensuring the overall robustness of the sampling process.

1.6.6 Workflow

Figure 1.1 shows the overall meta workflow for L-HNNs in efficient NUTS with online error monitoring. The overall workflow consists of four main stages: data generation, HMC sampling, HNN training, and sample generation with online monitoring. Each stage is described below:

1. **Training Data Generation:** The data is generated based on the target Hamiltonian system as described in 1.6.3. Specifically:
   - Initialize the system with random initial positions and momenta, sampled uniformly or from a Gaussian distribution.
   - Simulate the dynamics of the system over a time span using leapfrog integration to compute trajectories of positions and momenta.
   - Save the trajectories as the dataset for subsequent stages.
2. **HMC Sampling:** Using the leapfrog integration function, Hamiltonian Monte Carlo (HMC) is performed to sample from the target distribution. As demonstrated in Algorithm 2 and 3, the steps are as follows:

- Use the initial dataset to define the target potential energy function $U(q)$ and kinetic energy function $K(p)$.
- Perform leapfrog integration to simulate the Hamiltonian dynamics and generate candidate samples.
- Use the Metropolis acceptance criterion to decide whether to accept or reject the proposed samples.
- Save the accepted samples as training data for the Hamiltonian Neural Network (HNN).

3. **HNN Training:** The HNN is trained on the previously generated samples as shown in Algorithm 1 . The input data consists of $2d$-dimensional vectors, where:
   - The first d dimensions correspond to the positions q (associated with potential energy).
   - The last d dimensions correspond to the momenta p (associated with kinetic energy).

   The HNN learns to predict the derivatives of the Hamiltonian, enabling it to simulate dynamics and generate new samples efficiently.
4. **Sample Generation with NUTS and Online Monitoring:** After training the HNN, the model is used to generate new samples. This stage incorporates the No-U-Turn Sampler (NUTS) with online monitoring to improve sampling efficiency:
   - Use the trained HNN to define the Hamiltonian for the system.
   - Apply NUTS to adaptively determine trajectory lengths and step sizes during sampling through *BuildTree* iteratively function as shown in Algorithm 4 and 5
   - Generate high-quality samples that can be used for downstream tasks or further analysis.

This workflow ensures that the data generation, sampling, training, and sample generation processes are seamlessly integrated, leveraging online monitoring to maximize efficiency and robustness.

1.7 Unit test and Integration Test Implementation

This section outlines the unit tests and integration tests conducted for the replication code implemented using *TensorFlow*. It is crucial to highlight that, regardless of the type of testing—whether unit testing, integration testing, or manual testing—the configuration files must be updated beforehand to incorporate any required parameter changes. Ensuring the configuration files are up-to-date guarantees consistency across all testing scenarios and prevents discrepancies arising from outdated or incorrect settings.

1.7.1 Manual Test

To manually run the replicated functions, please perform:

- For the standard Deep Neural Network (DNN) models, execute the script `train_dnn.py` first, followed by `dnn_hmc.py`, `dnn_lmm.py`, and `dnn_nuts_online.py`, all located in the ‘dnns‘ directory.
- For the standard Hamiltonian Neural Network (HNN) models, execute the script `train_hnn.py` first, followed by `hnn_hmc.py`, `hnn_lmm.py`, and `hnn_nuts_online.py`, all located in the `hnns` directory.

1.7.2 Integration Test

Alternatively, integration tests can be directly conducted in their respective directories using the following command:

```
python -m unittest
```

1.7.3 Unit Test

The unit test can also be easily applied through command lines. To execute the corresponding tests for DNN, simply replace `hnn` with `dnn` in the command under the directory `dnns`. For example:

- For `test_00_train_dnn` which is the function of `train_dnn.py`:

  ```
  python -m unittest test_integration.TestIntegration.test_00_train_dnn
  ```

- For `test_01_dnn_hmc` which is the function of `dnn_hmc.py` :

```
python -m unittest test_integration.TestIntegration.test_01_dnn_hmc
```

- For `test_02_dnn_lmc` which is the function of `dnn_lmc.py`:

```
python -m unittest test_integration.TestIntegration.test_02_dnn_lmc
```

- For `test_03_dnn_nuts_online` which is the function of `dnn_nuts_online.py`:

```
python -m unittest test_integration.TestIntegration.test_03_dnn_nuts_online
```

1.7.4 Updating Configuration Files

To modify parameters, follow these steps:

1. Edit the necessary parameters in the `generate_configs.py` file. This file serves as the central configuration generator for all testing scenarios, including unit tests, integration tests, and manual tests.
2. Execute the script to generate the updated configuration files:

```
python generate_configs.py
```

3. Verify that the updated configuration files have been correctly generated in the designated directory (e.g., `configs/`).

1.8 Experiment Results for Part 1

Due to computational limitations, we set $n = 2$ as the default dimensionality in all experiments. While higher-dimensional experiments might provide a more comprehensive evaluation, the selected cases allow for meaningful insights while maintaining computational feasibility. To illustrate key conclusions, we present results for three representative distributions: Neal's funnel, the standard Gaussian, and the Rosenbrock distribution. These distributions were chosen because they exhibit distinct characteristics in terms of geometry and sampling difficulty, making them well-suited for evaluating the effectiveness of different sampling methods. The results of the three distributions are shown in Table 1.1, 1.2 and 1.3.

1.8.1 Comparison Between NUTS and Non-NUTS Methods

A key observation from our results is that, in most cases, methods incorporating the **No-U-Turn Sampler (NUTS) outperform those that do not**. Specifically, for the standard Gaussian distribution and the Rosenbrock distribution, the effective sample sizes (ESS) for NUTS-based methods are consistently higher than those for Hamiltonian Monte Carlo (HMC) without NUTS. This finding aligns with the conclusions drawn in Dhulipala et al. [2023], which highlight the advantages of NUTS in terms of adaptive step-size tuning and improved exploration of the target distribution.

However, a notable exception occurs in the case of Neal's funnel distribution, where the performance difference between NUTS and non-NUTS methods is less pronounced. This suggests that, for highly non-Gaussian and hierarchical structures such as Neal's funnel, the advantages of NUTS may not be as substantial. Further investigation is required to determine whether this is an artifact of the low-dimensional setting or a more general property of funnel-like distributions.

1.8.2 Effectiveness of L-HNN Across Different Distributions

Another critical question is whether the use of Latent Hamiltonian Neural Networks (L-HNNs) provides a significant advantage across different distributions. Our results indicate that while **L-HNN can improve performance in some cases, it does not consistently yield gains across all tested distributions**. Specifically:

- In the standard Gaussian distribution, L-HNN in NUTS demonstrates a slight improvement over traditional NUTS.
- In the Rosenbrock distribution, L-HNN does not show a significant improvement, and in some cases, its performance is slightly worse than traditional NUTS.

- In Neal's funnel, L-HNN in HMC performs worse than traditional HMC, suggesting that its benefits may be distribution-dependent.

These observations contrast with the conclusions of Dhulipala et al. [2023], which primarily focused on Gaussian-like distributions and did not evaluate L-HNN across a broader set of complex distributions. Our findings suggest that effectiveness of L-HNN may be highly dependent on the structure of the target distribution, and further studies are required to understand the generalizability.

1.8.3 Summary of Findings

Based on the above observations, we summarize our key findings as follows:

1. NUTS generally improves performance compared to traditional HMC, except in Neal's funnel.
2. L-HNN does not consistently improve performance across all distributions, indicating that its benefits may be problem-dependent.
3. Further research is needed to evaluate L-HNN on a wider range of distributions beyond the standard Gaussian, as prior work Dhulipala et al. [2023] only provides the relevant results focused on Gaussian-like settings.

Table 1.1: Effective Sample Sizes (ESS) and Gradient Evaluations for nD Funnel

Method	**ESS**	**# Gradients of Target Density**	**Avg. ESS per Gradient**
Traditional HMC	(52.18, 31.15)	40,000	0.00208
Traditional NUTS	(18.70, 12.91)	42,157	0.00075
L-HNN in HMC	(8.67, 7.89)	40,000	0.00041
L-HNN in NUTS	(20.40, 24.09)	31,213	0.00143

Table 1.2: Effective Sample Sizes (ESS) and Gradient Evaluations for nD Standard Gaussian

Method	**ESS**	**# Gradients of Target Density**	**Avg. ESS per Gradient**
Traditional HMC	(900, 900)	400,000	0.00450
Traditional NUTS	(422.78, 296.16)	105,763	0.00680
L-HNN in HMC	(5.39, 4.57)	40,000	0.00025
L-HNN in NUTS	(401.89, 337.10)	100,487	0.00735

Table 1.3: Effective Sample Sizes (ESS) and Gradient Evaluations for nD Rosenbrock

Method	**ESS**	**# Gradients of Target Density**	**Avg. ESS per Gradient**
Traditional HMC	(28.46, 66.84)	400,000	0.00024
Traditional NUTS	(169.67, 78.03)	77,363	0.00320
L-HNN in HMC	(14.19, 12.45)	400,000	0.00007
L-HNN in NUTS	(4.03, 6.94)	29,493	0.00037

Chapter 2

Part Two

ABSTRACT

This chapter presents a comprehensive exploration of Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC), a state-of-the-art Bayesian sampling methodology that synergizes Hamiltonian Monte Carlo (HMC) with pseudo-marginal techniques. Traditional MCMC methods often struggle with computational inefficiency in high-dimensional or latent variable-rich posterior distributions, particularly when exact density evaluations are intractable. The pseudo-marginal framework addresses this by employing unbiased stochastic estimators for target densities, enabling efficient exploration of complex distributions. PM-HMC extends HMC into this paradigm, leveraging gradient information to improve mixing rates while maintaining theoretical guarantees. A key innovation discussed is the integration of Hamiltonian Neural Networks (HNNs) into PM-HMC, which automates the learning of system dynamics and enhances sampling efficiency through data-driven Hamiltonian vector fields. The paper systematically reviews theoretical foundations, algorithmic implementations, and practical considerations for PM-HMC, emphasizing the advantages over classical HMC. Rigorous validation via unit and integration tests ensures the reliability of the implementation. Structured into four sections, the discussion progresses from foundational concepts (pseudo-marginal methods, importance sampling) to advanced integration strategies (HNN-enhanced PM-HMC), culminating in empirical validation protocols with `TensorFlow`. This work bridges theoretical statistics and machine learning, offering insights into scalable Bayesian inference for modern complex models.

Keywords Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC) · Hamiltonian Neural Network · Bayesian inference · *TensorFlow* Replication

2.1 Introduction

Markov Chain Monte Carlo (MCMC) methods are widely used for sampling from complex posterior distributions in Bayesian inference. While traditional MCMC methods like the Metropolis-Hastings algorithm are effective in many cases, their efficiency can degrade when the target distribution requires marginalization over latent variables or is computationally expensive to evaluate. The *pseudo-marginal* approach addresses these challenges by using unbiased estimators of the target density in place of exact calculations. Building on this foundation, the *Pseudo-Marginal Hamiltonian Monte Carlo* (PM-HMC) method extends Hamiltonian Monte Carlo (HMC) to the pseudo-marginal framework, providing a powerful tool for exploring high-dimensional and structured posterior distributions. This literature review summarizes recent developments in pseudo-marginal methods, focusing on PM-HMC and its connections to related techniques.

This document provides a clear discussion of the motivation of Pseudo-Marginal Hamiltonian Monte Carlo and how to integrate Hamiltonian Neural Network (HNN) into Pseudo-Marginal Hamiltonian Monte Carlo Sampling. The remainder of this paper is organized as follows. Section 2.2 introduces the literature of Pseudo-Marginal methods, Pseudo-Marginal Hamiltonian Monte Carlo and Importance Sampling, including the application and extensions of PM-HMC. Section 2.3 demonstrates the motivation for Pseudo-Marginal Hamiltonian Monte Carlo, including theoretical foundations and numerical integration algorithms. Section 2.4 illustrates the training of Pseudo-Marginal Hamiltonian Neural Network while shows how HNN integrate into PM-HMC sampling. Section 2.5 addresses the implementation of unit tests and integration tests.

2.2 Literature Review

Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC) represents a significant breakthrough in Bayesian computation, as it combines the pseudo-marginal framework with the momentum-based dynamics of Hamiltonian Monte Carlo (HMC). This integration offers a powerful approach for efficient sampling from complex and high-dimensional posterior distributions, particularly in scenarios where exact marginalization is computationally infeasible. Recent developments by Alenlöv et al. [2021] and Deligiannidis et al. [2018], among others, have established a solid theoretical and practical foundation for PM-HMC, demonstrating its potential in both machine learning and physical sciences. These advancements open new avenues for improving sampling efficiency, reducing estimator variance, and tackling increasingly sophisticated models.

2.2.1 Pseudo-Marginal Methods

The pseudo-marginal method was introduced to handle cases where the posterior distribution cannot be computed exactly but has an unbiased estimator available. The theoretical foundations of pseudo-marginal methods are discussed by Deligiannidis et al. [2018], who highlight the importance of controlling the variance of the estimator to ensure efficient sampling. They propose the *correlated pseudo-marginal method*, which improves computational efficiency by introducing correlation in successive evaluations of the unbiased estimator.

Schmon et al. [2021] further explore the large-sample asymptotics of the pseudo-marginal method, providing theoretical insights into its behavior for complex models. They show that the variance of the log of the estimator plays a critical role in determining the efficiency of the algorithm. These results have significant implications for the design of pseudo-marginal methods in practice.

2.2.2 Pseudo-Marginal Hamiltonian Monte Carlo

Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC) extends the pseudo-marginal framework to incorporate the momentum-based dynamics of HMC. This approach was developed by Alenlöv et al. [2021], who provide a comprehensive treatment of the method in the context of high-dimensional Bayesian inference. By leveraging Hamiltonian dynamics, PM-HMC achieves superior exploration of the posterior distribution compared to traditional pseudo-marginal methods. Alenlöv et al. [2021] also address the challenge of maintaining unbiasedness in the presence of numerical integration errors inherent in HMC.

The efficiency of PM-HMC depends on the choice of kinetic energy and the properties of the unbiased estimator. Livingstone et al. [2019] investigate the impact of kinetic energy choice in Hamiltonian Monte Carlo methods, providing guidelines that can also inform the implementation of PM-HMC. Their work emphasizes the need for careful tuning of hyperparameters to balance exploration and computational cost.

2.2.3 Importance Sampling

Importance sampling plays a critical role in pseudo-marginal methods by providing unbiased estimates of intractable posterior distributions, which are essential for ensuring the validity and efficiency of sampling algorithms. The performance of importance sampling depends heavily on the choice of proposal distribution and the ability to reduce variance in the resulting estimates.

Kloster Osmundsen et al. [2018] propose combining PM-HMC with efficient importance sampling strategies to enhance the performance of the method in challenging settings. By leveraging advanced importance sampling techniques, they demonstrate significant improvements in computational efficiency and sampler stability. This approach enables PM-HMC to handle high-dimensional problems more effectively, where poor choices of proposal distributions can otherwise lead to high variance and degraded performance.The importance of efficient proposal distributions is further emphasized by Kleppe and Liesenfeld [2014], who develop strategies for optimizing importance sampling within mixture frameworks. Their work underscores the need for adaptive techniques that dynamically refine the proposal distribution to match the target posterior more closely. Such adaptations are particularly relevant in the PM-HMC context, where the interplay between importance sampling and Hamiltonian dynamics must be carefully balanced.

In addition to their application in PM-HMC, importance sampling techniques have been successfully integrated into other sampling frameworks. For example, Grothe et al. [2019] demonstrate the use of particle-efficient importance sampling in Gibbs samplers for state-space models, achieving substantial computational gains. These advances highlight the versatility and broad applicability of importance sampling in Bayesian computation. Alenlöv et al. [2021] also discuss the role of importance sampling in the pseudo-marginal context, noting that careful tuning of the proposal distribution and the variance of the importance weights is crucial for maintaining the efficiency of the PM-HMC

algorithm. Their analysis emphasizes the importance of integrating importance sampling as a core component of the pseudo-marginal framework, rather than treating it as a standalone technique.

2.2.4 Applications and Extensions

The pseudo-marginal framework has been applied to a range of problems, demonstrating its versatility and effectiveness. Gadd et al. [2021] apply pseudo-marginal Bayesian inference to Gaussian process latent variable models, highlighting its ability to deal with intractable likelihoods in complex hierarchical models. Their results underscore the potential of PM-HMC for handling non-Gaussian latent variable models.

Kloster Osmundsen et al. [2018] explore PM-HMC with efficient importance sampling, proposing strategies to enhance the performance of the method in challenging settings. By combining importance sampling with PM-HMC, they achieve significant gains in computational efficiency.

In the context of physical sciences, Vandecasteele and Samaey [2024] use pseudo-marginal methods to approximate the free energy in micro-macro systems, demonstrating the applicability of these techniques in diverse domains. Their work highlights the ability of pseudo-marginal methods to tackle problems involving complex energy landscapes and high-dimensional state spaces.

2.3 Motivation: Why Pseudo Marginal Hamiltonian Monte Carlo Sampling?

2.3.1 Limitations of Pure HMC Compared to Pseudo-Marginal and Marginal Variants

Hamiltonian Monte Carlo (HMC) is a powerful sampling method for Bayesian inference, particularly effective for high-dimensional probability distributions. However, pure HMC has notable limitations when compared to pseudo-marginal or marginal variants. Below, we explore these limitations and explain how pseudo-marginal and marginal approaches address them.

Dependence on Exact Likelihood Evaluations

Pure HMC relies on the availability of the exact log-likelihood and its gradient. For a Hamiltonian defined as:

$$H(\theta, p) = U(\theta) + K(p), \tag{2.1}$$

where the potential energy $U(\theta) = -\log p(y \mid \theta) - \log p(\theta)$ represents the negative log-posterior, and $K(p)$ denotes the kinetic energy. The Hamiltonian dynamics are governed by:

$$\frac{d\theta}{dt} = \frac{\partial H}{\partial p}, \tag{2.2}$$

$$\frac{dp}{dt} = -\frac{\partial H}{\partial \theta}. \tag{2.3}$$

Simulating these dynamics necessitates repeated evaluation of $\nabla_\theta \log p(y \mid \theta)$ during integration steps. However, numerous probabilistic models Blei et al. [2003] (e.g., latent variable models and stochastic differential equations) involve intractable marginal likelihoods $p(y \mid \theta)$ due to high-dimensional integration:

$$p(y \mid \theta) = \int p(y, \mathbf{u} \mid \theta) du \tag{2.4}$$

This intractability precludes explicit computation of $\nabla_\theta \log p(y \mid \theta)$, rendering standard Hamiltonian Monte Carlo (HMC) inapplicable for such models. The requirement for exact gradient information fundamentally limits HMC's applicability to models with closed-form likelihood expressions.

$$H(\theta, p) = U(\theta) + K(p), \tag{2.5}$$

where $U(\theta) = -\log p(y \mid \theta) - \log p(\theta)$ is the potential energy (negative log-posterior) and $K(p)$ is the kinetic energy, the dynamics are simulated using:

$$\frac{d\theta}{dt} = \frac{\partial H}{\partial p}, \tag{2.6}$$

$$\frac{dp}{dt} = -\frac{\partial H}{\partial \theta}. \tag{2.7}$$

This requires evaluating $\nabla_\theta \log p(y \mid \theta)$ at each simulation step. For many real-world problems, such as latent variable models or stochastic differential equations, the likelihood $p(y \mid \theta)$ is intractable because it involves high-dimensional integrals. The deterministic gradient dependency inherent in Pure HMC fundamentally restricts the applicability to Bayesian inverse problems involving stochastic systems, where exact differentiability proves theoretically unobtainable.

2.3.2 Foundations of Pseudo-Marginal Hamiltonian Monte Carlo

Measure-Theoretic Distinction

Pseudo-marginal methods fundamentally differ from exact likelihood approaches through their construction of surrogate measures while require unbiased measure preservation (exact invariant distribution). In addition, while exact HMC requires pointwise evaluation of the true marginal likelihood $p(y|\theta)$, pseudo-marginal HMC operates on an extended probability space where:

$$\mathbb{E}_{\mathbf{U}\sim m}[\hat{p}(y|\theta, \mathbf{U})] = p(y|\theta) \quad \forall \theta \in \Theta$$

This equality in expectation – *not* in pathwise agreement – preserves the invariant measure while permitting Monte Carlo approximations. The pseudo-marginal framework substitutes exact likelihood evaluations with unbiased stochastic estimates, trading deterministic accuracy for computational tractability in high-dimensional integration settings. Hence, to achieve these goals, the pseudo-marginal Hamiltonian Monte Carlo (PM-HMC) framework requires two foundational assumptions to reconcile intractable likelihoods with Hamiltonian dynamics:

Assumption 1 (Auxiliary Variable Representation). For parameters $\theta \in \Theta$ and auxiliary variables $\mathbf{u} \in \mathcal{U} \subseteq \mathbb{R}^D$, there exists a non-negative estimator $\hat{p}(y \mid \theta, \mathbf{u})$ and reference measure $m(\mathbf{u})$ satisfying

$$p(y \mid \theta) = \int_{\mathcal{U}} \hat{p}(y \mid \theta, \mathbf{u}) m(\mathbf{u}) d\mathbf{u}. \tag{2.8}$$

This decomposition enables probabilistic lifting of the marginal likelihood into a product space $\Theta \times \mathcal{U}$, circumventing direct evaluation of the intractable integral. The extended target distribution

$$\bar{\pi}(\theta, \mathbf{u}) \propto p(\theta)\hat{p}(y \mid \theta, \mathbf{u})m(\mathbf{u}) \tag{2.9}$$

preserves the original posterior $\pi(\theta)$ as its θ-marginal, forming the measure-theoretic basis for pseudo-marginal methods.

Assumption 2 (Differentiable Embedding). The auxiliary space satisfies $\mathcal{U} = \mathbb{R}^D$ with $m(\mathbf{u}) = \mathcal{N}(\mathbf{u}; \mathbf{0}, I_D)$, and the estimator $\hat{p}(y \mid \theta, \mathbf{u})$ is continuously differentiable in both θ and $\mathbf{u}$. The gradient operator

$$(\theta, \mathbf{u}) \mapsto \nabla_{(\theta,\mathbf{u})} \log \hat{p}(y \mid \theta, \mathbf{u})$$

must be Lipschitz continuous, guaranteeing well-posed Hamiltonian trajectories.

Hamiltonian Reformulation with Momentum Augmentation

Building on these assumptions, we construct an augmented Hamiltonian system by introducing momentum variables $\boldsymbol{\rho} \in \mathbb{R}^d$ for parameters and $\mathbf{p} \in \mathbb{R}^D$ for auxiliary variables:

$$H(\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p}) = \underbrace{-\log p(\theta) - \log \hat{p}(y \mid \theta, \mathbf{u})}_{\text{Extended Potential Energy}} + \underbrace{\frac{1}{2}\left(\|\boldsymbol{\rho}\|^2 + \|\mathbf{u}\|^2 + \|\mathbf{p}\|^2\right)}_{\text{Kinetic Energy}}. \tag{2.10}$$

The corresponding joint distribution on the phase space becomes:

$$\bar{\pi}(\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p}) \propto \exp(-H(\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p})). \tag{2.11}$$

The Hamiltonian dynamics generate flows that preserve the symplectic structure:

$$\frac{d}{dt}\begin{pmatrix}\theta\\ \boldsymbol{\rho}\\ \mathbf{u}\\ \mathbf{p}\end{pmatrix} = \begin{pmatrix}\nabla_{\boldsymbol{\rho}} H\\ -\nabla_\theta H\\ \nabla_{\mathbf{p}} H\\ -\nabla_{\mathbf{u}} H\end{pmatrix} = \begin{pmatrix}\boldsymbol{\rho}\\ \nabla_\theta \log p(\theta) + \nabla_\theta \log \hat{p}(y \mid \theta, \mathbf{u})\\ \mathbf{p}\\ -\mathbf{u} + \nabla_{\mathbf{u}} \log \hat{p}(y \mid \theta, \mathbf{u})\end{pmatrix}. \tag{2.12}$$

Foundational Implications

- **Measure-Theoretic Completeness:** Assumption 1 establishes a Radon-Nikodym derivative relationship between the extended measure $\bar{\pi}(\theta, \mathbf{u})$ and the original posterior $\pi(\theta)$. This ensures that marginal inference performed using $\bar{\pi}(\theta, \mathbf{u})$ remains valid and consistent with the original posterior $\pi(\theta)$. Essentially, Assumption 1 highlights that Pseudo-marginal methods differ fundamentally from exact likelihood approaches by constructing surrogate measures that approximate the original target distribution. Despite the inherent stochasticity in transition kernels, the pseudo-marginal approach guarantees exactness in measure, making it a rigorous method. This measure-theoretic exactness is the defining feature of Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC), setting it apart from heuristic approximations. Under standard ergodicity conditions, this property guarantees asymptotic convergence to the true posterior distribution.
- **Dynamical Feasibility:** Assumption 2 ensures that the coupled differential equations in (2.8) satisfy the Picard-Lindelöf conditions, which guarantee the existence and uniqueness of solutions. This permits the use of numerical integration techniques, particularly geometric integrators, to solve these equations with high accuracy and stability. By satisfying these conditions, the dynamics of system remain well-defined and numerically tractable throughout the sampling process.
- **Momentum Synchronization and Kinetic Energy:** The auxiliary momentum $\mathbf{p}$ and the associated quadratic kinetic energy function,

  $$K(\boldsymbol{\rho}, \mathbf{u}, \mathbf{p}) = \frac{1}{2}(\|\boldsymbol{\rho}\|^2 + \|\mathbf{u}\|^2 + \|\mathbf{p}\|^2),$$

  play critical roles in ensuring the robustness and efficiency of the framework. These components contribute to the dynamics of system in several key ways:
  - **Symplectic Volume Conservation**: The momentum variable $\mathbf{p}$ ensures that the Hamiltonian dynamics preserve symplectic volume during state transitions. This conservation guarantees that the phase space volume remains invariant, maintaining the correctness and consistency of the sampling process.
  - **Efficient Posterior Exploration**: The auxiliary variable $\boldsymbol{\rho}$ facilitates efficient exploration of the parameter space by aligning the momentum with principal curvature directions. This alignment allows the sampler to adaptively navigate regions of high curvature in the posterior geometry, improving both sampling efficiency and convergence.
  - **Kinetic Energy Properties**: The quadratic kinetic energy function $K(\boldsymbol{\rho}, \mathbf{u}, \mathbf{p})$ provides three essential properties:
    - **Separability**: Its separable structure permits explicit symplectic integration using leapfrog decomposition, ensuring numerical stability and accuracy during state transitions.
    - **Gaussian Resampling**: The quadratic form enables straightforward Gibbs sampling of the momentum variables $\mathbf{p}$, $\boldsymbol{\rho}$, and $\mathbf{u}$ from Gaussian distributions. This preserves the extended measure $\bar{\pi}(\theta, \mathbf{u})$ during the sampling process since the potential energy will not be affected by the kinetic energy sampling.
    - **Dimensional Balancing**: The kinetic energy automatically adapts the scales of the momentum variables to the geometry of the auxiliary variables. This ensures efficient exploration of the latent variable space while avoiding numerical instabilities.
  - **Regularization and Stability**: The $\|\mathbf{u}\|^2$ term in the kinetic energy function acts as a regularization mechanism, controlling the magnitude of the auxiliary variables $\mathbf{u}$ and preventing divergence during stochastic gradient updates. Similarly, the $\|\mathbf{p}\|^2$ term couples with the gradients of $\mathbf{u}$ to enable coordinated exploration of the latent space curvature. Together, these terms ensure that the dynamics remain well-behaved and efficient across complex posterior geometries.

This framework extends the geometric convergence properties of Hamiltonian Monte Carlo (HMC) to latent variable models through three interrelated mechanisms:

1. **Measure-Theoretically Valid Target Distribution:** The extended measure $\bar{\pi}(\theta, \mathbf{u})$ ensures that the sampling process is consistent with the original posterior $\pi(\theta)$.

2. **Smooth Gradient-Based Transitions:** The numerical integration of the Hamiltonian dynamics allows for smooth transitions across the parameter space, leveraging the gradient information for efficient exploration.
3. **Momentum-Adapted Exploration:** The auxiliary variables $\mathbf{u}$ and the momentum variables $\mathbf{p}$ facilitate adaptive exploration of complex posterior geometries, navigating regions of high curvature while preserving computational tractability.

Therefore, the auxiliary variables $\mathbf{u}$ serve as stochastic sufficient statistics, preserving critical information about the latent structures of the posterior distribution while maintaining the computational efficiency necessary for high-dimensional sampling tasks.

2.3.3 Estimating the Gradients

Latent Variable Model

Consider the latent variable model

$$X_k \overset{\text{i.i.d.}}{\sim} f_\theta(\cdot), \quad Y_k \mid X_k \sim g_\theta(\cdot \mid X_k), \tag{2.13}$$

where the sequence $(X_k)_{k\geq 1}$ consists of latent variables that take values in $\mathbb{R}^n$, and the sequence $(Y_k)_{k\geq 1}$ represents observed variables that take values in $\mathbb{Y}$. For any integers $i \leq j$, the notation $i : j$ is defined as the set $\{i, i+1, \ldots, j\}$. When the observations $\mathbf{Y}_{1:T} = \mathbf{y}_{1:T} = \mathbf{y}$ are given (hence, $\mathbf{Y}$ is equivalent to $\mathbf{Y}^T$), the likelihood is expressed as

$$p(\mathbf{y}_{1:T} \mid \theta) = \prod_{k=1}^{T} p(y_k \mid \theta), \tag{2.14}$$

where each term $p(y_k \mid \theta)$ is defined by the following integral:

$$p(y_k \mid \theta) = \int_{\mathbb{R}^n} f_\theta(x_k) g_\theta(y_k \mid x_k) dx_k. \tag{2.15}$$

When the integral in Equation (2.15) cannot be evaluated in closed form, the likelihood $p(\mathbf{y}_{1:T} \mid \theta)$ is analytically intractable.

Computing the Relevant Gradients

In scenarios where the likelihood is intractable, unbiased estimates can be obtained through importance sampling. An importance density $q_\theta(x_k \mid y_k)$ is introduced for the latent variable X_k. It is assumed that the latent variable X_k can be generated using the representation $X_k = \gamma_k(\theta, V)$, where $\gamma_k : \Theta \times \mathbb{R}^p \to \mathbb{R}^n$ is a deterministic mapping, and the auxiliary variable V is sampled from a multivariate normal distribution $V \sim \mathcal{N}(0_p, I_p)$. Using N samples for each index k, the likelihood can be approximated as follows:

$$\hat{p}(y_{1:T} \mid \theta, \mathbf{U}) = \prod_{k=1}^{T} \hat{p}(y_k \mid \theta, \mathbf{U}_k),$$

where the approximate likelihood for each k is defined as

$$\hat{p}(y_k \mid \theta, \mathbf{U}_k) = \frac{1}{N} \sum_{i=1}^{N} \omega_\theta(y_k, \mathbf{U}_{k,i}),$$

and the importance weight $\omega_\theta(y_k, \mathbf{U}_{k,i})$ is expressed as

$$\omega_\theta(y_k, \mathbf{U}_{k,i}) = \frac{g_\theta(y_k \mid \mathbf{X}_{k,i}) f_\theta(\mathbf{X}_{k,i})}{q_\theta(\mathbf{X}_{k,i} \mid y_k)}.$$

The variable $\mathbf{X}_{k,i}$ is computed using the deterministic mapping $\mathbf{X}_{k,i} = \gamma_k(\theta, \mathbf{U}_{k,i})$, where the auxiliary variable $\mathbf{U}_{k,i}$ is independently sampled from a multivariate normal distribution $\mathbf{U}_{k,i} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0_p, I_p)$. In this scenario , the user has the freedom to specify the importance sampling distribution such that Assumption 2 is satisfied. Proposed by the Kloster Osmundsen et al. [2018], one natural selection of the importance sampling distribution density $q_\theta(\mathbf{X}_{k,i} \mid y_k)$ is the prior distribution density $f_\theta(\mathbf{X}_{k,i})$.

In this setup, the total computational cost is proportional to $D = TNp$, where T represents the number of observed data points, N refers to the number of samples used for each index k, and p denotes the dimensionality of the parameter θ. The gradient of the approximate log-likelihood with respect to the parameter θ is given by

$$\nabla_\theta \log \hat{p}(y_{1:T} \mid \theta, \mathbf{U}) = \sum_{k=1}^{T} \sum_{i=1}^{N} \frac{\omega_\theta(y_k, \mathbf{U}_{k,i})}{\sum_{j=1}^{N} \omega_\theta(y_k, \mathbf{U}_{k,j})} \nabla_\theta \log \omega_\theta(y_k, \mathbf{U}_{k,i}), \tag{2.16}$$

and the gradient with respect to the auxiliary variable $\mathbf{u}_{k,i}$ is expressed as

$$\nabla_{\mathbf{u}_{k,i}} \log \hat{p}(y_{1:T} \mid \theta, \mathbf{U}) = \frac{\omega_\theta(y_k, \mathbf{U}_{k,i})}{\sum_{j=1}^{N} \omega_\theta(y_k, \mathbf{U}_{k,j})} \nabla_{\mathbf{u}_{k,i}} \log \omega_\theta(y_k, \mathbf{U}_{k,i}). \tag{2.17}$$

The vector $\mathbf{U}_k$ is defined as $\mathbf{U}_k := (\mathbf{U}_{k,1}, \ldots, \mathbf{U}_{k,N})$, and the importance weight $\omega_\theta(y_k, \mathbf{U}_{k,i})$ is expressed as

$$\omega_\theta(y_k, \mathbf{U}_{k,i}) = \frac{g_\theta(y_k \mid X_{k,i}) f_\theta(X_{k,i})}{q_\theta(X_{k,i} \mid y_k)},$$

where the variable $X_{k,i}$ is computed as $X_{k,i} = \gamma_k(\theta, \mathbf{U}_{k,i})$.

2.3.4 Splitting Hamiltonian Components

The challenge of solving the Hamiltonian system associated with the extended target can be addressed by employing a splitting technique that takes advantage of the structure of the extended target distribution. This approach has been explored in prior works, including Beskos et al. [2011], Neal [2012], Matthews [2015], Shahbaba et al. [2014] The key idea is to decompose the Hamiltonian H, as defined in Equation (2.10), into two components $H = A + B$, where the two components are specified as follows:

$$A(\boldsymbol{\rho}, \mathbf{u}, \mathbf{p}) = \frac{1}{2}\Big(\boldsymbol{\rho}^\top \boldsymbol{\rho} + \mathbf{u}^\top \mathbf{u} + \mathbf{p}^\top \mathbf{p}\Big), \quad B(\theta, \mathbf{u}) = -\log p(\theta) - \log \hat{p}(y_{1:T} \mid \theta, \mathbf{u}). \tag{2.18}$$

The splitting of the Hamiltonian into two components A and B facilitates analytical integration of the Hamiltonian systems associated with each component. Specifically, for the component A, the explicit solution of the dynamical system is given by the mapping $\Phi_h^A : \mathbb{R}^{2d+2D} \to \mathbb{R}^{2d+2D}$, which evolves the system for h units of time from an initial state. The solution to the system defined by A is expressed as follows:

$$\Phi_h^A : \begin{cases} \theta(h) = \theta(0) + h\boldsymbol{\rho}(0), \\ \boldsymbol{\rho}(h) = \boldsymbol{\rho}(0), \\ \mathbf{u}(h) = \mathbf{p}(0)\sin(h) + \mathbf{u}(0)\cos(h), \\ \mathbf{p}(h) = \mathbf{p}(0)\cos(h) - \mathbf{u}(0)\sin(h). \end{cases} \tag{2.19}$$

The total integration time is specified as hL, where h is the step size and L represents the number of integration steps. To approximate the solution to the original Hamiltonian system associated with the vector field $\hat{\Psi}$ in Equation (2.12), a symmetric Strang splitting technique is employed. This method, as described in Matthews [2015], is defined as follows:

$$\widehat{\Phi}_{hL} = \Big(\Phi_{h/2}^A \circ \Phi_h^B \circ \Phi_{h/2}^A\Big)^L.$$

For practical implementation, consecutive half-steps of the integration of system A are combined to enhance numerical efficiency. This approach is analogous to the standard Verlet integrator and is expressed as:

$$\widehat{\Phi}_{hL} = \Phi_h^A/2 \circ \big\{\Phi_h^B \circ \Phi_h^A\big\}^{L-1} \circ \Phi_h^B \circ \Phi_h^A/2. \tag{2.20}$$

When computing the numerical integration takes into consideration, we define $\widehat{\Theta}[\ell]$ as the full parameter vector associated with the PM-HMC algorithm after ℓ steps of the integrator. We let $\widehat{\Theta}[0]$ be the initial values of the full parameter vector. Taking one step of the integrator $\widehat{\Phi}_h = \Phi_h^A \circ \Phi_h^B \circ \Phi_h^A$ offers us the updating scheme from $\widehat{\Theta}[\ell] = (\widehat{\theta}[\ell], \widehat{\rho}[\ell], \widehat{\mathbf{u}}[\ell], \widehat{\mathbf{p}}[\ell])$ to $\widehat{\Theta}[\ell+1]$ through the following equations:

$$\widehat{\theta}[\ell+1] = \widehat{\theta}[\ell] + h\widehat{\rho}[\ell] + \frac{h^2}{2}\nabla_\theta\Big\{\log p(\theta) + \log \hat{p}(y \mid \theta, \widehat{\mathbf{p}}[\ell]\sin\left(\frac{h}{2}\right) + \widehat{\mathbf{u}}[\ell]\cos\left(\frac{h}{2}\right))\Big\}\Big|_{\theta=\widehat{\theta}[\ell]+\frac{h}{2}\widehat{\rho}[\ell]}, \tag{2.21}$$

$$\widehat{\rho}[\ell+1] = \widehat{\rho}[\ell] + h\nabla_\theta\Big\{\log p(\theta) + \log \hat{p}(y \mid \theta, \widehat{\mathbf{p}}[\ell]\sin\left(\frac{h}{2}\right) + \widehat{\mathbf{u}}[\ell]\cos\left(\frac{h}{2}\right))\Big\}\Big|_{\theta=\widehat{\theta}[\ell]+\frac{h}{2}\widehat{\rho}[\ell]}, \tag{2.22}$$

$$\widehat{\mathbf{u}}[\ell+1] = \widehat{\mathbf{p}}[\ell]\sin(h) + \widehat{\mathbf{u}}[\ell]\cos(h) + \sin\left(\frac{h}{2}\right)h\nabla_{\mathbf{u}}\log\hat{p}(y \mid \widehat{\theta}[\ell] + \frac{h}{2}\widehat{\rho}[\ell], \mathbf{u})\Big|_{\mathbf{u}=\widehat{\mathbf{p}}[\ell]\sin\left(\frac{h}{2}\right)+\widehat{\mathbf{u}}[\ell]\cos\left(\frac{h}{2}\right)}, \tag{2.23}$$

$$\widehat{\mathbf{p}}[\ell+1] = \widehat{\mathbf{p}}[\ell]\cos(h) - \widehat{\mathbf{u}}[\ell]\sin(h) + \cos\left(\frac{h}{2}\right)h\nabla_{\mathbf{u}}\log\hat{p}(y \mid \widehat{\theta}[\ell] + \frac{h}{2}\widehat{\rho}[\ell], \mathbf{u})\Big|_{\mathbf{u}=\widehat{\mathbf{p}}[\ell]\sin\left(\frac{h}{2}\right)+\widehat{\mathbf{u}}[\ell]\cos\left(\frac{h}{2}\right)}. \tag{2.24}$$

Hence, we can perform Pseudo-marginal Hamiltonian Monte Carlo Sampling through algorithm 6 iteratively.

Algorithm 6 Pseudo-marginal HMC (Single Iteration)

1: Current Markov chain state: (θ, u)
2: **Execute:**
Sample auxiliary variables:

$$\boldsymbol{\rho} \sim \mathcal{N}(\mathbf{0}_d, I_d),$$
$$\mathbf{p} \sim \mathcal{N}(\mathbf{0}_D, I_D)$$

Compute new state via numerical integrator (See equation (2.21)-(2.24)):

$$(\theta', \boldsymbol{\rho}', u', \mathbf{p}') = \hat{\Phi}_{hL}(\theta, \boldsymbol{\rho}, u, \mathbf{p})$$

Accept new state with probability:

$$\min\{1,\ \exp\left(-\left[H(\theta, \boldsymbol{\rho}, u, \mathbf{p}) - H(\theta', \boldsymbol{\rho}', u', \mathbf{p}')\right]\right)\}$$

2.4 Perform the Pseudo-marginal Hamiltonian Neural Network

The implementation framework of Hamiltonian Neural Networks (HNNs) was systematically detailed in Chapter 1. This section serves two primary objectives: (1) to justify the adoption of Pseudo-Marginal Hamiltonian Neural Networks (PM-HNNs) as a superior alternative to Pseudo-Marginal Hamiltonian Monte Carlo (PM-HMC) for high-dimensional Bayesian sampling, and (2) to present a comprehensive procedural workflow for training PM-HNNs, including their unique operational mechanics.

2.4.1 Why use the Pseudo-marginal Hamiltonian Neural Network?

This complexity separation emerges from three fundamental mechanisms: Let n denote the sample size and $\tilde{D}$ the overall parameter dimension. Three fundamental aspects shows that the Pseudo-marginal Hamiltonian Neural Network(PM-HNN) is superior:

1. **Computing Complexity(at least)**: Traditional PM-HMC requires processing all N observations for gradient calculation:

$$\text{PM-HMC Complexity} = \underbrace{n}_{\substack{\text{Data}\\ \text{size}}} \times \underbrace{C * \tilde{D}^2}_{\substack{\text{Matrix}\\ \text{inversion}}} \tag{2.25}$$

PM-HNN decouples complexity through neural parameterization:

$$\text{PM-HNN Complexity} = \underbrace{C_{\text{net}}}_{\substack{\text{Network}\\ \text{operations}}} \quad (\text{independent of } n) \tag{2.26}$$

2. **Amortized Inference Architecture**: PM-HNN decouples data processing into distinct phases:

$$\underbrace{\text{Offline Learning}}_{\substack{\text{Data-dependent}\\ \text{(one-time cost)}}} \Rightarrow \underbrace{\text{Online Sampling}}_{\substack{\text{Data-agnostic}\\ \text{(constant per-iteration cost)}}}$$

Subsequent sampling iterations require only forward propagation through the pre-trained Hamiltonian network $H_\phi(\theta, \rho)$, eliminating recurrent data access.

3. **Stochastic Training Protocol**: During offline phase, mini-batch stochastic optimization:

$$\min_{\phi} \mathbb{E}_{\mathcal{B}\sim\mathcal{D}} \left[\|\nabla_\theta H_\phi - \frac{\partial}{\partial\theta} \log p(\mathcal{B}|\theta)\|^2 \right], \quad |\mathcal{B}| \ll n$$

enables full data distribution capture without persistent bulk data storage requirements.

The computational invariance to observational data scale n positions PM-HNN as superior for modern large-n Bayesian inference tasks, whereas PM-HMC remains constrained by linear data complexity scaling. The computational superiority of PM-HNN over conventional PM-HMC becomes particularly pronounced with large-scale datasets. While PM-HMC requires full-data scanning during every iteration - causing prohibitive computational costs when processing million-scale samples ($n > 10^6$) - PM-HNN employs a strategic two-phase approach:

- **Knowledge Acquisition Phase**: Trains neural networks using randomized mini-batches (e.g., 1% data subsamples) to learn parameter dynamics patterns. Cost Profile: Data-dependent but single upfront investment Operational Phase
- **Operational Phase**: Executes parameter updates through neural network inferences, completely decoupled from original observational data. Cost Profile: Determined solely by network architecture This paradigm shift transforms PM-HNN from a data-bound algorithm into architecture-bound computation, achieving orders-of-magnitude efficiency gains for big data applications while maintaining theoretical rigor.

2.4.2 Workflow of Using PM-HNN in PM-HMC Sampling

The procedure of PM-HNN, namely Extended HNN, in PM-HMC closely resembles the procedure of HNN in HMC. Figure 2.3 demonstrates the meta workflow of using PM-HNN in PM-HMC sampling. The key steps in the workflow are summarized as follows:

- **Data Generation**: Begin by generating the training data and observed data y according to the specified setup. This step ensures the availability of data required to construct the sample paths and train the PM-HNN.
- **Training Sample Path Construction**: Construct the sample path following the procedure described in Section 2.3. This involves integrating the Hamiltonian dynamics using the method outlined in Algorithm 6. The integration is performed over a series of time steps to simulate the dynamics of the extended Hamiltonian system.
- **Extended HNN Training**: Train the extended HNN following the procedure in Algorithm 8. The training process incorporates the potential term, the kinetic term, and two auxiliary terms as inputs to the Hamiltonian neural network. This step enables the model to learn the underlying dynamics of the system.
- **Sampling the path**: The initial value of θ is set to zero, while $\boldsymbol{\rho}$, $\mathbf{u}$, and $\mathbf{p}$ are initialized by sampling from a standard Gaussian distribution. This initialization reflects the stochastic nature of the system and provides a starting point for the simulation. After initialization, use the trained model to compute the time derivatives of the system. Subsequently, generate the sample path using the leapfrog integration method described in Algorithm 7. This step ensures accurate simulation of the extended Hamiltonian dynamics.

This workflow highlights the integration of PM-HNN into the PM-HMC framework, leveraging both data-driven learning and Hamiltonian dynamics to perform efficient sampling.

2.5 Unit test and Integration Test Implementation

This section outlines the unit tests and integration tests conducted for the replication code implemented using TensorFlow. It is crucial to highlight that, regardless of the type of testing—whether unit testing, integration testing, or manual testing—the configuration files must be updated beforehand to incorporate any required parameter changes. Ensuring the configuration files are up-to-date guarantees consistency across all testing scenarios and prevents discrepancies arising from outdated or incorrect settings. All the tests in this section should be run under the directories `phnn`.

2.5.1 Manual Test

To manually run the replicated functions, please perform:

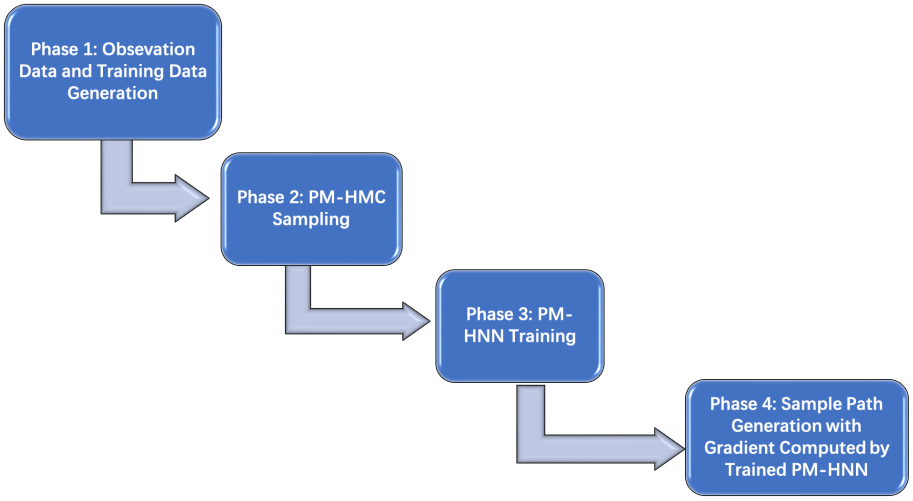


Figure 2.1: The meta workflow for PM-HNN in PM-HMC Sampling

Algorithm 7 Leapfrog Integration for $\{\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p}\}$

Hamiltonian: $H_{\boldsymbol{\Theta}}(\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p})$
Initial conditions: $\{\theta(0), \boldsymbol{\rho}(0), \mathbf{u}(0), \mathbf{p}(0)\}$
Step size: Δt; **Number of steps:** N
for $j = 0$ to $N - 1$ **do**
    Compute the current time step as $t = j\Delta t$
    Step 1: Update $\boldsymbol{\rho}$ and $\mathbf{p}$ (half step):

$$\boldsymbol{\rho}(t + \frac{\Delta t}{2}) = \boldsymbol{\rho}(t) + \frac{\Delta t}{2} \nabla_\theta \Big(\log p(\theta) + \log \hat{p}(y \mid \theta, \mathbf{u}(t)) \Big),$$
$$\mathbf{p}(t + \frac{\Delta t}{2}) = \mathbf{p}(t) + \frac{\Delta t}{2} \nabla_{\mathbf{u}} \log \hat{p}(y \mid \theta(t), \mathbf{u}(t)).$$

    Step 2: Update θ and $\mathbf{u}$ (full step):

$$\theta(t + \Delta t) = \theta(t) + \Delta t \boldsymbol{\rho}(t + \frac{\Delta t}{2}),$$
$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \Delta t \mathbf{p}(t + \frac{\Delta t}{2}).$$

    Step 3: Update $\boldsymbol{\rho}$ and $\mathbf{p}$ (another half step):

$$\boldsymbol{\rho}(t + \Delta t) = \boldsymbol{\rho}(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2} \nabla_\theta \Big(\log p(\theta) + \log \hat{p}(y \mid \theta, \mathbf{u}(t + \Delta t)) \Big),$$
$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2} \nabla_{\mathbf{u}} \log \hat{p}(y \mid \theta(t + \Delta t), \mathbf{u}(t + \Delta t)).$$

end for

Algorithm 8 Pseudo-Marginal Hamiltonian neural network training

PM-HNN parameters: $\boldsymbol{\Theta}$; **Training data:** $\{\theta, \boldsymbol{\rho}, \mathbf{u}, \mathbf{p}\}$
Evaluate gradients of the training data $\{\frac{\Delta\theta}{\Delta t}, \frac{\Delta\boldsymbol{\rho}}{\Delta t}, \frac{\Delta\mathbf{u}}{\Delta t}, \frac{\Delta\mathbf{p}}{\Delta t}\}$
Initialize HNN parameters $\boldsymbol{\Theta}$
Compute HNN Hamiltonian $H_{\boldsymbol{\Theta}}$
Compute $H_{\boldsymbol{\Theta}}$ gradients $\{\frac{\partial H_{\boldsymbol{\Theta}}}{\partial \boldsymbol{\rho}}, \frac{\partial H_{\boldsymbol{\Theta}}}{\partial \theta}, \frac{\partial H_{\boldsymbol{\Theta}}}{\partial \mathbf{p}}, \frac{\partial H_{\boldsymbol{\Theta}}}{\partial \mathbf{u}}\}$
Compute loss

$$\mathcal{L} = \left\| \frac{\partial H_{\boldsymbol{\Theta}}}{\partial \boldsymbol{\rho}} - \frac{\Delta\theta}{\Delta t} \right\| + \left\| -\frac{\partial H_{\boldsymbol{\Theta}}}{\partial \theta} - \frac{\Delta\boldsymbol{\rho}}{\Delta t} \right\| + \left\| \frac{\partial H_{\boldsymbol{\Theta}}}{\partial \mathbf{p}} - \frac{\Delta\mathbf{u}}{\Delta t} \right\| + \left\| -\frac{\partial H_{\boldsymbol{\Theta}}}{\partial \mathbf{u}} - \frac{\Delta\mathbf{p}}{\Delta t} \right\|$$

Minimize $\mathcal{L}$ with respect to $\boldsymbol{\Theta}$

Algorithm 9 Pseudo-Marginal Hamiltonian neural networks evaluation in leapfrog integration)

Hamiltonian: H; **Initial conditions:** $\mathbf{z}(0) = \{\theta(0), \boldsymbol{\rho}(0), \mathbf{u}(0), \mathbf{p}(0)\}$; **Dimensions:** d; **Steps:** N; **End time:** T
$\Delta t = \frac{T}{N}$
for $j = 0 : N - 1$ **do**
    $t = j\Delta t$
    Compute HNN output gradients: $\{\frac{\partial H}{\partial \theta(t)}, \quad \frac{\partial H}{\partial \boldsymbol{\rho}(t)}, \quad \frac{\partial H}{\partial \mathbf{u}(t)}, \quad \frac{\partial H}{\partial \mathbf{p}(t)}\}$
    for $i = 1 : d$ **do**
        Update $\theta_i(t + \Delta t)$:

$$\theta_i(t + \Delta t) = \theta_i(t) + \Delta t \boldsymbol{\rho}_i(t) - \frac{\Delta t^2}{2} \frac{\partial H}{\partial \theta_i(t)}$$

        Update $\mathbf{u}_i(t + \Delta t)$:

$$\mathbf{u}_i(t + \Delta t) = \mathbf{u}_i(t) + \Delta t \mathbf{p}_i(t) - \frac{\Delta t^2}{2} \frac{\partial H}{\partial \mathbf{u}_i(t)}$$

    end for
    Recompute HNN output gradients: $\{\frac{\partial H}{\partial \theta(t+\Delta t)}, \quad \frac{\partial H}{\partial \mathbf{u}(t+\Delta t)}\}$
    for $i = 1 : d$ **do**
        Update $\boldsymbol{\rho}_i(t + \Delta t)$:

$$\boldsymbol{\rho}_i(t + \Delta t) = \boldsymbol{\rho}_i(t) - \frac{\Delta t}{2} \left(\frac{\partial H}{\partial \theta_i(t)} + \frac{\partial H}{\partial \theta_i(t + \Delta t)} \right)$$

        Update $\mathbf{p}_i(t + \Delta t)$:

$$\mathbf{p}_i(t + \Delta t) = \mathbf{p}_i(t) - \frac{\Delta t}{2} \left(\frac{\partial H}{\partial \mathbf{u}_i(t)} + \frac{\partial H}{\partial \mathbf{u}_i(t + \Delta t)} \right)$$

    end for
end for

- For training phase, you can run the following command to test `PHNN_train.py`

  `python PHNN\_train.py`

- For sampling phase, you can run the following command to test `phnn_hmc.py`

  `python phnn\_hmc.py`

2.5.2 Integration Test

Alternatively, integration tests can be directly conducted in `phnn` respective directories using the following command:

```
python -m unittest test_integration.py -k test_workflow
```

2.5.3 Unit Test

For training phase, you can run the following command to test `PHNN_train.py`

```
python -m unittest test_integration.py -k test_train
```

For sampling phase, you can run the following command to test `phnn_hmc.py`

```
python -m unittest test_integration.py -k test_sampling
```

In addition, direct path can also used for both unit test and integration test:

```
# Integration test
python -m unittest test_integration.TestIntegration.test_workflow

# Training unit test
python -m unittest test_integration.TestIntegration.test_train

# Sampling unit test
python -m unittest test_integration.TestIntegration.test_sampling
```

2.5.4 Updating Configuration Files

If you want to customize the parameters, try

```
python script.py --json_config config.json --num_samples 2000 --lambda1 20.0
```

where `script.py` is the file you want to run, while the priorities of parameter assignment operations in descending order are `command line`, `json file` and `default parameters`. Specifically, to replicate the results of Section 4.3 in Alenlöv et al. [2021]. Try

```
    python -m unittest test_integration.py -k test_workflow -- --model glmm --N 7000
```

or try:

```
    python script.py  --model glmm --N 7000
```

To construct `config.json`, please run

```
    python construct_json.py
```

Key Features:

- `-k` option: Filters tests using substring matching
- Direct addressing: Enables precise test case selection
- Execution order: Sampling test **must** follow training test

2.6 Experiment Results for Part 2.e

The original experiment was planned for **7000 iterations**, but due to the **large Hamiltonian network and memory constraints (no GPU)**, only **1000 iterations** were completed. Despite this limitation, efforts were made to **replicate the original study**, and command-line instructions were provided in the previous section to reproduce the results.

The convergence behavior of the parameters is analyzed as follows:

The plot of $\frac{1}{\lambda_1}$ and $\frac{1}{\lambda_2}$ shows a **clear decreasing trend**, suggesting that λ_1 and λ_2 are approaching convergence. The red dashed line at $y = 0.1$ and the green dashed line at $y = 0.33$ represent expected values, but the curves have not fully stabilized at these levels. Given more iterations, they may eventually reach convergence.

In contrast, the plot of μ_1 and μ_2 indicates a **slow upward trend**, but both parameters are far from a stable state. The blue dashed line at $y = 0$ and the orange dashed line at $y = 3$ serve as reference points, showing that μ_1 and μ_2 have not yet reached their expected values. Since only 1000 iterations were performed, μ remains uncertain and may require further iterations to assess its convergence.

Conclusion: While λ appears to be approaching convergence, μ remains uncertain due to the limited number of iterations. The lack of GPU resources has prevented the completion of 7000 iterations, but efforts have been made to replicate the original study.

Next Steps:

- Further iterations are required to confirm the convergence of λ and μ.
- Computational optimizations (e.g., reducing batch size, using more efficient optimization algorithms) should be explored.
- Additional computing resources (e.g., cloud GPU servers) could help complete the full 7000 iterations.

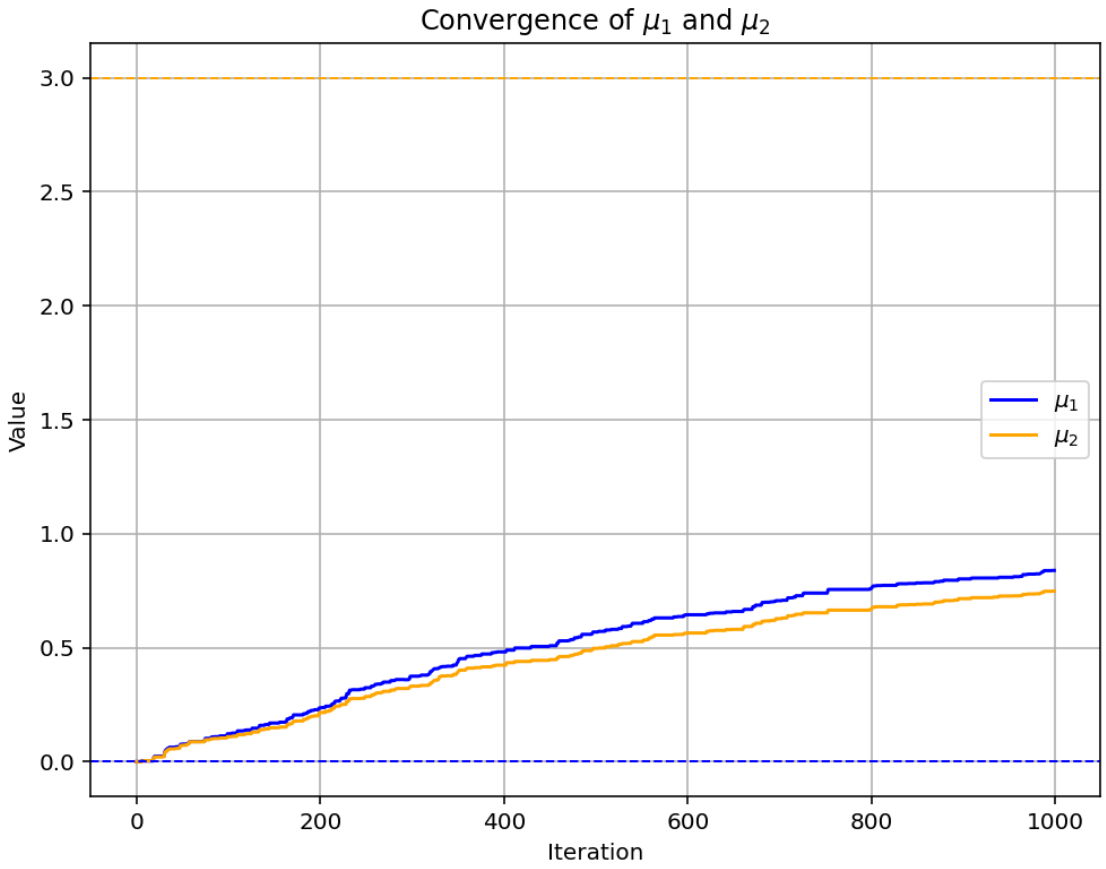


Figure 2.2: Convergence of μ Parameters

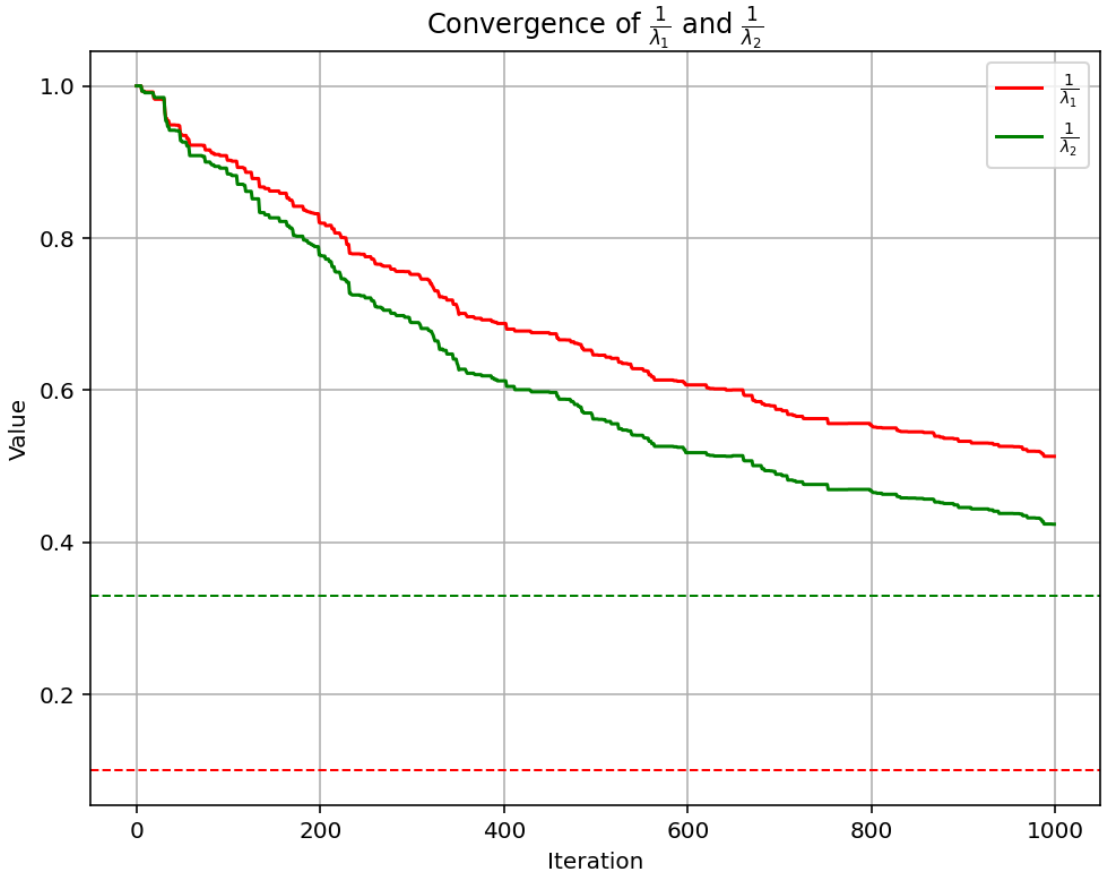


Figure 2.3: Convergence of Inverse λ Parameters

Chapter 3

Empirical Study: Bayesian Inference for VaR and CVaR Using HMC

ABSTRACT

This chapter assumes the reader has a fundamental understanding of VaR estimation and backtesting. We conduct a simple empirical analysis to explore the suitability of Bayesian HMC methods for estimating VaR and CVaR under different return distribution assumptions. The study compares Bayesian and non-Bayesian approaches, highlighting the strengths and limitations of each across different time horizons and asset classes.Results indicate that non-Bayesian methods generally underestimate risk, leading to higher exceedance ratios, while Bayesian methods, particularly those employing HMC sampling, produce more conservative VaR estimates. The Parametric t-Distribution performs better than the Normal distribution but still struggles to model extreme losses effectively. Bayesian GEV, with consistently zero exceedance ratios, demonstrates superior tail risk capture.For short-term horizons (1–3 months), market fluctuations are driven by short-term noise, liquidity changes, and market sentiment. Non-Bayesian Parametric t-Distribution responds quickly to market changes, providing a balance between accuracy and capital efficiency. In contrast, Bayesian methods tend to be overly conservative, leading to inefficient capital allocation. Over longer horizons (6–12 months), economic cycles, policy shifts, and systemic risks become dominant, increasing the probability of extreme events. Bayesian GEV effectively incorporates historical data and prior distributions to capture long-term tail risks, making it more suitable for extended periods. Overall, this chapter explores potential financial applications of Bayesian HMC methods for VaR and CVaR estimation.

Keywords Hamiltonian Monte Carlo (HMC) · Value at Risk(VaR) · Conditional Value at Risk(CVaR) · Bayesian inference · Backtesting

3.1 Introduction

In this chapter, we conduct an empirical analysis of Value-at-Risk (VaR) and Conditional Value-at-Risk (CVaR) estimation using Bayesian inference with Hamiltonian Monte Carlo (HMC). Our study compares seven different methods for estimating the 95% VaR and CVaR:

- Three Bayesian inference models using different distributional assumptions:
  - Normal distribution
  - Student-t distribution
  - Generalized Extreme Value (GEV) distribution
- Three parametric estimation methods.
- Historical simulation.

The estimation period spans from 2007 to 2019, while the out-of-sample backtesting period covers 2020 to 2025. The dataset comprises monthly total USD returns of four broad asset class indices:

- Merrill Lynch Corporate Master index (Bonds)
- S&P Goldman Sachs Commodity index (Commodities)
- S&P 500 index (Equities)
- National Association of Real Estate Investment Trusts Index (NAREIT) (Real Estate)

The data are obtained from Wind.

3.2 Bayesian Models for VaR and CVaR Estimation

In Bayesian inference, we define a joint posterior distribution over model parameters and use HMC to sample from this posterior. Below, we describe the Bayesian models corresponding to different return distributional assumptions.

3.2.1 Normal Distribution Model

We assume that returns r_t follow a normal distribution:

$$r_t \sim \mathcal{N}(\mu, \sigma^2) \tag{3.1}$$

where μ is the mean return and σ is the standard deviation. The Bayesian prior distributions for the parameters are:

$$\mu \sim \mathcal{N}(0, 0.1^2), \quad \sigma \sim \text{HalfNormal}(0.1) \tag{3.2}$$

The joint log-probability function is given by:

$$\log p(\mu, \sigma | r) = \log p(\mu) + \log p(\sigma) + \sum_t \log p(r_t | \mu, \sigma) \tag{3.3}$$

where

$$p(r_t | \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(r_t - \mu)^2}{2\sigma^2}\right) \tag{3.4}$$

3.2.2 Student-*t* Distribution Model

To account for heavy tails in financial returns, we assume a Student-*t* distribution:

$$r_t \sim t_\nu(\mu, \sigma) \tag{3.5}$$

where ν (degrees of freedom) controls the heaviness of the tails. The priors are:

$$\mu \sim \mathcal{N}(0, 0.1^2), \quad \sigma \sim \text{HalfNormal}(0.1), \quad \nu \sim \text{Exponential}(1) \tag{3.6}$$

The joint log-probability function is:

$$\log p(\mu, \sigma, \nu | r) = \log p(\mu) + \log p(\sigma) + \log p(\nu) + \sum_t \log p(r_t | \mu, \sigma, \nu) \tag{3.7}$$

where the likelihood function is:

$$p(r_t | \mu, \sigma, \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{\pi\nu}\sigma} \left(1 + \frac{(r_t - \mu)^2}{\nu\sigma^2}\right)^{-\frac{\nu+1}{2}} \tag{3.8}$$

3.2.3 Generalized Extreme Value (GEV) Distribution Model

To better capture extreme tail risks, we assume a GEV distribution:

$$r_t \sim \text{GEV}(\mu, \sigma, \xi) \tag{3.9}$$

where ξ (shape parameter) controls the tail behavior. The priors are:

$$\mu \sim \mathcal{N}(0, 0.1^2), \quad \sigma \sim \text{HalfNormal}(0.1), \quad \xi \sim \mathcal{N}(0, 0.1^2) \tag{3.10}$$

The joint log-probability function is:

$$\log p(\mu, \sigma, \xi | r) = \log p(\mu) + \log p(\sigma) + \log p(\xi) + \sum_t \log p(r_t | \mu, \sigma, \xi) \tag{3.11}$$

where the likelihood function is:

$$p(r_t | \mu, \sigma, \xi) = \frac{1}{\sigma} \left(1 + \xi \frac{r_t - \mu}{\sigma}\right)^{-\frac{1}{\xi} - 1} \exp\left(-\left(1 + \xi \frac{r_t - \mu}{\sigma}\right)^{-\frac{1}{\xi}}\right), \quad \text{if } \xi \neq 0 \tag{3.12}$$

When $\xi = 0$, the GEV distribution reduces to the Gumbel distribution:

$$p(r_t | \mu, \sigma) = \frac{1}{\sigma} \exp\left(-\frac{r_t - \mu}{\sigma} - \exp\left(-\frac{r_t - \mu}{\sigma}\right)\right) \tag{3.13}$$

3.2.4 Conclusion

In this section, we have formulated Bayesian inference models for estimating VaR and CVaR using HMC under different return distribution assumptions. These models allow us to incorporate prior beliefs and provide uncertainty quantification for risk measures. In the next section, we will present the empirical results and compare the performance of different estimation methods.

3.3 Empirical Analysis

As shown from Figure A.1 to A.8 and Table A.1 to A.4, our results shows that:

- Non-Bayesian methods generally underestimate risk, resulting in higher exceedance ratios.
- Bayesian methods with HMC sampling tend to estimate higher VaR values, making them more conservative.
- Parametric t-Distribution performs better than Normal, but both fail to model extreme losses effectively.
- Bayesian GEV, whose exceedance ratios always equal zero, consistently estimates the lowest VaR and CVaR, indicating better tail risk capture.

3.4 Performance Across Different Asset Classes and Horizons

3.4.1 Bonds

For bond assets, non-Bayesian methods generally underestimate risk, as evidenced by the high exceedance ratios observed in Historical, Parametric Normal, and t-Distribution methods.

- Short-term horizons (1-month and 3-month):
  - Non-Bayesian Parametric t-Distribution is preferable since Bayesian methods (especially Bayesian GEV) tend to be overly conservative, leading to 0% exceedance ratio.
  - Historical VaR underestimates risk, with exceedance ratios reaching 17-20%, much higher than expected.
- Long-term horizons (6-month and 12-month):

  - Bayesian methods (Bayesian GEV and Bayesian t-Distribution) perform better as they provide a more conservative risk estimate, avoiding excessive exceedances observed in non-Bayesian methods.
  - Parametric t-Distribution still underestimates risk, while Bayesian GEV ensures a safer risk measure with near-zero exceedance.

3.4.2 Commodities

For commodities, non-Bayesian methods (Historical and Parametric Normal) consistently underestimate risk, while Bayesian methods provide more stable and conservative estimates.

- Short-term horizons (1-month and 3-month):
  - Non-Bayesian methods (Parametric Normal and t-Distribution) perform adequately, with exceedance ratios around 4-8%, which is reasonable.
  - Bayesian Normal and Bayesian t-Distribution are overly conservative, leading to very low exceedance ratios.
- Long-term horizons (6-month and 12-month):
  - Bayesian GEV is the best choice, capturing extreme risks more effectively, while non-Bayesian methods continue to underestimate risk.
  - The exceedance ratio for Bayesian GEV is higher (16-23%), indicating a more realistic estimation of extreme losses.

3.4.3 Equities

Equity returns exhibit higher volatility and more extreme losses, making Bayesian methods generally more suitable.

- Short-term horizons (1-month and 3-month):
  - Bayesian t-Distribution provides the best balance between conservatism and accuracy, while non-Bayesian methods tend to underestimate risks.
  - Historical and Parametric t-Distribution methods show exceedance ratios above 8-9%, indicating risk underestimation.
- Long-term horizons (6-month and 12-month):
  - Bayesian GEV is the most conservative, with exceedance ratios close to zero, effectively capturing extreme risks.
  - Non-Bayesian methods (Parametric Normal and t-Distribution) continue to underestimate risk, making them less suitable.

3.4.4 REITs (Real Estate Investment Trusts)

REITs exhibit moderate risk, but non-Bayesian methods often underestimate extreme losses.

- Short-term horizons (1-month and 3-month):
  - Non-Bayesian methods (Parametric Normal and t-Distribution) perform adequately, with moderate exceedance ratios.
  - Bayesian methods exhibit similar performance, but Bayesian GEV method have very low exceedance ratio.
- Long-term horizons (6-month and 12-month):
  - Bayesian GEV provides the most reliable risk estimate, capturing extreme tail risks effectively.
  - Non-Bayesian methods (Normal and t-Distribution) underestimate risks, making them unsuitable for long-term forecasts.

3.5 Final Recommendations

Risk estimation objectives vary depending on the time horizon. For short-term VaR (1-month and 3-month), the primary goal is to balance accuracy and efficiency while avoiding excessive conservatism, which may lead to unnecessary capital allocation. In contrast, for long-term VaR (6-month and 12-month), it is more critical to adopt conservative models to ensure sufficient risk coverage, as extreme losses are more likely to occur over extended periods.

1. Short-term VaR (1-month and 3-month):
    - The primary objective is to balance precision and conservatism while maintaining sensitivity to recent market changes.
    - Non-Bayesian t-Distribution is the most balanced option for most asset classes except bond while Non-Bayesian GEV is suitable for bond VaR estimation.
    - Bayesian methods tend to be overly conservative in the short term, especially for Bonds and Commodities.
    - Equities benefit from Bayesian t-Distribution, as it better captures volatility.
2. Long-term VaR (6-month and 12-month):
    - The key objective is to ensure robust risk estimation by prioritizing conservatism, as extreme losses become more significant over longer horizons.
    - Bayesian GEV provides the most accurate tail risk estimation for all asset classes.
    - Non-Bayesian methods (especially Normal and t-Distribution) underestimate long-term risks, contributing to less suitable estimation.

Therefore, for short-term risk estimation (1-3 months), non-Bayesian t-Distribution is preferred, while for long-term risk estimation (6-12 months), Bayesian GEV is the most reliable choice.

3.6 Discussion: Comparison of Short-Term and Long-Term VaR Estimation

In the short term (1–3 months), market fluctuations primarily result from short-term noise, liquidity changes, and shifts in market sentiment. Non-Bayesian methods, particularly the Parametric t-Distribution, demonstrate rapid responsiveness to market changes, generating VaR estimates that effectively reflect recent data. In contrast, Bayesian methods, which incorporate prior information, exhibit excessive conservatism, producing inflated VaR estimates that lead to inefficient capital allocation. For short-term horizons, the Parametric t-Distribution within the non-Bayesian framework offers a preferable balance between accuracy and capital efficiency.

Over longer horizons (6–12 months), market risks increasingly stem from economic cycles, policy shifts, and systemic risks, with extreme events becoming more probable. Bayesian methods, especially Bayesian GEV, integrate historical data and prior distributions to enhance the capture of long-term tail risks. The GEV distribution, explicitly designed for modeling extreme losses, provides a more suitable approach for long-term VaR estimation. In contrast, non-Bayesian approaches, including the t-Distribution, systematically underestimate tail risks over extended periods, reducing their effectiveness in long-term risk assessment.

Overall, non-Bayesian Parametric t-Distribution remains optimal for short-term VaR estimation, while Bayesian GEV enhances long-term risk assessment by ensuring both accuracy and robustness in managing extreme risks.

Chapter 4

Self-Assessment and Future Plans

4.1 Extent of Code Conversion to TensorFlow Probability

After receiving feedback and making necessary modifications, I have successfully converted almost all of the original code into TensorFlow Probability (TFP) in Part One of this report. In Part Two, most of the pseudo-Hamiltonian network training and parts of the sampling process have been implemented using TFP. While some minor components still rely on other frameworks, the overall transformation has been largely completed.

4.2 Challenges Encountered During the Conversion

The primary difficulty I faced was my prior experience with PyTorch during my Ph.D. research, as I was more accustomed to PyTorch than TensorFlow. This lack of familiarity led to the following challenges:

- **Understanding TensorFlow's Computational Graph:** Unlike PyTorch, TensorFlow requires explicit management of computational graphs, and I encountered issues with `tape.gradient()` due to incorrect `watch()` placement, which resulted in gradients not being computed.
- **Differences in Neural Network Initialization:** TensorFlow initializes neural networks differently from PyTorch, which required additional debugging to match results.
- **Platform-Specific Issues:** When using `tfp.nuts` for sampling, I found that on macOS M1/M2 systems, failing to include `with tf.device('/cpu:0')` resulted in incorrect random number generation. This was a TensorFlow Probability-specific issue that took significant time to resolve.
- **Memory Limitations and Lack of GPU Access:** Due to the unavailability of a GPU (as the university's GPUs have been inaccessible for the past year), managing memory efficiently became a challenge. This constraint prevented me from fully reproducing all results from the original paper, though I attempted to address the key issues to the best of my ability.
- **Lack of Official Code Implementation:** The research paper did not provide an official codebase, and some implementation details were unclear. Since this topic is not my primary research focus, I had to invest additional effort in understanding and reconstructing the methodology.

Despite these challenges, I have made substantial progress in adapting to TFP and implementing the required functionalities.

4.3 Short-Term Improvement Plan

In the short term, I am committed to strengthening my coding practices by enforcing strict modularization, unit testing, and parameter management across all my projects, including both past and ongoing research works. This structured approach has already proven effective, and I will continue to refine my workflow.

- **Enforcing Modularization and Parameter Management:** To ensure reusability and maintainability, I will systematically modularize all model components and adopt a standardized parameter management system.

- **Expanding Unit Testing Coverage:** I will integrate rigorous unit tests into all critical components of my implementations to improve reliability.
- **Implementing TensorFlow Versions Alongside PyTorch:** Whenever developing AI models in PyTorch, I will concurrently implement and validate a TensorFlow version to enhance my proficiency with both frameworks.
- **Optimizing GPU Utilization for Large-Scale Data Processing:** If GPU resources become available, I will focus on optimizing computational efficiency, particularly for handling large datasets that require significant processing power.

Timeline: Moving forward, I will progressively refine my implementations, focusing on enforcing best coding practices while improving computational performance. The exact schedule will be aligned with my research progress and computational resource availability.

4.4 Long-Term Plan for Improving Coding Skills

Beyond the short-term improvements, I aim to integrate best coding practices into my long-term research workflow while maintaining a strong focus on financial AI applications. My plan includes:

- **Participating in Kaggle Competitions and Open-Source Projects:** If research time permits, I will engage in Kaggle competitions to gain hands-on experience with state-of-the-art AI techniques. Additionally, I will contribute to open-source projects, particularly in areas related to TensorFlow Probability and financial AI.
- **Open-Sourcing My Own Implementations:** For models that I have developed and published, I will consider open-sourcing my implementations. This will not only help others but also encourage me to maintain high-quality, well-documented code.
- **Learning from and Discussing Large-Scale AI Models:** I will study well-structured AI projects on GitHub, analyze how large-scale AI models are organized, and engage in discussions with researchers and engineers working on such models, including direct interactions with the models themselves.

Timeline: Over time, I will integrate these practices into my research workflow, ensuring a balance between coding proficiency and high-impact research contributions.

4.5 Integration of Research and Technical Development

As I advance in my Ph.D. research and internship projects, my approach to technical development will remain closely aligned with my academic and professional objectives. While coding proficiency is essential, my primary focus is on developing and applying advanced AI models to address real-world challenges in financial markets and investment practice, leveraging my interdisciplinary expertise in mathematics, finance, and AI.

By adhering to best coding practices—such as writing modular, well-tested, and scalable code—I will naturally refine my technical skills while prioritizing financial market-driven problem-solving. Rather than viewing coding as an isolated skill, I integrate it as a fundamental component of my broader quantitative research methodology for financial applications.

Additionally, implementing AI models in both PyTorch and TensorFlow will deepen my understanding of different frameworks and enhance my adaptability to various computational environments. This dual-framework approach ensures a seamless transition between academic research and industry applications, particularly in algorithmic trading, risk management, and portfolio optimization.

In the long term, my goal is to contribute to financial AI research and investment strategies by developing models that are both theoretically robust and practically applicable. A structured coding practice will support this objective by promoting reproducibility, scalability, and collaboration. By continuously refining my workflow and actively engaging with the AI research community, I will ensure that my technical development remains not only relevant and impactful but also directly beneficial to financial decision-making and investment practice.

Chapter 5

Response to the Questions of Interviews in TFP Part

ABSTRACT

This chapter presents my updated understanding of TensorFlow Probability (TFP), shaped by an interview with Jodie Lu and a follow-up email that included five key questions. Four of these were discussed in detail during the interview, while the fourth question—regarding the Leapfrog Integrator—was not covered. Accordingly, **the improvements in this chapter focus on the four questions that were actually addressed**.Following the interview, I revisited relevant documentation and literature, and reflected carefully on the feedback provided. Rather than applying superficial fixes, I re-examined the logic and structure of my original implementation, identified specific shortcomings, and rewrote key components with an emphasis on clarity and correctness. Although computational limitations remain and prevent full replication of the original experiments, this revised version reflects a clearer structure and a deeper understanding of the core ideas. I hope this demonstrates a willingness to confront weaknesses directly and a commitment to thoughtful, continuous improvement.

5.1 What are the built-in Python data structures? Could you explain how different data structures are used in the *tensorflow_probability.mcmc* package, and why these design choices are made?

In the recommended book *Python Object-Oriented Programming*, several built-in Python data structures are discussed, including tuples, named tuples, dictionaries, lists, sets, and different types of queues. After studying these structures, I realized that using **named tuples** in the `tensorflow_probability.mcmc` package—especially in the `trace_fn` functions—is a preferable choice over dictionaries or lists.

This is because **named tuples are immutable and hashable**, which makes them safer and more efficient when used in functional-style code. In contrast, dictionaries and lists are mutable and unhashable. This mutability means that their contents can be changed after creation, which is undesirable in the context of MCMC sampling. For example, in a Markov chain, the correctness of the algorithm depends on the fact that the previous state is fixed and cannot be altered after transitioning to the next state. Using dictionaries or lists might inadvertently allow such modification, leading to subtle bugs.

In `trace_fn`, the purpose is often to record and return intermediate kernel results during the sampling process. Using a `namedtuple` here provides the following advantages:

- **Immutability**: Ensures that the trace data or kernel results from previous steps are not accidentally modified, which is crucial for maintaining the correctness and reproducibility of the sampling process. The previous sampling information may be accidentally changed when dictionaries are used in the iteration.
- **Hashability**: Enables the results to be used in caching, memoization, or as dictionary keys—which is essential in some advanced sampling workflows.
- **Clarity and readability**: Named fields like `acceptance_rate` and `log_accept_ratio` improve code readability and reduce the likelihood of errors compared to index-based access in lists.

Therefore, in the context of TensorFlow Probability's MCMC framework, adopting named tuples in `trace_fn` functions aligns well with the functional programming style of the library and protects against unintended side effects caused by mutable data structures.

In the implementation of the custom transition kernel `HNN_HMC_Kernel`, which subclasses `tfp.mcmc.TransitionKernel`, I previously returned a list to store intermediate results such as acceptance rate and log acceptance ratio. However, this approach is not ideal because lists are mutable and do not offer clear field names.As a result, **instead of returning a list as in the previous version, I modified the implementation to return a** `namedtuple` **for better readability, immutability, and compatibility.**The key lines from both the original and the modified code are shown below for comparison.

Original version:

```
def one_step():
     return [next_q, next_p], HNNKernelResults(
                accept=accept,
                log_accept_ratio=log_accept_ratio,
                grad_evals=total_grad_evals
            )
 def bootstrap_results(self, init_state):
        chains = init_state[0].shape[0]
        return [tf.zeros([1,chains], dtype=tf.bool)]
```

Modified version:

```
def one_step():
        return [next_q, next_p], HNNKernelResults(
            accept=accept,
            log_accept_ratio=log_accept_ratio,
            grad_evals=total_grad_evals
        )

 def bootstrap_results(self, init_state):
        chains = init_state[0].shape[0]
        return HNNKernelResults(
            accept=tf.zeros([chains], dtype=tf.bool),
            log_accept_ratio=tf.zeros([chains], dtype=tf.float32),
            grad_evals=tf.zeros([], dtype=tf.int32)
        )
```

Definition of `KernelResults`:

```
from collections import namedtuple

HNNKernelResults = namedtuple("HNNKernelResults", [
    "accept",
    "log_accept_ratio",
    "grad_evals"
])
```

Another advantage of using a well-structured `namedtuple` in the `trace_fn` is that it allows for easy access to multiple diagnostic metrics during or after sampling. Each component of the tuple can be clearly named and directly accessed, making downstream analysis and visualization much more convenient.

For example, if the `KernelResults` namedtuple includes a Boolean tensor `accept` indicating whether each proposal was accepted, we can easily compute the average acceptance rate per chain:

```
accept_ratio_per_chain = tf.reduce_mean(
    tf.cast(kernel_results.accept, tf.float32), axis=0)
```

This level of clarity and modularity would be harder to achieve using a list, where accessing elements by index can be error-prone and less readable. By contrast, the use of named fields such as `accept` or `log_accept_ratio` improves both the maintainability and interpretability of the code.

5.2 How does TensorFlow Probability (TFP) handle distribution shapes? What changes could you make to your library to align with TFP's design?

TensorFlow Probability (TFP) distinguishes between three important shape concepts when working with probability distributions: **sample shape**, **batch shape**, and **event shape**. Understanding these shapes is essential for correctly constructing models and performing efficient sampling.

- **Sample shape** represents the number of i.i.d. samples drawn from the distribution. For example, drawing 1000 samples from a standard normal distribution gives a sample shape of `[1000]`.
- **Batch shape** indicates how many independent distributions are being handled in parallel. For example, a `Normal(loc=[0., 1.], scale=1.)` has `batch_shape = [2]`, representing two independent normal distributions.
- **Event shape** describes the shape of a single sample from a single distribution. For scalar distributions like Normal, the event shape is `[]`, while for multivariate distributions like `MultivariateNormalDiag`, it can be `[d]` for d-dimensional vectors.

TFP combines these shapes to construct the full shape of a sampled tensor as:

```
[sample_shape, batch_shape, event_shape]
```

This hierarchical shape structure enables TensorFlow Probability (TFP) to flexibly and efficiently model complex probabilistic systems, such as vectorized MCMC chains, batched likelihoods, and structured latent variables. By leveraging this shape semantics, TFP internally manages broadcasting and vectorized computation, allowing sampling functions (e.g., `sample()` or MCMC kernels) to automatically replicate and batch operations as needed. This design not only simplifies probabilistic programming but also ensures that models can scale effectively and execute efficiently on parallel hardware such as GPUs and TPUs.

In my original implementation, I drew samples from a single chain sequentially, which limited computation throughput and convergence diagnostics. To improve performance and scalability, I modified the code to perform **parallel sampling using multiple chains**. This was done by specifying the number of chains as part of the `current_state` batch shape and configuring the MCMC driver accordingly. For example:

```
initial_q = tf.zeros([chains, input_dim // 2], dtype=tf.float32)
initial_p = tf.random.normal([chains, input_dim // 2], dtype=tf.float32)
initial_state = [initial_q, initial_p]
samples, kernel_results = tfp.mcmc.sample_chain(
            num_results=N,
            num_burnin_steps=burn,
            current_state=initial_state,
            kernel=hmc_kernel,
            trace_fn=lambda current_state, kernel_results: kernel_results,
            seed = seed
        )
```

The way TFP manages sample shapes also facilitates the computation of effective sample sizes, which can be evaluated separately for each chain when using parallel sampling.

```
    ess_hnn_tf = tfp.mcmc.effective_sample_size(q_samples)
```

5.3 How does TFP manage randomness?

TensorFlow Probability (TFP) supports two distinct approaches to randomness: **stateful** and **stateless** random number generation.

Stateful randomness, as used in early TensorFlow APIs such as `tf.random.normal`, relies on a combination of a Python integer seed and an optional graph-level seed set via `tf.random.set_seed()`. These seeds are embedded as attributes in the TensorFlow graph node. Internally, a dedicated **C++ kernel object** is created for each such op, maintaining a persistent pseudorandom random number generator(PRNG) state that advances on each execution. This makes the behavior inherently stateful and often leads to nondeterminism, especially when used across devices or in control flow.

To overcome these limitations, TensorFlow introduced **stateless** random operations such as `tf.random.stateless_normal`. These operations are functionally pure: their outputs are entirely determined by an explicit two-element `int32` seed tensor, with no internal state. This makes them suitable for use in dynamic control flow (such as `tf.while_loop`) and for reproducible sampling across hardware platforms. TFP adopted support for stateless randomness in 2020, and it is now the default mechanism in modules such as `tfp.distributions` and `tfp.mcmc`.

In particular, the `sample_chain` function uses stateless seeding internally. Each call to the transition kernel is handled by a helper function `_seeded_one_step`, which splits the input seed using `tfp.random.split_seed()` to derive a per-step seed and a pass-along seed for the next iteration. This ensures that each sampling step is independent and deterministic:

```
step_seed, passalong_seed = tfp.random.split_seed(seed)
```

Here, `step_seed` is used to generate randomness for the current MCMC step, while `passalong_seed` is passed forward to derive seeds for future steps. This design ensures that randomness is no longer tied to global state, and allows for parallel, reproducible sampling across chains.

A key improvement in the current version is the explicit adoption of this pattern. Previously, this aspect of seed management was overlooked. In this version, we explicitly construct a two-element seed tensor and sanitize it before passing it into the sampler:

```
seed_tf = tf.constant(args.seed_tf, dtype=tf.int32)
seed = tfp.random.sanitize_seed(seed_tf)
```

This change ensures that the seed conforms to the expected shape and dtype for stateless operations. It also makes the sampling process fully reproducible and compatible with vectorized execution. By using this approach, each chain can independently derive its own sub-seed from a common global seed, without any reliance on `tf.random.set_seed()` or shared RNG state.

5.4 What are the different types of tests in Python? What are the criteria for effective tests in each category? How could you rewrite your tests to make your program more robust?

In Python, testing is commonly divided into:

- **Unit tests** – These check the behavior of individual functions or classes in isolation. They are fast, local, and help detect low-level implementation bugs early.
- **Integration tests** – These validate that components operate correctly when combined, such as ensuring that a model is correctly called within a kernel step and that trained parameters are appropriately used.

To make the test suite more robust and maintainable, we extended our unit testing to cover key functions such as `integrate_model`, `one_step`, and `leapfrog`. These tests check not only data types and tensor shapes, but also logical correctness in simplified scenarios (e.g., reversibility of velocity updates, conservation of momentum structure).

The three functions play distinct roles in the inference pipeline. `integrate_model` serves as a wrapper that calls the learned dynamics model to compute time derivatives, ensuring that the model is invoked and evaluated properly. `leapfrog` implements a symplectic integrator for numerically solving Hamiltonian dynamics, which is critical for ensuring reversibility and energy conservation. `one_step` is the core update routine in the `HNN_HMC_Kernel` class, responsible for simulating a single step of the Markov chain by sampling momentum, integrating the trajectory, and applying the Metropolis acceptance criterion.

This level of granularity helps catch subtle mathematical or implementation errors before they propagate into higher-level logic. Although not all edge cases or execution paths are currently covered, the existing tests lay a solid foundation

by validating essential logic and component interactions. Future refinements may explore property-based testing with `hypothesis` and statistical tools for evaluating long-term MCMC behavior.

To illustrate the functionality being tested, we include representative excerpts from the implementations of `integrate_model`, `leapfrog`, and `one_step` below.

Testing `one_step` function:

```
class TestHNNHMCKernel(unittest.TestCase):
    def test_one_step(self):
        num_chains = 2
        q0 = tf.random.normal([num_chains, args.input_dim // 2])
        p0 = tf.random.normal([num_chains, args.input_dim // 2])
        initial_state = [q0, p0]

        kernel = HNN_HMC_Kernel(
            hnn_model=hnn_model,
            integrate_model=fake_integrate_model,
            functions=fake_hamiltonian,
            step_size=0.1,
            args=args,
            L=1
        )

        kernel_results = kernel.bootstrap_results(initial_state)

        seed = tf.constant([123, 456], dtype=tf.int32)

        [next_q, next_p], next_results = kernel.one_step(initial_state, kernel_results, seed=seed)

        self.assertEqual(next_q.shape, (num_chains, args.input_dim // 2))
        self.assertEqual(next_p.shape, (num_chains, args.input_dim // 2))

        self.assertTrue(next_results.accept.dtype == tf.bool)
        self.assertEqual(next_results.accept.shape, (num_chains,))

        self.assertTrue(tf.reduce_all(next_results.log_accept_ratio >= -50.0))
        self.assertTrue(tf.reduce_all(next_results.log_accept_ratio <= 50.0))

        self.assertEqual(next_results.grad_evals.numpy(), 2)

        print("one_step executed successfully")
        print("Initial q:\n", q0.numpy())
        print("Next q:\n", next_q.numpy())
        print("Next p:\n", next_p.numpy())
        print("Accept:\n", next_results.accept.numpy())
        print("Log acceptance ratio:\n", next_results.log_accept_ratio.numpy())
        print("Grad evals:\n", next_results.grad_evals.numpy())
```

Testing `integrate_model` function:

```
class TestIntegrateModelTF(unittest.TestCase):
    def test_integrate_model_tf(self):
        model = DummyModel()
        t_span = [0.0, 1.0]
        y0 = tf.constant([1.0, 0.0, 0.0, 0.0], dtype=tf.float32)
        n = 5

        result = integrate_model_tf(model, t_span, y0, n, args)
```

```
        self.assertEqual(result.shape, (n + 1, args.input_dim))

        self.assertIsInstance(result, tf.Tensor)

        diffs = result[1:, 0] - result[:-1, 0]
        self.assertTrue(tf.reduce_all(diffs > 0))

        print("integrate_model_tf  test passed")
        print("Output Results:\n", result.numpy())
```

Testing `leapfrog` function:

```
class TestLeapfrogTF(unittest.TestCase):
    def test_leapfrog_simple_oscillator(self):
        # dq/dt = p, dp/dt = -q
        def dydt(t, y):
            q, p = y[0], y[1]
            return tf.convert_to_tensor([p, -q], dtype=tf.float32)

        y0 = tf.constant([1.0, 0.0], dtype=tf.float32)
        tspan = [0.0, 10.0]
        n = 1000
        dim = 2

        result = leapfrog_tf(dydt, tspan, y0, n, dim)

        self.assertEqual(result.shape, (n + 1, dim))

        self.assertIsInstance(result, tf.Tensor)

        q_vals = result[:, 0]
        p_vals = result[:, 1]
        energy = 0.5 * (q_vals**2 + p_vals**2)

        energy_diff = tf.reduce_max(tf.abs(energy - energy[0]))
        self.assertLess(energy_diff.numpy(), 1e-2)

        print("Leapfrog_tf test passed")
        print("Initial Energy:", energy[0].numpy())
        print("Maximum Energy Difference:", energy_diff.numpy())
```

5.5 Summary

1. **What are all the questions to be addressed?**
   - What are the built-in Python data structures? Could you explain how different data structures are used in the `tensorflow_probability.mcmc` package, and why these design choices are made?
   - How does TensorFlow Probability (TFP) handle distribution shapes? What changes could you make to your library to align with TFP's design?
   - How does TFP manage randomness?
   - What are the different types of tests in Python? What are the criteria for effective tests in each category? How could you rewrite your tests to make your program more robust?
   - What is the function of the Leapfrog Integrator in HMC, and how is it implemented?
2. **What are all the questions that I have remedied in the submission?**
   - The use of built-in Python data structures was addressed by analyzing the advantages of `namedtuple` in the `trace_fn`, and updating the implementation accordingly.
   - The handling of distribution shapes in TFP was explained, and the code was revised to support batch sampling with multiple chains.

- The management of randomness in TFP was explored in detail, and the code was updated to use stateless seeds and `tfp.random.sanitize_seed()` for reproducibility.
- The structure and purpose of unit and integration tests were clarified, and concrete tests for `integrate_model`, `leapfrog`, and `one_step` were added with assertions on shape, data type, and logical consistency.

3. **What are ones that are still to be addressed?**
   - The question regarding the Leapfrog Integrator was not covered in the original interview and remains only partially addressed. Although the updated tests verify its behavior in a simplified oscillator system, its implementation as a class rather than a function in the HMC framework could be further expanded upon in future revisions.
   - Other aspects, including writing effective unit tests, also require more hands-on experience and refinement. As this is my first time integrating such tests into a project of this scale, I recognize that my current implementation still has room for improvement.

The modifications made in this revised version were based either on direct discussions with Ms. Lu during the interview, or on my own reflections after studying the official TensorFlow Probability documentation more carefully. All changes aim to improve correctness, efficiency, or maintainability of the codebase.

Specifically, the following points were discussed during the interview and have been directly reflected in this updated submission:

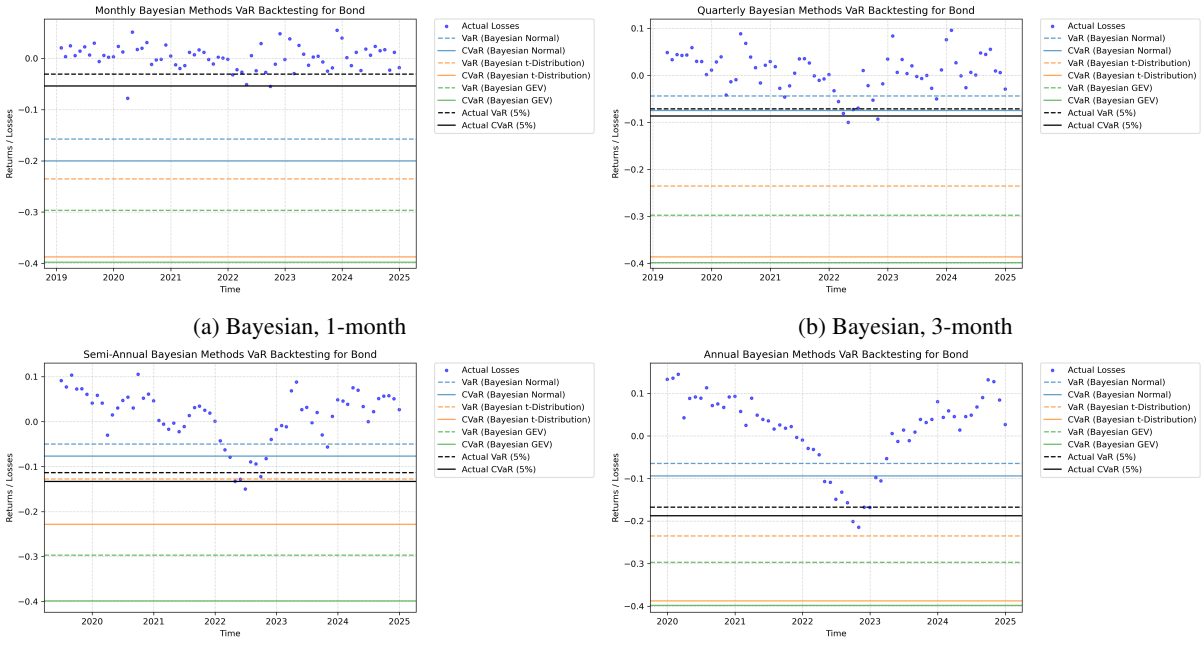
1. Ms. Lu pointed out the issue of using `namedtuple` in the `trace_fn` argument of `sample_chain`. In my original submission, I mistakenly returned a list. I have now corrected this by introducing a properly structured `HNNKernelResults` namedtuple.
2. Our discussion about TFP shape conventions inspired me to revise the sampling interface to support parallel chains. In the improved version, I explicitly reshape the initial state to include the chain dimension, enabling batch-wise sampling.
3. I initially overlooked the importance of stateless random number generation. This was raised during the interview, and I have since adopted `tfp.random.sanitize_seed()` to ensure deterministic, reproducible, and parallel-safe randomness in all sampling routines.
4. Ms. Lu emphasized that unit tests should not merely test training or sampling outcomes, but should verify the logic and I/O behavior of individual components. In response, I rewrote the test suite to include standalone tests for key functions such as `integrate_model`, `leapfrog`, and `one_step`, using minimal working examples to isolate and validate core logic.

In the previous sections, I have provided detailed explanations of what was changed, why the changes were made, and how they align with the core principles of TensorFlow Probability. I have made a conscious effort to address every issue raised by Ms. Lu during the interview, either through direct code revisions or by clarifying my understanding in writing. These revisions demonstrate that I have taken Ms. Lu's feedback seriously, and that I am capable of identifying root causes, consulting relevant documentation, and implementing effective improvements. This process also reflects my ability to learn through discussion with team members and to apply that learning independently, which I believe is essential for long-term growth and collaboration in a technical environment.

Appendix A

Some Tables and Figures

A.1 Figures



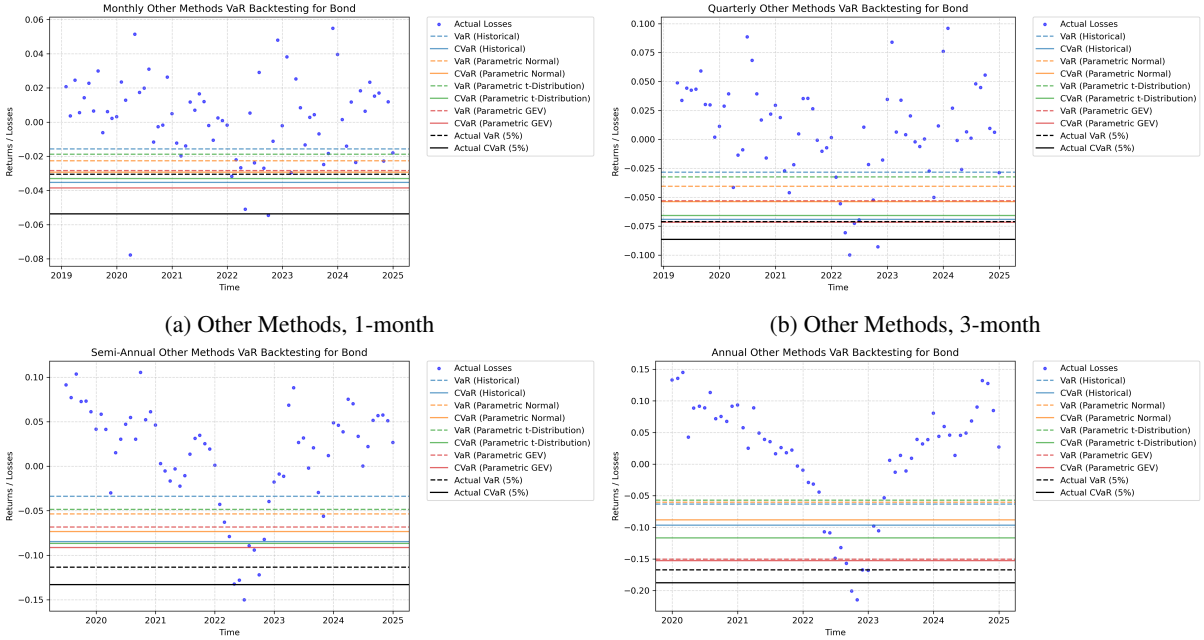
(a) Bayesian, 1-month

(b) Bayesian, 3-month

(c) Bayesian, 6-month

(d) Bayesian, 12-month

Figure A.1: Bayesian VaR and CVaR Estimates for Bond Index



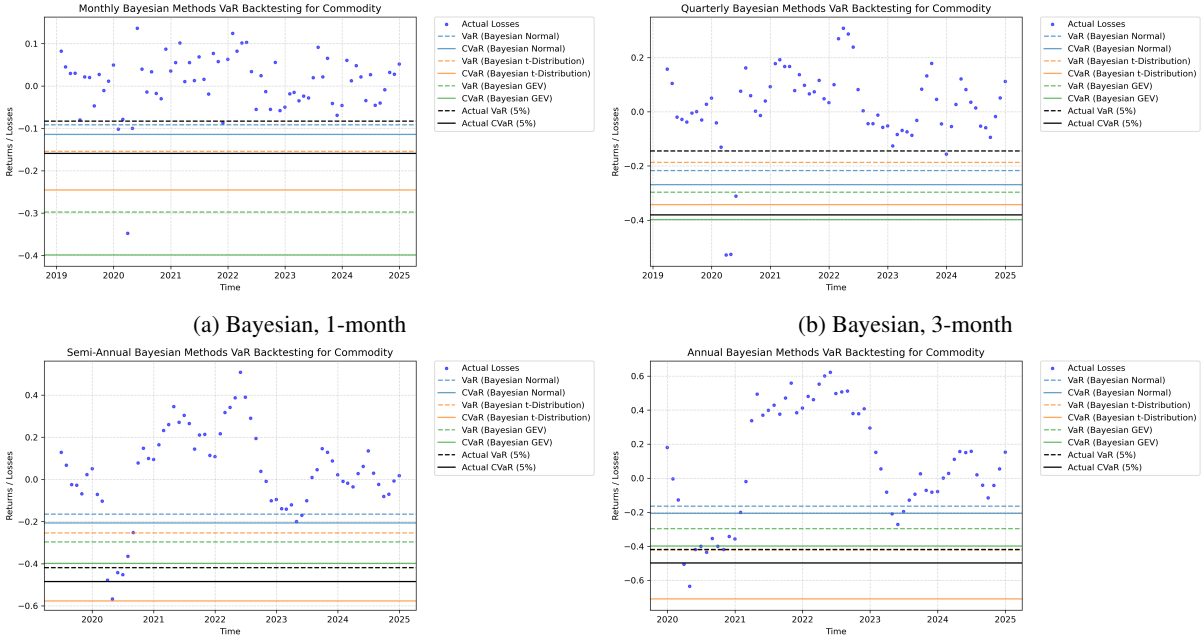
(a) Other Methods, 1-month

(b) Other Methods, 3-month

(c) Other Methods, 6-month

(d) Other Methods, 12-month

Figure A.2: Other Methods' VaR and CVaR Estimates for Bond Index



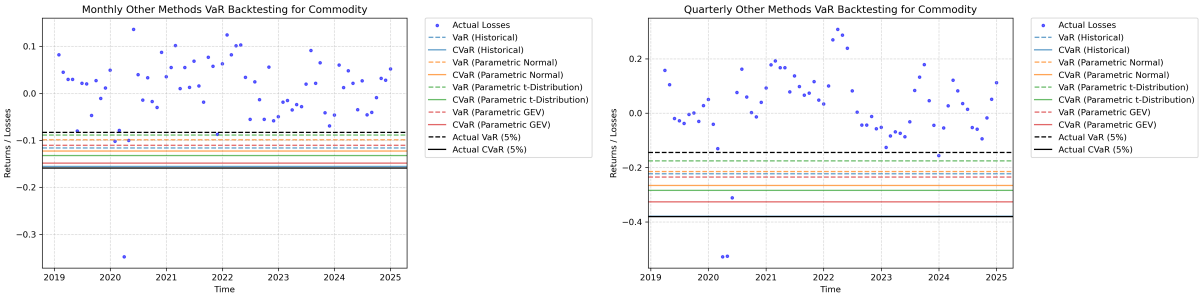
(a) Bayesian, 1-month

(b) Bayesian, 3-month

(c) Bayesian, 6-month

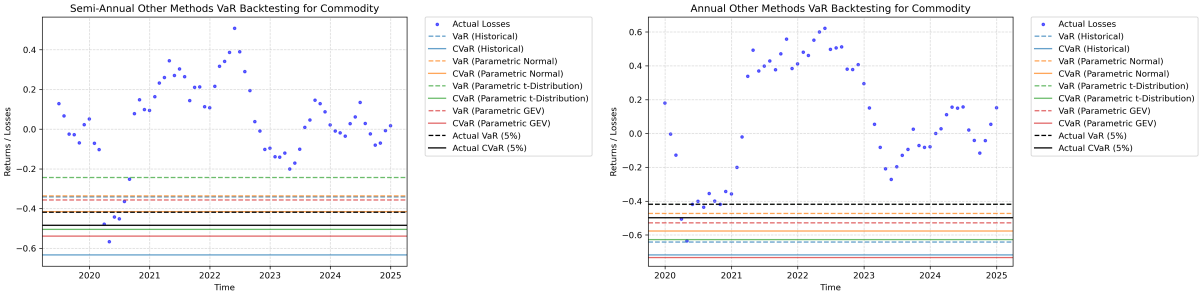
(d) Bayesian, 12-month

Figure A.3: Bayesian VaR and CVaR Estimates for Commodity Index



(a) Other Methods, 1-month

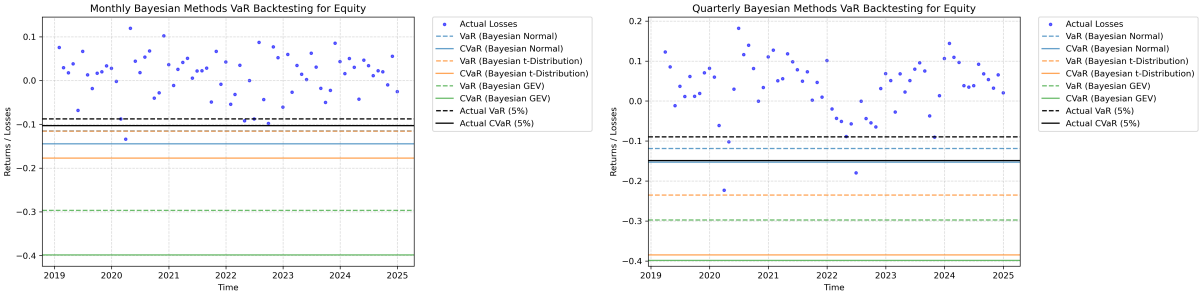
(b) Other Methods, 3-month



(c) Other Methods, 6-month

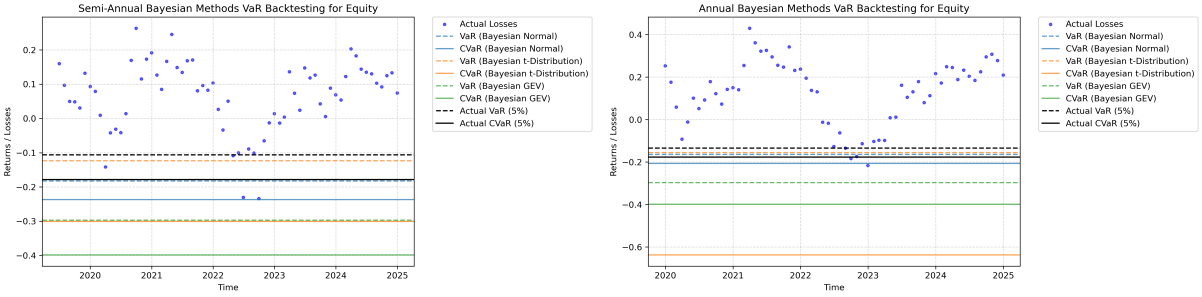
(d) Other Methods, 12-month

Figure A.4: Other Methods' VaR and CVaR Estimates for Commodity Index



(a) Bayesian, 1-month

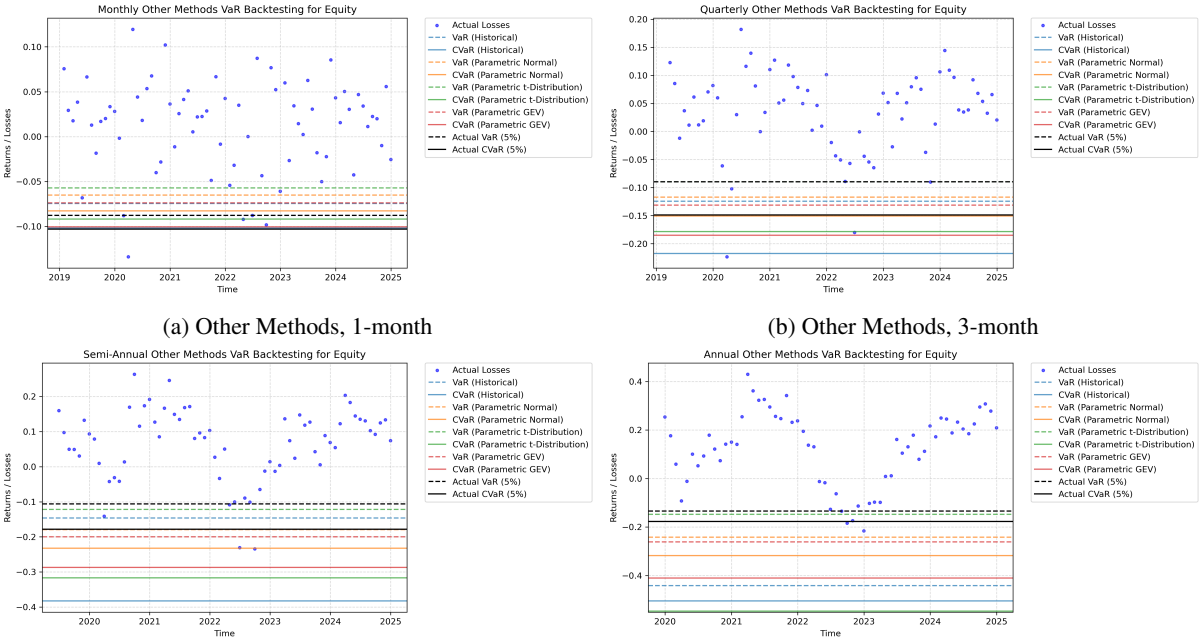
(b) Bayesian, 3-month



(c) Bayesian, 6-month

(d) Bayesian, 12-month

Figure A.5: Bayesian VaR and CVaR Estimates for Equity Index



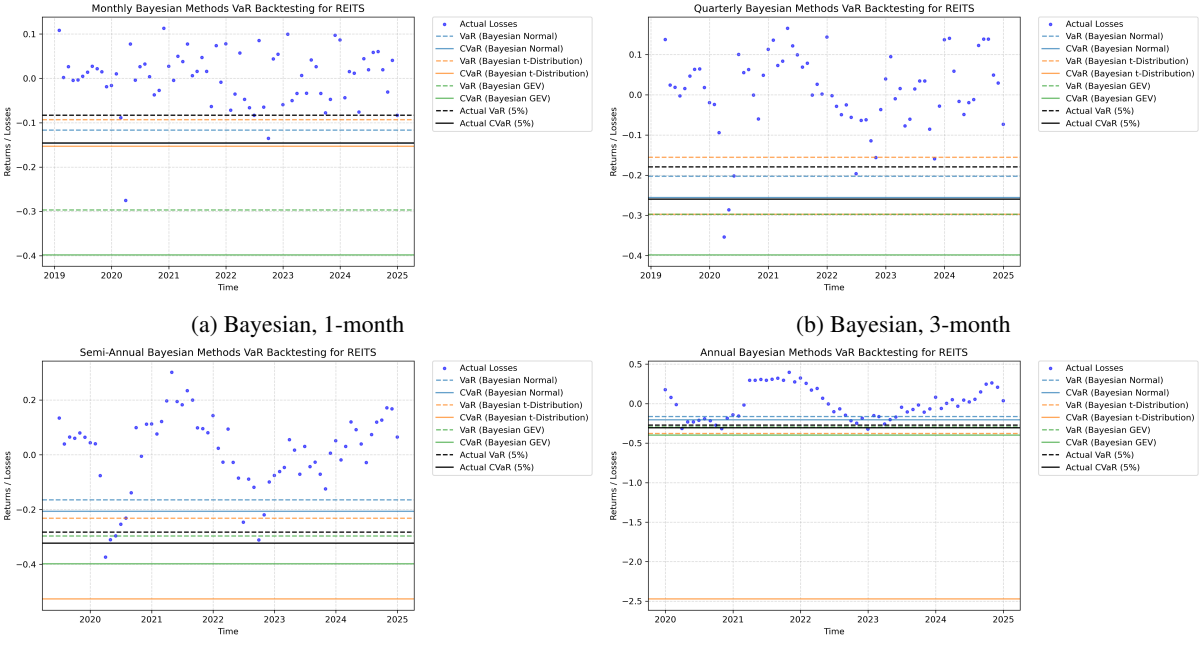
(a) Other Methods, 1-month

(b) Other Methods, 3-month

(c) Other Methods, 6-month

(d) Other Methods, 12-month

Figure A.6: Other Methods' VaR and CVaR Estimates for Equity Index



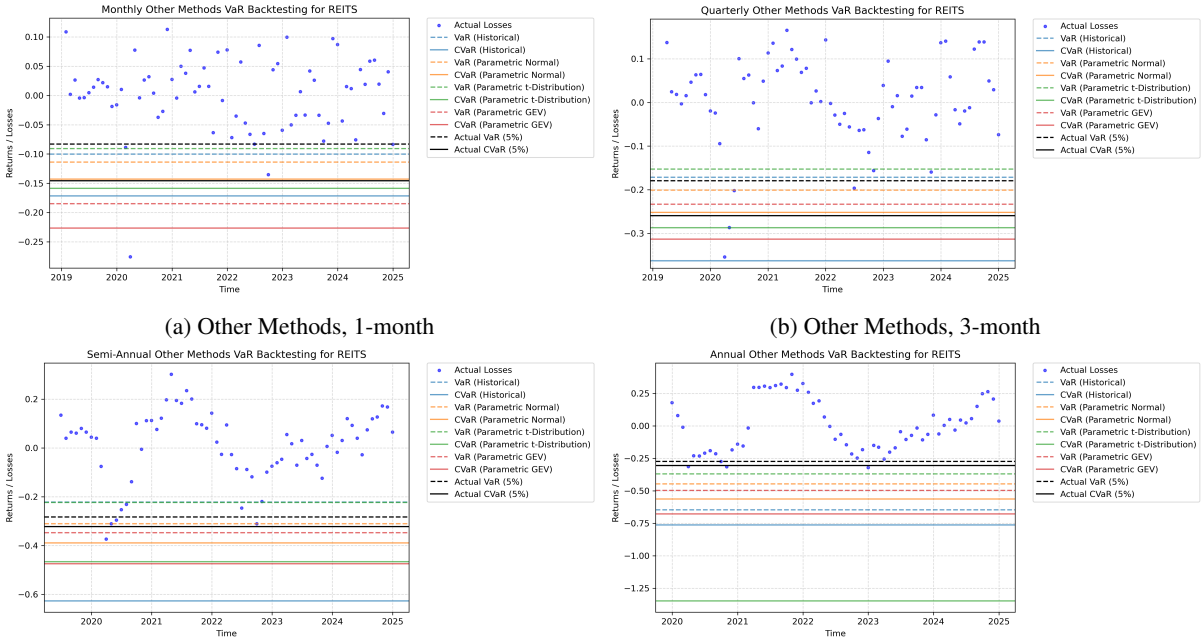
(a) Bayesian, 1-month

(b) Bayesian, 3-month

(c) Bayesian, 6-month

(d) Bayesian, 12-month

Figure A.7: Bayesian VaR and CVaR Estimates for REITs Index



(a) Other Methods, 1-month

(b) Other Methods, 3-month

(c) Other Methods, 6-month

(d) Other Methods, 12-month

Figure A.8: Other Methods' VaR and CVaR Estimates for REITS Index

A.2 Tables

Table A.1: Monthly VaR Backtesting Results

Method	Asset Class	Exceedance Ratio	Kupiec p-value	Christoffersen p-value
Bayesian GEV	Bond	0.000000	1.000000	1.000000
	Commodity	0.013889	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000
Bayesian Normal	Bond	0.000000	1.000000	1.000000
	Commodity	0.041667	1.000000	1.000000
	Equity	0.013889	1.000000	1.000000
	REITS	0.027778	1.000000	1.000000
Bayesian t-Distribution	Bond	0.000000	1.000000	1.000000
	Commodity	0.013889	1.000000	1.000000
	Equity	0.013889	1.000000	1.000000
	REITS	0.027778	1.000000	1.000000
Historical	Bond	0.208333	1.000000	1.000000
	Commodity	0.013889	1.000000	1.000000
	Equity	0.069444	1.000000	1.000000
	REITS	0.027778	1.000000	1.000000
Parametric GEV	Bond	0.069444	1.000000	1.000000
	Commodity	0.013889	1.000000	1.000000
	Equity	0.069444	1.000000	1.000000
	REITS	0.013889	1.000000	1.000000
Parametric Normal	Bond	0.152778	1.000000	1.000000
	Commodity	0.041667	1.000000	1.000000
	Equity	0.083333	1.000000	1.000000
	REITS	0.027778	1.000000	1.000000
Parametric t-Distribution	Bond	0.180556	1.000000	1.000000
	Commodity	0.041667	1.000000	1.000000
	Equity	0.097222	1.000000	1.000000
	REITS	0.027778	1.000000	1.000000

Table A.2: Quarterly VaR Backtesting Results

Method	Asset Class	Exceedance Ratio	Kupiec p-value	Christoffersen p-value
Bayesian GEV	Bond	0.000000	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.014286	1.000000	1.000000
Bayesian Normal	Bond	0.128571	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.028571	1.000000	1.000000
	REITS	0.028571	1.000000	1.000000
Bayesian t-Distribution	Bond	0.000000	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.085714	1.000000	1.000000
Historical	Bond	0.171429	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.028571	1.000000	1.000000
	REITS	0.057143	1.000000	1.000000
Parametric GEV	Bond	0.085714	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.028571	1.000000	1.000000
	REITS	0.028571	1.000000	1.000000
Parametric Normal	Bond	0.142857	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.028571	1.000000	1.000000
	REITS	0.042857	1.000000	1.000000
Parametric t-Distribution	Bond	0.157143	1.000000	1.000000
	Commodity	0.042857	1.000000	1.000000
	Equity	0.057143	1.000000	1.000000
	REITS	0.085714	1.000000	1.000000

Table A.3: Semi-Annual VaR Backtesting Result

Method	Asset Class	Exceedance Ratio	Kupiec p-value	Christoffersen p-value
Bayesian GEV	Bond	0.000000	1.000000	1.000000
	Commodity	0.074627	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.044776	1.000000	1.000000
Bayesian Normal	Bond	0.149254	1.000000	1.000000
	Commodity	0.119403	1.000000	1.000000
	Equity	0.029851	1.000000	1.000000
	REITS	0.119403	1.000000	1.000000
Bayesian t-Distribution	Bond	0.044776	1.000000	1.000000
	Commodity	0.074627	1.000000	1.000000
	Equity	0.044776	1.000000	1.000000
	REITS	0.089552	1.000000	1.000000
Historical	Bond	0.179104	1.000000	1.000000
	Commodity	0.074627	1.000000	1.000000
	Equity	0.029851	1.000000	1.000000
	REITS	0.104478	1.000000	1.000000
Parametric GEV	Bond	0.119403	1.000000	1.000000
	Commodity	0.074627	1.000000	1.000000
	Equity	0.029851	1.000000	1.000000
	REITS	0.014925	1.000000	1.000000
Parametric Normal	Bond	0.149254	1.000000	1.000000
	Commodity	0.074627	1.000000	1.000000
	Equity	0.029851	1.000000	1.000000
	REITS	0.014925	1.000000	1.000000
Parametric t-Distribution	Bond	0.149254	1.000000	1.000000
	Commodity	0.089552	1.000000	1.000000
	Equity	0.044776	1.000000	1.000000
	REITS	0.104478	1.000000	1.000000

Table A.4: Annual VaR Backtesting Results

Method	Asset Class	Exceedance Ratio	Kupiec p-value	Christoffersen p-value
Bayesian GEV	Bond	0.000000	1.000000	1.000000
	Commodity	0.163934	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.049180	1.000000	1.000000
Bayesian Normal	Bond	0.180328	1.000000	1.000000
	Commodity	0.229508	1.000000	1.000000
	Equity	0.049180	1.000000	1.000000
	REITS	0.278689	1.000000	1.000000
Bayesian t-Distribution	Bond	0.000000	1.000000	1.000000
	Commodity	0.049180	1.000000	1.000000
	Equity	0.049180	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000
Historical	Bond	0.180328	1.000000	1.000000
	Commodity	0.000000	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000
Parametric GEV	Bond	0.081967	1.000000	1.000000
	Commodity	0.016393	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000
Parametric Normal	Bond	0.180328	1.000000	1.000000
	Commodity	0.032787	1.000000	1.000000
	Equity	0.000000	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000
Parametric t-Distribution	Bond	0.180328	1.000000	1.000000
	Commodity	0.081967	1.000000	1.000000
	Equity	0.049180	1.000000	1.000000
	REITS	0.000000	1.000000	1.000000

Table A.5: Monthly CVaR Error Analysis

Method	Asset Class	Absolute Error	MSE
Bayesian GEV	Commodity	0.239447	0.057335
	Equity	0.295398	0.087260
	Bond	0.344241	0.118502
	REITS	0.252503	0.063758
Bayesian Normal	Bond	0.146272	0.021396
	REITS	0.000752	0.000001
	Equity	0.041649	0.001735
	Commodity	0.044743	0.002002
Bayesian t-Distribution	Bond	0.333643	0.111318
	Equity	0.074260	0.005515
	Commodity	0.086467	0.007477
	REITS	0.007487	0.000056
Historical	REITS	0.026022	0.000677
	Equity	0.001098	0.000001
	Bond	0.018470	0.000341
	Commodity	0.003655	0.000013
Parametric GEV	Bond	0.015244	0.000232
	Equity	0.002501	0.000006
	REITS	0.080844	0.006536
	Commodity	0.010517	0.000111
Parametric Normal	Commodity	0.036396	0.001325
	Bond	0.024403	0.000596
	REITS	0.002677	0.000007
	Equity	0.020197	0.000408
Parametric t-Distribution	Equity	0.011206	0.000126
	Commodity	0.026572	0.000706
	REITS	0.013184	0.000174
	Bond	0.020648	0.000426

Table A.6: Quarterly CVaR Error Analysis

Method	Asset Class	Absolute Error	MSE
Bayesian GEV	Commodity	0.017665	0.000312
	Equity	0.249952	0.062476
	Bond	0.311858	0.097256
	REITS	0.138712	0.019241
Bayesian Normal	Bond	0.011795	0.000139
	REITS	0.003984	0.000016
	Equity	0.004487	0.000020
	Commodity	0.111010	0.012323
Bayesian t-Distribution	Bond	0.299527	0.089717
	Equity	0.235962	0.055678
	Commodity	0.037159	0.001381
	REITS	0.037910	0.001437
Historical	REITS	0.103233	0.010657
	Equity	0.068634	0.004711
	Bond	0.017211	0.000296
	Commodity	0.001020	0.000001
Parametric GEV	Bond	0.014718	0.000217
	Equity	0.036317	0.001319
	REITS	0.053852	0.002900
	Commodity	0.053982	0.002914
Parametric Normal	Commodity	0.114658	0.013146
	Bond	0.032557	0.001060
	REITS	0.007722	0.000060
	Equity	0.001703	0.000003
Parametric t-Distribution	Equity	0.029619	0.000877
	Commodity	0.095936	0.009204
	REITS	0.027281	0.000744
	Bond	0.020655	0.000427

Table A.7: Semi-Annual CVaR Error Analysis

Method	Asset Class	Absolute Error	MSE
Bayesian GEV	Commodity	0.085819	0.007365
	Equity	0.219595	0.048222
	Bond	0.265551	0.070517
	REITS	0.075558	0.005709
Bayesian Normal	Bond	0.056352	0.003176
	REITS	0.116098	0.013479
	Equity	0.058194	0.003387
	Commodity	0.278183	0.077386
Bayesian t-Distribution	Bond	0.095291	0.009080
	Equity	0.121990	0.014882
	Commodity	0.092095	0.008481
	REITS	0.203792	0.041531
Historical	REITS	0.304231	0.092557
	Equity	0.203799	0.041534
	Bond	0.048323	0.002335
	Commodity	0.148433	0.022032
Parametric GEV	Bond	0.041582	0.001729
	Equity	0.108681	0.011811
	REITS	0.152038	0.023116
	Commodity	0.054480	0.002968
Parametric Normal	Commodity	0.069755	0.004866
	Bond	0.059616	0.003554
	REITS	0.067297	0.004529
	Equity	0.053687	0.002882
Parametric t-Distribution	Equity	0.138465	0.019173
	Commodity	0.019772	0.000391
	REITS	0.143518	0.020597
	Bond	0.046190	0.002134

Table A.8: Annual CVaR Error Analysis

Method	Asset Class	Absolute Error	MSE
Bayesian GEV	Commodity	0.126602	0.016028
	Equity	0.207252	0.042953
	Bond	0.204143	0.041675
	REITS	0.081962	0.006718
Bayesian Normal	Bond	0.100190	0.010038
	REITS	0.110342	0.012175
	Equity	0.015404	0.000237
	Commodity	0.318557	0.101479
Bayesian t-Distribution	Bond	0.193256	0.037348
	Equity	0.445844	0.198777
	Commodity	0.183995	0.033854
	REITS	2.155229	4.645010
Historical	REITS	0.447005	0.199813
	Equity	0.313434	0.098241
	Bond	0.097738	0.009553
	Commodity	0.192740	0.037149
Parametric GEV	Bond	0.041522	0.001724
	Equity	0.219295	0.048090
	REITS	0.361389	0.130602
	Commodity	0.209254	0.043787
Parametric Normal	Commodity	0.052213	0.002726
	Bond	0.106019	0.011240
	REITS	0.246534	0.060779
	Equity	0.126897	0.016103
Parametric t-Distribution	Equity	0.355444	0.126340
	Commodity	0.102909	0.010590
	REITS	1.030785	1.062518
	Bond	0.077734	0.006043

Bibliography

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

Radford M Neal. Mcmc using hamiltonian dynamics. *arXiv preprint arXiv:1206.1901*, 2012.

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

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76, 2017.

Motohide Nishio and Aisaku Arakawa. Performance of hamiltonian monte carlo and no-u-turn sampler for estimating genetic parameters and breeding values. *Genetics Selection Evolution*, 51:1–12, 2019.

Nisheeth K Vishnoi. An introduction to hamiltonian monte carlo method for sampling. *arXiv preprint arXiv:2108.12107*, 2021.

Yifeng Che, Xu Wu, Giovanni Pastore, Wei Li, and Koroush Shirvan. Application of kriging and variational bayesian monte carlo method for improved prediction of doped uo2 fission gas release. *Annals of Nuclear Energy*, 153:108046, 2021.

Somayajulu LN Dhulipala, Michael D Shields, Benjamin W Spencer, Chandrakanth Bolisetti, Andrew E Slaughter, Vincent M Labouré, and Promit Chakroborty. Active learning with multifidelity modeling for efficient rare event simulation. *Journal of Computational Physics*, 468:111506, 2022.

Daniel Levy, Matthew D Hoffman, and Jascha Sohl-Dickstein. Generalizing hamiltonian monte carlo with neural networks. *arXiv preprint arXiv:1711.09268*, 2017.

Yiting Yin, Wenxia Xu, Liping Zhu, Wen Cheng, Yani Mo, and Ren Lin. Bayesian sampling method design based on hierarchical model. 2023.

Richard Gerlach, Chris Carter, and Robert Kohn. Efficient bayesian inference for dynamic mixture models. *Journal of the American Statistical Association*, 95(451):819–828, 2000.

Shmuel Kandel, Robert McCulloch, and Robert F Stambaugh. Bayesian inference and portfolio efficiency. *The Review of Financial Studies*, 8(1):1–53, 1995.

Gregor Kastner. stochvol: Efficient bayesian inference for stochastic volatility (sv) models. 2017.

Nick Whiteley, Christophe Andrieu, and Arnaud Doucet. Efficient bayesian inference for switching state-space models using discrete particle markov chain monte carlo methods. *arXiv preprint arXiv:1011.2437*, 2010.

Gregor Kastner, Sylvia Fr"uhwirth-Schnatter, and Hedibert Freitas Lopes. Efficient bayesian inference for multivariate factor stochastic volatility models. *Journal of Computational and Graphical Statistics*, 26(4):905–917, 2017.

Qiang Han, Pinghe Ni, Xiuli Du, Hongyuan Zhou, and Xiaowei Cheng. Computationally efficient bayesian inference for probabilistic model updating with polynomial chaos and gibbs sampling. *Structural Control and Health Monitoring*, 29(6):e2936, 2022.

Somayajulu LN Dhulipala, Yifeng Che, and Michael D Shields. Efficient bayesian inference with latent hamiltonian neural networks in no-u-turn sampling. *Journal of Computational Physics*, 492:112425, 2023.

Johan Alenlöv, Arnoud Doucet, and Fredrik Lindsten. Pseudo-marginal hamiltonian monte carlo. *Journal of Machine Learning Research*, 22(141):1–45, 2021.

George Deligiannidis, Arnaud Doucet, and Michael K Pitt. The correlated pseudomarginal method. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 80(5):839–870, 2018.

Sebastian M Schmon, George Deligiannidis, Arnaud Doucet, and Michael K Pitt. Large-sample asymptotics of the pseudo-marginal method. *Biometrika*, 108(1):37–51, 2021.

Samuel Livingstone, Michael F Faulkner, and Gareth O Roberts. Kinetic energy choice in hamiltonian/hybrid monte carlo. *Biometrika*, 106(2):303–319, 2019.

Kjartan Kloster Osmundsen, Tore Selland Kleppe, and Roman Liesenfeld. Pseudo-marginal hamiltonian monte carlo with efficient importance sampling. *Available at SSRN 3304077*, 2018.

Tore Selland Kleppe and Roman Liesenfeld. Efficient importance sampling in mixture frameworks. *Computational statistics & data analysis*, 76:449–463, 2014.

Oliver Grothe, Tore Selland Kleppe, and Roman Liesenfeld. The gibbs sampler with particle efficient importance sampling for state-space models. *Econometric Reviews*, 38(10):1152–1175, 2019.

C Gadd, Sara Wade, and Akeel A Shah. Pseudo-marginal bayesian inference for gaussian process latent variable models. *Machine Learning*, 110:1105–1143, 2021.

Hannes Vandecasteele and Giovanni Samaey. Pseudo-marginal approximation to the free energy in a micro–macro markov chain monte carlo method. *The Journal of Chemical Physics*, 160(10), 2024.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

Alexandros Beskos, Frank J Pinski, Jesús Marıa Sanz-Serna, and Andrew M Stuart. Hybrid monte carlo on hilbert spaces. *Stochastic Processes and their Applications*, 121(10):2201–2230, 2011.

BLC Matthews. Molecular dynamics: With deterministic and stochastic numerical methods, 2015.

Babak Shahbaba, Shiwei Lan, Wesley O Johnson, and Radford M Neal. Split hamiltonian monte carlo. *Statistics and Computing*, 24:339–349, 2014.