# Nonparametric Involutive Markov Chain Monte Carlo: a MCMC algorithm for universal probabilistic programming

Carol Mak

Magdalen College
University of Oxford

*A thesis submitted for the degree of*
*Doctor of Philosophy*

Trinity 2022

## Abstract

Probabilistic programming, the idea to write probabilistic models as computer programs, has proven to be a powerful tool for statistical analysis thanks to the computation power of built-in inference algorithms. Developing suitable inference algorithms that work for arbitrary programs in a Turing-complete probabilistic programming language (PPL) has become increasingly important. This thesis presents the Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC) framework for the construction of MCMC inference machines for nonparametric models that can be expressed in Turing-complete PPLs. Relying on the tree representable structure of probabilistic programs, the NP-iMCMC algorithm automates the trans-dimensional movement in the sampling process and only requires the specification of proposal distributions and mappings on fixed dimensional spaces which are provided by inferences like the popular Hamiltonian Monte Carlo (HMC). We gave a theoretical justification for the NP-iMCMC algorithm and put NP-iMCMC into action by introducing the Nonparametric HMC (NP-HMC) algorithm, a nonparametric variant of the HMC sampler. This NP-HMC sampler works out-of-the-box and can be applied to virtually all useful probabilistic models. We further improved NP-HMC by applying the techniques specified for NP-iMCMC to construct irreversible extensions that have shown significant performance improvements against other existing inference methods.

# Nonparametric Involutive Markov Chain Monte Carlo: a MCMC algorithm for universal probabilistic programming


Carol Mak

Magdalen College

University of Oxford

A thesis submitted for the degree of

*Doctor of Philosophy*

Trinity 2022

# Acknowledgements

First and foremost, I would like to thank my supervisor, Luke Ong, whose guidance, encouragement, and support have been instrumental throughout my doctoral journey. Your supervision has shaped not just this thesis but also my academic career.

I am grateful to my thesis examiners, Hongseok Yang and Arnaud Doucet, for their expertise and constructive feedback that have enriched the quality of this work.

I am indebted to University of Oxford and Croucher Foundation for providing financial support for this research. This funding has enabled me to freely pursue my intellectual ideas, and grow and become an independent researcher.

My heartfelt gratitude go to my colleagues and mentors at Department of Computer Science, Magdalen College, St Catherine's College, whose research discussions have been invaluable, in particular Gavin Lowe, Thomas Mattinson, Angus Taylor, Toby Cathcart Burn, Mario Alvarez Picallo, Dominik Wagner, Rolf Morel, Hugo Paquet, Fabian Zaiser and Maria Nicoleta Craciun.

I extend my appreciation to the staff of the Department of Computer Science, whose administrative assistance, technical support, and resources have been indispensable in the completion of this thesis.

I am deeply grateful to my friends for their encouragement which have sustained me through the highs and lows of this doctoral journey, including the Oxford Catholic Chaplaincy chaplains and residents, members of the Newman Society: the University of Oxford Catholic Society, the 11am and 9pm Mass Choristers, the Oxford Leg Student Cross pilgrims, and all those participated in the God in the Quad discussion group.

I would like to thank my family for their unwavering belief in me, their patience, and their unconditional love and support.

Lastly, I dedicate this thesis to God, whose loving mercy continue to inspire and motivate me every day.

# Abstract

Probabilistic programming, the idea to write probabilistic models as computer programs, has proven to be a powerful tool for statistical analysis thanks to the computation power of built-in inference algorithms. Developing suitable inference algorithms that work for arbitrary programs in a Turing-complete probabilistic programming language (PPL) has become increasingly important. This thesis presents the Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC) framework for the construction of MCMC inference machines for nonparametric models that can be expressed in Turing-complete PPLs. Relying on the tree representable structure of probabilistic programs, the NP-iMCMC algorithm automates the trans-dimensional movement in the sampling process and only requires the specification of proposal distributions and mappings on fixed dimensional spaces which are provided by inferences like the popular Hamiltonian Monte Carlo (HMC). We gave a theoretical justification for the NP-iMCMC algorithm and put NP-iMCMC into action by introducing the Nonparametric HMC (NP-HMC) algorithm, a nonparametric variant of the HMC sampler. This NP-HMC sampler works out-of-the-box and can be applied to virtually all useful probabilistic models. We further improved NP-HMC by applying the techniques specified for NP-iMCMC to construct irreversible extensions that have shown significant performance improvements against other existing inference methods.

# Contents

# List of Figures

*Dominus illuminatio mea.*

— Psalm 26

# 1 Introduction

The inclusion of randomness in programming languages dates back to the earliest version of FORTRAN (Backus et al., 1957). Kozen in (Kozen, 1979) presented the foundation of such language which proves to be a versatile and elegant system for the study of probabilistic complexity compared to low-level models such as decision trees, Turing machines, directed graphs and finite automata. This line of research was further explored in the early 2000s (Ramsey and Pfeffer, 2002; Park et al., 2005). However, these languages lack the ability to record observations, which limits their applicability towards statistical analysis.

Goodman et al. in (Goodman et al., 2008) designed Church, a so-called universal probabilistic programming language (PPL) which is a Turing-complete functional language augmented with two probabilistic operators, mimicking the probabilities in Bayes' Law. Typically, randomness is introduced in a universal PPL via a command called sampling and denoted as `sample(D)`. Intuitively, `sample(D)` returns a randomly drawn sample from the distribution denoted by `D`. In statistical analysis, it is typically used to describe the prior knowledge of a task. In addition, the authors introduced the idea of scoring: it gives the samplings results so far a score which reflects how likely they are to have happened. It is usually called scoring, sometimes known as soft constraints, and denoted as `score(c)`. Scoring is a powerful feature in statistical analysis as it allows users to input the likelihood of the observed data to the program. With these two new constructs, probabilistic models can be described as computer programs. For example, nonparametric models like the infinite Gaussian mixture model (iGMM) can be expressed using branching and recursion in universal PPL.

The introduction of universal probabilistic programming led to an explosive development of many practical probabilistic languages, such as Anglican (Wood et al., 2014), Venture (Mansinghka et al., 2014), Web PPL (Goodman and Stuhlmüller, 2014), Hakaru (Narayanan and Shan, 2020), Pyro (Bingham et al., 2019), Turing (Ge et al., 2018), Gen (Cusumano-Towner et al., 2019), Stan (Gelman et al., 2015), Infer.NET (Minka et al., 2018) and PyMC (Salvatier et al., 2016); and the study of probability theory in programming languages (Staton, 2017; Borgström et al., 2016; Culpepper and Cobb, 2017; Wand et al., 2018; Vákár et al., 2019; Heunen et al., 2017; Ścibior et al., 2017; Staton et al., 2016; Ehrhard et al., 2014, 2018, 2015; Danos and Ehrhard, 2011; Mak et al., 2021a; Zhou et al., 2019).

Typically in a practical probabilistic language, after a model is defined as a computer program, the computation of the posterior is then carried out by some back-end inference engines. A common example of these back-end engines is the Markov Chain Monte Carlo (MCMC) inference method, where the posterior is simulated via a Markov chain of samples. Unfortunately, popular black-box inference algorithms like Hamiltonian Monte Carlo (HMC) (Neal, 2011) cannot always be applied to models defined as probabilistic programs since it has a finite dimensional state space. Other existing inference algorithms designed for nonparametric models are either not reliable (LMH (Wingate et al., 2011), RMH, particle Gibbs etc) or require a high-level customisation (RJMCMC (Green, 1995)).

The purpose of this thesis is to (1) simplify the design process of inference algorithms for probabilistic programming and (2) construct a concrete inference algorithm that has an adequate trade-off amongst reliability, efficiency, applicability, adaptability, simplicity of implementation, easiness to extend, and automation.

We first understand probabilistic programming by discussing the Statistical Programming Computable Functions (SPCF), a probabilistic variant of the infamous functional PCF language invented by Scott in 1969 (Plotkin, 1977; Scott, 1993), where all computable probabilistic models can be specified. We give a simple small-step reduction system for SPCF programs and importantly identify a key characteristic of the densities, namely that they must be *tree representable*. We then study the recently suggested involutive Markov Chain Monte Carlo (iMCMC) framework (Neklyudov et al., 2020), which can describe many existing MCMC algorithms (MH, HMC, Gibbs) and the list of tricks that can be applied to iMCMC.

The main contribution of this thesis is the Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC) algorithm, a general framework to design MCMC algorithms for nonparametric probabilistic models specified in the SPCF language. Relying on the tree representable structure of their density functions, the NP-iMCMC algorithm automates the trans-dimensional movement in the sampling process and only requires

the specification of proposal distributions and mappings on fixed dimensional spaces which are provided by iMCMC methods like HMC. As SPCF can specify all computable probabilistic models, NP-iMCMC is applicable to virtually all useful probabilistic models. Furthermore, techniques identified for iMCMC can also be applied on the NP-iMCMC sampler to facilitates powerful extensions. With some minor assumptions, we justify the NP-iMCMC algorithm and prove that the generated Markov chain preserves the distribution of the given SPCF program.

After that, we put NP-iMCMC into action and design the Nonparametric HMC (NP-HMC) algorithm, a nonparametric variant of the HMC algorithm. Furthermore, we improve NP-HMC by applying the techniques specified for NP-iMCMC and form the Nonparametric Discontinuous HMC (NP-DHMC), NP-Generalised DHMC and NP-Look Ahead DHMC. These inferences work out-of-the-box and thanks to (Mak et al., 2021a) can be applied to many useful probabilistic models. We justify them empirically by comparing them with other existing inference methods.

**Outline** We first study the SPCF language and show that it can be represented by a class of tree representable function in Chapter 3. After that in Chapter 4, we learn about a few prominent MCMC methods to sample from a probabilistic model using the recently suggested iMCMC framework. We then introduce the Nonparametric iMCMC (NP-iMCMC) method in Chapter 5 which can be seen as a generalisation of the iMCMC algorithm for nonparametric models described by tree representable functions. We further develop the system by considering some techniques that can be applied to NP-iMCMC to improve flexibility and form irreversible chains. After that, we show that under mild conditions, these algorithms are correct in Chapter 6. Last by not least, in Chapter 7 we form a nonparametric variant of HMC using the NP-iMCMC framework, develop some variants and run experiments on them.

*Judica me, Deus, et discerne causam meam de gente non sancta, ab homine iniquo et doloso erue me.*

— Psalm 42:1

# 2

# A Brief Introduction to Measure Theory

This chapter provides the essential mathematical foundations on which this thesis is based, and outlines conventions that will be used throughout this thesis. We start with a gentle introduction to basic measure theory, which will equip us with the necessary tools and techniques to explore the research questions and key concepts in the rest of this thesis. Whenever appropriate, we show how these measure theoretical definitions can be applied to probability theory. Readers familiar with measurable spaces, measurable functions, measures, and integration can start with Chapter 3 and refer back to clarify notations and conventions as needed. For more details on the propositions and theorems, we direct interested readers to the excellent account in (Williams, 1991), from which much of this section is borrowed.

## 2.1 Measurable space

A measurable space specifies a set $S$ and a collection $\Sigma_S$ of subsets of $S$ called the $\sigma$-algebra, containing $S$ and is closed under complementation and countable unions. In a probabilistic programming language, every sampler has an associated measurable space $(S, \Sigma_S)$. The set $S$ contains all possible outcomes of the sampler, and the $\sigma$-algebra $\Sigma_S$ contains all events the sampler can describe. Hence each event is a subset of possible outcomes. The definition of $\sigma$-algebra insists that the set of all possible outcome is an event and the complement and countable union of events are also events. Formally, it is defined as follows.

**Definition 1.** A ***measurable space*** is a pair $(S, \Sigma_S)$ containing a set $S$ and a ***$\sigma$-algebra*** $\Sigma_S$ on $S$, i.e. $\Sigma_S$ is a collection of subsets of $S$ satisfying

- $S \in \Sigma_S$;
- $A \in \Sigma_S$ implies $A^c := S \setminus A \in \Sigma_S$; and
- for all $n \in \mathbb{N}$, $A_n \in \Sigma_S$ implies $\bigcup_{n\in\mathbb{N}} A_n \in \Sigma_S$.

We denote elements in the $\sigma$-algebra $\Sigma_S$ as ***measurable sets*** on $S$.

**Example 1.** Let $S$ be a set.

(1) The set containing the empty set $\varnothing$ and $S$ is a $\sigma$-algebra on $S$. We call it the ***trivial $\sigma$-algebra*** on $S$.

(2) The power set $\wp(S)$ of $S$, which contains all subsets of $S$, is a $\sigma$-algebra on $S$. We call $\wp(S)$ the ***discrete $\sigma$-algebra***.

(3) Let $C$ be a collection of subsets in $S$. The $\sigma$-algebra $\sigma(C)$ ***generated by*** $C$ is the smallest $\sigma$-algebra which consists of every set in $C$. Formally, it is the intersection of all $\sigma$-algebra containing $C$.

**Example 2.** The $\sigma$-algebra $\mathcal{B} := \sigma(\{(r, q) \mid r, q \in \mathbb{R}\})$ generated by the set of all open intervals in the Euclidean space $\mathbb{R}$ is called the ***Borel $\sigma$-algebra***. This is an important example and includes all subsets of Reals of interest to this thesis. Note that the Borel $\sigma$-algebra on $\mathbb{R}$ is not the same as the discrete $\sigma$-algebra, i.e. it does *not* contain all subsets of $\mathbb{R}$. Unless otherwise specified, the Euclidean space $\mathbb{R}$ is always equipped with the Borel $\sigma$-algebra $\mathcal{B}$, and we call $(\mathbb{R}, \mathcal{B})$ the ***Borel measurable space***.

**Example 3.** The normal sampler `normal` in a typical probabilistic programming language randomly draws a value in the Borel measurable space $(\mathbb{R}, \mathcal{B})$. Similarly the Boolean sampler `coin` draws from the measurable space $2 := \{\mathsf{T}, \mathsf{F}\}$ with the discrete $\sigma$-algebra $\Sigma_2 := \wp(2)$.

In probabilistic programming, samplers are used as building blocks in the specification of probabilistic models. This thesis studies probabilistic models that are *non-parametric* — models where the number of random variables are not determined *a priori*. An example of nonparametric models is the geometric distribution which returns the number of Bernoulli trials needed to get one success as the number of Bernoulli trials is not determined. Nonparametric models can be specified by probabilistic programs with an unbounded number of calls to sample using recursion. For instance the geometric distribution with parameter $p = 0.5$ can be described by the program `count = 1; while coin: count += 1; return count`.

We now present the measurable space that describes such programs. First we look at those spaces that describe programs with multiple sample calls such as `coin; coin`, then we move to those that allow us to describe programs with an uncertain number of sample calls such as `if coin: coin;coin else: coin`. Using these measurable spaces, we construct the ***list measurable space*** that can describe programs with an unbounded number of sample calls.

**Definition 2.** Let $(S_1, \Sigma_{S_1})$ and $(S_2, \Sigma_{S_2})$ be measurable spaces. The ***product measurable space*** is the Cartesian product $S_1 \times S_2$ equipped with the ***product $\sigma$-algebra*** $\Sigma_{S_1 \times S_2} := \sigma(\{A_1 \times A_2 \mid A_1 \in \Sigma_{S_1}, A_2 \in \Sigma_{S_2}\})$ generated by the set of all Cartesian products of measurable sets on $S_1$ and $S_2$.

**Proposition 1.** *The product of $n$ copies of the Borel measurable space $(\mathbb{R}, \mathcal{B})$ coincides with the measurable space on the $n$-dimensional Euclidean space $\mathbb{R}^n$ equipped with the Borel $\sigma$-algebra $\mathcal{B}_n$ generated by all open sets in $\mathbb{R}^n$.*

*Remark* 1. For ease of reference, we denote $(S^n, \Sigma_{S^n})$ to be the product measurable space of $n$ copies of $(S, \Sigma_S)$.

**Example 4.** The product measurable space $2^2$ equipped with the product $\sigma$-algebra $\Sigma_{2^2} := \sigma(\{A \times B \mid A, B \in \wp(2)\})$ can describe the probabilistic program `coin; coin`. For instance, the event of 'first flipping true' is described by the measurable set $\{(\mathsf{T}, \mathsf{T}), (\mathsf{T}, \mathsf{F})\}$ and the event of 'flipping both true and false' is described by the measurable set $\{(\mathsf{T}, \mathsf{F}), (\mathsf{F}, \mathsf{T})\}$.

**Definition 3.** Let $(S_1, \Sigma_{S_1})$ and $(S_2, \Sigma_{S_2})$ be measurable spaces. The ***union measurable space*** is the union $S_1 \cup S_2$ equipped with the ***union $\sigma$-algebra*** $\Sigma_{S_1 \cup S_2} := \sigma(\Sigma_{S_1} \cup \Sigma_{S_2})$ generated by the union of the $\sigma$-algebra of $S_1$ and $S_2$.

**Example 5.** The union measurable space $2^3 \cup 2^2$ equipped with the union $\sigma$-algebra $\Sigma_{2^3 \cup 2^2}$ allows us to describe the probabilistic program `if coin: coin;coin else: coin`, which either performs three coin flips or two depending on the result of the first coin flip.

**Definition 4.** The ***list measurable space*** of a measurable space $(S, \Sigma_S)$ describes all finite lists that contain elements in $S$ and is constructed by taking the (countable) union of the $n$ product measurable space $(S^n, \Sigma_{S^n})$ over $n \in \mathbb{N}$. This gives us a measurable space with set $\bigcup_{n \in \mathbb{N}} S^n$ and $\sigma$-algebra generated by the union of $\Sigma_{S^n}$, namely $\sigma(\bigcup_{n \in \mathbb{N}} \Sigma_{S^n})$. It is easy to check that measurable set takes the form $\bigcup_{n \in \mathbb{N}} A_n$ for $A_n \in \Sigma_{S^n}$. We use the list notation to describe elements in the list measurable space, e.g. $[]$, $[s_1, \ldots, s_n]$ for $s_1, \ldots, s_n \in S$.

**Example 6.** Consider the list measurable space $(\bigcup_{n\in\mathbb{N}} 2^n, \sigma(\bigcup_{n\in\mathbb{N}} \Sigma_{2^n}))$ of the Boolean measurable space. This can be used to describe probabilistic programs with an unbounded number of calls to the Boolean sampler `coin`. For instance, the list of samples drawn in a particular run of the probabilistic program `count = 1; while coin: count += 1; return count` are elements in $\bigcup_{n\in\mathbb{N}} 2^n$.

**Definition 5.** Let $(S, \Sigma_S)$ be a measurable space and $S'$ a subset of $S$. The ***sub-measurable space*** consists of the set $S'$ and the ***sub-$\sigma$-algebra*** $\Sigma_{S'} := \{A \cap S' \mid A \in \Sigma_S\}$.

It is easy to check that the sub-$\sigma$-algebra is indeed a $\sigma$-algebra.

## 2.2 Measurable function

After defining measurable spaces, we can now present measurable functions, which are functions between measurable spaces. A measurable function from $S_1$ to $S_2$ can be viewed as a variable that takes different values in $S_2$ depending on the random outcome in $S_1$.

**Definition 6.** Let $(S_1, \Sigma_{S_1})$ and $(S_2, \Sigma_{S_2})$ be measurable spaces. A map $X : S_1 \to S_2$ is a ***measurable function*** if the inverse image of any measurable set in its codomain is measurable in its domain, i.e.

$$A \in \Sigma_{S_2} \quad \Rightarrow \quad X^{-1}(A) := \{s \in S_1 \mid X(s) \in A\} \in \Sigma_{S_1}.$$

**Example 7.** We look at some notable measurable and non-measurable functions.

- All functions from a measurable space with a discrete $\sigma$-algebra is measurable.
- Let $\mathbb{R}$ be equipped with the Borel $\sigma$-algebra. Then, all continuous functions with the type $\mathbb{R} \to \mathbb{R}$ are measurable.
- Not all measurable functions with the type $\mathbb{R} \to \mathbb{R}$ are continuous. For instance, the characteristic function $\mathbb{1}_{\mathbb{Q}} : \mathbb{R} \to \mathbb{R}$ of the set $\mathbb{Q}$ of rational numbers is not continuous but is measurable.
- Not all functions with the type $\mathbb{R} \to \mathbb{R}$ are measurable. Let $A$ be a non-measurable subset in $\mathbb{R}$. Then, the characteristic function $\mathbb{1}_A$ of $A$ is not measurable.

We can form more sophisticated measurable functions as they are closed under composition and product.

**Proposition 2.** *Let $(S_1, \Sigma_{S_1}), (S_2, \Sigma_{S_2}), (S_3, \Sigma_{S_3})$ and $(S_4, \Sigma_{S_4})$ be measurable spaces.*

*(i)* *If* $X : S_1 \to S_2$ *and* $Y : S_2 \to S_3$ *are measurable functions, then so is their composite* $Y \circ X : S_1 \to S_3$ *defined as* $(Y \circ X)(s) := Y(X(s))$

*(ii)* *If* $X : S_1 \to S_2$ *and* $Y : S_3 \to S_4$ *are measurable functions, then so is their product* $X \times Y : S_1 \times S_3 \to S_2 \times S_4$ *defined as* $(X \times Y)(s_1, s_3) := (X(s_1), Y(s_3))$

A key observation (Kozen, 1979) in the development of probabilistic programming is that given a list of samples drawn in a particular execution of a probabilistic program, the program ceases to be probabilistic. Instead we can run the probabilistic program like any deterministic program.

**Example 8.** Consider the probabilistic program `count = 1; while coin: count += 1; return count`. Take the list $[\mathsf{T}, \mathsf{T}, \mathsf{F}]$, which lies in the list measurable space of the Boolean measurable space $(\mathbb{2}, \Sigma_{\mathbb{2}})$, the program can be executed determinedly by taking the first unused value in the list for each `coin` flip and returning `2`. The function from the list measurable space $\bigcup_n \mathbb{2}^n$ to the union $\mathbb{R} \cup \{\bot\}$ of the Borel measurable space and the singleton $\{\bot\}$ that returns different values in $\mathbb{R}$ (or $\bot$ if the given list does not match the structure of the program such as $[\mathsf{T}, \mathsf{F}, \mathsf{T}]$) depending on given list of random coin flips is measurable.

## 2.3 Measure

Measure theory gets exciting when we start assigning a non-negative Real number to every measurable set, giving it a 'weight'. The function which defines such assignment is called a measure.

**Definition 7.** A function $\mu_S : \Sigma_S \to [0, \infty]$ is a ***measure*** on the measurable space $(S, \Sigma_S)$ if it is countably additive, i.e.

- $\mu_S(\varnothing) = 0$, and
- for any countable sequence of disjoint measurable sets $\{A_n\}_{n \in \mathbb{N}}$ on $S$, $\mu_S(\bigcup_{n \in \mathbb{N}} A_n) = \sum_{n \in \mathbb{N}} \mu_S(A_n)$.

A ***measure space*** is a tripe $(S, \Sigma_S, \mu_S)$ where $(S, \Sigma_S)$ is a measurable space and $\mu_S$ is a measure.

For ease of reference, we often drop the subscript in the measure when the measurable space is unambiguous. We now give some simple properties of measure which are direct consequences of the definition.

**Proposition 3.** *Let $(S, \Sigma_S, \mu_S)$ be a measure space. The measure $\mu_S$ always satisfy the followings. Let $A, B, A_n$ be measurable sets for all $n \in \mathbb{N}$.*

- *(Additive) $A \cap B = \varnothing$ implies that $\mu_S(A \cup B) = \mu_S(A) + \mu_S(B)$.*
- *(Monotone) $A \subseteq B$ implies that $\mu_S(A) \leq \mu_S(B)$.*
- *(Countably subadditive) $\mu_S(\bigcup_{n\in\mathbb{N}} A_n) \leq \sum_{n\in\mathbb{N}} \mu_S(A_n)$.*

*Let $\mu_1$ and $\mu_2$ be measures on $(S, \Sigma_S)$. Then, there exists a unique measure $\mu_1 + \mu_2$ such that $(\mu_1 + \mu_2)(A) := \mu_1(A) + \mu_2(A)$ for all measurable sets $A$.*

The (value) measure of a probabilistic program takes a measurable set of returned values, i.e. an event, and gives the "likelihood" of such event according to the program. For instance, the value measure of the probabilistic program `count = 1; while coin: count += 1; return count` takes a subset $A$ in $\mathbb{N}$ and returns $\sum_{k\in A}(1/2)^k$, which coincides with the geometric distribution with $p = 0.5$.

What makes probabilistic programming different from randomized programming is its ability to give a *score* to the current execution. In other words, one can (artificially) change the weight of a particular execution in a probabilistic program. This proves to be very useful in Bayesian inference (as we will see in Chapter 3).

For now, we consider scoring to be a way to multiply the current likelihood with a non-negative Real number. For example, say we observe that the `count` in the above program must be larger than three. We can add this piece of information to the program by assigning a zero score to the runs with `count` less than or equals to three. Putting it together, we get the following program.

```
count = 1;
while coin:
  count += 1;
if count <= 3:
  score(0);
return count
```

This program will now have a value measure given by

$$A \mapsto \begin{cases} \sum_{k\in A}(1/2)^k & \text{if } k > 3 \\ 0 & \text{otherwise.} \end{cases}$$

An important remark is that the value measure of a probabilistic program does *not* necessarily "sum to one". But it does fall under one of the following four classes of measures.

**Definition 8.** Let $(S, \Sigma_S, \mu_S)$ be a measure space.

- The measure $\mu_S$ is a ***probability*** measure if $\mu_S(S) = 1$. A measure space with a probability measure is called a ***probability space***.
- The measure $\mu_S$ is ***finite*** if $\mu_S(S) < \infty$.
- The measure $\mu_S$ is ***$\sigma$-finite*** if there is a countable cover[1] $\{A_n\}_{n\in\mathbb{N}}$ of measurable sets on $S$ where $\mu_S(A_n) < \infty$ for all $n \in \mathbb{N}$.
- The measure $\mu_S$ is ***s-finite*** if there is a countable sequence $\{\mu_n\}_{n\in\mathbb{N}}$ of finite measures on $S$ such that $\mu_S = \sum_{n\in\mathbb{N}} \mu_n$.

We have the following chain of inclusions for measures:

probability measures $\subset$ finite measures $\subset$ $\sigma$-finite measures $\subset$ s-finite measures

In the following example, we show that these inclusions are strict.

**Example 9.** We consider some common measures that will be used throughout this thesis. Let $(S, \Sigma_S)$ be a measurable space.

- Take an element $s \in S$. The ***Dirac measure*** $\delta_s(A)$ in $s$ is defined to be the characteristic function $\mathbb{1}_{\{s\}}(A)$ on the singleton set $\{s\}$. It is easy to see that the Dirac measure is a probability measure.
- The ***counting measure*** on $S$ is set to be

$$\mu(A) := \begin{cases} |A| & \text{if } A \text{ is finite} \\ \infty & \text{otherwise} \end{cases} \qquad \text{for measurable set } A \in \Sigma_S$$

  where $|A|$ returns the number of elements in the measurable set $A$, if it is finite. Given different measurable spaces, the counting measure has different types. For instance, on the finite space $\{1, \ldots, n\}$ with the discrete $\sigma$-algebra, the counting measure is finite; whereas on the countably finite space $\mathbb{N}$ (with the discrete $\sigma$-algebra), it is no longer finite but $\sigma$-finite. On the Borel measurable space $(\mathbb{R}, \mathcal{B})$, the counting measure is not even $\sigma$-finite. This is because any *countable* cover of $\mathbb{R}$ must contain an uncountable (and infinite) subset of $\mathbb{R}$.
- Fix $s \in S$. Consider the measure $\mu$ that sums over a countable number of Dirac measure in $s$, i.e. $\mu = \sum_{n\in\mathbb{N}} \delta_s$. By construction, $\mu$ is s-finite. But it is *not* $\sigma$-finite, because all countable covers for $S$ must contain a measurable set $A$ consisting of $s$ which has infinite measure.

[1] A ***cover*** of a set $S$ is a collection $C$ of subsets of $S$ such that $S = \bigcup C$.

**Example 10.** Here are some probabilistic programs with different types of value measures.

- Probability measures: `return 3` (Dirac measure in three), `return coin` (Boolean measure), `return normal + normal` (Sum of two Gaussian measures).
- Finite measures: `score(2); return 3` (Point measure in three with weight two), `if coin: score(1); else: score(0)` (Half measure on T and zero on F).
- $\sigma$-finite measures: `x = normal; score(1/pdfnormal(x)); return x` (Lebesgue measure on $\mathbb{R}$) where `pdfnormal` is a primitive that returns the probability density of the Gaussian measure.
- S-finite measures: `x = normal; score(1/pdfnormal(x)); return 0` (Point measure in zero with infinite weight).

Undoubtedly, the most important measure on the Borel measurable space is the ***Lebesgue measure***. It captures the idea of length and is used extensively in measure theory. The justification of its existence and uniqueness is outside of the scope of this thesis and we point interested readers to (Williams, 1991).

**Theorem 1.** *There exists a unique measure* $\mathsf{Leb}$ *on the Borel measurable space* $(\mathbb{R}, \mathcal{B})$ *such that for any* $r, q \in \mathbb{R}$ *where* $r < q$, $\mathsf{Leb}((r, q)) = q - r$.

With the countable cover $\{(n, n+1] \mid n \in \mathbb{Z}\}$ of the Euclidean space $\mathbb{R}$, we see that Lebesgue measure is $\sigma$-finite.

In a measure-theoretical context, a statement is "true enough" if the set on which it does not hold is "neglectable". We define these terms formally as follows.

**Definition 9.** Let $(S, \Sigma_S, \mu_S)$ be a measure space. A measurable set $A$ is said to be $\mu_S$***-null*** (or simply ***null***) if $\mu_S(A) = 0$. A statement about $S$ is true ***almost everywhere (a.e.)*** if the statement can be written as a measurable function $\mathsf{statement} : S \to 2$ and the set $\{s \in S \mid \mathsf{statement}(s) = \mathsf{F}\}$ is is $\mu_S$-null.

### 2.3.1 Pushforward measure

Up till now, we have considered measures that are directly constructed by assigning weights to each measurable set. Another way of defining measures is to use measurable functions.

**Definition 10.** Let $(S_1, \Sigma_{S_1})$ and $(S_2, \Sigma_{S_2})$ be measurable spaces. Given a measurable function $X$ from $S_1$ to $S_2$ and measure $\mu_{S_1}$ on $S_1$, The ***pushforward measure*** $X_*\mu_{S_1}$ of $\mu_{S_1}$ along $X$ is a measure on $S_2$ given by $X_*\mu_{S_1} := \mu_{S_1} \circ X^{-1}$. Note that since $X$ is measurable, its inverse $X^{-1} : \Sigma_{S_2} \to \Sigma_{S_1}$ is well-defined.

Essentially the pushforward measure $X_*\mu_{S_1}$ transforms a measurable set $A$ in $S_2$ to one in $S_1$ via $X^{-1}$ and computes the measure of that set according to $\mu_{S_1}$. It is routine to show that the pushforward measure is indeed a measure.

**Example 11** (Marginal measure)**.** Let $\mu$ be a measure on the $n$-product measurable space $(\prod_{i=1}^n S_i, \Sigma_{\prod_{i=1}^n S_i})$. The ***k-th marginal measure*** of $\mu$ is the pushforward measure $\pi_{k*}\mu$ of $\mu$ along the projection $\pi_k : \prod_{i=1}^n S_i \to S_k$ which returns the $k$-th component. Note that for any measurable set $A$ in $S_k$,

$$\pi_{k*}\mu(A) = \mu(S_1 \times \cdots \times S_{k-1} \times A \times S_{k+1} \times \cdots \times S_n).$$

*Remark* 2. In general, measures on a product measurable space are *not* uniquely determined by its marginals.

We end this subsection with a proposition that links s-finite and $\sigma$-finite measures using pushforward.

**Proposition 4** ((Staton, 2017), Proposition 7)**.** *A measure is s-finite if and only if it is a pushforward of a $\sigma$-finite measure.*

### 2.3.2 Product measure

Similar to measurable spaces and functions, we can form more sophisticated measures as they are closed under product.

**Proposition 5.** *Let $(S_i, \Sigma_{S_i}, \mu_{S_i})$ be measure spaces where $\mu_{S_i}$ are $\sigma$-finite measures for all $i = 1, \ldots, n$. There exists a unique measure $\prod_{i=1}^n \mu_{S_i}$ on the $n$ product measurable space $(\prod_{i=1}^n S_i, \prod_{i=1}^n \Sigma_{S_i})$ such that for all $A_i \in \Sigma_{S_i}$,*

$$(\prod_{i=1}^n \mu_{S_i})(A_1 \times \cdots \times A_n) = \mu_{S_1}(A_1) \times \cdots \times \mu_{S_n}(A_n).$$

*$\prod_{i=1}^n \mu_{S_i}$ is called the* **product measure** *on $\prod_{i=1}^n S_i$.*

*Remark* 3. The product measure for general measures is *not* unique. See (Vákár and Ong, 2018) for a discussion on the product of s-finite measures, which are not unique.

*Remark* 4. In contrast to general measures on a product measurable space (see Rem. 2), the $k$-th marginal measures of a product measure $\prod_{i=1}^{n} \mu_{S_i}$ is given by $\mu_{S_k}$ and the product measure is uniquely determined by its marginals.

**Example 12.** The ***Lebesgue measure*** $\mathsf{Leb}^n$ on the $n$-dimensional Euclidean measurable space $(\mathbb{R}^n, \mathcal{B}_n)$ is defined to be the product of $n$ copies of the Lebesgue measure $\mathsf{Leb}$.

## 2.4 Integration

In this section, we will state the key definitions and properties of integration and discuss how integration is used in measure theory.

### 2.4.1 Definition

We discuss the existence and definition of integrals on three classes of measurable functions. The first class is the set of simple functions, which are finite sums of scaled characteristic functions on measurable sets. Simple functions are like "step functions" where each "step" has a "width" determined by a measurable set and "height" determined by a scalar.

**Definition 11.** Let $(S, \Sigma_S, \mu_S)$ be a measure space. The ***integral*** of a non-negative ***simple function*** $f : S \to [-\infty, \infty]$ where $f = \sum_{k=1}^{n} a_k \cdot \mathbb{1}_{A_k}$ for some $n \geq 1$ where $a_k \in \mathbb{R}$ and $A_k \in \Sigma_S$, w.r.t. $\mu_S$ is defined to be

$$\int_S f(s)\, \mu_S(\mathrm{d}s) := \sum_{k=1}^{n} a_k \cdot \mu_S(A_k).$$

If we see simple functions as step functions, then the integral of a simple function is simply the sum of the "width" of each step, w.r.t. the measure $\mu_S$, times the "height" given by the scalar $a_k$.

Now the integral of non-negative measurable function can be defined as the supremum of the integrals of simple functions that is smaller than it.

**Definition 12.** Let $(S, \Sigma_S, \mu_S)$ be a measure space. The ***integral*** of a non-negative measurable function $f : S \to [-\infty, \infty]$ w.r.t. $\mu_S$ is defined to be

$$\int_S f(s)\, \mu_S(\mathrm{d}s) := \sup \Big\{ \int_S g(s)\, \mu_S(\mathrm{d}s) \mid g \text{ is a simple function such that } 0 \leq g \leq f \Big\}.$$

This definition matches the intuition that the integral of a non-negative measurable function is the area under the function. Finally we extend the above definition to all measurable functions.

**Definition 13.** Let $(S, \Sigma_S, \mu_S)$ be a measure space. A measurable function $f : S \to [-\infty, \infty]$ is ***integrable*** if $\int_S |f(s)| \, \mu_S(\mathrm{d}s) < \infty$. If so, the integral of $f$ w.r.t. $\mu_S$ is defined to be

$$\int_S f(s) \, \mu_S(\mathrm{d}s) := \int_S \max\{f(s), 0\} \, \mu_S(\mathrm{d}s) - \int_S \max\{-f(s), 0\} \, \mu_S(\mathrm{d}s).$$

*Remark* 5. We have considered integral of measurable function $f : S \to [-\infty, \infty]$ over the whole measurable space $S$. The definition can easily be extended to integration over some measurable set $A$ as $\int_A f(s) \, \mu_S(\mathrm{d}s) := \int_S \mathbb{1}_A(s) \cdot f(s) \, \mu_S(\mathrm{d}s)$

**Example 13.** The integral of Real-valued measurable functions on $(\mathbb{R}, \mathcal{B})$ w.r.t. the Lebesgue measure $\mathsf{Leb}$ is called the ***Lebesgue integral***.

Now we state the most important result in integration theory.

**Theorem 2** (Monotone Convergence Theorem (MCT))**.** *Let $(S, \Sigma_S, \mu_S)$ be a measure space. If $f_n : S \to [-\infty, \infty]$ is a countable sequence of non-negative measurable functions, then*

$$\int_S \sum_{n=1}^{\infty} f_n(s) \, \mu_S(ds) = \sum_{n=1}^{\infty} \int_S f_n(s) \, \mu_S(ds).$$

MCT allows us to define the integral of a measurable function $f$ by considering the limit of the integrals of a countable sequence of measurable functions that tends towards $f$.

### 2.4.2 Integration by substitution

Integration by substitution (also called change of variables) is a technique where variables in an integral are changed to simplify the integration. We will use this technique in proving invariance of MCMC inferences in Chapter 4.

**Theorem 3** ((Fremlin, 2010), Theorem 24D)**.** *Let $U$ be an open set in $\mathbb{R}^n$ and $g : U \to \mathbb{R}^n$ be an injective (i.e. one-to-one) differentiable function. Let $f : g(U) \to \mathbb{R}$ be a measurable function. Then, $f$ is integrable if and only if $(f \circ g) \cdot |\det \nabla g|$ is integrable on $U$, where $\det \nabla g$ is the Jacobian determinant of $g$. In which case,*

$$\int_{g(U)} f(r) \, \mathsf{Leb}^n(dr) = \int_U (f \circ g)(q) \cdot |\det \nabla g(q)| \, \mathsf{Leb}^n(dq).$$

*Remark* 6. If $g$ is measurable and is ***volume preserving***, i.e. $g_* \mathsf{Leb}^n = \mathsf{Leb}^n$, then the Jacobian determinant of $g$ is always equal to one. Hence, the integration by substitution can be simplified to

$$\int_{g(U)} f(r) \, \mathsf{Leb}^n(\mathrm{d}r) = \int_U (f \circ g)(q) \, \mathsf{Leb}^n(\mathrm{d}q).$$

### 2.4.3 Radon-Nikodym Derivative

An important observation in integration theory is that integrals define measures. Let $(S, \Sigma_S, \mu)$ be a measure space and $f : S \to [-\infty, \infty]$ be a positive integrable function. Then, we can define a measure $\nu : \Sigma_S \to [0, \infty)$

$$\nu(A) \coloneqq \int_A f(s)\, \mu(\mathrm{d}s) \text{ for measurable set } A \in \Sigma_S.$$

Applying MCT (Thm. 2), it is routine to check that $\nu$ is indeed a measure.

It is natural to ask if we can reverse engineer this method. In other words, can we construct the integrable function $f$ from measures $\mu$ and $\nu$ on $S$?

The ***Radon-Nikodym Theorem*** tells us that such is possible if $\mu$ and $\nu$ are $\sigma$-finite and $\nu$ is ***absolutely continuous*** with respect to $\mu$, i.e. every $\mu$-null measurable set is also $\nu$-null. We denote this relation as $\nu \ll \mu$.

*Remark* 7. It is easy to see that the relation $\ll$ is reflective and transitive but not symmetric.

**Theorem 4** (Radon-Nikodym Theorem)**.** *Let $\mu$ and $\nu$ be $\sigma$-finite measures on a measurable space $(S, \Sigma_S)$. If $\nu$ is absolutely continuous with respect to $\mu$ ($\nu \ll \mu$), then there is a measurable function $f : S \to [0, \infty]$ where*

$$\nu(A) = \int_A f(s)\, \mu(ds) \quad \textit{for all } A \in \Sigma_S.$$

*This measurable function is often referred to as the* **Radon-Nikodym derivative**.

*Remark* 8. The Radon-Nikodym Theorem can be extended to s-finite measures with some additional condition as discussed in (Vákár and Ong, 2018). Let $\mu$ and $\nu$ be s-finite measures on the measurable space $(S, \Sigma_S)$. Say a measurable set $A$ is a $\mu$-0-$\infty$ set if $\nu(A \cap B) = 0$ or $\infty$ for all $B \in \Sigma_S$. Then, the Radon-Nikodym Theorem holds for $\mu$ and $\nu$ if $\nu \ll \mu$ and every $\mu$-0-$\infty$ set is also a $\nu$-0-$\infty$ set.

In probabilistic models, it is often more natural to consider the probability of a single outcome instead of an event. The ***probability density function (pdf)*** of a probability measure $\nu$ coincides with the Radon-Nikodym derivative of $\nu$ w.r.t. the measure $\mu$.

**Example 14.** We list some common probability measures and their probability density functions (pdfs) w.r.t. the Lebesgue measure (for continuous distributions) and the counting measure (for discrete distributions).

- The ***uniform*** measure $\mathcal{U}(a, b)$ with endpoints $a$ and $b$ such that $a \leq b$ has pdf $\mathbb{1}_{[a,b]}(x)$ for $x \in \mathbb{R}$.

- The ***Gaussian*** measure $\mathcal{N}(m, v)$ with mean $m$ and variance $v$ has pdf $\frac{1}{\sqrt{2v\pi}} \exp\left(-\frac{(x-m)^2}{2v}\right)$ for $x \in \mathbb{R}$.

- The ***Bernoulli*** measure $\mathsf{Bern}(p)$ with probability $p \in [0,1]$ has pdf[2] $p \cdot \mathbb{1}_{\{\mathsf{T}\}}(a) + (1-p) \cdot \mathbb{1}_{\{\mathsf{F}\}}(a)$ for $a \in \mathbb{2}$.

- The ***geometric*** measure $\mathsf{Geo}(p)$ with probability $p \in (0,1]$ has pdf $(1-p)^n p$ for $n \in \mathbb{N}$.

- The ***Dirac*** measure $\delta_s$ in $s \in S$ has pdf $\mathbb{1}_{\{s\}}(x)$ for $x \in S$.

### 2.4.4 Commutativity of Integrals

In proving the invariance result for MCMC inferences, we reply on the Fubini/Tonelli Theorem which tells us that integrals on a product space can be rearranged as long as all measures are $\sigma$-finite.

**Theorem 5** (Fubini/Tonelli Theorem)**.** *Let $(S_1, \Sigma_{S_1}, \mu_{S_1})$ and $(S_2, \Sigma_{S_2}, \mu_{S_2})$ be measure spaces where $\mu_{S_1}$ and $\mu_{S_2}$ are $\sigma$-finite. If $f : S_1 \times S_2 \to [0, \infty)$ is an integrable function, then*

$$\begin{aligned}\int_{S_1 \times S_2} f(s_1, s_2)\, \mu_{S_1 \times S_2}(d(s_1, s_2)) &= \int_{S_1} \int_{S_2} f(s_1, s_2)\, \mu_{S_2}(ds_2)\, \mu_{S_1}(ds_1) \\ &= \int_{S_2} \int_{S_1} f(s_1, s_2)\, \mu_{S_1}(ds_1)\, \mu_{S_2}(ds_2).\end{aligned}$$

Recall in Prop. 5 we have only given the existence of the product measure. With the above theorem, we can *define* the product measure.

**Definition 14.** Let $(S_1, \Sigma_{S_1}, \mu_{S_1})$ and $(S_2, \Sigma_{S_2}, \mu_{S_2})$ be measure spaces where $\mu_{S_1}$ and $\mu_{S_2}$ are $\sigma$-finite measures. The unique product measure $\mu_{S_1} \times \mu_{S_2}$ on the product measurable space $(S_1 \times S_2, \Sigma_{S_1 \times S_2})$ is defined as

$$\begin{aligned}(\mu_{S_1} \times \mu_{S_2})(A) &:= \int_{S_1} \mu_{S_2}(\{s_2 \in S_2 \mid (s_1, s_2) \in A\})\, \mu_{S_1}(\mathrm{d}s_1) \\ &= \int_{S_2} \mu_{S_1}(\{s_1 \in S_1 \mid (s_1, s_2) \in A\})\, \mu_{S_2}(\mathrm{d}s_2).\end{aligned}$$

[2]For ease of reference, we do not make a distinction between probability mass functions on discrete distributions and probability density functions on continuous distributions.

### 2.4.5 Kernel

Last but not least, we present ***kernels*** which are functions that connect each element in the measurable space to a measure. Kernels are useful in describing stochastic processes, and will be used to define MCMC inferences in Chapter 4.

**Definition 15.** Let $(S_1, \Sigma_{S_1})$ and $(S_2, \Sigma_{S_2})$ be measurable spaces. A ***kernel*** $k$ from $S_1$ to $S_2$, denoted as $k : S_1 \rightsquigarrow S_2$ is a function $k : S_1 \times \Sigma_{S_2} \to [0, \infty)$ where

- for any $s \in S_1$, the map $k(s, \cdot) : \Sigma_{S_2} \to [0, \infty)$ is a measure; and
- for any $A \in \Sigma_{S_2}$, the map $k(\cdot, A) : S_1 \to [0, \infty)$ is a measurable function.

A ***probability kernel*** $k$ is a kernel where $k(s, \cdot) : \Sigma_{S_2} \to [0, \infty)$ is a probability measure for all $s \in S_1$.

**Example 15.** Let $(S, \Sigma_S)$ be a measurable space. Then, the map $k(s, A) := \delta_s(A)$ is a kernel that returns a Dirac measure on the input $s$.

Integration allows us to form composition of kernels.

**Definition 16.** Let $(S_1, \Sigma_{S_1})$, $(S_2, \Sigma_{S_2})$ and $(S_3, \Sigma_{S_3})$ be measurable spaces. The composition of kernel $k_1 : S_1 \rightsquigarrow S_2$ and kernel $k_2 : S_2 \rightsquigarrow S_3$ is defined as $(k_2 \circ k_1)(s, A) := \int_{S_2} k_2(s', A)\, k_1(s, \mathrm{d}s')$ for all $s \in S_1$ and $A \in \Sigma_{S_3}$.

*Remark* 9. The integral in the above definition of the composition of kernels is well-defined as by definition $k_2(\cdot, A)$ is a non-negative measurable function and $k_1(s, \cdot)$ is a measure for all $s \in S_1$ and $A \in \Sigma_{S_3}$.

**Example 16.** We often treat a probability kernel $k : S \rightsquigarrow S$ as a stochastic step where $k(s, A)$ represents the probability of reaching $A$ from the point $s$. Hence, the composition of $n$ number of $k$, commonly written as $k^n$ represents the application of $n$ stochastic steps, and $k^n(s, A)$ returns the probability of reaching $A$ in $n$ steps starting at $s$. Formally for all $s \in S$ and $A \in \Sigma_S$, we have

$$k^0(s, A) := [s \in A]$$
$$k^{n+1}(s, A) := (k^n \circ k)(s, A).$$

*Quia tu es, Deus, fortitudo mea, quare me repulisti? et quare tristis incedo, dum affligit me inimicus?*

— Psalm 42:2

# 3 Probabilistic Programming for Bayesian Inference

In this chapter, we study probabilistic programming, a programming paradigm in which Bayesian inference, an approach to infer properties of a probabilistic model from data, can be applied in a flexible and expressive manner. To aid our discussion, we present the Statistical Programming Computable Functions (SPCF), a Turing-complete language that supports discrete and continuous sampling and scoring. SPCF can be seen as the foundational core of many practical universal probabilistic programming languages including Church (Goodman et al., 2008), Anglican (Wood et al., 2014), Venture (Mansinghka et al., 2014), Hakaru (Narayanan and Shan, 2020), Pyro (Bingham et al., 2019), Turing (Ge et al., 2018), and Gen (Cusumano-Towner et al., 2019). We explore the properties of SPCF, setting the stage for the central contribution of this thesis—the design of Markov chain Monte Carlo inference algorithms for probabilistic programming.

## 3.1 Bayesian Machine Learning

Machine Learning studies methods which 'learn' from previously observed data in order to improve the performance of the task at hand. The Bayesian approach for Machine Learning represents our degree of belief by assigning probabilities to each uncertainties and updating them using the observations via Bayes' Law.

### 3.1.1 Bayesian Framework

Given a task and some observed data, the first step in Bayesian Machine Learning is to specify a *probabilistic model* describing the task with random variables that represent

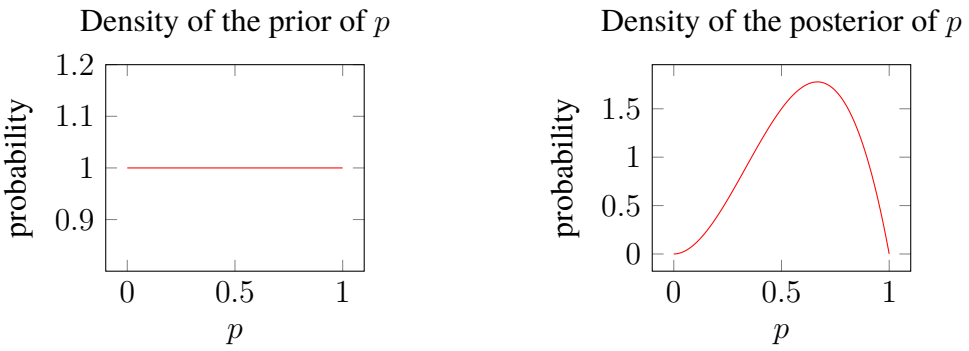


**Figure 3.1:** *Densities of the prior and posterior of $p$*

uncertainties in the model. A probability distribution is then assigned to each random variable, representing our prior belief on these uncertainties (before seeing the data). The product of these distributions is called the *prior* distribution. Note that the prior distribution on the random variables give the probability of a range of models that can describe our task. Give the observed data, we compute how likely the data are generated from each of these models and update the distribution. The updated distribution on the random variables is called the *posterior* distribution. This process of computing the posterior distribution is called *Bayesian inferencing* as it utilises the Bayes' Law.

For example, Bayesian inference can be used to model the bias of an unfair coin. The Bernoulli distribution with parameter $p$ is a suitable probabilistic model for an unfair coin with the unknown random variable $p \in [0,1]$ as the uncertainty. Suppose we believe the bias could be any value from $0$ to $1$. Then, our prior belief on $p$ is the standard uniform distribution $\mathcal{U}(0,1)$. To update the probabilistic model, say tail, head and head are observed from the last three tosses. We compute how likely these tosses might be generated from the Bernoulli distribution with $p$, namely $(1-p)p^2$, and multiply it to the density $\mathbb{1}_{[0,1]}(p)$ of the uniform distribution for $p \in [0,1]$. The resulting posterior distribution on $p$ then has density

$$\mathbb{1}_{[0,1]}(p) \cdot \frac{(1-p)p^2}{\int_0^1 (1-p)p^2 \,\mathsf{Leb}(\mathrm{d}p)} = \mathbb{1}_{[0,1]}(p) \cdot \frac{(1-p)p^2}{1/3 - 1/4} = \mathbb{1}_{[0,1]}(p) \cdot 12(1-p)p^2$$

where the denominator $\int_0^1 (1-p)p^2 \,\mathsf{Leb}(\mathrm{d}p)$ is the normalising constant. Fig. 3.1 gives the densities of the prior and posterior distributions of $p$. We see that after observing three tosses, the posterior gives a better presentation of the unfair coin. It makes sense that $p$ can neither be $0$ nor $1$ as both tail and head are observed. Moreover the right skew of the density shows us that it is more likely the next toss will be a head.

### 3.1.2 Challenges of Bayesian Machine Learning

Though Bayesian inference provides a simple yet compelling framework for Machine Learning, it is not always straightforward to execute. The probabilistic model needs to be defined; the prior distributions on the unknown parameters specified; and the posterior distribution computed. These are no easy tasks.

Given a prior and probabilistic model, the posterior is uniquely defined but finding the exact posterior can be computationally intractable on a probabilistic model with many parameters. Hence approximate methods like Markov chain Monte Carlo and variational inference are used instead to estimate the posterior distribution. We discuss these algorithms in Chapter 4.

By contrast, specifying the probabilistic model and prior distribution are naturally subjective (Neal, 1998) and require domain expertise in the given task. This thesis leaves the discussion of model and prior construction to domain experts, and instead studies probabilistic programming as a tool for specifying probabilistic models.

## 3.2 Statistical PCF

The idea behind universal probabilistic programming (Goodman et al., 2008) is to express probabilistic models in a Turing-complete functional language. For our discussion, we present the Statistical Programming Computable Functions (SPCF), a probabilistic extension of the call-by-value Turing-complete PCF (Scott, 1993; Sieber, 1990) with Real and Boolean ground types. SPCF distils the main concepts of probabilistic programming, making it easy to study the properties of probabilistic programming.

Our SPCF can be seen as adding a discrete sampler to the language given in (Vákár et al., 2019; Mak et al., 2021a) or a simply-typed variant of the language given in (Borgström et al., 2016).

### 3.2.1 Syntax

Types, terms and the typing system of SPCF are presented in Fig. 3.2. Following the convention, we assume there is a countable sequence $\mathcal{V}$ of variables with meta-variables $x, y, z$ and a set $\mathcal{F}$ of primitive functions with meta-variables $f, g, h$. The set of all terms is denoted as $\Lambda$ with meta-variables $M, N, L$, the set of free variables of a term $M$ is denoted as $\mathsf{FV}(M)$ and the set of all closed terms (also called programs) is denoted as $\Lambda^0$. In the interest of readability, we use pseudocode in the style of Python (e.g. Ex. 17) to express SPCF terms.

***Types*** (typically denoted $\sigma, \tau$) and ***terms*** (typically $M, N, L$):

$$
\begin{aligned}
\sigma, \tau &::= \mathsf{R} \mid \mathsf{B} \mid \sigma \Rightarrow \tau \\
M, N, L &::= y \mid \underline{r} \mid \underline{a} && \text{(Variables and constants)} \\
&\mid \lambda y.M \mid M\,N && \text{(Higher-order)} \\
&\mid \underline{f}(M_1, \ldots, M_\ell) && \text{(Primitive functions)} \\
&\mid \mathsf{if}(L, M, N) \mid \mathsf{Y}M && \text{(Branching and recursion)} \\
&\mid \mathsf{normal} \mid \mathsf{coin} \mid \mathsf{score}(M) && \text{(Probabilistic)}
\end{aligned}
$$

***Typing system***:

$$
\frac{y \in \mathcal{V}}{\Gamma \cup \{y : \sigma\} \vdash y : \sigma} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash \underline{r} : \mathsf{R}} \qquad \frac{a \in 2}{\Gamma \vdash \underline{a} : \mathsf{B}}
$$

$$
\frac{\Gamma \cup \{y : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda y.M : \sigma \Rightarrow \tau} \qquad \frac{\Gamma \vdash M : \sigma \Rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}
$$

$$
\frac{\{\Gamma \vdash M_i : \mathsf{R}\}_{i=0}^{n} \qquad \{\Gamma \vdash N_j : \mathsf{B}\}_{j=n+1}^{\ell} \qquad f : \mathbb{R}^n \times 2^{\ell-n} \rightharpoonup G}{\Gamma \vdash \underline{f}(M_1, \ldots, M_n, N_{n+1}, \ldots, N_\ell) : \begin{cases} \mathsf{R} & \text{if } G = \mathbb{R} \\ \mathsf{B} & \text{if } G = 2 \end{cases}}
$$

$$
\frac{\Gamma \vdash L : \mathsf{B} \qquad \Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{if}(L, M, N) : \sigma} \qquad \frac{\Gamma \vdash M : (\sigma \Rightarrow \tau) \Rightarrow (\sigma \Rightarrow \tau)}{\Gamma \vdash \mathsf{Y}M : \sigma \Rightarrow \tau}
$$

$$
\frac{}{\Gamma \vdash \mathsf{normal} : \mathsf{R}} \qquad \frac{}{\Gamma \vdash \mathsf{coin} : \mathsf{B}} \qquad \frac{\Gamma \vdash M : \mathsf{R}}{\Gamma \vdash \mathsf{score}(M) : \mathsf{R}}
$$

**Figure 3.2:** *Syntax of SPCF, where $r, q, p \in \mathbb{R}$, $a, b \in 2$, $x, y, z \in \mathcal{V}$, and $f, g, h \in \mathcal{F}$.*

*Remark* 10 (Church Encodings). Similar to PCF, we can represent pairs and lists in SPCF using Church encoding as follows:

$$
\begin{aligned}
&\mathsf{Pair}(\sigma, \tau) := \sigma \to \tau \to (\sigma \to \tau \to \mathsf{R}) \to \mathsf{R} && \mathsf{List}(\sigma) := (\sigma \to \mathsf{R} \to \mathsf{R}) \to \mathsf{R} \to \mathsf{R} \\
&\langle M, N \rangle \equiv \lambda z.z\,M\,N && [M_1, \ldots, M_\ell] \equiv \lambda f x.f\,M_1(f\,M_2 \ldots (f\,M_\ell\,x))
\end{aligned}
$$

Moreover standard primitives on pairs and lists, such as `proj`, `len`, `append` and `sum`, can be defined easily.

SPCF enriched PCF with

(i) sampling constructs (normal and coin) introducing randomness;

(ii) scoring construct score$(M)$ enabling Bayesian inference; and

(iii) a set $\mathcal{F}$ of primitive functions, extending the expressiveness of the language.

**Primitive Functions** Like (Staton et al., 2016; Staton, 2017), all partial measurable functions of type $\mathbb{R}^n \times 2^{\ell-n} \rightharpoonup \mathbb{R}$ or $\mathbb{R}^n \times 2^{\ell-n} \rightharpoonup 2$ for some $\ell \geq n \in \mathbb{N}$, are assumed to be primitives $\mathcal{F}$ in SPCF. Examples include addition +, division /, comparison <, equality =, cumulative distribution functions (cdf) and probability density functions (pdf) of distributions.

**Continuous Sampler** The continuous sampler normal (or `normal`) of our SPCF language draws from the standard Gaussian distribution $\mathcal{N}$ with mean $0$ and variance $1$, unlike (Culpepper and Cobb, 2017; Wand et al., 2018; Ehrhard et al., 2018; Vákár et al., 2019; Mak et al., 2021a) whose continuous sampler draws from the standard uniform distribution $\mathcal{U}(0,1)$. For our purposes, it is more convenient to include a sampler that that can draw any Real number like the standard normal distribution because many inference algorithms have a Real-valued target density, e.g. Hamiltonian Monte Carlo. The following example shows that our choice of sampler does not restrict nor extend our language as long as suitable primitives are present.

**Example 17.** Distributions specified in SPCF using only the continuous sampler `normal`.

(i) Let `cdfnormal` be the cumulative distribution function (cdf) of the standard normal distribution. Then, the uniform distribution $\mathcal{U}(a,b)$ with endpoints $a$ and $b$ can be described as `def uniform(a,b): return a + cdfnormal(normal)*(b-a)`.

(ii) Say `f` is a function where `f(p)` gives the inverse cumulative density of a distribution at `p`. Then this distribution can be described as `f[uniform/p]`. For instance, the inverse cdf of the exponential distribution (with rate 1) is $f(p) := -\ln(1-p)$ and hence `-ln(1-uniform)` describes the exponential distribution in SPCF.

(iii) Following (Devroye, 1986), the (discrete) Poisson distribution can be specified using the standard uniform distribution

```
def poisson(rate):
  x = 0; p = exp(-rate); s = p
  while s < uniform:
    x += 1; p *= rate/x; s += p
  return x
```

**Discrete Sampler** Like (Danos and Ehrhard, 2011; Ścibior et al., 2017), we choose the fair coin coin as our discrete sampler for its simplicity. Again, this is not limiting as many sophisticated discrete distributions can be expressed using the discrete sampler coin.

**Example 18.** Distributions specified in SPCF using the discrete sample `coin`.

(i) The Bernoulli distribution with probability $p \in [0,1] \cap \mathbb{D}$, where $\mathbb{D} := \{\frac{n}{2^m} \mid n, m \in \mathbb{N}\}$ is the set of all Dyadic numbers, can be specified by

```
def bernoulli(p):
  if p == 0: return False
  elif p == 1: return True
  elif p < 0.5:
    if coin: bernoulli(2*p) else: return False
  else:
    if coin: return True else: bernoulli(2*(p-0.5))
```

(ii) The geometric distribution with rate $p \in [0,1] \cap \mathbb{D}$ can be specified by

```
def geometric(p):
  count = 0;
  while bernoulli(1-p):
    count += 1;
  return count
```

(iii) The binomial distribution with $n \in \mathbb{N}$ trails and probability $p \in [0,1] \cap \mathbb{D}$ can be specified by `sum([1 for i in range(n)if bernoulli(p)])`.

*Remark* 11 (Multiple Samplers). Though many practical PPLs support both continuous and discrete distributions, most purified languages only study one or the other, (Staton, 2017; Ścibior et al., 2017; Zhou et al., 2019) being the exceptions. As shown in Ex. 17, one *can* describe discrete distributions using a continuous sampler. However, some inference algorithms (e.g. Mixed HMC (Zhou, 2020)) apply special treatment to discrete variables. For this reason, we include both samplers in SPCF.

**Example 19.** With recursion, it is easy to express nonparametric models in SPCF. For instance, the random walk experiment can be describe by

```
position = normal; while position < 0: position += normal; position
```

The step size is normally distributed and importantly the number of steps is unbounded.

### 3.2.2 SPCF for Bayesian Inference

SPCF shines when it specifies probabilistic models for Bayesian inference. Recall Bayesian inference is the process of updating a probabilistic model using the likelihood of the observed data. This can be achieved by the scoring construct score($M$) (or `score(c)`), which multiplies the weight of the current execution with the Real number denoted by $M$.

**Coin toss** Consider the coin toss example discussed in Sec. 3.1.1. The task is to infer the bias $p$ of an unfair coin using the result of three toss (tail, head and head). The probabilistic model is the Bernoulli distribution parametrised with a uniformly distributed random variable $p$. To update the model, we input the "score" of each toss to the program by running `score(1-p); score(p); score(p)`. Putting them together, the following pseudocode describes the Bayesian inference on the unfair coin.

```
p = uniform;                    # prior of the bias of the unfair coin
score(1-p); score(p); score(p)  # observe THH upon three tosses
return p                        # posterior
```

**Random walk example** A typical nonparametric model is the one-sided random walk described in (Mak et al., 2021a). The story is as follows. Starting from a random location in $[0, 3]$, Alice passes her destination at $0$ by repeatedly walking forwards or backwards of a distance of at most one. Say the total distance travelled is nosily measured to be $1.1$, where is Alice's starting point? This can be described by the following pseudocode.

```
start = uniform(0,3)                # prior of the starting point
position = start; distance = 0
while position > 0:
  step = uniform(-1,1)
  position += step
  distance += abs(step)
score(pdfnormal(1.1,0.1)(distance)) # observe a total distance of 1.1
return start                        # posterior
```

## 3.3 Properties of SPCF

The simplicity of the SPCF syntax allows us to more readily discuss the theoretical properties of probabilistic programming. Here we present an *operational semantics* of SPCF which gives meaning to a probabilistic program by reducing it to some values. We use this reduction system to show that all closed SPCF terms have a *tree representable* density function, an important assumption of the inference algorithms we will discuss in Chapter 5.

### 3.3.1 Operational Semantics

Operational semantics studies the evaluation of programs. In a probabilistic context, this is typically done by first identifying the source of randomness, which we call the trace space, and then defining a reduction system that specifies how terms are evaluated. By analysing all execution paths of a particular SPCF term, a (value) measure can be defined as the operational semantics of the term.

**Trace Space** Randomness in probabilistic programming is introduced by the samplers, hence the ***sample space*** $(\Omega, \Sigma_\Omega, \mu_\Omega)$ of SPCF is the union of the Real measure space $(\mathbb{R}, \mathcal{B}, \mathcal{N})$ and boolean measure space $(2, \wp(2), \mu_2)$. Formally it is the union measurable space $\Omega := \mathbb{R} \cup 2$ with the union $\sigma$-algebra $\Sigma_\Omega := \sigma(\mathcal{B} \cup \wp(2))$ and the measure $\mu_\Omega(V \cup W) := \mathcal{N}(V) + \mu_2(W)$ where $V \in \mathcal{B}$ and $W \in \wp(2)$.

A ***trace*** is a record of the values sampled in the course of an execution of a SPCF term. Hence, the ***trace space*** $(\mathbb{T}, \Sigma_\mathbb{T}, \mu_\mathbb{T})$ is the union of sample spaces of varying dimensions. Formally it is the list measurable space $\mathbb{T} := \bigcup_{n\in\mathbb{N}} \Omega^n$ with $\sigma$-algebra $\Sigma_\mathbb{T} := \{\bigcup_{n\in\mathbb{N}} U_n \mid U_n \in \Sigma_{\Omega^n}\}$ and measure $\mu_\mathbb{T}(\bigcup_{n\in\mathbb{N}} U_n) := \sum_{n\in\mathbb{N}} \mu_{\Omega^n}(U_n)$. We write traces as lists, e.g. $[-0.2, \mathsf{T}, \mathsf{T}, 3.1, \mathsf{F}]$ and $[]$.

*Remark* 12. Another way of recording the sampled value in the course of an execution of a SPCF term is to have separate records for the values of the continuous and discrete samples. In this case, the trace space becomes $\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \times \bigcup_{m\in\mathbb{N}} 2^m$. We find separating the continuous and discrete samples inconvenient for our inference algorithm as the trace alone cannot tell us the type of samples (discrete or continuous) drawn in the course of an execution of a probabilistic program. Hence follow the more conventional definition of trace space.

**Small-step Reduction** Fig. 3.3 gives a rewrite system of ***configurations***, which are triples of the form $\langle M, w, \boldsymbol{t}\rangle$ where $M$ is a closed SPCF term, $w > 0$ is a ***weight***, and $\boldsymbol{t} \in \mathbb{T}$ is a trace. This is the so-called ***small-step reduction***, which tells us how closed SPCF terms are evaluated. This rewrite system can be seen as a typed variant of the small-step sampling-based operational semantics in (Borgström et al., 2016).

In the rule for normal, a random value $r \in \mathbb{R}$ is generated and recorded in the trace, while the weight remains unchanged: even though the program samples from a normal distribution, the weight does not factor in Gaussian densities as they are already accounted for by the trace measure $\mu_\mathbb{T}$. Similarly, in the rule for coin, a random boolean $a \in 2$ is sampled and recorded in the trace with an unchanged weight. In the rule for score($\underline{r}$),

***Values*** (typically denoted $V$), ***redexes*** (typically $R$) and ***evaluation contexts*** (typically $E$):

$$
\begin{aligned}
V &::= \underline{r} \mid \underline{a} \mid \lambda y.M \\
R &::= \underline{f}(\underline{c_1}, \ldots, \underline{c_\ell}) \mid (\lambda y.M)\, V \mid \mathsf{if}(\underline{a}, M, N) \mid \mathsf{Y}(\lambda y.M) \\
&\quad \mid \mathsf{normal} \mid \mathsf{coin} \mid \mathsf{score}(\underline{r}) \\
E &::= [\,] \mid E\, M \mid (\lambda y.M)\, E \mid \mathsf{if}(E, M, N) \mid \underline{f}(\underline{c_1}, \ldots, \underline{c_{i-1}}, E, M_{i+1}, \ldots, M_\ell) \mid \mathsf{Y}E \\
&\quad \mid \mathsf{score}(E)
\end{aligned}
$$

***Redex contractions***:

$$
\begin{aligned}
\langle \underline{f}(\underline{c_1}, \ldots, \underline{c_\ell}), w, \boldsymbol{t} \rangle &\longrightarrow \begin{cases} \langle \underline{f(c_1, \ldots, c_\ell)}, w, \boldsymbol{t} \rangle & \text{if } (c_1, \ldots, c_\ell) \in \mathsf{Dom}(f), \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\langle (\lambda y.M)\, V, w, \boldsymbol{t} \rangle &\longrightarrow \langle M[V/y], w, \boldsymbol{t} \rangle \\
\langle \mathsf{if}(\underline{a}, M, N), w, \boldsymbol{t} \rangle &\longrightarrow \begin{cases} \langle M, w, \boldsymbol{t} \rangle & \text{if } a, \\ \langle N, w, \boldsymbol{t} \rangle & \text{otherwise} \end{cases} \\
\langle \mathsf{Y}(\lambda y.M), w, \boldsymbol{t} \rangle &\longrightarrow \langle \lambda z.M[\mathsf{Y}(\lambda y.M)/y]\, z, w, \boldsymbol{t} \rangle \qquad \text{(for fresh variable } z) \\
\langle \mathsf{normal}, w, \boldsymbol{t} \rangle &\longrightarrow \langle \underline{r}, w, \boldsymbol{t} +\!\!+ [r] \rangle \qquad \text{(for some } r \in \mathbb{R}) \\
\langle \mathsf{coin}, w, \boldsymbol{t} \rangle &\longrightarrow \langle \underline{a}, w, \boldsymbol{t} +\!\!+ [a] \rangle \qquad \text{(for some } a \in 2) \\
\langle \mathsf{score}(\underline{r}), w, \boldsymbol{t} \rangle &\longrightarrow \begin{cases} \langle \underline{r}, r \cdot w, \boldsymbol{t} \rangle & \text{if } r > 0, \\ \mathsf{fail} & \text{otherwise.} \end{cases}
\end{aligned}
$$

***Evaluation contexts***:

$$
\frac{\langle R, w, \boldsymbol{t} \rangle \longrightarrow \langle \Delta, w', \boldsymbol{t}' \rangle}{\langle E[R], w, \boldsymbol{t} \rangle \longrightarrow \langle E[\Delta], w', \boldsymbol{t}' \rangle} \qquad \frac{\langle R, w, \boldsymbol{t} \rangle \longrightarrow \mathsf{fail}}{\langle E[R], w, \boldsymbol{t} \rangle \longrightarrow \mathsf{fail}}
$$

**Figure 3.3:** *Small-step reduction of SPCF, where $r, q, p \in \mathbb{R}$, $a, b \in 2$, $c \in \mathbb{R} \cup 2$, $x, y, z \in \mathcal{V}$, and $f, g, h \in \mathcal{F}$.*

the current weight is multiplied by $r \in \mathbb{R}$: typically this reflects the likelihood of the current execution given some observed data. Similar to (Borgström et al., 2016) we reduce terms which cannot be reduced in a reasonable way (i.e. scoring with non-positive constants or evaluating functions outside their domain) to fail.

We write $\longrightarrow^+$ for the transitive closure and $\longrightarrow^*$ for the reflexive and transitive closure of the small-step reduction. It is easy to see that every closed SPCF term can either be uniquely expressed in the form $E[R]$ or is a value $V$. Hence all closed SPCF terms can either be evaluated via the small-step reduction or is a value. We say a closed SPCF term $M$ terminates in the value $V$ and weight $w$ with the trace specified by $\boldsymbol{t}$ if $\langle M, 1, [\,] \rangle \longrightarrow^* \langle V, w, \boldsymbol{t} \rangle$.

**Value and Weight Functions** With the small-step reduction system, we can now describe the behaviour of a SPCF term given a particular trace $\boldsymbol{t} \in \mathbb{T}$, namely whether it terminates and if so, in which value and what weight it terminates. These can be specified by the value and weight functions of a term.

First, we construct a measurable space on the set $\Lambda$ of terms. Following (Borgström et al., 2016), we view the set $\Lambda$ of all SPCF terms as $\bigcup_{n,m\in\mathbb{N}} (\mathsf{SK}_{n,m} \times \mathbb{R}^n \times 2^m)$ where $\mathsf{SK}_{n,m}$ is the set of SPCF terms with exactly $n$ Real-valued and $m$ boolean-valued placeholders, which are holes for missing Real or boolean values. The measurable space of terms is equipped with the $\sigma$-algebra $\Sigma_\Lambda$ generated by all open sets in the countable disjoint union topology of the product topology of the discrete topology on $\mathsf{SK}_{n,m}$, the standard topology on $\mathbb{R}^n$ and the discrete topology on $2^m$. Similarly the subspace $\Lambda_v^0$ of closed values inherits the Borel algebra on $\Lambda$.

Given a trace, the ***value function*** $\mathsf{value}_M : \mathbb{T} \to \Lambda_v^0 \cup \{\bot\}$ of a closed SPCF term $M$ returns the value the program $M$ reduces to if it terminates; otherwise $\bot$; and the ***weight function*** $\mathsf{weight}_M : \mathbb{T} \to [0, \infty)$ of $M$ returns the final weight of the corresponding execution of $M$ if it terminates; otherwise zero. Formally:

$$\mathsf{value}_M(\boldsymbol{t}) := \begin{cases} V & \text{if } \langle M, 1, [] \rangle \longrightarrow^* \langle V, w, \boldsymbol{t} \rangle \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathsf{weight}_M(\boldsymbol{t}) := \begin{cases} w & \text{if } \langle M, 1, [] \rangle \longrightarrow^* \langle V, w, \boldsymbol{t} \rangle \\ 0 & \text{otherwise.} \end{cases}$$

It follows readily from (Borgström et al., 2016) that the functions $\mathsf{value}_M$ and $\mathsf{weight}_M$ are measurable. With the value and weight functions, the ***value measure*** $\langle\!\langle M \rangle\!\rangle$ of the closed SPCF term $M$ on $\Lambda_v^0$ is defined as

$$\begin{aligned} \langle\!\langle M \rangle\!\rangle : \Sigma_{\Lambda_v^0} &\longrightarrow [0, \infty) \\ U &\longmapsto \int_{{\mathsf{value}_M}^{-1}(U)} \mathsf{weight}_M \, \mathrm{d}\mu_{\mathbb{T}} \end{aligned}$$

*Remark* 13. A trace is in the support of the weight function if and only if the value function returns a (closed) value when given this trace. i.e. $\mathsf{Supp}(\mathsf{weight}_M) = {\mathsf{value}_M}^{-1}(\Lambda^0)$ for all closed SPCF term $M$.

The value and weight functions give the behaviour of a SPCF term. In particular, the weight function defined here is the density of the target distribution to which an inference algorithm typically samples from. Hence, we now study the properties of the weight function.

*Remark* 14. In this work, we call this the "weight function" when considering semantics following (Culpepper and Cobb, 2017; Vákár et al., 2019; Mak et al., 2021a), and use the notion "density" when referring it in an inference algorithm similar to (Zhou et al., 2019, 2020; Cusumano-Towner et al., 2020).

### 3.3.2 Tree Representable Functions

Notice that not every function of type $\mathbb{T} \to [0, \infty)$ makes sense as a weight function of a SPCF term. Consider the program in Listing 3.1. It executes successfully with the trace $[\mathsf{T}, 0.5, \mathsf{F}]$. This immediately tells us that upon sampling $\mathsf{T}$ and $0.5$, the program must sample at least one extra value, moreover this third sample must be a boolean. In other words, the program does *not* terminate with a trace specified by any proper prefix of $[\mathsf{T}, 0.5, \mathsf{F}]$ such as $[\mathsf{T}, 0.5]$, nor any traces of the form $[\mathsf{T}, 0.5, r]$ for $r \in \mathbb{R}$.

Formally, we call a measurable function $w : \mathbb{T} \to [0, \infty)$ that satisfies

- the ***prefix property*** if whenever $\boldsymbol{t} \in \mathsf{Supp}(w)$ then for all $k < |\boldsymbol{t}|$, we have $\boldsymbol{t}^{1\ldots k} \notin \mathsf{Supp}(w)$; and
- the ***type property*** if whenever $\boldsymbol{t} \in \mathsf{Supp}(w)$ then for all $k < |\boldsymbol{t}|$ and for all $t \in \Omega \setminus \mathsf{Type}(\boldsymbol{t}_{k+1})$[1] we have $\boldsymbol{t}^{1\ldots k} +\!\!+ [t] +\!\!+ \boldsymbol{t}' \notin \mathsf{Supp}(w)$ for any $\boldsymbol{t}' \in \mathbb{T}$

***tree representable (TR)*** because any such function can be represented as a (possibly) infinite but finitely branching tree, which we call *program tree*.

A program tree is an undirected graph where any two nodes are connected by exactly one path, consisting of four types of nodes:

- Real node (represented by a circle) denotes a random variable in $\mathbb{R}$;
- boolean node (represented by a square) denotes a random variable in $\mathbb{2}$;
- decision node (represented by a triangle) determines which one of the two paths the program will take given the values of the Real and boolean nodes before it;
- leaf node (represented by a star) gives the value of the applying the tree representable function $w$ to the list of values given by the Real and boolean nodes from the root.

[1]The ***type*** $\mathsf{Type}(t)$ of a sample $t \in \Omega$ is $\mathbb{R}$ if $t \in \mathbb{R}$ and is $\mathbb{2}$ if $t \in \mathbb{2}$.

**Listing (3.1)** *Pseudocode for Counting*

```
count = 0;
while coin: count += normal;
return count
```

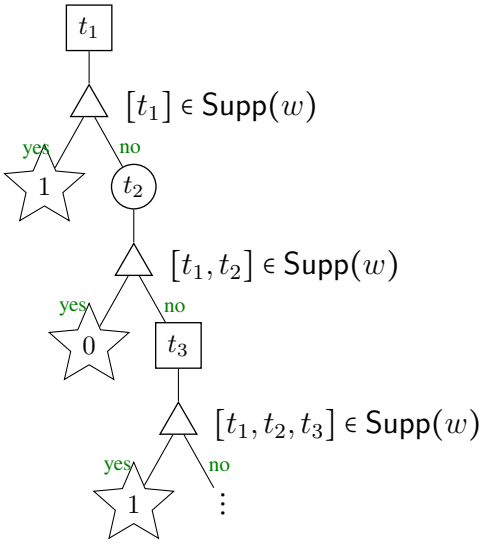


**Figure 3.4:** *Program tree and program*

Every Real and boolean nodes in the program tree must be followed by a decision node determining whether the list $[t_1, \ldots, t_n]$ of random values given by the path from the root to this decision node is in the support. Moreover, the right child of every decision node must be a leaf node returning the value of of $w([t_1, \ldots, t_n])$.

For instance, Fig. 3.4 gives the program tree of the program in Listing 3.1 with weight function

$$w : \mathbb{T} \to [0, \infty)$$
$$w(\boldsymbol{t}) := \begin{cases} 1 & \text{if } \boldsymbol{t} = [\mathsf{T}, r_1, \ldots, \mathsf{T}, r_n, \mathsf{F}] \\ 0 & \text{otherwise.} \end{cases}$$

The following proposition ties SPCF terms and TR functions together.

**Proposition 6.** *Every closed SPCF term has a tree representable weight function.*

*Proof.* Assume $M$ is a closed SPCF term and $\boldsymbol{t} \in \mathsf{Supp}(\mathsf{weight}_M)$. Then, there must exists such run $\langle M, 1, [] \rangle \longrightarrow^* \langle V, w, \boldsymbol{t} \rangle$ for some value $V$ and weight $w > 0$. Assume $|\boldsymbol{t}| > 0$. (Otherwise, the prefix and type properties hold trivially.)

Assume for contradiction that the prefix property does not hold and there is some $k < |\boldsymbol{t}|$, value $V'$ and weight $w' > 0$ where $\langle M, 1, [] \rangle \longrightarrow^* \langle V', w', \boldsymbol{t}^{1\ldots k} \rangle$. Since $\boldsymbol{t}^{1\ldots k}$ is a proper prefix of $\boldsymbol{t}$ and the small-step reduction $\longrightarrow$ is deterministic, we must have $\langle M, 1, [] \rangle \longrightarrow^* \langle V', w', \boldsymbol{t}^{1\ldots k} \rangle \longrightarrow^+ \langle V, w, \boldsymbol{t} \rangle$, which contradicts the fact that $V'$ is a value.

Consider the type property and some $k < |\boldsymbol{t}|$. WLOG assume $\boldsymbol{t}^{k+1} \in \mathbb{R}$. Then we must have

$$\langle M, 1, [] \rangle \longrightarrow^* \langle E[\mathsf{normal}], w', \boldsymbol{t}^{1\ldots k} \rangle \longrightarrow \langle E[\underline{\boldsymbol{t}^{k+1}}], w', \boldsymbol{t}^{1\ldots k+1} \rangle \longrightarrow^* \langle V, w, \boldsymbol{t} \rangle.$$

For any $a \in 2$ and $\boldsymbol{t}' \in \mathbb{T}$, $\langle E[\mathsf{normal}], w', \boldsymbol{t}^{1\ldots k}\rangle \not\to \langle E[a], w', \boldsymbol{t}^{1\ldots k} +\!\!+ [a]\rangle$. Hence $\boldsymbol{t}^{1\ldots k} +\!\!+ [a] +\!\!+ \boldsymbol{t}' \notin \mathsf{Supp}(w)$. □

We will see in Chapter 5 how TR functions are instrumental in the design of the inference algorithm.

### 3.3.3 Almost Sure Termination and Integrability

Since SPCF is Turing-complete, terms can diverge or specify a non-normalisable distribution. Hence, we restrict our class of SPCF terms to those that terminate almost surely and are integrable.

We say a SPCF term $M$ ***terminates almost surely*** if $M$ is closed and $\mu_{\mathbb{T}}(\{\boldsymbol{t} \in \mathbb{T} \mid$ for some $V, w, \langle M, 1, []\rangle \longrightarrow^* \langle V, w, \boldsymbol{t}\rangle\}) = 1$.

The set of traces which a closed SPCF term $M$ terminates on, i.e. $\{\boldsymbol{t} \in \mathbb{T} \mid$ for some $V, w, \langle M, 1, []\rangle \longrightarrow^* \langle V, w, \boldsymbol{t}\rangle\}$, can be understood as the support $\mathsf{Supp}(\mathsf{weight}_M)$ of its weight function or, as discussed in Rem. 13, the traces for which the value function returns a value, i.e. ${\mathsf{value}_M}^{-1}(\Lambda_v^0)$. Hence, $M$ almost surely terminates if and only if $\mu_{\mathbb{T}}(\mathsf{Supp}(\mathsf{weight}_M)) = \mu_{\mathbb{T}}({\mathsf{value}_M}^{-1}(\Lambda_v^0)) = 1$.

The following proposition is used to prove the correctness proof.

**Proposition 7.** *The value measure $\langle\!\langle M\rangle\!\rangle$ of a closed almost surely terminating SPCF term $M$ which does not contain* $\mathsf{score}(-)$ *as a subterm is probabilistic.*

*Proof.* Let $M$ be such a term and let $A$ be the set of traces which $M$ terminates on, i.e. $A = \{\boldsymbol{t} \in \mathbb{T} \mid$ for some $V, w, \langle M, 1, []\rangle \longrightarrow^* \langle V, w, \boldsymbol{t}\rangle\}$. By assumption, $\mu_{\mathbb{T}}(A) = 1$. Since $\mathsf{score}(-)$ is not a subterm in $M$, $\langle M, 1, []\rangle \longrightarrow^* \langle V, w, \boldsymbol{t}\rangle$ implies $w = 1$. So, the weight function is $\mathbb{1}_A$, and the value measure of $M$ of all values must be $\langle\!\langle M\rangle\!\rangle(\Lambda_v^0) = \int_{{\mathsf{value}_M}^{-1}(\Lambda_v^0)} \mathsf{weight}_M \, \mathrm{d}\mu_{\mathbb{T}} = \int_A \mathsf{weight}_M \, \mathrm{d}\mu_{\mathbb{T}} = \int_A 1 \, \mathrm{d}\mu_{\mathbb{T}} = \mu_{\mathbb{T}}(A) = 1$. □

We say a SPCF term $M$ is ***integrable*** if $M$ is closed and its value measure is finite, i.e. $\langle\!\langle M\rangle\!\rangle(\Lambda_v^0) < \infty$;

**Proposition 8.** *Integrable term gives integrable weight function.*

*Proof.* Let $M$ be an integrable SPCF term. Then,

$$\langle\!\langle M\rangle\!\rangle(\Lambda_v^0) = \int_{{\mathsf{value}_M}^{-1}(\Lambda_v^0)} \mathsf{weight}_M \, \mathrm{d}\mu_{\mathbb{T}} < \infty.$$

Hence $\int_{\mathbb{T}} \mathsf{weight}_M \, \mathrm{d}\mu_{\mathbb{T}} < \infty$. □

**Example 20.** The following SPCF terms show that almost sure termination and integrability specify two distinct classes of SPCF terms.

(i) Consider the term $M_1$ defined as `count = 0; while coin: count += 1; return score(2**count)`. It almost surely terminates since it only diverges on the infinite trace $[\mathsf{T}, \mathsf{T}, \dots]$ which has zero probability. However, it is *not* integrable as the value measure applied to all closed values

$$\begin{aligned}\langle\!\langle M_1 \rangle\!\rangle(\Lambda_v^0) &= \int_{\mathsf{value}_{M_1}{}^{-1}(\Lambda_v^0)} \mathsf{weight}_{M_1}\, \mathrm{d}\mu_{\mathbb{T}} = \int_{\{[\mathsf{T},\dots,\mathsf{T},\mathsf{F}]\}} \mathsf{weight}_{M_1}\, \mathrm{d}\mu_{\mathbb{T}} \\ &= \sum_{n=0}^{\infty} \int_{\{[\mathsf{T}]*n+[\mathsf{F}]\}} \mathsf{weight}_{M_1}\, \mathrm{d}\mu_{2^n} = \sum_{n=0}^{\infty} (\frac{1}{2})^{n+1} \cdot 2^n \\ &= \sum_{n=0}^{\infty} \frac{1}{2}\end{aligned}$$

is infinite.

(ii) Consider the term $M_2$ defined as `if coin: Y (lambda x:x)0 else: 1`. Since it reduces to the diverging term `Y (lambda x:x)0` with non-zero probability, it does not terminate almost surely. However, it is integrable, since $\langle\!\langle M_2 \rangle\!\rangle(\Lambda_v^0) = \int_{\{[\mathsf{F}]\}} \mathsf{weight}_{M_2}\, \mathrm{d}\mu_{\mathbb{T}} = \frac{1}{2} < \infty$.

(iii) Consider the term $M_3$ defined as $\mathsf{if}(\mathsf{coin}, M_1, M_2)$. It neither terminates almost surely nor integrable, since $M_1$ is not integrable and $M_2$ is not almost surely terminating.

(iv) All terms considered previously in Ex. 17 to 19 are both almost surely terminating and integrable.

## 3.4 Other Probabilistic Programming Languages

Various programming paradigms have been studied with probabilistic programming including

- first-order (Staton, 2017; Zhou et al., 2019) and higher-order (Borgström et al., 2016; Culpepper and Cobb, 2017; Wand et al., 2018; Vákár et al., 2019; Heunen et al., 2017; Staton et al., 2016; Ehrhard et al., 2018; Mak et al., 2021a; Danos and Ehrhard, 2011; Ścibior et al., 2017; Ehrhard et al., 2014) terms;
- simply-typed (Staton et al., 2016; Staton, 2017; Culpepper and Cobb, 2017; Ehrhard et al., 2018; Mak et al., 2021a; Danos and Ehrhard, 2011; Zhou et al., 2019; Ehrhard et al., 2014), inductively-typed (Vákár et al., 2019; Ścibior et al., 2017) and untyped (Borgström et al., 2016; Wand et al., 2018) systems;

- call-by-value (Borgström et al., 2016; Culpepper and Cobb, 2017; Staton, 2017; Ścibior et al., 2017; Wand et al., 2018; Staton et al., 2016; Vákár et al., 2019; Mak et al., 2021a; Zhou et al., 2019) and call-by-name (Ehrhard et al., 2014, 2018; Danos and Ehrhard, 2011) sampling; and
- recursion (Borgström et al., 2016; Wand et al., 2018; Ehrhard et al., 2015, 2014, 2018; Vákár et al., 2019; Mak et al., 2021a; Danos and Ehrhard, 2011; Ścibior et al., 2017).

The probabilistic constructs these languages support include

- discrete sampler(s) (Danos and Ehrhard, 2011; Ehrhard et al., 2014),
- continuous sampler(s) (Vákár et al., 2019; Mak et al., 2021a; Culpepper and Cobb, 2017; Wand et al., 2018; Ehrhard et al., 2018; Borgström et al., 2016; Staton et al., 2016),
- both discrete and continuous samplers (Zhou et al., 2019; Staton, 2017; Ścibior et al., 2017), and
- scoring (Borgström et al., 2016; Staton et al., 2016; Staton, 2017; Vákár et al., 2019; Culpepper and Cobb, 2017; Wand et al., 2018; Mak et al., 2021a; Ścibior et al., 2017; Zhou et al., 2019).

An important class of these probabilistic programming languages are those extended from the simply-typed Turing-complete Programming Computable Functions (PCF) invented by Scott in 1969 (Plotkin, 1977; Scott, 1993). Examples include the Probabilistic PCF (Ehrhard et al., 2014, 2018), an extension of PCF with a random number sampler and Statistical PCF (Vákár et al., 2019; Mak et al., 2021a), an extension of PCF with a continuous sampler and a scoring construct.

**Operational semantics** There are two main styles of operational semantics in the literature, each gives a different view of the evaluation of probabilistic programs. The *sampling-based operational semantics* first considered in (Kozen, 1979) associates each probabilistic program with a (deterministic) function from the space of randomness to the value space. It has been used to study the contextual equivalence of probabilistic programs (Wand et al., 2018; Culpepper and Cobb, 2017), their differentiability properties (Mak et al., 2021a) and to design inference algorithms (Zhou et al., 2019). Contrasting to sampling-based semantics, *distribution-based operational semantics* interprets every step in the evaluation of a program as a measure directly. It has been used to study full abstraction of probabilistic PCF (Ehrhard et al., 2014) and adequacy of statistical PCF (Vákár et al., 2019). Both styles give the same value measure (Borgström et al., 2016).

**Denotational semantics** We can also reason about probabilistic programs by assigning each type and term to an mathematical object. This is the denotational approach. First-order probabilistic languages such as those considered by (Gordon et al., 2014; Staton, 2017; Staton et al., 2016) interpret types as measurable spaces and term as measurable kernels using the probability monad (Giry, 1982) on the category $\mathbf{Meas}$ of measurable spaces and measurable functions. Moreover, (Staton, 2017) describes probabilistic programs as s-finite kernels and shows that they are commutative. Since the category $\mathbf{Meas}$ of measurable spaces is not Cartesian closed (Aumann, 1961), various models have been developed for higher-order probabilistic languages such as a functor category that embeds $\mathbf{Meas}$ (Staton et al., 2016) and the category of quasi-Borel spaces (Heunen et al., 2017). To support full recursion, the category of probabilistic coherence spaces (Danos and Ehrhard, 2011; Ehrhard et al., 2014) can be used for languages with discrete sampling and the category of measurable cones (Ehrhard et al., 2018) and the category of omega quasi-Borel space (Vákár et al., 2019) for continuous sampling.

**Probabilistic $\lambda$-calculus** Another approach of probabilistic programming is the so-called *probabilistic $\lambda$-calculus* (Saheb-Djahromi, 1978), which is an extension of pure (untyped) $\lambda$-calculus with a probabilistic choice operator $+_p$, indexed by a Real number $p \in [0, 1]$, where $M +_p N$ reduces to $M$ with $p$ probability and $N$ with $1 - p$ probability. Probabilistic Böhm trees (Leventis, 2018) and weighted relational model (Ehrhard et al., 2011) are models of the probabilistic $\lambda$-calculus.

*Emitte lucem tuam et veritatem tuam; ipsa me deduxerunt, et adduxerunt in montem sanctum tuum, et in tabernacula tua.*

— Psalm 42:3

# 4 Markov Chain Monte Carlo

A major challenge in Bayesian inference is to compute the posterior distribution of a probabilistic model. In this chapter, we study the Markov Chain Monte Carlo (MCMC) method of approximating the posterior. Using the recently suggested involutive MCMC framework, we give examples and techniques used in many MCMC algorithms. As case studies, we explore the popular Hamiltonian Monte Carlo (HMC) algorithm and its discontinuous and irreversible variants; and the Reversible Jump MCMC algorithms as instance of the involutive MCMC framework.

## 4.1 An Introduction to Markov Chains Monte Carlo

Borrowed from the famous casino in Monaco, the term Monte Carlo in statistics describes the simulation of random process. Hence Markov chain Monte Carlo (MCMC) refers to the simulation of random process using Markov chains.

A ***Markov chain*** is a sequence $\{x_i\}_{i\in\mathbb{N}}$ of elements, commonly called ***states***, in a measurable space $(S, \Sigma_S)$ where

- the first element $x_0$ is sampled from some probability measure $\mu$ on $(S, \Sigma_S)$ called the ***initial distribution***; and
- given the chain $x_0, \ldots, x_i$, the next element $x_{i+1}$ is sampled from a probability measure $\iota(x_i, \cdot)$ given by a probability kernel $\iota : S \rightsquigarrow S$ commonly called the ***transition kernel***.

**Example 21.** Let $(S, \Sigma_S)$ be a measurable space and $\mu_S$ be a probability measure on $S$.

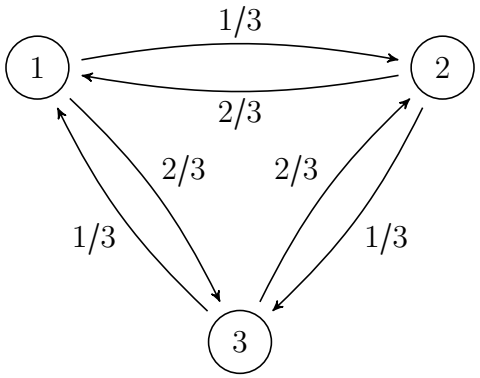


**Figure 4.1:** *Directed graph of the transition kernel $\iota$*

(i) Markov chains generated by the initial distribution $\mu_S$ and the transition kernel $\iota(s, A) := [s \in A]$ are simply sequences consisting of the same element.

(ii) A slightly more interesting example would be Markov chains generated by the initial distribution $\mu_S \times \mathbb{R}$ and the transition kernel $\iota((s, r), A) := [(s, -r) \in A]$, which are sequences of the form $(s, r), (s, -r), (s, r), (s, -r), \ldots$.

It is common to present a transition kernel as a weighted directed graph with states as nodes and the probability of sampling $s'$ from $\iota(s, \cdot)$ given by the weight of an edge from $s$ to $s'$. Edges with zero weight are omitted.

**Example 22.** Consider the measurable space $(\mathbb{3} := \{1, 2, 3\}, \wp(\mathbb{3}))$ with the Dirac distribution $\delta_1$ in $1$ as the initial distribution and transition kernel $\iota : \mathbb{3} \rightsquigarrow \mathbb{3}$ given by

$$\begin{aligned}
\iota(1, A) &:= \frac{1}{3}[2 \in A] + \frac{2}{3}[3 \in A]; \\
\iota(2, A) &:= \frac{2}{3}[1 \in A] + \frac{1}{3}[3 \in A]; \\
\iota(3, A) &:= \frac{1}{3}[1 \in A] + \frac{2}{3}[2 \in A].
\end{aligned}$$

The graph in Fig. 4.1 represents the transition kernel $\iota$. Examples of Markov chains generated using this kernel are $1, 2, 3, 2, 1, 2, 3, 2, 1, \ldots$ and $1, 3, 2, 3, 1, 3, 2, 1, 2, \ldots$. Note the initial distribution $\delta_1$ ensures the chain must start with $1$ and the transition kernel $\iota$ ensures there are no consecutive states of the same kind in the chain.

It is natural to ask whether these generated Markov chains simulate any random process. More specifically, can we determine the occurrence of every state in the Markov chain generated by the same transition kernel?

A transition kernel $\iota : S \rightsquigarrow S$ on a measurable space $(S, \Sigma_S)$ ***preserves*** a probability measure $\pi$ on $S$ if $\int_S \iota(x, A)\ \pi(\mathrm{d}x) = \pi(A)$ for all $A \in \Sigma_S$. We call $\pi$ the ***invariant*** (or equilibrium or stationary) distribution of the transition kernel $\iota$.

*Remark* 15. The invariant distribution of a transition kernel need *not* be unique. For instance, the transition kernel considered in Ex. 21.i preserves *all* probability measures on $S$ and that considered in Ex. 21.ii preserves *all* probability measure of the form $\mu_S \times \nu$ where $\nu$ is a probability measure on $\mathbb{R}$ such that $\mathsf{pdf}_\nu(r) = \mathsf{pdf}_\nu(-r)$ for all $r \in \mathbb{R}$.

**Example 23.** Consider the transition kernel $\iota$ given in Ex. 22. Taking $p_i$ to be the probability of state $i$, we can figure out the invariant distribution of $\iota$ using the following system of equations:

$$p_1 = \frac{2}{3}p_2 + \frac{1}{3}p_3; \qquad p_2 = \frac{1}{3}p_1 + \frac{2}{3}p_3; \qquad p_3 = \frac{2}{3}p_1 + \frac{1}{3}p_2; \qquad 1 = p_1 + p_2 + p_3$$

which has solution $p_1 = p_2 = p_3 = 1/3$. Hence, the transition kernel $\iota$ preserves the probability measure $\pi$ which assigns $1/3$ weight to each state, and every state has the same probability of occurrence in all Markov chains generated by $\iota$.

Let $\pi$ be a probability measure on a measurable space $(S, \Sigma_S)$. A transition kernel $\iota : S \rightsquigarrow S$ is $\pi$***-reversible*** (or simply reversible) if it satisfies ***detailed balanced***: $\int_B \iota(x, A)\ \pi(\mathrm{d}x) = \int_A \iota(x, B)\ \pi(\mathrm{d}x)$ for all $A, B \in \Sigma_S$.

**Proposition 9.** *A $\pi$-reversible transition kernel preserves $\pi$.*

*Proof.* The result follows by considering $B = S$ and the fact that transition kernel is a probability kernel. □

It is often easier to show reversibility rather than invariance. Indeed, most MCMC algorithms deduce their invariance result by proving reversibility.

*Remark* 16. The reverse of Prop. 9 is not true in general. For instance the transition kernel given in Ex. 22 is not $\pi$-reversible since the probability of moving from 1 to 2 is not the same as the reverse.

**Proposition 10.** *Composition of transition kernels preserves invariant distribution.*

*Proof.* Let $T_1, T_2$ be transition kernels on the measure space $(X, \Sigma_X, \mu_X)$ with invariant measure $\pi$. Then for any $U \in \Sigma_X$,

$$\begin{aligned}
\pi(U) &= \int_X T_2(x', U)\ \pi(\mathrm{d}x') \\
&= \int_X T_2(x', U) \int_X T_1(x, \mathrm{d}x')\ \pi(\mathrm{d}x) \\
&= \int_X \int_X T_2(x', U) \cdot T_1(x, \mathrm{d}x')\ \pi(\mathrm{d}x) \\
&= \int_X (T_2 \circ T_1)(x, U)\ \pi(\mathrm{d}x).
\end{aligned}$$

The composition $T_2 \circ T_1$ preserves the distribution $\pi$. □

If an invariant distribution $\pi$ exists, we can then study the circumstances in which the generated Markov chain converges to $\pi$. Recall given a probability kernel $\iota$ on $S$, the probability kernel $\iota^n$ on $S$ represents the application of $n$ stochastic steps given by $\iota$, and $\iota^n(x, A)$ gives the probability of a Markov chain that starts off at $x$ and ends up in $A$ after $n$ 'steps'.

Let $\iota : S \rightsquigarrow S$ be a transition kernel and $\pi$ be a probability measure on a measurable space $(S, \Sigma_S)$.

- The transition kernel $\iota$ is $\pi$***-irreducible*** if for all $x \in S$ and $A \in \Sigma_S$, there is $n \in \mathbb{N}_{>0}$ such that $\pi(A) > 0$ implies $\iota^n(x, A) > 0$.

- The transition kernel $\iota$ is $\pi$***-aperiodic*** if there do not exist $m \geq 2$ and disjoint measurable sets $B_1, \ldots, B_m$ where $\pi(B_1) > 0$ and $x \in B_i$ implies $\iota(x, B_{i+1}) = 1$ for all $i$

- The transition kernel $\iota$ ***converges*** to $\pi$ if for any $x \in S$, $\lim_{n\to\infty} \|\iota^n(x, -) - \pi\| = 0$ with $\|\cdot\|$ denoting the total variation distance.

The following lemma states that as long as the transition kernel is able to move to any positively weighted area (according to $\pi$) in a finite number of steps regardless of its starting point ($\pi$-irreducible) and does not form perpetual loops ($\pi$-aperiodic), it converges to its invariant distribution $\pi$, if it exists.

**Lemma 1** ((Tierney, 1994), Theorem 1 and Corollary 2)**.** *All $\pi$-irreducible and $\pi$-aperiodic transition kernels that preserve $\pi$ also converge to $\pi$.*

**Example 24.** The transition kernel $\iota$ considered in Ex. 21.i defined as $\iota(s, A) := [s \in A]$ is $\pi$-irreducible if the space $S$ is equipped with the trivial $\sigma$-algebra $\Sigma_S := \{\varnothing, S\}$ and $\pi(\varnothing) := 0$ and $\pi(S) := 1$. This is not true in general. For instance, $\iota$ is not irreducible with respect to any probability measure on the Borel measurable space.

**Example 25.** The transition kernel $\iota$ in Ex. 22 which is shown to preserve the probability measure $\pi$ in Ex. 23, is also $\pi$-irreducible and $\pi$-aperiodic. Hence by Lem. 1, it converges to $\pi$.

## 4.2 Involutive MCMC Algorithms

As Markov chains are generated by transition kernels, and random processes can be described using probability distributions, the study of MCMC focuses on the construction of transition kernels (usually in the form of stochastic algorithms) that preserve and converge to a specified target distribution. This is because by setting the target distribution to be the posterior distribution of a probabilistic program, MCMC methods can generate a Markov chain that approximates the posterior.

Many MCMC inferences introduced in the last half century are inspired by the seminal work by Metropolis et al. (Metropolis et al., 1953) and Hastings (Hastings, 1970). Hence, almost all of them have a similar design: perform some stochastic and deterministic steps to the current sample in order to figure out a proposal for the next sample which is accepted with some probability that ensures the transition kernel preserves the target distribution.

The ***involutive Markov chain Monte Carlo*** (iMCMC) algorithm, introduced by (Neklyudov et al., 2020; Cusumano-Towner et al., 2020), is a framework that aims to describe these steps formally. Given a target density $\rho$ on the measure space $(X, \Sigma_X, \mu_X)$, the iMCMC algorithm constructs a Markov chain of samples $\{\boldsymbol{x}^{(i)}\}_{i\in\mathbb{N}}$ in three steps: Given the current sample $\boldsymbol{x}_0$,

1. (Stochastic step) Sample a value $\boldsymbol{v}_0$ on the measure space $(Y, \Sigma_Y, \mu_Y)$ from a probability measure $K(\boldsymbol{x}_0, \cdot)$ given by a probability kernel $K : X \rightsquigarrow Y$;

2. (Deterministic step) Compute the new state $(\boldsymbol{x}, \boldsymbol{v})$ by applying a continuously differentiable involution $\Phi$ on $X \times Y$ to $(\boldsymbol{x}_0, \boldsymbol{v}_0)$;

3. (Accept/reject step) Accept this new sample $\boldsymbol{x}$ with probability

$$\alpha(\boldsymbol{x}_0, \boldsymbol{v}_0) := \min\left\{1, \frac{\rho(\boldsymbol{x}) \cdot \mathsf{pdf}_K(\boldsymbol{x}, \boldsymbol{v})}{\rho(\boldsymbol{x}_0) \cdot \mathsf{pdf}_K(\boldsymbol{x}_0, \boldsymbol{v}_0)} \left|\det\left(\nabla\Phi(\boldsymbol{x}_0, \boldsymbol{v}_0)\right)\right|\right\}$$

   otherwise repeat with the current sample $\boldsymbol{x}_0$.

The iMCMC algorithm advocates for a reduction and separation of the stochastic and deterministic elements in a MCMC algorithm using the ***auxiliary kernel*** $K : X \rightsquigarrow Y$ to capture all randomness and the ***involution*** $\Phi$ on $X \times Y$ to determine the proposal sample. We now discuss the advantages and limits of such treatment.

### 4.2.1 Correctness of iMCMC Algorithm

It will be pointless to discuss the iMCMC algorithm unless it does preserve the target distribution

$$\nu(U) := \frac{1}{Z}\int_U \rho \,\mathrm{d}\mu_X \quad \text{where } Z := \int_X \rho \,\mathrm{d}\mu_X$$

given by the target density $\rho$ on $(X, \Sigma_X, \mu_X)$.

To do this, we first figure out the transition kernel of the algorithm. Let $\boldsymbol{x}_0$ be the current sample. The probability that the next sample $\boldsymbol{x}$ is in some measurable set $A$ is $\alpha(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot [\Phi(\boldsymbol{x}_0, \boldsymbol{v}_0) \in A \times Y]$ if the algorithm accepts the proposal or $(1 - \alpha(\boldsymbol{x}_0, \boldsymbol{v}_0)) \cdot [(\boldsymbol{x}_0, \boldsymbol{v}_0) \in A \times Y]$ if the algorithm rejects the proposal but $\boldsymbol{x}_0 \in A$, given that $\boldsymbol{v}_0$ is sampled from $K(\boldsymbol{x}_0, \cdot)$. Hence, the transition kernel $\iota$ on the parameter space $X$ of the iMCMC algorithm is formally defined to be

$$\begin{aligned}\iota(\boldsymbol{x}_0, A) := \int_Y \Big(&\alpha(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot [\Phi(\boldsymbol{x}_0, \boldsymbol{v}_0) \in A \times Y] \\ &+ (1 - \alpha(\boldsymbol{x}_0, \boldsymbol{v}_0)) \cdot [(\boldsymbol{x}_0, \boldsymbol{v}_0) \in A \times Y]\Big) \, K(\boldsymbol{x}_0, \mathrm{d}\boldsymbol{v}_0)\end{aligned}$$

for all $\boldsymbol{x}_0 \in X$ and $A \in \Sigma_X$.

A nice thing about iMCMC is that there are *hardly any* conditions for this correctness result. The following proposition shows that as long as the input kernel $K$ is a probability kernel and the input function $\Phi$ is indeed involutive, the iMCMC algorithm is reversible and hence preserves the target distribution.

**Proposition 11** ((Neklyudov et al., 2020), Proposition 2)**.** *The Markov chain generated by the transition kernel $\iota$ is $\nu$-reversible.*

**Corollary 1.** *The iMCMC algorithm preserves the target distribution.*

With Cor. 1, the correctness proof of any MCMC algorithm can be simplified to a formulation of the algorithm as an instance of the iMCMC algorithm.

### 4.2.2 Pseudocode of iMCMC Algorithm

Some might rightly argue that iMCMC is *too* general in the sense that formulating a sophisticated MCMC algorithm as an instance of iMCMC would result in unnecessarily complicated auxiliary kernel and involution. This is indeed the case for some MCMC algorithms given in Appendix B of (Neklyudov et al., 2020). Therefore, like (Cusumano-Towner et al., 2020), we specify all MCMC algorithms (including the iMCMC algorithm) using the SPCF language introduced in Chapter 3.

**Listing 4.1:** *Pseudocode of the iMCMC algorithm*

```
def iMCMC(x0):
  v0 = auxkernel(x0)                          # stochastic step
  (x,v) = involution(x0,v0)                   # deterministic step
  return x if uniform < min{1, w(x)/w(x0) *   # accept/reject step
                              pdfauxkernel(x,v)/pdfauxkernel(x0,v0) *
                              absdetjacinv(x0,v0)}
           else x0
```

**Listing 4.2:** *Pseudocode of the MH algorithm*

```
def MH(x0):
  v0 = proposal(x0)                           # stochastic step
  (x,v) = (v0,x0)                             # deterministic step
  return x if uniform < min{1, w(x)/w(x0) *   # accept/reject step
                              pdfproposal(x,v)/pdfproposal(x0,v0)}
           else x0
```

The `iMCMC` function in Listing 4.1 is an implementation of the iMCMC algorithm in SPCF. We assume there are SPCF programs `auxkernel` of type `X -> Y` that implements the auxiliary kernel $K : X \rightsquigarrow Y$; `pdfauxkernel` of type `X*Y -> R` that implements the probability density function $\mathrm{pdf}_K : X \times Y \to \mathbb{R}$ of the auxiliary kernel; `involution` of type `X*Y -> X*Y` that implements the involution $\Phi$ on $X \times Y$; `absdetjacinv` of type `X*Y -> R` that implements the absolute value of the Jacobian determinant of $\Phi$; and `w` of type `X -> R` that implements the target density $\rho$.

*Remark* 17. Similar to many MCMC algorithms, the implementation `iMCMC` in Listing 4.1 assumes the existence of both a sampler `auxkernel` and a probability density function `pdfauxkernel` of the auxiliary kernel.

The term `absdetjacinv` is not strictly necessary in a programming language that supports the computation of derivatives of deterministic programs.

### 4.2.3 Unified View of MCMC Algorithms

As discussed at the start of this section, Metropolis and Hastings played key roles in the development of MCMC methods for Bayesian inference. The algorithm that bears both their names is the classic *Metropolis-Hastings* (MH) sampler. It can be implemented as `MH` in Listing 4.2.

It is clear that the `MH` function is identical to the `iMCMC` function in Listing 4.1 with `auxkernel` replaced by the proposal distribution `proposal` and `involution` replaced

by a swap function, as the Jacobian determinant of a swap is always one. Cor. 1 tells us that `MH` indeed preserves the target distribution.

## 4.3 Techniques on iMCMC Algorithms

Viewing MCMC algorithms through the iMCMC framework helps distil the techniques employed. In this section, we generalise and discuss some of the techniques identified in (Neklyudov et al., 2020) that aim to improve the flexibility and efficiency of iMCMC samplers for the target distribution given by a density $\rho$ on $(X, \Sigma_X, \mu_X)$.

### 4.3.1 State-dependent iMCMC Mixture

Perhaps there is a range of iMCMC samplers suitable for different area in the parameter space. The following technique allows us to form a 'mixture' of these samplers. The key is to randomly choose one of the samplers in such a way that the 'mixture' preserves the target distribution.

Given a family of iMCMC algorithms, indexed by $a \in A$, each with auxiliary kernel $K_a : X \rightsquigarrow Y$ and involution $\Phi_a$ on $X \times Y$, a ***State-dependent iMCMC Mixture*** generates a new sample from the current sample $\boldsymbol{x}_0$ by drawing a value $a$ from a probability measure $K_M(\boldsymbol{x}_0, \cdot)$ on $A$ given by a probability kernel $K_M : X \rightsquigarrow A$, and then proposing the state $(\boldsymbol{x}, \boldsymbol{v})$ which is the result of applying the involution $\Phi_a$ to $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ where $\boldsymbol{v}_0$ is sampled from $K_a(\boldsymbol{x}_0, \cdot)$ (as in the iMCMC algorithm) and is accepted with probability

$$\min\left\{1, \frac{\rho(\boldsymbol{x}) \cdot \mathsf{pdf}_{K_a}(\boldsymbol{x}, \boldsymbol{v}) \cdot \mathsf{pdf}_{K_M}(\boldsymbol{x}, a)}{\rho(\boldsymbol{x}_0) \cdot \mathsf{pdf}_{K_a}(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot \mathsf{pdf}_{K_M}(\boldsymbol{x}_0, a)} \left|\det\left(\nabla \Phi_a(\boldsymbol{x}_0, \boldsymbol{v}_0)\right)\right|\right\}.$$

**Pseudocode** This sampler can be implemented in SPCF as the `MixtureiMCMC` function in Listing 4.3. (Terms specific to this technique are highlighted.) We assume there are SPCF programs `mixkernel` of type `X -> A` that implements the mixture kernel $K_M : X \rightsquigarrow A$; and for each $a \in A$, `auxkernel(a)` that implements the auxiliary kernel $K_a$ with pdf implemented by `pdfauxkernel(a)` and `involution(a)` that implements the involution $\Phi_a$ with the absolute value of the Jacobian determinant of $\Phi_a$ implemented by `absdetjacinv(a)`.

*Remark* 18. Mixture distribution (Trick 1 in (Neklyudov et al., 2020)) can be formed using `MixtureiMCMC` if the involution stays the same for each $a \in A$; similarly a mixture of involutions (Trick 2) is simply `MixtureiMCMC` where the same auxiliary kernel is used for each $a \in A$.

**Listing 4.3:** *Pseudocode of the State-dependent iMCMC Mixture algorithm*

```
def MixtureiMCMC(x0):
  a = mixkernel(x0)                              # mixture step
  v0 = auxkernel(a)(x0)                          # stochastic step
  (x,v) = involution(a)(x0,v0)                   # deterministic step
  return x                                       # accept/reject step
    if uniform < min{1, w(x)/w(x0) *
                        pdfmixkernel(x,a)/pdfmixkernel(x0,a) *
                        pdfauxkernel(a)(x,v)/pdfauxkernel(a)(x0,v0)*
                        absdetjacinv(a)(x0,v0)}
    else x0
```

**Listing 4.4:** *Pseudocode for* `mixauxkernel` *and* `mixinvolution`

```
def mixauxkernel(x0):
  a = mixkernel(x0)
  v0 = auxkernel(a)(x0)
  return (a,v0)

def mixinvolution(x0,a,v0):
  (x,v) = involution(a)(x0,v0)
  return (x,a,v)
```

**Correctness** We show that the State-dependent iMCMC Mixture preserves the target distribution by formulating `MixtureiMCMC` as an instance of `iMCMC` in Listing 4.1. This means specifying the terms `auxkernel` and `involution` and arguing that the resulting `iMCMC` function is equivalent to `MixtureiMCMC`.

The SPCF terms `mixauxkernel` and `mixinvolution` given in Listing 4.4 should suffice. Notice that the density of `mixauxkernel` at `(x,a,v)` is equal to the product of `pdfmixkernel(a,x)` and `pdfauxkernel(a)(x,v)`, so the `iMCMC` function with `auxkernel` replaced by `mixauxkernel` and `involution` replaced by `mixinvolution` is equivalent to `MixtureiMCMC`. By Cor. 1, the State-dependent iMCMC Mixture given by `MixtureiMCMC` preserves the target distribution.

### 4.3.2 Direction iMCMC Algorithm

It is sometimes difficult to specify an involution that explores the parameter space. The following technique tells us that a bijection is enough. The key is to add an additional direction variable $d_0$ to the sampler.

Given a non-involutive bijection $f$ on the state space $X \times Y$, the ***Direction iMCMC*** algorithm generates the next sample by following the standard iMCMC algorithm with

**Listing 4.5:** *Pseudocode of the Direction iMCMC algorithm*

```
def DirectioniMCMC(x0):
  d0 = normal                                  # direction step
  v0 = auxkernel(x0)                           # stochastic step
  (x,v) = bijection(d0)(x0,v0)                 # deterministic step
  d = -d0
  return x if uniform < min{1,w(x)/w(x0) *     # accept/reject step
                            pdfauxkernel(x,v)/pdfauxkernel(x0,v0) *
                            absdetjacbij(d0)(x0,v0)}
          else x0
```

**Listing 4.6:** *Pseudocode for* `dirauxkernel` *and* `dirinvolution`

```
def dirauxkernel(x0)
  d0 = normal
  v0 = auxkernel(x0)
  return (d0,v0)

def dirinvolution(x0,(d0,v0))
  (x,v) = bijection(d0)(x0,v0)
  d = -d0
  return (x,(d,v))
```

the involution replaced by either $f$ or its inverse $f^{-1}$, depending on the sign of a random normally-distributed direction variable $d_0 \in \mathbb{R}$.

**Pseudocode** This algorithm can be expressed in SPCF as the `DirectioniMCMC` function in Listing 4.5. (Terms specific to this technique are highlighted.) In addition to the SPCF terms assumed in `iMCMC`, we assume there is a SPCF term `bijection(d)` which implements the bijection $f$ when `d` is positive and the inverse $f^{-1}$ otherwise and `absdetjacbij(d)` that implements the absolute value of the Jacobian determinant of the bijection $f$ if `d` is positive and the inverse $f^{-1}$ otherwise.

*Remark* 19. `DirectioniMCMC` corresponds to Trick 3 in (Neklyudov et al., 2020).

**Correctness** We show that the Direction iMCMC algorithm preserves the target distribution by formulating `DirectioniMCMC` as an instance of `iMCMC` in Listing 4.1 with a specification of the terms `auxkernel` and `involution`.

The SPCF terms `dirauxkernel` and `dirinvolution` in Listing 4.6 would work. The density of `dirauxkernel(x)` at `(d,v)` is equal to the product of `pdfnormal(d)` and `pdfauxkernel(x,v)` and the absolute value of the Jacobian determinant of `dirinvolution` at `(d,x,v)` is equal to `absdetjacbij(d)(x,v)`. Hence the `iMCMC` function with `auxkernel`

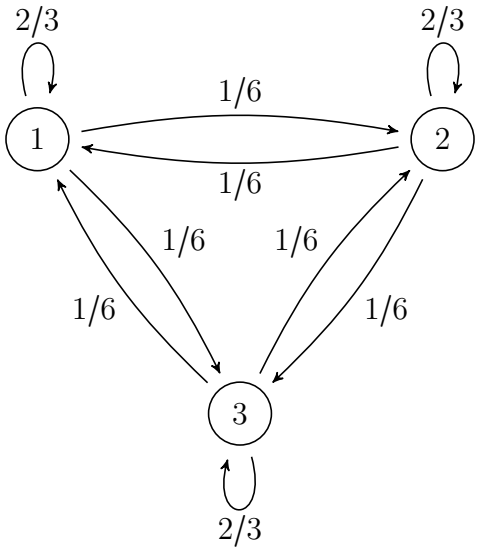


**Figure 4.2:** *Directed graph of the transition kernel described in Ex. 26*

replaced by `dirauxkernel` and `involution` replaced by `dirinvolution` is equivalent to `DirectioniMCMC`. By Cor. 1, the Direction iMCMC algorithm given by `DirectioniMCMC` preserves the target distribution.

**Example 26.** Let's consider the `DirectioniMCMC` algorithm with `auxkernel` given by the probability kernel discussed in Ex. 22 and `bijection` given by the function

$$f(s, v) := ((s \mod 3) + 1, v).$$

Say we want to simulate the target distribution with density $\rho(s) := 1/3$ on the measurable space $(\mathbb{3} := \{1, 2, 3\}, \mathcal{P}(\mathbb{3}))$. What would the transition kernel of `DirectioniMCMC` look like?

Say the current sample `x0` is `2`. If the direction variable `d0` is drawn to be positive and the auxiliary variable `v0` to be `1`, then the proposal state `(x,v)` becomes `(3,1)` and `3` is accepted with half probability. This is the only possible way to move from `2` to `3` and has probability 1/6.

Similar arguments can be used to work out the transition kernel of the `DirectioniMCMC` algorithm which is given as a graph in Fig. 4.2. Note the transition kernel is reversible and hence preserves the target distribution given by the target density $\rho$.

### 4.3.3 Persistent iMCMC Algorithm

It is known that irreversible transition kernels (those that do not satisfy detailed balance) have better mixing times, i.e. converge more quickly to the target distribution, compared

to reversible ones. However, most MCMC samplers (including iMCMC) are reversible. The following technique gives us a method to transform reversible iMCMC algorithms to irreversible ones that still preserve the target distribution. The key is to compose the iMCMC algorithm with some transition kernel so that the resulting algorithm does not satisfy detailed balance but still preserves the invariant distribution.

The ***Persistent iMCMC*** algorithm is an MCMC algorithm similar to the Direction iMCMC algorithm in which the stochastic step (given by the auxiliary kernel) and the deterministic step (given by the bijection) both depend on the value of a direction variable $d_0$. But instead of sampling a new direction in each iteration, Persistent iMCMC keeps track of and negates the previous direction $d_0$ strategically to make the resulting algorithm irreversible.

Given the current sample $\boldsymbol{x}_0$ and direction $d_0$, a proposal state $(\boldsymbol{x}, \boldsymbol{v})$ is obtained by applying a bijection $f$ or its inverse $f^{-1}$ depending on the value of $d_0$ to $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ where $\boldsymbol{v}_0$ is sampled from a probability measure conditioned on $\boldsymbol{x}_0$ and $d_0$. Then $\boldsymbol{x}$ is accepted with an appropriate probability alongside an *unchanged* direction $d_0$ or rejected and the current sample $\boldsymbol{x}_0$ is returned with a *negated* direction $-d_0$.

**Pseudocode** The algorithm can be expressed in SPCF as `PersistentiMCMC` in Listing 4.7. (Terms specific to this technique are highlighted.) In addition to the SPCF terms in `DirectioniMCMC`, we assume there is a SPCF term `auxkernel` that implements the auxiliary kernel $K_P : X \times \mathbb{R} \rightsquigarrow Y$ with pdf implemented by `pdfauxkernel`. Note that the Markov chain generated by the `PersistentiMCMC` algorithm has state space $X \times \mathbb{R}$ instead of $X$, which can easily be marginalised by taking the first component of each state in the chain.

**Correctness** We first show that the Persistent iMCMC algorithm preserves the distribution $\nu \times \mathcal{N}$ on $X \times \mathbb{R}$ by formulating the `PersistentiMCMC` function as a composition of an instance of the `iMCMC` function (by specifying the terms `auxkernel` and `involution`) and the transition kernel $\iota$ defined by $\iota((s, r), A) := [(s, -r) \in A]$ in Ex. 21.ii.

Consider the `iMCMC` algorithm with `involution` replaced by `perinvolution` in Listing 4.8 (and the `auxkernel` stays the same.) The resulting `iMCMC` is almost equivalent to `PersistentiMCMC` except with a negated direction. Hence, composing this instance of `iMCMC` with the transition kernel $\iota$ gives us a sampler that is equivalent to `PersistentiMCMC`. By construction, the composition preserves the distribution $\nu \times \mathcal{N}$ on $X \times \mathbb{R}$. It is easy to see that the marginalised Markov chain obtained by taking the $X$-component of that generated by iterating `PersistentiMCMC` preserves the target distribution $\nu$.

**Listing 4.7:** *Pseudocode of the Persistent iMCMC algorithm*

```
def PersistentiMCMC(x0,d0):
  v0 = auxkernel(x0,d0)                          # stochastic step
  (x,v) = bijection(d0)(x0,v0)                   # deterministic step
  d = -d0
  return (x,-d)                                  # accept/reject step
    if uniform < min{1,w(x)/w(x0) *
                     pdfauxkernel((x,d),v)/pdfauxkernel((x0,d0),v0) *
                     absdetjacbij(d0)(x0,v0)}
    else (x0,-d0)
```

**Listing 4.8:** *Pseudocode for* `perinvolution`

```
def perinvolution((x0,d0),v0)
  (x,v) = bijection(d0)(x0,v0)
  d = -d0
  return ((x,d),v)
```

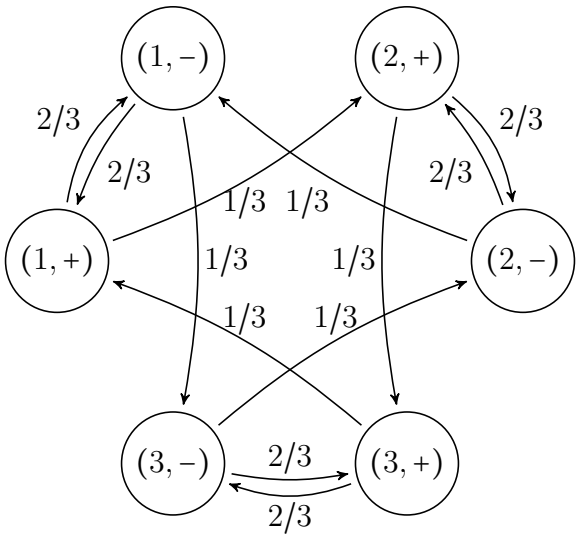


**Figure 4.3:** *Directed graph of the transition kernel described in Ex. 27*

**Example 27.** Follow the setup in Ex. 26 but with the `PersistentiMCMC` algorithm in Listing 4.7. The transition kernel on $\mathbb{3} \times \mathbb{R}$ can be illustrated as the graph in Fig. 4.3. (We simplified the graph to give the sign of the Real-component of each state.)

It is easy to check that the resulting transition kernel preserves the target distribution given by the target density $\rho(s, r) := 1/6$ but is irreversible as there is non-zero probability to move from state $(1, +)$ to $(2, +)$ but none the other way round.

*Remark* 20. Persistent iMCMC corresponds to Tricks 5 and 6 in (Neklyudov et al., 2020).

## 4.4 Case Study: Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo (also known as Hybrid Monte Carlo) (HMC) algorithm is a popular MCMC inference that uses Hamiltonian dynamics to sample from a target distribution on the measure space $(\mathbb{R}^n, \mathcal{B}_n, \mathsf{Leb}_n)$. In this case study, we explore the standard HMC algorithm as presented in (Neal, 2011) and its variants, namely the Generalised HMC (Horowitz, 1991) and Look Ahead HMC (Sohl-Dickstein et al., 2014) (equivalent to the Extra Chance Generalised HMC (Campos and Sanz-Serna, 2015)) using the iMCMC framework and techniques introduced in Sec. 4.2 and 4.3.

### 4.4.1 The Hamiltonian Monte Carlo Algorithm

Let $\nu$ be the target distribution on $\mathbb{R}^n$ given by a continuously differentiable target density $\rho : \mathbb{R}^n \to [0, \infty)$. The HMC algorithm generates a Markov chain $\{(\boldsymbol{q}_i, \boldsymbol{p}_i)\}_{i \in \mathbb{N}}$ with the invariant distribution $\pi$ on the measure space $(\mathbb{R}^n \times \mathbb{R}^n, \mathcal{B}_n \times \mathcal{B}_n, \mathsf{Leb}_n \times \mathsf{Leb}_n)$ given by the probability density function

$$\zeta(\boldsymbol{q}, \boldsymbol{p}) := \frac{1}{Z} \exp\left(-U(\boldsymbol{q}) - K(\boldsymbol{p})\right)$$

where $U : \mathbb{R}^n \to \mathbb{R}$ is the *potential energy*, given by $U(\boldsymbol{q}) := -\log \rho(\boldsymbol{q})$, and $K : \mathbb{R}^n \to \mathbb{R}$ is the *kinetic energy*, given by $K(\boldsymbol{p}) := -\log \mathsf{pdf}_D(\boldsymbol{p})$ where $D$ is the *momentum distribution*, a probability measure on $(\mathbb{R}^n, \mathcal{B}_n, \mathsf{Leb}_n)$ typically chosen to be the (unnormalised) multivariate Gaussian distribution $\mathcal{N}_n(0, \boldsymbol{I}_n)$, in which case $K(\boldsymbol{p}) := \boldsymbol{p}^\intercal \boldsymbol{p}/2$, and $Z$ is the normalising constant.

Clearly the target distribution $\nu$ on $\mathbb{R}^n$ is the $q$-marginal of $\pi$ on $\mathbb{R}^n \times \mathbb{R}^n$. Hence if the Markov chain generated by the HMC algorithm indeed preserves its invariant distribution $\pi$ as defined above, the corresponding marginal chain $\{\boldsymbol{q}_i\}_{i \in \mathbb{N}}$ preserves the target distribution $\nu$.

**Hamiltonian Dynamics** To generate a proposal with a high acceptance probability, HMC tracks the *Hamiltonian* motion of a particle on the surface defined by the potential energy $U$ with momentum defined by the kinetic energy $K$. Intuitively, this makes sense as an area with a high probability will have low potential energy and is more likely to be visited by the simulated particle.

The Hamiltonian $H : \mathbb{R}^n \times \mathbb{R}^n \to [0, \infty)$ of a system is defined quite simply to be the sum of the potential and kinetic energies, i.e.

$$H(\boldsymbol{q}, \boldsymbol{p}) \coloneqq U(\boldsymbol{q}) + K(\boldsymbol{p}).$$

The trajectories $\{(\boldsymbol{q}^t, \boldsymbol{p}^t)\}_{t \geq 0}$, where $\boldsymbol{q}^t$ and $\boldsymbol{p}^t$ are the position and momentum of the particle at time $t$ respectively, defined by the Hamiltonian $H$, can be determined by the Hamiltonian equations

$$\frac{\partial q_i(t)}{\partial t} = \frac{\partial H}{\partial p_i}(q(t), p(t)) \quad \text{and} \quad \frac{\partial p_i(t)}{\partial t} = -\frac{\partial H}{\partial q_i}(q(t), p(t)) \qquad \text{for } i = 1, \ldots, n$$

with initial conditions $(q(0), p(0)) = (\boldsymbol{q}^0, \boldsymbol{p}^0)$. Since computers cannot simulate continuous motions like Hamiltonian, the equations of motion are typically numerically integrated by the leapfrog method:

$$\phi^M_{\epsilon/2} \circ \phi^P_{\epsilon} \circ \phi^M_{\epsilon/2}$$

where $\phi^M_k(\boldsymbol{q}, \boldsymbol{p}) \coloneqq (\boldsymbol{q}, \boldsymbol{p} - k\nabla U(\boldsymbol{q}))$ is the $k$-sized momentum step; $\phi^P_k(\boldsymbol{q}, \boldsymbol{p}) \coloneqq (\boldsymbol{q} + k\nabla K(\boldsymbol{p}), \boldsymbol{p})$ is the $k$-sized position step; and $\epsilon$ is the time step.

We define the $(L, \epsilon)$-***leapfrog*** (or simply leapfrog) function $\mathbf{L}$ to be a map on $\mathbb{R}^n \times \mathbb{R}^n$ such that $\mathbf{L}(\boldsymbol{q}, \boldsymbol{p})$ is the result of $L$ iterations of the above leapfrog method with initial condition $(\boldsymbol{q}^0, \boldsymbol{p}^0) \coloneqq (\boldsymbol{q}, \boldsymbol{p})$, i.e. $\mathbf{L} \coloneqq (\phi^M_{\epsilon/2} \circ \phi^P_{\epsilon} \circ \phi^M_{\epsilon/2})^L$.

**Proposition 12.** *The leapfrog function* $\mathbf{L}$ *is bijective and has inverse* $\mathbf{L}^{-1} = M \circ \mathbf{L} \circ M$ *where* $M(\boldsymbol{q}, \boldsymbol{p}) \coloneqq (\boldsymbol{q}, -\boldsymbol{p})$*, and the leapfrog function* $\mathbf{L}$ *and its inverse are volume preserving (i.e.* $\mathbf{L}_* \mathsf{Leb}_{2n} = \mathsf{Leb}_{2n}$ *and* $\mathbf{L}^{-1}{}_* \mathsf{Leb}_{2n} = \mathsf{Leb}_{2n}$*).*

*Proof.* It is easy to see that $(\phi^M_k)^{-1} = M \circ \phi^M_k \circ M$ and $(\phi^P_k)^{-1} = M \circ \phi^P_k \circ M$. Hence,

$$\mathbf{L}^{-1} = \left((\phi^M_{\epsilon/2})^{-1} \circ (\phi^Q_{\epsilon})^{-1} \circ (\phi^M_{\epsilon/2})^{-1}\right)^L = M \circ \left(\phi^M_{\epsilon/2} \circ \phi^Q_{\epsilon} \circ \phi^M_{\epsilon/2}\right)^L \circ M = M \circ \mathbf{L} \circ M.$$

Moreover, $\phi^M_k$, $\phi^P_k$ and the momentum flip $M$ are sheaf transformations and preserves measure on $\mathbb{R}^{2n}$, i.e. $\phi^M_k(D)$, $\phi^P_k(D)$, $M(D)$ and $D$ all have the same weight w.r.t. $\mathsf{Leb}_{2n}$ for all measurable set $D$ in $\mathbb{R}^{2n}$. Hence,

$$\begin{aligned}(\mathbf{L}_* \mathsf{Leb}_{2n})(D) &= \mathsf{Leb}_{2n}(\mathbf{L}^{-1}(D)) = \mathsf{Leb}_{2n}(M(\mathbf{L}(M(D)))) = \mathsf{Leb}_{2n}(D) \\ (\mathbf{L}^{-1}{}_* \mathsf{Leb}_{2n})(D) &= \mathsf{Leb}_{2n}(\mathbf{L}(D)) = \mathsf{Leb}_{2n}(D)\end{aligned}$$

and $\mathbf{L}$ and $\mathbf{L}^{-1}$ are volume preserving. □

**Pseudocode of HMC Algorithm** We can formulate the HMC algorithm as an instance of the `DirectioniMCMC` algorithm in Listing 4.5, with the momentum distribution (typically the $n$-dimensional multivariate Gaussian distribution) `[normal]*len(q)` as the auxiliary kernel, the leapfrog function `leapfrog` as the bijection, and the pdf `w` of the target distribution $\nu$ on $\mathbb{R}^n$.

The resulting `HMC` algorithm in Listing 4.9 can then given the current state `q0` returns a new state as follows.

1. (Direction step) Sample a direction `d0` from `normal`.
2. (Stochastic step) Generate a (fresh) $n$-dimensional momentum vector `p0` from the momentum distribution `[normal]*len(q0)`.
3. (Deterministic step) Find `q` by evolving Hamilton's equation with initial condition `(q0,p0)` and the leapfrog function `leapfrog` if `d0` is positive, otherwise with its inverse `momflip leapfrog momflip`.
4. (Accept/reject step) Return `q` as the next state with probability

$$\min\left\{1, \frac{\rho(\boldsymbol{q}) \cdot \mathsf{pdf}_{\mathcal{N}_n}(\boldsymbol{p})}{\rho(\boldsymbol{q}_0) \cdot \mathsf{pdf}_{\mathcal{N}_n}(\boldsymbol{p}_0)}\right\}.$$

   Otherwise return the current state `q0`.

Since the leapfrog step $\mathbf{L}$ and its inverse $\mathbf{L}^{-1}$ are volume preserving (Prop. 12), the absolute value of the Jacobian determinant of $\mathbf{L}$ and its inverse $\mathbf{L}^{-1}$ is always one. Hence, the term `absdetjacbij(d0)(q0,p0)` for all `(q0,p0)` and `d0` not necessary in the accept/reject step.

*Remark* 21. Most presentations of the HMC algorithm (Neal, 2011; Cances et al., 2007) do not contain the direction variable `d0`, miss the direction step and use the involution $M \circ \mathbf{L}$ in the deterministic step to find the proposal state. This undoubtedly makes for a cleaner presentation of HMC. However, as we will see in Sec. 4.4.3, viewing HMC as an instance of `DirectioniMCMC` makes it easier to extend.

**Correctness of HMC** Since `leapfrog` is bijective and the momentum distribution is a probability measure, the statement that the HMC algorithm preserves the distribution $\pi$ is induced from the correctness of the Direction iMCMC algorithm given in Sec. 4.3.2.

**Listing 4.9:** *Pseudocode of the HMC algorithm with the SPCF terms* `HMC`, `leapfrog` *and* `momflip`

```
def HMC(q0):
  d0 = normal                                          # direction
  p0 = [normal]*len(q0)                                # stochastic
  (q,p) = leapfrog(q0,p0) if d0 > 0                    # deterministic
          else momflip(leapfrog(momflip(q0,p0)))
  return q if uniform < min{1, w(q) / w(q0) *          # accept/reject
                               pdfnormal(p) / pdfnormal(p0)}
              else q0

def leapfrog(q0,p0):                       # implements the function L
  (q,p) = (q0,p0)
  for i in range(L):
    p = p - ep/2 * grad(U)(q)
    q = q + ep * grad(K)(p)
    p = p - ep/2 * grad(U)(q)
  return (q,p)

def momflip(q0,p0): return (q0,-p0)        # implements the function M
```

### 4.4.2 Discontinuous Hamiltonian Monte Carlo

The HMC algorithm relies on the conservation of Hamiltonian energy to propose samples with high acceptance ratio. However, energy is not preserved if the target density function $\rho$ is not continuously differentiable, which is often the case for probabilistic program containing branching or recursion commands. As a result, applying HMC on these programs has low acceptance ratio.

Instead Nishimura et al. (2020) presents the *Discontinuous Hamiltonian Monte Carlo* (DHMC) sampler, which pairs the discontinuous variables with the Laplace momentum instead of the Gaussian momentum, preserving the energy and thus having a high acceptance ratio.

Say the target density $\rho : \mathbb{R}^n \to [0, \infty)$ is not differentiable on the $j$-th coordinate for $j \in D \subseteq \{1, \ldots, n\}$ and is differentiable on $i$-th coordinates for $i \in C := \{1, \ldots, n\} \setminus D$. Then, the equations of motion for the continuous variables in $C$ are the same as that in the conventional HMC algorithm, whereas the discontinuous variables in $D$ are numerically integrated as follows: the state $(\boldsymbol{q}, \boldsymbol{p})$ is updated as $(\boldsymbol{q}^*, \boldsymbol{p}^*) = \chi_\epsilon^D(\boldsymbol{q}, \boldsymbol{p})$ where for each $j \in D$,

$$\boldsymbol{q}^* = \boldsymbol{q} + \epsilon \mathrm{sign}(\boldsymbol{p}^j)\boldsymbol{e}_j \quad \text{and} \quad \boldsymbol{p}^* = \boldsymbol{p} - \mathrm{sign}(\boldsymbol{p}^j)(\Delta U)\boldsymbol{e}_j$$

if $|\boldsymbol{p}^j| > \Delta U := U(\boldsymbol{q}^*) - U(\boldsymbol{q})$ and otherwise

$$\boldsymbol{q}^* = \boldsymbol{q} \quad \text{and} \quad \boldsymbol{p}^* = R_j \cdot \boldsymbol{p}$$

where $\boldsymbol{e}_j$ is the $j$-th standard basis vector, $R_j := \mathsf{diag}(1, \ldots, 1, -1, 1, \ldots, 1)$ is the diagonal matrix with diagonal entries 1 everywhere except in the $j$-th position, where it is -1 and $\epsilon$ is the time step.

We define the $(L, \epsilon)$-***Integrator*** function $\mathbf{Int}$ to be a map

$$(\phi^M_{\epsilon/2} \circ \phi^P_{\epsilon/2} \circ \chi^D_\epsilon \circ \phi^P_{\epsilon/2} \circ \phi^M_{\epsilon/2})^L$$

on $\mathbb{R}^n \times \mathbb{R}^n$ such that $\mathbf{Int}(\boldsymbol{q}, \boldsymbol{p})$ is the result of $L$ iterations of the leapfrog method and the above discontinuous step with initial condition $(\boldsymbol{q}, \boldsymbol{p})$.

**Proposition 13** (Lemma 1 of (Nishimura et al., 2020))**.** *The integrator function* $\mathbf{Int}$ *is bijective and has inverse* $\mathbf{Int}^{-1} = M \circ \mathbf{Int} \circ M$ *where* $M(\boldsymbol{q}, \boldsymbol{p}) := (\boldsymbol{q}, -\boldsymbol{p})$*, and the integrator function* $\mathbf{Int}$ *and its inverse are volume preserving (i.e.* $\mathbf{Int}_* \mathsf{Leb}_{2n} = \mathsf{Leb}_{2n}$ *and* $\mathbf{Int}^{-1}{}_* \mathsf{Leb}_{2n} = \mathsf{Leb}_{2n}$*).*

**Pseudocode of DHMC Algorithm** We can again formulate the DHMC algorithm as an instance of the `DirectioniMCMC` algorithm in Listing 4.5, with the momentum distribution (typically the $n$-dimensional multivariate Gaussian distribution) `[normal]*len(q)` as the auxiliary kernel, the integrator function `integrator` as the bijection, and the pdf `w` of the target distribution $\nu$ on $\mathbb{R}^n$. The resulting algorithm is given by the `DHMC` function in Listing 4.10.

*Remark* 22. The DHMC sampler given by (Nishimura et al., 2020) randomly permutes and updates (all) the discontinuous variables in $D$. Here we present a simplified version of it.

**Correctness of DHMC** Since `integrator` is bijective and the momentum distribution is a probability measure, the statement that the DHMC algorithm preserves the distribution $\pi$ is induced from the correctness of the Direction iMCMC algorithm given in Sec. 4.3.2.

### 4.4.3 Irreversible HMC Algorithms

With the catalogue of techniques explored in Sec. 4.3, different irreversible variants of the HMC algorithm can be formed. In this case study, we focus on two such variants, the Generalised HMC algorithm (Horowitz, 1991) and the Look Ahead HMC algorithm (Sohl-Dickstein et al., 2014), which is shown to be equivalent to the Extra Chance Generalised HMC (Campos and Sanz-Serna, 2015).

**Listing 4.10:** *Pseudocode of the DHMC algorithm*

```
def DHMC(q0,C,D):
  d0 = normal                                          # direction
  p0 = [normal]*len(q0)                                # stochastic
  (q,p) = integrator(q0,p0,C,D) if d0 > 0              # deterministic
          else momflip(integrator(momflip(q0,p0),C,D))
  return q if uniform < min{1, w(q) / w(q0) *          # accept/reject
                            pdfnormal(v) / pdfnormal(p0)}
           else q0

def integrator(q0,p0,C,D):                # implements the function Int
  (q,p) = (q0,p0)
  for i in range(L):
    p[C] = p[C] - ep/2 * grad(U)(q[C])
    q[C] = q[C] + ep/2 * grad(K)(p[C])
    for j in D:
      q* = q
      q*[j] = q[j] + ep*sign(p[j])
      DeltaU = U(q*) - U(q)
      if abs(p[j]) > DeltaU:   # refract
        q = q*
        p[j] = p[j] - sign(p[j])*DeltaU
      else:                    # reflect
        p[j] = -p[j]
    q[C] = q[C] + ep/2 * grad(K)(p[C])
    p[C] = p[C] - ep/2 * grad(U)(q[C])
  return (q,p)

momflip(q0,p0) = (q0,-p0)         # implements the function M
```

**Generalised HMC** The Generalised HMC algorithm (Horowitz, 1991) makes two changes to the conventional HMC algorithm in order to generate irreversible Markov chains on $\mathbb{R}^n \times \mathbb{R}^n$, namely

1. "corrupts" the current momentum $\boldsymbol{p_0}$ (instead of samples for a fresh one) in such a way that preserves the state distribution and

2. persists with the direction $d_0$ (i.e. skipping the direction step) if the proposal is accepted; otherwise persists with the negated direction.

As shown in (Neklyudov et al., 2020), the Generalised HMC algorithm can be nicely presented as a composition of an iMCMC algorithm that "corrupts" the momentum (implemented as `CorruptMom`) and a Persistent iMCMC algorithm (implemented as `PerHMC`) that uses Hamiltonian dynamics to find a new state with persisting direction. The resulting algorithm is implemented as the `GenHMC` function in Listing 4.11.

Given the current state `((q0,p0),d0)`, `GenHMC` generates a new state as follows.

**Listing 4.11:** *Pseudocode for the Generalised HMC algorithm*

```
GenHMC((q0,p0),d0) = PerHMC(CorruptMom((q0,p0),d0))

def HMCw(q,p): return w(q)

def CorruptMom((q0,p0),d0):
  v0 = [normal(p0[i]*sqrt(1-alpha^2), alpha^2) for i in range(len(p0))]
  (((q,p),d),v) = (((q0,v0),d0),p0)
  return ((q,p),d)

def PerHMC((q1,p1),d1):
  v1 = normal                                          # stochastic
  ((q,p),v) = (leapfrog(q1,p1),v1) if d1 > 0           # deterministic
              else (momflip(leapfrog(momflip(q1,p1))),v1)
  d = -d1
  return ((q,p),-d) if uniform <
                       min{1,HMCw(q,p)/HMCw(q1,p1)}    # accept/reject
                    else ((q1,p1),-d1)
```

1. (`CorruptMom`) Return `((q0,p),d0)` where the $i$-th component of `p` is sampled from `normal(p0[i]*sqrt(1-alpha^2), alpha^2)` with a hyperparameter `alpha`.

   The `CorruptMom` function is given in the `iMCMC` format where the auxiliary kernel is `[normal(p0[i]*sqrt(1-alpha^2), alpha^2)for i in range(len(p0))]`, the involution is a swap, and (hence) the acceptance ratio is

$$\min\left\{1, \frac{\rho(\boldsymbol{q}_0)\cdot\varphi_n(\boldsymbol{v}_0)\cdot\mathsf{pdf}_{\mathcal{N}}(d_0)\cdot\varphi_n(\boldsymbol{p}_0 \mid \boldsymbol{v}_0\sqrt{1-\alpha^2},\alpha^2)}{\rho(\boldsymbol{q}_0)\cdot\varphi_n(\boldsymbol{p}_0)\cdot\mathsf{pdf}_{\mathcal{N}}(d_0)\cdot\varphi_n(\boldsymbol{v}_0 \mid \boldsymbol{p}_0\sqrt{1-\alpha^2},\alpha^2)}\right\} = 1.$$

2. (`PerHMC`) Taking the output state `((q1,p1),d1)` from `CorruptMom`, simulate Hamiltonian dynamics using the `leapfrog` function implemented in Listing 4.9 if `d1` is positive, otherwise its inverse `momflip leapfrog momflip`, with initial condition `(q1,p1)`. The computed proposal state `(q,p)` is then accepted with probability

$$\min\left\{1, \frac{\rho(\boldsymbol{q})\cdot\mathsf{pdf}_{\mathcal{N}_n}(\boldsymbol{p})}{\rho(\boldsymbol{q}_0)\cdot\mathsf{pdf}_{\mathcal{N}_n}(\boldsymbol{p}_0)}\right\},$$

   and returned with an unchanged direction `d1`. Otherwise, the state `(q1,p1)` with a negated direction `-d1` is returned instead.

   The `PerHMC` is given as an instance of the `PersistentiMCMC` function with the target distribution `HMCw`, a dummy auxiliary kernel `normal` and the product of `leapfrog` and `identity` as the bijection.

The Generalised HMC algorithm preserves the target distribution as both `CorruptMom`, an instance of iMCMC, and `PerHMC`, an instance of Persistent iMCMC, are shown to preserve the target distribution in Sec. 4.2.1 and 4.3.3.

**Look Ahead HMC** Last but not least, we consider the Look Ahead HMC algorithm (Sohl-Dickstein et al., 2014), which is equivalent to the Extra Chance Generalised HMC algorithm (Campos and Sanz-Serna, 2015). This algorithm modifies the Generalised HMC algorithm by performing extra leapfrog steps when the proposal state is rejected. This has the effect of increasing the acceptance rate for the algorithm as the chance that the algorithm accepts a proposal has increase.

Similar to Generalised HMC, we present the Look Ahead HMC algorithm as a composition of the iMCMC algorithm that "corrupts" the momentum (implemented as `CorruptMom` in Listing 4.11) and a Persistent iMCMC algorithm that applies a random number of leapfrog function (or its inverse) to the current state.

There are two different ways of implementing this Persistent iMCMC sampler. We can either draw a random number in the stochastic step and then perform said number of steps; or after each set of leapfrog steps, determine whether to perform an extra set of leapfrog steps in the acceptance step. We call the first implementation the *Multiple HMC* sampler and the second the *Extra Chance HMC* sampler.

**Multiple HMC Sampler** Listing 4.13 gives the implementation of the Multiple HMC sampler in SPCF.

Given the returned state `((q1,p1),d1)` from `CorruptMom`, simulate Hamiltonian dynamics using the leapfrog function `nleapfrog` in the direction `d1` to the initial condition `(q1,p1)` and obtain the next state `(q,p)`. The duration of the simulation is determined by the value of the auxiliary variable `n` sampled from `Multikernel((q1,p1),d1)`. The proposal state `(q,p)` is always accepted alongside an unchanged direction `d1` if `n` is non-zero, otherwise the direction is negated.

The `MultiHMC` is given as an instance of the `PersistentiMCMC` function with the auxiliary kernel `Multikernel` and a bijection that finds the proposal state `(q,p)` by evolving Hamiltonian equation using the `nleapfrog(n)` function with the initial condition `(q0,p0)`.

The heart of `MultiHMC` is the definition of the auxiliary kernel `Multikernel` that determines the number `n` of applications of the leapfrog function (or its inverse). Given the state $((\boldsymbol{q}_0, \boldsymbol{p}_0), d_0)$ and $K \in \mathbb{N}_{>0}$, `Multikernel` returns $n > 0$ with probability

**Listing 4.12:** *Pseudocode for Look Ahead HMC Sampler with Multiple HMC*

```
LAHMC((q0,p0),d0) = MultiHMC(CorruptMom((q0,p0),d0)))
```

**Listing 4.13:** *Pseudocode for Multiple HMC Sampler*

```
def MultiHMC((q1,p1),d1):
  n = LAkernel((q1,p1),d1)                          # stochastic
  ((q,p),n) = (leapfrog(n)(q1,p1),n) if d1 > 0      # deterministic
              else (momflip(leapfrog(n)(momflip(q1,p1))),n)
  d = -d1 if n > 0 else d1
  return ((q,p),-d)                                 # always accept

def LAkernel((q1,p1),d1):
  u = uniform
  n = 1
  acc = min{1,w(nleapfrog(n)(q1,p1))/w(q1,p1)} if d1 > 0
        else min{1,w(momflip(nleapfrog(n)(momflip(q1,p1))))/w(q1,p1)}
  while acc < u && n <= K:
    n = n+1
    acc = min{1,w(nleapfrog(n)(q1,p1))/w(q1,p1)} if d1 > 0
          else min{1,w(momflip(nleapfrog(n)(momflip(q1,p1))))/w(q1,p1)}
  return 0 if n = K+1
         else n

def nleapfrog(n)(q1,p1) =                  # apply leapfrog n times
  (q,p) = (q1,p1)
  for i in range(n):
    (q,p) = leapfrog(q,p)
  return (q,p)
```

$\max\{0, \min\{1, \sigma_n\} - \max\{\sigma_\ell \mid \ell < n\}\}$ and returns $0$ with probability $\max\{0, 1 - \max\{\sigma_\ell \mid \ell \leq K\}\}$ where

$$\sigma_k := \begin{cases} \dfrac{\zeta(\mathbf{L}^k(\boldsymbol{q}_0, \boldsymbol{p}_0))}{\zeta(\boldsymbol{q}_0, \boldsymbol{p}_0)} & \text{if } d_0 > 0 \\ \dfrac{\zeta(M(\mathbf{L}^k(M(\boldsymbol{q}_0, \boldsymbol{p}_0))))}{\zeta(\boldsymbol{q}_0, \boldsymbol{p}_0)} & \text{otherwise.} \end{cases}$$

Hence the likelihood of performing $n$ numbers of $\mathbf{L}$s is the distance between $\min\{1, \sigma_n\}$ and the highest of $\sigma_\ell$ for $\ell < n$, if it is non-negative. Moreover, it has the interesting consequence that the `MultiHMC` function always accepts the proposal because by Prop. 14, the acceptance ratio is

$$\min\{1, \frac{\zeta(\boldsymbol{q}, \boldsymbol{p}) \cdot \mathsf{pdf}_{\texttt{Multikernel}}((\boldsymbol{q}, \boldsymbol{p}), -d_0)(k)}{\zeta(\boldsymbol{q}_0, \boldsymbol{p}_0) \cdot \mathsf{pdf}_{\texttt{Multikernel}}((\boldsymbol{q}_0, \boldsymbol{p}_0), d_0)(k)}\} = \min\{1, \sigma_k \cdot \frac{1}{\sigma_k}\} = 1$$

**Listing 4.14:** *Pseudocode for Look Ahead HMC Sampler with Extra Chance HMC*

```
LAHMC((q0,p0),d0) = ExtraChanceHMC(CorruptMom((q0,p0),d0)))
```

**Listing 4.15:** *Pseudocode for Extra Chance HMC Sampler*

```
def ExtraChanceHMC((q1,p1),d1):
  (q,p) = (q1,p1)
  u = uniform
  stop = False
  j = 1
  while not stop:
    # perform a set of leapfrog steps
    (q,p) = leapfrog(q,p) if d1 > 0
            else momflip(leapfrog(momflip(q,p)))
    if u > min{1,w(q)/w(q1) *
                 pdfnormal(p)/pdfnormal(p1)}:
      if j <= J:
        # perform an extra set of leapfrog steps
        j = j + 1
      else:
        # no leapfrog steps is performed
        (q,p) = (q1,p1)
        stop = True
        d = d0
    else:
      # enough leapfrog steps are performed
      stop = True
      d = -d0
  return ((q,p), -d)
```

for $k > 0$.

**Proposition 14.** *Let* $(\boldsymbol{q}_0, \boldsymbol{p}_0) \in \mathbb{R}^n \times \mathbb{R}^n$, $d_0 \in \mathbb{R}$ *and* $k > 1$. *Let*

$$(\boldsymbol{q}, \boldsymbol{p}) = \begin{cases} \mathbf{L}^k(\boldsymbol{q}_0, \boldsymbol{p}_0) & \text{if } d_0 > 0 \\ (\mathbf{L}^{-1})^k(\boldsymbol{q}_0, \boldsymbol{p}_0) & \text{otherwise} \end{cases}$$

*Then,*

$$\frac{\mathsf{pdf}_{\texttt{Multikernel}}((\boldsymbol{q}, \boldsymbol{p}), -d_0)(k)}{\mathsf{pdf}_{\texttt{Multikernel}}((\boldsymbol{q}_0, \boldsymbol{p}_0), d_0)(k)} = \frac{1}{\sigma_k} \text{ or is undefined.}$$

With the `CorruptMom` from Listing 4.11, the Look Ahead HMC sampler is implemented as the `LAHMC` function in Listing 4.12. Since both `CorruptMom`, an instance of iMCMC, and `MultiHMC`, an instance of Persistent iMCMC, preserve the target distribution (Sec. 4.2.1 and 4.3.3), so does the Look Ahead HMC sampler.

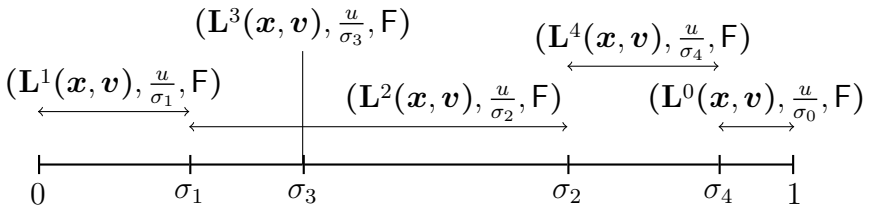


**Figure 4.4:** *Result of* $\Phi(\boldsymbol{x},\boldsymbol{v},u,\mathsf{T})$ *for varying* $u \in [0,1]$.

**Extra Chance HMC** Another way of describing Look Ahead HMC is to check, after each set of leapfrog steps, whether to perform an extra set of leapfrog steps in the acceptance step.

We consider the involution $\Phi$ on $\mathbb{R}^n \times \mathbb{R}^n \times [0,1) \times 2$ given by

$$\Phi(\boldsymbol{x},\boldsymbol{v},u,\mathsf{T}) := \begin{cases} (\mathbf{L}^j(\boldsymbol{x},\boldsymbol{v}),\frac{u}{\sigma_j},\mathsf{F}) & \text{if } \max\{\sigma_i \mid i<j\} \le u < \min\{1,\sigma_j\} \\ (\boldsymbol{x},\boldsymbol{v},u,\mathsf{T}) & \text{if } \max\{\sigma_j \mid j \le J\} \le u \end{cases}$$

$$\Phi(\boldsymbol{x},\boldsymbol{v},u,\mathsf{F}) := \begin{cases} (\mathbf{L}^{-j}(\boldsymbol{x},\boldsymbol{v}),\frac{u}{\sigma'_j},\mathsf{T}) & \text{if } \max\{\sigma'_i \mid i<j\} \le u < \min\{1,\sigma'_j\} \\ (\boldsymbol{x},\boldsymbol{v},u,\mathsf{F}) & \text{if } \max\{\sigma'_j \mid j \le J\} \le u \end{cases}$$

where

$$\sigma_j := \frac{\zeta(\mathbf{L}^j(\boldsymbol{x},\boldsymbol{v}))}{\zeta(\boldsymbol{x},\boldsymbol{v})}, \qquad \sigma'_j := \frac{\zeta(\mathbf{L}^{-j}(\boldsymbol{x},\boldsymbol{v}))}{\zeta(\boldsymbol{x},\boldsymbol{v})}, \qquad \mathbf{L}^{-j} := (\mathbf{L}^{-1})^j$$

and $J$ is the maximum set of leapfrog steps.

The first two parameters of $\Phi$, namely $(\boldsymbol{x},\boldsymbol{v})$, give the state of the algorithm, and the last parameter indicates the direction of the Hamiltonian simulation. The third parameter of $\Phi$ is the most interesting: $u \in [0,1)$ is a uniformly distributed variable that determines how many sets $(j)$ of leapfrog steps are to be performed. The idea is to perform an extra set of leapfrog steps if the proposal is rejected. Hence if all the acceptance ratios $\min\{1,\sigma_j\}$ for $j = 1,\dots,J$ are marked on the unit interval $[0,1]$, the probability of performing $j$ sets of leapfrog steps is given by the distance between $\min\{1,\sigma_j\}$ and the highest of $\sigma_i$ for $i < j$, if it is non-negative. Sec. 4.4.3 gives an example of the result of $\Phi(\boldsymbol{x},\boldsymbol{v},u,\mathsf{T})$ for varying $u \in [0,1]$.

The Look Ahead HMC sampler can be formulated as a Persistent iMCMC sampler with the auxiliary kernel $K((\boldsymbol{x},\boldsymbol{v}),\cdot) := \mathcal{U}[0,1)$ and the above involution $\Phi$. Note that the sampler always accepts the proposal as the acceptance ratio is for $u \in [\max\{\sigma_i \mid i < j\}, \min\{1,\sigma_j\})$

$$\min\{1, \frac{\zeta(\mathbf{L}^j(\boldsymbol{x},\boldsymbol{v}))}{\zeta(\boldsymbol{x},\boldsymbol{v})} \cdot |\det \nabla\Phi(\boldsymbol{x},\boldsymbol{v},u,\mathsf{T})|\} = \min\{1, \sigma_j \cdot |(\det \nabla\mathbf{L}^j(\boldsymbol{x},\boldsymbol{v})) \cdot \frac{1}{\sigma_j}|\} = 1$$

and similarly for $u \in [\max\{\sigma_j \mid j \leq J\}, 1]$. A similar argument can be made when the direction is F.

Listing 4.15 gives a SPCF implementation of the Extra Chance HMC sampler. After a random variable `u` is drawn from the uniform distribution, the first set of leapfrog steps (or its inverse) is performed on `(q,p)`. To determine whether to do an extra set of leapfrog steps, the variable `u` is compared to the acceptance ratio. If `u` is lower than the ratio, the state is returned as the proposal. Otherwise, a new set of leapfrog steps (or its inverse) is performed. This is repeated for at most `J` times. By then, if `u` is still higher than the acceptance ratio, the initial state `(q1,p1)` is returned.

Putting `ExtraChanceHMC` and `CorruptMom` from Listing 4.11 together, we get the Look Ahead HMC sampler as the `LAHMC` function in Listing 4.14.

## 4.5 Case Study: Reversible Jump MCMC

Transdimensional inferences aim to sample from a posterior distribution where the number of parameters (and hence the dimension of the model) is unknown. A transdimensional MCMC algorithm has to move between dimensions in order to explore the whole parameter space. Different transdimensional samplers have been suggested (Carlin and Chib, 1995; Grenander and Miller, 1994) based on different processes, but the most widely applied one is the infamous Reversible Jump MCMC (RJMCMC) (Green, 1995) suggested by Green in 1995.

One can see RJMCMC as a natural generalisation of the standard Metropolis-Hastings algorithm (Green and Hastie, 2009). In this case study, we explore a simplified RJMCMC algorithm from (Gagnon and Doucet, 2020) using the iMCMC framework.

### 4.5.1 The Reversible Jump MCMC Algorithm

Let $\nu$ be the target distribution on $\bigcup_{j\in\mathbb{N}} \mathbb{R}^j$ with density $\rho : \bigcup_{j\in\mathbb{N}} \mathbb{R}^j \to [0, \infty)$. Given the current state $\boldsymbol{x} \in \mathbb{R}^k$, the RJMCMC algorithm generates a new state as follows:

1. Propose a model candidate $k'$ from some probability distribution $g(k, \cdot)$ on $\mathbb{N}$;
2. Generate $u_{k\to k'}$ from some probability distribution $q_{k\to k'}$;
3. Apply the function $T_{k\to k'}$ to $(\boldsymbol{x}, u_{k\to k'})$ to obtain $(\boldsymbol{y}, u_{k'\to k})$ where $T_{k\to k'}$ is a diffeomorphism with inverse $T_{k'\to k}$;

**Listing 4.16:** *Pseudocode of the RJMCMC algorithm*

```
def RJMCMC(x0):
  k0 = len(x0)                                    # stochastic step
  k = g(k0)
  u0 = proposal(k0)(k)(x0)
  (x,u) = T(k0)(k)(x0,u0)                         # deterministic step
  return x if uniform < min{1, w(x)/w(x0) *       # accept/reject step
                        pdfg(k,k0)/pdfg(k0,k)*
                        pdfproposal(k)(k0)(u)/pdfproposal(k0)(k)(u0)*
                        absdetjacT(k0)(k)(x0,u0)}
    else x0
```

**Listing 4.17:** *Auxiliary kernel and involution of the RJMCMC algorithm*

```
def auxkernel(x0):
  k0 = len(x0)
  k = g(k0)
  u0 = proposal(k0)(k)(x0)
  return (u0,k0,k)

def involution(x0,u0,k0,k):
  (x,u) = T(k0)(k)(x0,u0)
  return (x,u,k,k0)
```

4. Accept the proposal $\boldsymbol{y}$ with probability

$$\min\{1, \frac{g(k',k)\cdot\rho(\boldsymbol{y})\cdot q_{k'\to k}(u_{k'\to k})}{g(k,k')\cdot\rho(\boldsymbol{x})\cdot q_{k\to k'}(u_{k\to k'})}\cdot|\det\nabla T_{k\to k'}(\boldsymbol{x},u_{k\to k'})|\};$$

   otherwise return the initial state $\boldsymbol{x}$.

Note that the state "jumps" from dimension $k$ to $k'$ if the proposal is accepted.

**Pseudocode of RJMCMC Algorithm** We can formulate the RJMCMC algorithm as an instance of the `iMCMC` algorithm (Listing 4.1) with the pdf `w` as the target density function, `auxkernel` and `involution` as given in Listing 4.17. The resulting algorithm is given by the `RJMCMC` function in Listing 4.16.

**Correctness of RJMCMC** Since `involution` is indeed an involution, the correctness of RJMCMC is deduced from that of the iMCMC algorithm (Cor. 1).

### 4.5.2 Instances and Generalisations of RJMCMC

The jump-diffusion sampler of (Grenander and Miller, 1994) and the birth-death sampler of (Stephens, 2000), can be seen as instances or a sequence of RJMCMC samplers. Based on the product state space approach of (Carlin and Chib, 1995), Godsill presented a sampler that generalises RJMCMC for the product state space (Godsill, 2001). The RJMCMC sampler has also be extended to construct a non-reversible Markov chain (Gagnon and Doucet, 2020). (Chavis et al., 2021) gave a parallelised extension of RJMCMC.

### 4.5.3 Automating RJMCMC

The performance of RJMCMC replies on its proposal distributions, namely $g(k, \cdot)$ and $q_{k\to k'}$, and its between-model mappings $T_{k\to k'}$. Considerable efforts have been put to automate these distributions and mappings in hope to construct a fully automatic transdimensional algorithm. For instance, between-model mappings are suggested by (Green, 2003; Godsill, 2003) in order to form an automatic generic trans-dimensional sampler. Recently, (Heck et al., 2019) suggested a design on the estimated stationary distribution $g$ of the discrete parameter indexing the competing models.

## 4.6 Approximate Inferences for Probabilistic Programming

Since the inception of probabilistic programming, inference algorithms have played an important role. Here we discuss common inference algorithms found in popular probabilistic programming, including some that are designed specifically for probabilistic programming.

### 4.6.1 Importance Sampling

Importance Sampling (IS) is a basic, simple to implement Monte Carlo method that can sample from all probabilistic models, including those described by probabilistic programs. It is conceptually elegant, comprising of three steps:

1. Draw samples from a proposal distribution;

2. Compute the ratio of the weight of each sample w.r.t. the posterior and the proposal;

3. Perform inference using the weighted samples.

**Listing 4.18:** *State Space Model*

```
def StateSpaceModel(x, data):
  for t in range(len(data)):
    observe data[t] from Dist(x[0:t])
```

Various extensions of IS have been developed to find a good proposal distribution that can locate the posterior. *Multiple IS* methods (Elvira et al., 2019) draw samples from multiple proposal distributions; whereas *adaptive IS* methods (Bugallo et al., 2017) update the proposal distribution by iterating the sampling and weighting process, resulting in more accurate samples.

IS is commonly found in probabilistic programming languages as a baseline inference algorithm for models. However designing a suitable proposal distribution is challenging. The recently suggested adaptive IS has been applied to probabilistic programs in (Luo et al., 2021), though no correctness proof was provided in their account.

### 4.6.2 Particle Methods

Sequential Monte Carlo (SMC) (also known as particle filter) methods (Gordon, 1993; Doucet et al., 2001) are a class of algorithms that sequentially sample from the posterior. It is particularly suitable for recursive estimation problems such as state space models in Listing 4.18 where new data points are considered sequentially. For instance, Sequential Importance Resampling with $N$ particles samples from `StateSpaceModel` with dataset `data` of length $T$ as follows:

1. Draw $N$ weighted samples `(x[0],weight[0])` from the distribution given by `y = normal; StateSpaceModel(y, data[0])` using IS;
2. For each subsequent parameters $t = 1, \ldots, T-1$,
   (a) Draw $N$ weighted (temporary) samples `(tempx[t],tempweight[t])` from the distribution given by `y = normal; StateSpaceModel(x[0:t-1]+[y], data[0:t])` using IS;
   (b) Draw $N$ samples `(x[t],weight[t])` using the weighted samples `(tempx[t],tempweight[t])`.

Similar to IS, the selection of proposal distributions in SMC can be tricky and the SMC approximation deteriorates as components might not be rejuvenated at subsequent time steps in high dimensional models.

Attempts to combine the strengths of SMC and MCMC lead to the development of Particle MCMC methods such as Particle Metropolis-Hastings and Particle Gibbs (Andrieu et al., 2010). These algorithms use SMC approximations to build efficient high dimensional proposal distributions for MCMC schemes and have been applied successfully to a non-linear state space model and a Lévy-driven stochastic volatility model.

Many probabilistic programming languages have implemented some kind of particle methods. Designing a suitable proposal distribution for probabilistic programming remains difficult. Staton et al. (2016) applied their semantics to validate the correctness of SMC simulation.

### 4.6.3 Optimisation Methods

Besides sampling methods, variational inference (VI) provides another method for approximating probability densities. The main idea behind it is to optimise

$$q^*(z) = \arg\min_{q \in Q} \{KL(q(z) \parallel p(z|x))\}$$

where $Q$ is a family of approximate densities. In this case, the optimised member $q^*$ minimises the Kullback-Leibler (KL) divergence to the exact posterior density $p$ and approximates the posterior $p$.

Unlike Monte Carlo methods, variational inference (VI) (Blei et al., 2017) solves the Bayesian inference problem by treating it as an optimisation problem. When adapted to models expressed as probabilistic programs, the score function VI (Ranganath et al., 2014) can in principle be applied to a large class of branching and recursive programs because only the variational density functions need to be differentiable. Existing implementations of VI algorithms in probabilistic programming systems are however far from automatic: in the main, the guide programs (that express variational distributions) still need to be hand-coded.

### 4.6.4 Lightweight Metropolis-Hastings

The standard MCMC algorithm for probabilistic programming that is widely implemented (for example, in Anglican, Venture, and Web PPL) is the Lightweight Metropolis-Hastings (LMH) algorithm (Wingate et al., 2011) and its extensions (Yang et al., 2014; Tolpin et al., 2015; Ritchie et al., 2016), which performs single-site updates on the current sample and re-executes the program. It works well in simple cases but suffers from a lack of predictive accuracy in its proposal for high dimensional models.

### 4.6.5 Divide, Conquer and Combine

Recently, Zhou et al. (2020) introduced the Divide, Conquer, and Combine (DCC) algorithm, which is applicable to programs definable using branching and recursion. As a hybrid algorithm, DCC solves the problem of designing a proposal that can efficiently transition between configurations by performing local inferences on submodels, and returning an appropriately weighted combination of the respective samples. Thanks to a judicious resource allocation scheme, it exhibits strong performance on multimodal distributions.

### 4.6.6 MCMC Methods

In this chapter, we studied the HMC and RJMCMC samplers. Here we discuss how they might or might not be suitable for probabilistic programming.

**HMC** The HMC algorithm and its variants, notably the No-U Turn Sampler, are the workhorse inference methods in the influential PPL Stan (Gelman et al., 2015). The challenges posed by stochastic branching in probabilistic programming are the focus of reflective/refractive HMC (Afshar and Domke, 2015); discontinuous HMC (Nishimura et al., 2020); mixed HMC (Zhou, 2020); and the first-order PPL in (Zhou et al., 2019) which is equipped with an implementation of discontinuous HMC. However, none of these variants addresses the transdimensional movement in the parameter space posed by recursion in probabilistic programming.

**RJMCMC** The Reversible Jump Markov chain Monte Carlo (RJMCMC) algorithm (Green, 1995) is a trans-dimensional MCMC sampler. However, it additionally requires the user to specify a transition kernel which is labour-intensive. Various RJMCMC transition kernels have been suggested for specific models, e.g. split-merge proposal for infinite Gaussian mixture models.

Some practical probabilistic programming languages such as Hakaru, Pyro and Gen give users the flexibility to hand-code the proposal in a MCMC setting. For instance, Cusumano-Towner et al. (2019) implement the split-merge proposal (Richardson and Green, 1997) of RJMCMC in Gen. Though this line of research is orthogonal to ours, probabilistic programming such as Gen could play a useful role in the implementation of trans-dimensional samplers for probabilistic programming.

*Et introibo ad altare Dei, ad Deum qui lætificat juventutem meam.*

— Psalm 42:4

# 5 Nonparametric Involutive MCMC

Recall the two main tasks in Bayesian Machine Learning — the specification of probabilistic models and the computation of the posterior distribution. As discussed in Sec. 3.1.2, this thesis leaves the discussion of model specification to domain experts and instead demonstrates that probabilistic programs are fitting presentations of probabilistic models in Bayesian inference (Sec. 3.2.2). Hence, we are left with the task of computing the posterior of a probabilistic program, in other words, the design of inference algorithms for probabilistic programming.

In this chapter, we present the ***Nonparametric Involutive Markov chain Monte Carlo*** (NP-iMCMC), an inference algorithm that simulates the probabilistic model specified by a given SPCF program. As its name suggests, the NP-iMCMC sampler gives a general framework for the design of MCMC algorithms, similar to the *involutive MCMC* sampler, and is specially developed to compute the posteriors of *nonparametric* probabilistic programs.

To start, we study how the Metropolis-Hastings (MH) inference algorithm fails to sample from the infinite Gaussian mixture model (iGMM), and investigate the challenge faced by all MCMC samplers in simulating nonparametric probabilistic models. After that, we detail the NP-iMCMC inference algorithm: its state space, conditions on the inputs and steps to generate the next sample; and study how the sampler tackles the aforementioned challenge and returns new samples of a nonparametric probabilistic program. We then give an implementation of NP-iMCMC in SPCF and demonstrate how the NP-iMCMC method extends the MH sampler and samples from iGMM. Finally, we provide some techniques to improve the NP-iMCMC sampler.

We will discuss the correctness of NP-iMCMC in Chapter 6 and will report the empirical experiments on the sampler in Chapter 7.

**Listing 5.1:** *Infinite Gaussian mixture model*

```
def iGMM:
  K = normal
  means = []
  for i in range(abs(K)+1):
    means.append(normal)
  for d in data:
    score(sum[pdfnormal(x,1)(d) for x in means]/len(means))
  return K
```

## 5.1 The Challenge MCMC Samplers Face

The goal of a MCMC sampler is to generate a Markov chain that simulates the target distribution given by a target density $\rho$ w.r.t. the Lebesgue measure. This is typically done by iterating a stochastic algorithm that updates the current sample. For example, the classic Metropolis-Hastings (MH) algorithm updates the current sample $\boldsymbol{x}_0$ with a proposal $\boldsymbol{x}$ randomly drawn from a proposal distribution $q(\boldsymbol{x}_0, \cdot)$ with probability

$$\min\{1, \frac{\rho(\boldsymbol{x}) \cdot \mathsf{pdf}_q(\boldsymbol{x}, \boldsymbol{x}_0)}{\rho(\boldsymbol{x}_0) \cdot \mathsf{pdf}_q(\boldsymbol{x}_0, \boldsymbol{x})}\}$$

where $\mathsf{pdf}_q$ is the Radon-Nikodym derivative of $q$ w.r.t. the Lebesgue measure. Otherwise, it repeats with the current sample $\boldsymbol{x}_0$.

We demonstrate the challenge of updating samples of a nonparametric model by considering the MH sampler on the infinite Gaussian mixture model `iGMM` (Listing 5.1). `iGMM` describes a mixture of $\max\{0, \lfloor$ `K` $\rfloor\}$ Gaussian distributions with normally-distributed random variables `K` and `means`. Notice that the total number of calls to *`normal`* in `iGMM` depends on the result of the first sample, i.e. the value of `K`, making the model `iGMM` nonparametric.

Say the current sample is $\boldsymbol{x}_0 := [3.4, -1.2, 1.0, 0.5]$. Then, the model `iGMM` with a trace specified by $\boldsymbol{x}_0$, describes a mixture of three Gaussian distributions, centred at $-1.2, 1.0$ and $0.5$ respectively. To update the current sample, the MH sampler draws say $\boldsymbol{x} = [4.3, -2.4, -0.1, 1.4]$ from the proposal distribution $\mathcal{N}_{|\boldsymbol{x}_0|}(\boldsymbol{x}_0, \boldsymbol{I})$. Unfortunately, we *cannot* simply propose $\boldsymbol{x}$. This is because with $\boldsymbol{x}$ as the trace, the model `iGMM` describes a mixture of *four* Gaussian distributions (as `K` is set to be $4.3$), with only *three* values provided for the means. What is the mean of the "extra" Gaussian? Or more generally, how should the state move from a sample $\boldsymbol{x}_0$ of length (or dimension) four to one of length (or dimension) five?

This is *the* challenge all MCMC inferences face when simulating a nonparametric model like `iGMM`. For instance, to design an iMCMC sampler that explores the model `iGMM` fully, the auxiliary kernel or the involution must jump between states of different lengths (or dimensions). Designing such a sampler could be labour-intensive and model specific. For example, we have to specify the kernels `g`, `proposal` and diffeomorphism `T` in the Reversible Jump MCMC sampler in Sec. 4.5.

This thesis presents the ***Nonparametric Involutive MCMC*** (NP-iMCMC) sampler which naturally moves between states of different dimensions and explores every dimensions of the nonparametric model. We start our discussion with the state space of the NP-iMCMC sampler, laying the groundwork for the rest of this chapter.

## 5.2 State Spaces

A *state* in the NP-iMCMC algorithm is a pair $(\boldsymbol{x}, \boldsymbol{v})$ of equal *dimension parameter* and *auxiliary* variables. The parameter variable $\boldsymbol{x}$ is used to store traces and the auxiliary variable $\boldsymbol{v}$ is used to record randomness. Both variables are vectors of *entropies*, i.e. Real-boolean pairs. This section gives the formal definitions of the entropy, parameter and auxiliary variables and the state, in preparation for the discussion of the NP-iMCMC sampler.

### 5.2.1 Entropy Space

As shown in Sec. 3.2, the reduction of a SPCF program is determined by the input trace $\boldsymbol{t} \in \mathbb{T}$, a record of drawn values in a particular run of the program. Hence in order to simulate a probabilistic model described by a SPCF program, the NP-iMCMC sampler should generate Markov chains on the trace space. However traversing through the trace space is a delicate business because the positions and numbers of discrete and continuous values in a trace given by a SPCF program may vary. (Consider `if coin: normal else: coin`.)

Instead, we pair each value $\boldsymbol{t}^i$ in a trace $\boldsymbol{t}$ with a random value $t$ of the other type to make a Real-boolean pair $(\boldsymbol{t}^i, t)$ (or $(t, \boldsymbol{t}^i)$). For instance, the trace $[\mathsf{T}, -3.1]$ can be made into a Real-boolean vector $[(1.5, \mathsf{T}), (-3.1, \mathsf{T})]$ with randomly drawn values $1.5$ and $\mathsf{T}$. In this case, the position of discrete and continuous random variables does not matter and the number of discrete and continuous random variables is fixed in each vector.

We call a Real-boolean pair an ***entropy*** and define the ***entropy space*** $\mathbb{E}$ to be the product space $\mathbb{R} \times 2$ of the Borel measurable space and the boolean measurable space, equipped with the $\sigma$-algebra $\Sigma_{\mathbb{E}} := \sigma\big(\{R \times B \mid R \in \mathcal{B}, B \in \Sigma_2\}\big)$, and the product measure $\mu_{\mathbb{E}} := \mathcal{N} \times \mu_2$. Note the Radon-Nikodym derivative $\varphi_{\mathbb{E}}$ of $\mu_{\mathbb{E}}$ can be defined as $\varphi_{\mathbb{E}}(r, a) := \frac{1}{2}\varphi(r)$. A $n$-length ***entropy vector*** is then a vector of $n$ entropies, formally an element in the product measurable space $(\mathbb{E}^n, \Sigma_{\mathbb{E}^n})$. We write $|\boldsymbol{x}|$ to mean the length of the entropy vector $\boldsymbol{x}$.

As mentioned earlier, the parameter variable of a state is an entropy vector that stores traces. Hence, it would be useless if a unique trace cannot be restored from an entropy vector. We found that such a recovery is possible if the trace is in the support of a tree representable function.

Say we would like to recover the trace $\hat{\boldsymbol{t}}$ that is used to form the entropy vector $\boldsymbol{x}$ by pairing each value in the trace with a random value of the other type. First we realise that traces can be made by selecting either the Real or boolean component of each pair in a prefix of $\boldsymbol{x}$. For example, traces like $[]$, $[\mathsf{T}]$, $[-0.2]$, $[\mathsf{T}, 2.9]$ and $[-0.2, \mathsf{T}, \mathsf{F}]$ can be made from the entropy vector $[(-0.2, \mathsf{T}), (2.9, \mathsf{T}), (1.3, \mathsf{F})]$. We call these traces ***instances*** of the entropy vector. Formally, a trace $\boldsymbol{t} \in \mathbb{T}$ is an instance of an entropy vector $\boldsymbol{x} \in \mathbb{E}^n$ if $|\boldsymbol{t}| \leq n$ and $\boldsymbol{t}^i \in \{r, a \mid (r, a) = \boldsymbol{x}^i\}$ for all $i = 1, \ldots, |\boldsymbol{t}|$. We denote the set of all instances of $\boldsymbol{x}$ as $\mathsf{instance}(\boldsymbol{x}) \subseteq \mathbb{T}$. Then, the trace $\hat{\boldsymbol{t}}$ must be an instance of $\boldsymbol{x}$. Moreover, if we can further assume that $\hat{\boldsymbol{t}}$ is in the support of a tree representable function, then Prop. 15 says we can uniquely identify $\hat{\boldsymbol{t}}$ amongst all instances of $\boldsymbol{x}$.

**Proposition 15.** *There is at most one trace that is both an instance of an entropy vector and in the support of a tree representable function.*

Finally, we consider differentiability on the multi-dimensional entropy space. We say a function $f : \mathbb{E}^{k_1} \to \mathbb{E}^{k_2}$ is ***differentiable almost everywhere*** if for all $\boldsymbol{i} \in 2^{k_1}$, $\boldsymbol{j} \in 2^{k_2}$, the partial function $f_{\boldsymbol{i} \to \boldsymbol{j}} : \mathbb{R}^{k_1} \to \mathbb{R}^{k_2}$ where

$$f_{\boldsymbol{i} \to \boldsymbol{j}}(\boldsymbol{r}) = \boldsymbol{q} \qquad \iff \qquad f(\mathsf{zip}(\boldsymbol{r}, \boldsymbol{i})) = (\mathsf{zip}(\boldsymbol{q}, \boldsymbol{j}))^1$$

is differentiable *almost everywhere* on its domain

$$\mathsf{Dom}(f_{\boldsymbol{i} \to \boldsymbol{j}}) := \{\boldsymbol{r} \in \mathbb{R}^{k_1} \mid \exists \boldsymbol{q} \in \mathbb{R}^{k_2} \,.\, f(\mathsf{zip}(\boldsymbol{r}, \boldsymbol{i})) = (\mathsf{zip}(\boldsymbol{q}, \boldsymbol{j}))\}.$$

The Jacobian of $f$ on $(\mathsf{zip}(\boldsymbol{r}, \boldsymbol{i}))$ is given by $\nabla f_{\boldsymbol{i} \to \boldsymbol{j}}(\boldsymbol{r})$, if it exists.

### 5.2.2 Parameter Space

A parameter variable $\boldsymbol{x}$ of dimension $n$ is an entropy vector of length $\iota_{\mathbb{X}}(n)$ where $\iota_{\mathbb{X}} : \mathbb{N} \to \mathbb{N}$ is a monotone map, i.e. $n_1 < n_2$ implies $\iota_{\mathbb{X}}(n_1) < \iota_{\mathbb{X}}(n_2)$ for all $n_1, n_2 \in \mathbb{N}$. We write $\dim(\boldsymbol{x})$ to mean the dimension of $\boldsymbol{x}$ and $|\boldsymbol{x}|$ to mean the length of $\boldsymbol{x}$. Hence, $\dim(\boldsymbol{x}) \leq |\boldsymbol{x}|$ and $\iota_{\mathbb{X}}(\dim(\boldsymbol{x})) = |\boldsymbol{x}|$. We extend the notion of dimension to traces and say a trace $\boldsymbol{t} \in \mathbb{T}$ has dimension $n$ if $|\boldsymbol{t}| = \iota_{\mathbb{X}}(n)$.

[1] We write $\mathsf{zip}(\ell_1, \ell_2)$ to be the $n$-length vector $[(\ell_1{}^1, \ell_2{}^1), (\ell_1{}^2, \ell_2{}^2), \ldots, (\ell_1{}^n, \ell_2{}^n)] \in (L_1 \times L_2)^n$ for any vectors $\ell_1 \in L_1^{n_1}$ and $\ell_2 \in L_2^{n_2}$ with $n := \min\{n_1, n_2\}$.

We distinguish the length $\iota_{\mathbb{X}}(n)$ and dimension $n$ of variables to better accommodate techniques in Sec. 5.5. The variable $\boldsymbol{x}$ in these techniques might contain extra information we would like to keep track of. These extra information would be stored in the first $\iota_{\mathbb{X}}(n) - n$ coordinates of $\boldsymbol{x}$, whereas the last $n$ coordinates of $\boldsymbol{x}$ gives the value of the variable.

Formally, the $n$-dimensional parameter space $(\mathbb{X}^{(n)}, \Sigma_{\mathbb{X}^{(n)}})$ is the product of $\iota_{\mathbb{X}}(n)$ copies of the entropy space $(\mathbb{E}, \Sigma_{\mathbb{E}})$ and the base measure $\mu_{\mathbb{X}^{(n)}}$ on $\mathbb{X}^{(n)}$ is the product of $\iota_{\mathbb{X}}(n)$ copies of the entropy measure $\mu_{\mathbb{E}}$ with the Radon-Nikodym derivative $\varphi_{\mathbb{X}^{(n)}}$. For ease of reference, we write $(\mathbb{X}, \Sigma_{\mathbb{X}}, \mu_{\mathbb{X}})$ for the one-dimensional parameter space.

### 5.2.3 Auxiliary Space

Similarly, an auxiliary variable $\boldsymbol{v}$ of dimension $n$ is an entropy vector of length $\iota_{\mathbb{Y}}(n)$ where $\iota_{\mathbb{Y}} : \mathbb{N} \to \mathbb{N}$ is a monotone map. The $n$-dimensional auxiliary space $(\mathbb{Y}^{(n)}, \Sigma_{\mathbb{Y}^{(n)}})$ is the product of $\iota_{\mathbb{Y}}(n)$ copies of the entropy space $(\mathbb{E}, \Sigma_{\mathbb{E}})$ and the base measure $\mu_{\mathbb{Y}^{(n)}}$ on $\mathbb{Y}^{(n)}$ is the product of $\iota_{\mathbb{Y}}(n)$ copies of the entropy measure $\mu_{\mathbb{E}}$ with the Radon-Nikodym derivative $\varphi_{\mathbb{Y}^{(n)}}$. For ease of reference, we write $(\mathbb{Y}, \Sigma_{\mathbb{Y}}, \mu_{\mathbb{Y}})$ for the one-dimensional auxiliary space.

### 5.2.4 State Space

A ***state*** is a pair of *equal dimension but not necessarily equal length* parameter and auxiliary variable. For instance with $\iota_{\mathbb{X}}(n) \coloneqq n+1$ and $\iota_{\mathbb{Y}}(n) \coloneqq n$, the parameter variable $\boldsymbol{x} \coloneqq [(-0.2, \mathsf{T}), (2.9, \mathsf{T}), (1.3, \mathsf{F})]$ and the auxiliary variable $\boldsymbol{v} \coloneqq [(1.5, \mathsf{T}), (-2.1, \mathsf{F})]$ are both of dimension two and $(\boldsymbol{x}, \boldsymbol{v})$ is a two-dimensional state.

Formally, the ***state space*** $\mathbb{S}$ is the list measurable space of the product of parameter and auxiliary spaces of equal dimension, i.e. $\mathbb{S} \coloneqq \bigcup_{n \in \mathbb{N}} (\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)})$, equipped with the $\sigma$-algebra $\Sigma_{\mathbb{S}} \coloneqq \sigma\{X_n \times V_n \mid X_n \in \Sigma_{\mathbb{X}^{(n)}}, V_n \in \Sigma_{\mathbb{Y}^{(n)}}, n \in \mathbb{N}\}$ and measure $\mu_{\mathbb{S}}(S) \coloneqq \sum_{n=1}^{\infty} \int_{\mathbb{Y}^{(n)}} \mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid (\boldsymbol{x}, \boldsymbol{v}) \in S\})\, \mu_{\mathbb{Y}^{(n)}}(\mathrm{d}\boldsymbol{v})$. We write $\mathbb{S}^{(n)}$ for the set consisting of all $n$-dimensional states.

We extend the notion of instances to states and say a trace $\boldsymbol{t}$ is an instance of a state $(\boldsymbol{x}, \boldsymbol{v})$ if it is an instance of the parameter component $\boldsymbol{x}$.

## 5.3 Inputs of the NP-iMCMC Algorithm

Besides the target density function, the NP-iMCMC sampler, like iMCMC, introduces randomness via ***auxiliary kernels*** and moves around the state space via ***involutions*** to propose the next sample. We now examine each of these inputs closely.

### 5.3.1 Target Density Function

Similar to other inference algorithms for probabilistic programming, the NP-iMCMC sampler takes the *weight function* $w : \mathbb{T} \to [0, \infty)$ as the target density function. Recall $w(\boldsymbol{t})$ gives the weight of the run of the probabilistic program indicated by the trace $\boldsymbol{t}$. By Prop. 6, the weight function $w$ is always tree representable. For the sampler to work properly, we also require weight function $w$ to satisfy the following assumptions.

(V1) $w$ is ***integrable***, i.e. $\int_{\mathbb{T}} w \, \mathrm{d}\mu_{\mathbb{T}} =: Z < \infty$ (otherwise, the inference problem is undefined).

(V2) $w$ is ***almost surely terminating (AST)***, i.e. $\mu_{\mathbb{T}}(\{\boldsymbol{t} \in \mathbb{T} \mid w(\boldsymbol{t}) > 0\}) = 1$ (otherwise, the loop in the NP-iMCMC algorithm may not terminate almost surely).

(V3) Every trace in the support of $w$ has a dimension (w.r.t. $\iota_{\mathbb{X}}$), i.e. $\mathsf{Supp}(w) = \bigcup_{n \in \mathbb{N}} \mathsf{Supp}^{\iota_{\mathbb{X}}(n)}(w)$.

Virtually all useful probabilistic models can be specified by SPCF programs with densities satisfying V 1 and 2. Exceptions are models that are not normalizable or diverge with non-zero probability. (See Sec. 3.3.3 for more details.) As $\iota_{\mathbb{X}}$ is used internally (to accommodate the techniques in Sec. 5.5), we expect all probabilistic models specified by SPCF programs satisfy V 3.

### 5.3.2 Auxiliary kernels

To introduce randomness, the NP-iMCMC sampler takes, for each $n \in \mathbb{N}$, a probability ***auxiliary kernel*** $K^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$ which gives a probability distribution $K^{(n)}(\boldsymbol{x}, \cdot)$ on $\mathbb{Y}^{(n)}$ for each $n$-dimensional parameter variable $\boldsymbol{x}$. We assume each auxiliary kernel $K^{(n)}$ has a probability density function (pdf) $\mathsf{pdf} K^{(n)} : \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)} \to [0, \infty)$ w.r.t. $\mu_{\mathbb{Y}^{(n)}}$.

### 5.3.3 Involutions

To move around the state space $\mathbb{S}$, the NP-iMCMC sampler takes, for each $n \in \mathbb{N}$, an endofunction $\Phi^{(n)}$ on $\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$ that is both involutive and differentiable almost everywhere. We require the set $\{\Phi^{(n)}\}_n$ of involutions to satisfy the *projection commutation property*:

(V4) For all $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}$ where $\dim(\boldsymbol{x}) = m$, if $\mathsf{Supp}^{\iota_{\mathbb{X}}(n)}(w) \cap \mathsf{instance}(\boldsymbol{x}) \neq \varnothing$ for some $n$, then for all $k = n, \ldots, m$, $\mathsf{take}_k(\Phi^{(m)}(\boldsymbol{x}, \boldsymbol{v})) = \Phi^{(k)}(\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}))$

where $\mathsf{take}_k$ is the projection that takes a state $(\boldsymbol{x}, \boldsymbol{v})$ and returns a $k$-dimensional state $(\boldsymbol{x}^{1\ldots\iota_{\mathbb{X}}(k)}, \boldsymbol{v}^{1\ldots\iota_{\mathbb{Y}}(k)})$ with the first $\iota_{\mathbb{X}}(k)$ coordinates of $\boldsymbol{x}$ and the first $\iota_{\mathbb{Y}}(k)$ coordinates of $\boldsymbol{v}$.

The projection commutation property ensures that the order of applying a projection and an involution to a state (which has an instance in the support of the target density function) does not matter.

## 5.4 The NP-iMCMC Algorithm

After identifying the state space and the necessary conditions on the inputs of the NP-iMCMC sampler, we have enough foundation to describe the algorithm.

Given a SPCF program $M$ with weight function $w$ on the trace space $\mathbb{T}$, the ***Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC)*** algorithm generates a Markov chain on $\mathbb{T}$ as follows. Given a current sample $\boldsymbol{t}_0$ of dimension $k_0$ (i.e. $|\boldsymbol{t}_0| = \iota_{\mathbb{X}}(k_0)$),

1. (Initialisation Step) Form a $k_0$-dimensional parameter variable $\boldsymbol{x}_0 \in \mathbb{X}^{(k_0)}$ by pairing each element ${\boldsymbol{t}_0}^i$ in the trace $\boldsymbol{t}_0$ with a randomly drawn value $t$ of the other type to make a pair $({\boldsymbol{t}_0}^i, t)$ or $(t, {\boldsymbol{t}_0}^i)$ in the entropy space $\mathbb{E} := \mathbb{R} \times \mathbb{2}$. Note that $\boldsymbol{t}_0$ is the unique instance of $\boldsymbol{x}_0$ that is in the support of $w$.

2. (Stochastic Step) Introduce randomness to the sampler by drawing a $k_0$-dimensional value $\boldsymbol{v}_0 \in \mathbb{Y}^{(k_0)}$ from the probability measure $K^{(k_0)}(\boldsymbol{x}_0, \cdot)$.

3. (Deterministic Step) Move around the $n$-dimensional state space $\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$ and compute the new state $(\boldsymbol{x}, \boldsymbol{v})$ by applying the involution $\Phi^{(n)}$ to the *initial state* $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ where $n = \dim(\boldsymbol{x}_0) = \dim(\boldsymbol{v}_0)$.

4. (Extend Step) Test whether any instance $\boldsymbol{t}$ of $\boldsymbol{x}$ is in the support of $w$. If so, proceed to the next step with $\boldsymbol{t}$ as the proposed sample; otherwise

    i. Extend the $n$-dimensional initial state to a state $(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)$ of dimension $n+1$ where $\boldsymbol{y}_0$ and $\boldsymbol{u}_0$ are values drawn randomly from $\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n+1)-\iota_{\mathbb{X}}(n)}}$ and $\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n+1)-\iota_{\mathbb{Y}}(n)}}$ respectively,

    ii. Go to Step 3 with an incremented $n$ and the initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ replaced by $(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)$.

5. (Accept/reject Step) Accept the proposed sample $\boldsymbol{t}$ as the next sample with probability

$$\min\left\{1;\ \frac{w(\boldsymbol{t})\cdot \mathsf{pdf}\, K^{(k)}(\mathsf{take}_k(\boldsymbol{x},\boldsymbol{v}))\cdot \varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x})\cdot \varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v})}{w(\boldsymbol{t}_0)\cdot \mathsf{pdf}\, K^{(k_0)}(\mathsf{take}_{k_0}(\boldsymbol{x}_0,\boldsymbol{v}_0))\cdot \varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0)\cdot \varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)}\cdot \right. \tag{5.1}$$
$$\left. |\det(\nabla \Phi^{(n)}(\boldsymbol{x}_0,\boldsymbol{v}_0))|\right\}$$

where $n = \dim(\boldsymbol{x}_0) = \dim(\boldsymbol{v}_0)$, $k$ is the dimension of $\boldsymbol{t}$ and $k_0$ is the dimension of $\boldsymbol{t}_0$; otherwise reject the proposal and repeat $\boldsymbol{t}_0$.

*Remark* 23. The integrable assumption on the target density (V 1) ensures the inference problem is well-defined. The assumption that all traces in the support of the target density has a dimension (V 3) confirms that the notion of dimension makes sense in the support of $w$. The almost surely terminating assumption on the target density (V 2) guarantees that the NP-iMCMC sampler almost surely terminates. (Lem. 2 for a concrete proof.) The projection commutation property on the involutions (V 4) allows us to define the invariant distribution.

### 5.4.1 Movement Between Samples of Varying Dimensions

As mentioned earlier (Sec. 5.1), all MCMC samplers that simulate a nonparametric model must decide how to move between samples of varying dimensions. We now discuss how the NP-iMCMC sampler achieves this.

**Form initial and new states in the same dimension** Say the current sample $\boldsymbol{t}_0$ has a dimension of $k_0$. Step 1 to 3 form a $k_0$-dimensional initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ and a new $k_0$-dimensional state $(\boldsymbol{x}, \boldsymbol{v})$.

**Move between dimensions** The novelty of NP-iMCMC is its ability to generate a proposed sample $\boldsymbol{t}$ in the support of the target density $w$ which may not be of same dimension as $\boldsymbol{t}_0$. This is achieved by Step 4.

**Propose a sample of a lower dimension** Step 4 first checks whether any instance of the parameter-component $\boldsymbol{x} \in \mathbb{X}^{(k_0)}$ of the new state (computed in Step 3) is in the support of $w$. If so, we proceed to Step 5 with that instance, say $\boldsymbol{t}$, as the proposed sample.

Say the dimension of $\boldsymbol{t}$ is $k$. Then, we must have $k \leq k_0$ as by the definition of instances, the instance $\boldsymbol{t} \in \mathbb{T}$ of a $k_0$-dimensional parameter $\boldsymbol{x} \in \mathbb{X}^{(k_0)}$ must have a dimension that is lower than or equal to $k_0$. Hence, the dimension of the proposed sample $\boldsymbol{t}$ is lower than or equal to the current sample $\boldsymbol{t}_0$.

**Listing 5.2:** *Pseudocode of the NP-iMCMC algorithm*

```
def NPiMCMC(t0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0](x0)                          # stochastic step
  (x,v) = involution[k0](x0,v0)                   # deterministic step
  n = k0                                          # extend step
  while not intersect(instance(x),support(w)):
    x0 = x0 + [(normal, coin)]*(indexX(n+1)-indexX(n))
    v0 = v0 + [(normal, coin)]*(indexY(n+1)-indexY(n))
    n = n + 1
    (x,v) = involution[n](x0,v0)
  t = intersect(instance(x),support(w))[0]        # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k](proj((x,v),k))/
                                 pdfauxkernel[k0](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               absdetjacinv[n](x0,v0)}
           else t0
```

**Propose a sample of a higher dimension** Otherwise (i.e. if none of the instances of $\boldsymbol{x} \in \mathbb{X}^{(k_0)}$ is in the support of $w$) Step 4 extends the initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{X}^{(k_0)} \times \mathbb{Y}^{(k_0)}$ to $(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0) \in \mathbb{X}^{(k_0+1)} \times \mathbb{Y}^{(k_0+1)}$; and computes a new $(k_0+1)$-dimensional state $(\boldsymbol{x} +\!\!+ \boldsymbol{y}, \boldsymbol{v} +\!\!+ \boldsymbol{u}) \in \mathbb{X}^{(k_0+1)} \times \mathbb{Y}^{(k_0+1)}$ (via Step 3). This process of incrementing the dimensions of both the initial and new states is repeated until an instance $\boldsymbol{t}$ of the new state, say of dimension $n$, is in the support of $w$. At which point, the proposed sample is set to be $\boldsymbol{t}$.

Say the dimension of $\boldsymbol{t}$ is $k$. Then, we must have $k > k_0$ as $\boldsymbol{t}$ is *not* an instance of the $k_0$-dimensional parameter $\boldsymbol{x} \in \mathbb{X}^{(k_0)}$ but one of $\boldsymbol{x} +\!\!+ \boldsymbol{y} \in \mathbb{X}^{(n)}$. Hence, the dimension of the proposed sample $\boldsymbol{t}$ is higher than the current sample $\boldsymbol{t}_0$.

**Accept or reject the proposed sample** Say the proposed sample $\boldsymbol{t}$ is of dimension $k$. With the probability given in Eq. (5.1), Step 5 accepts $\boldsymbol{t}$ as the next sample and NP-iMCMC updates the current sample $\boldsymbol{t}_0$ of dimension $k_0$ to a sample $\boldsymbol{t}$ of dimension $k$. Otherwise, the current sample $\boldsymbol{t}_0$ is repeated and the dimension remains unchanged.

### 5.4.2 Pseudocode of NP-iMCMC Algorithm

The `NPiMCMC` function in Listing 5.2 is an implementation of the NP-iMCMC algorithm in SPCF. We assume that the following SPCF types and terms exist. For each $n \in \mathbb{N}$, the SPCF types `T`, `X[n]` and `Y[n]` implement $\mathbb{T}$, $\mathbb{X}^{(n)}$ and $\mathbb{Y}^{(n)}$ respectively; the SPCF

term `w` of type `T -> R` implements the target density $w$; for each $n \in \mathbb{N}$, the SPCF terms `auxkernel[n]` of type `X[n] -> Y[n]` implements the auxiliary kernel $K^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$; `pdfauxkernel[n]` of type `X[n]*Y[n] -> R` implements the probability density function $\mathrm{pdf}K^{(n)} : \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)} \to \mathbb{R}$ of the auxiliary kernel; `involution[n]` of type `X[n]*Y[n] -> X[n]*Y[n]` implements the involution $\Phi^{(n)}$ on $\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$; and `absdetjacinv[n]` of type `X[n]*Y[n] -> R` implements the absolute value of the Jacobian determinant of $\Phi^{(n)}$.

We further assume that the following primitives are implemented: `dim` returns the dimension of a given trace; `indexX` and `indexY` implement the maps $\iota_{\mathbb{Y}}$ and $\iota_{\mathbb{X}}$ respectively; `pdfpar[n]` implements the derivative $\varphi_{\mathbb{X}^{(n)}}$ of the $n$-dimensional parameter space $\mathbb{X}^{(n)}$; `pdfaux[n]` implements the derivative $\varphi_{\mathbb{Y}^{(n)}}$ of the $n$-dimensional auxiliary space $\mathbb{Y}^{(n)}$; `instance` returns a set of all instances of a given entropy vector; `support` returns a set of traces in the support of a given function; and `proj` implements the projection function where `proj((x,v),k)=(x[:indexX(k)],v[:indexY(k)])`.

### 5.4.3 Nonparametric Metropolis-Hastings

Let us now extend the Metropolis-Hastings sampler to work on nonparametric models using the NP-iMCMC method.

As discussed in Sec. 4.2.3, the standard MH sampler can be seen as an instance of the iMCMC sampler with the proposal distribution $q$ as the auxiliary kernel and a swap function as the involution. What would be the resulting algorithm if we take the NP-iMCMC method described in Sec. 5.4 with these inputs?

To keep things simple, we assume the parameter and auxiliary variables do not hold extra information and set both $\iota_{\mathbb{X}}$ and $\iota_{\mathbb{Y}}$ to be identities. This means $\mathbb{X}^{(n)} = \mathbb{Y}^{(n)} = \mathbb{E}^n$ for all $n \in \mathbb{N}$ and V3 holds trivially. Moreover, we assume a proposal kernel $q^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$ exists for each $n \in \mathbb{N}$. Then, as long as the target density function $w$ is integrable (V1) and almost surely terminating (V2), we have a nonparametric extension of the MH sampler!

The `NPMH` function in Listing 5.3 is a SPCF implementation of this sampler. It can seen as an instance of the `NPiMCMC` function with `auxkernel[n]` replaced by the proposal distribution `q[n]`, `pdfauxkernel[n]` replaced by the pdf of the proposal distribution `pdfq[n]`, `involution[n]` replaced by a swap function, and `indexX` and `indexY` replaced by identities, alongside a simplified acceptance ratio as $(\boldsymbol{x}, \boldsymbol{v}) = (\boldsymbol{v}_0, \boldsymbol{x}_0)$, $\varphi_{\mathbb{X}^{(n)}} = \varphi_{\mathbb{Y}^{(n)}}$, and

$$\frac{\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x})}{\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0)} \cdot \frac{\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v})}{\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)} \cdot \left|\det \nabla \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0)\right| = 1.$$

**Listing 5.3:** *Pseudocode of the NP-MH algorithm*

```
def NPMH(t0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = q[k0](x0)                                  # stochastic step
  (x,v) = (v0,x0)                                 # deterministic step
  while not intersect(instance(x),support(w)):    # extend step
    x0 = x0 + [(normal, coin)]
    v0 = v0 + [(normal, coin)]
    (x,v) = (v0,x0)
  t = intersect(instance(x),support(w))[0]        # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfq[k](proj((x,v),k))/
                                 pdfq[k0](proj((x0,v0),k0))}
          else t0
```

To see `NPMH` in action, recall the example in Sec. 5.1 where we tried and failed to simulate the infinite Gaussian mixture model `iGMM` using the MH sampler.

Given the weight function `w` of the model `iGMM` as the target density, how does the `NPMH` sampler generate a new sample from the current sample `t0 = [3.4, -1.2, 1.0, 0.5]`? (For simplicity we only consider the Real-components of the entropy vectors.) Say after the initialisation, stochastic and deterministic steps in `NPMH`, we have `x0 = [3.4, -1.2, 1.0, 0.5]`, `v0 = [4.3, -3.4, -0.1, 1.4]` and `(x,v) = (v0,x0)`.

The extend step then tests whether any instance of `x = [4.3, -3.4, -0.1, 1.4]` is in the support of `w`. Since the `iGMM` term does not terminate with a trace specified by any prefixes of `[4.3, -3.4, -0.1, 1.4]`, the body of the while loop is executed. Say `-0.7` and `-0.3` are drawn and we have `x0 = [3.4, -1.2, 1.0, 0.5, -0.7]`, `v0 = [4.3, -3.4, -0.1, 1.4, -0.3]` and `(x,v) = (v0,x0)`.

Again the extend step checks whether any instance of `x = [4.3, -3.4, -0.1, 1.4, -0.3]` is in the support of `w`. Since the model `iGMM` does terminate with a trace specified by `x = [4.3, -3.4, -0.1, 1.4, -0.3]` (each of the four Gaussian distributions has a value for their mean), we exit the while loop and propose `[4.3, -3.4, -0.1, 1.4, -0.3]` to be the next sample `t`.

Lastly, the proposed sample `t` is accepted with probability

```
min{1, w(t)/w(t0) * pdfq[k](proj((x,v),k))/pdfq[k0](proj((x0,v0),k0))}
```

otherwise the current sample `t0` is returned.

Unlike the classic MH sampler, this new `NPMH` sampler is able to generate a new sample `t` of dimension five from a current sample `t0` of dimension four. Hence it is more suitable for nonparametric models like `iGMM`.

Let us consider another instance where after the initialisation, stochastic and deterministic steps, we have `x0 = [3.4, -1.2, 1.0, 0.5]`, `v0 = [2.5, -0.1, 2.1, -0.7]` and `(x,v) = (v0,x0)`. Since `[2.5, -0.1, 2.1]` is an instance of `x` that is in the support of `w`, the extend step will *not* increase the dimension and instead proceed to the accept/reject step with `[2.5, -0.1, 2.1]` as the proposal. If it is accepted, the `NPMH` sampler decreases the dimension (from four to three).

## 5.5 Techniques on NP-iMCMC Algorithms

In this section, we discuss how the techniques of the iMCMC sampler discussed in Sec. 4.3 can be applied on the NP-iMCMC sampler. We assume the input target density function $w : \mathbb{T} \to [0, \infty)$ is tree representable, integrable (V1) and almost surely terminating (V2).

### 5.5.1 State-dependent NP-iMCMC Mixture

Say we want to use multiple NP-iMCMC samplers to simulate the posterior given by the target density function $w$. The following technique allows us to 'mix' NP-iMCMC samplers in such a way that the resulting sampler still preserves the posterior.

Given a collection of NP-iMCMC samplers, indexed by $m \in \mathbb{E}^\ell$, each with auxiliary kernels $\{K_m^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}\}_{n\in\mathbb{N}}$ and involutions $\{\Phi_m^{(n)} : \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)} \to \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}\}_{n\in\mathbb{N}}$ satisfying the projection commutation property (V4), the ***State-dependent NP-iMCMC Mixture*** sampler determines which NP-iMCMC sampler to use by drawing an indicator $m \in \mathbb{E}^\ell$ from a probability measure $K_M(\boldsymbol{x}_0, \cdot)$ where $K_M : \bigcup_{n\in\mathbb{N}} \mathbb{X}^{(n)} \rightsquigarrow \mathbb{E}^m$ is a probability kernel and $\boldsymbol{x}_0$ is the parameter variable constructed from the current sample $\boldsymbol{t}_0$ at the initialisation step (Step 1 of NP-iMCMC). Then, using the $m$-indexed NP-iMCMC sampler, a proposal $\boldsymbol{t}$ is generated and accepted with a modified probability that includes the probability of picking $m$, namely

$$\min\left\{1;\ \frac{w(\boldsymbol{t})\cdot \mathsf{pdf}{K^{(k)}}_m(\mathsf{take}_k(\boldsymbol{x},\boldsymbol{v}))\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x})\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v})}{w(\boldsymbol{t}_0)\cdot \mathsf{pdf}{K^{(k_0)}}_m(\mathsf{take}_{k_0}(\boldsymbol{x}_0,\boldsymbol{v}_0))\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0)\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)}\cdot \frac{\mathsf{pdf}K_M({\boldsymbol{x}_0}^{1\ldots k_0}, m)}{\mathsf{pdf}K_M(\boldsymbol{x}^{1\ldots k}, m)}\cdot\left|\det(\nabla\Phi_m^{(n)}(\boldsymbol{x}_0,\boldsymbol{v}_0))\right|\right\}$$

where $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ is the (possibly extended) initial state, $(\boldsymbol{x}, \boldsymbol{v})$ is the new state, $n = \dim(\boldsymbol{x}_0) = \dim(\boldsymbol{v}_0)$, $k_0$ is the dimension of $\boldsymbol{t}_0$ (i.e. $|\boldsymbol{t}_0| = \iota_{\mathbb{X}}(k_0)$) and $k$ is the dimension of $\boldsymbol{t}$ (i.e. $|\boldsymbol{t}| = \iota_{\mathbb{X}}(k)$).

**Pseudocode** This sampler can be implemented in SPCF as the `MixtureNPiMCMC` function in Listing 5.4. (Terms specific to this technique are highlighted.) We assume the following SPCF terms exists: `mixkernel` of type `List(X) -> (R*B)^l` implements the mixture kernel $K_M : \bigcup_{n\in\mathbb{N}} \mathbb{X}^{(n)} \rightsquigarrow \mathbb{E}^\ell$; `pdfmixkernel` of type `List(X)*(R*B)^l -> R` implements the probability density function $\mathrm{pdf}K_M : \bigcup_{n\in\mathbb{N}} \mathbb{X}^{(n)} \times \mathbb{E}^\ell \to \mathbb{R}$ ; and for each $m \in \mathbb{E}^\ell$ and $n \in \mathbb{N}$, `auxkernel[n][m]` implements the auxiliary kernel $K_m^{(n)}$; `pdfauxkernel[n][m]` implements the pdf $\mathrm{pdf}{K^{(n)}}_m$; `involution[n][m]` implements the involution $\Phi_m^{(n)}$; and `absdetjacinv[n][m]` implements the absolute value of the Jacobian determinant of $\Phi_m^{(n)}$.

**Correctness** Similar to the correctness arguments in Sec. 4.3, we show that the State-dependent NP-iMCMC Mixture sampler is correct by formulating `MixtureNPiMCMC` as an instance of `NPiMCMC` (Listing 5.2). This means specifying `auxkernel[n]` and `involution[n]` in `NPiMCMC` and arguing that the resulting `NPiMCMC` function is equivalent to `MixtureNPiMCMC`.

The SPCF terms `mixauxkernel[n]` and `mixinvolution[n]` given in Listing 5.5 should suffice. The auxiliary space is expanded to embed the indicator `m` in such a way that the auxiliary variable `mixv` is in the space $\mathbb{E}^\ell \times \mathbb{Y}^{(n)}$ where its first $\ell$ components `mixv[:l]` give `m` and the rest `mixv[l:]` gives `v`. Since the auxiliary space is expanded to include the indicator, the maps `mixindexX` and `mixindexY` and the projection `mixproj` are modified accordingly.

To see how the `NPiMCMC` function with `auxkernel[n]` replaced by `mixauxkernel[n]` and `involution[n]` replaced by `mixinvolution[n]` is equivalent to `MixtureNPiMCMC`, we consider the probability density of `mixauxkernel[k]` at `mixproj((x,mixv),k)`.

```
  pdfmixauxkernel[k](x[:mixindexX(k)], mixv[:mixindexY(k)])
= pdfmixauxkernel[k](x[:indexX(k)], mixv[:l+indexY(k)])
= pdfmixkernel(x[:indexX(k)], mixv[:l]) * pdfauxkernel[k][mixv[:l]](x[:
    indexX(k)], mixv[l:l+indexY(k)])
= pdfmixkernel(x[:indexX(k)], m) * pdfauxkernel[k][m](x[:indexX(k)], v[:
    indexY(k)])
= pdfmixkernel(proj(x,k),m) * pdfauxkernel[k][m](proj((x,v),k))
```

where `m = mixv[:l]` and `v = mixv[l:]`. This shows that the acceptance probability in `NPiMCMC` is identical to that in `MixtureNPiMCMC` and hence the two algorithms are equivalent.

**Listing 5.4:** *Pseudocode of the State-dependent NP-iMCMC Mixture algorithm*

```
def MixtureNPiMCMC(t0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  m = mixkernel(x0)                               # mixture step
  v0 = auxkernel[k0][m](x0)                       # stochastic step
  (x,v) = involution[k0][m](x0,v0)                # deterministic step
  n = k0                                          # extend step
  while not intersect(instance(x),support(w)):
    x0 = x0 + [(normal, coin)]*(indexX(n+1)-indexX(n))
    v0 = v0 + [(normal, coin)]*(indexY(n+1)-indexY(n))
    n = n + 1
    (x,v) = involution[n][m](x0,v0)
  t = intersect(instance(x),support(w))[0]        # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k][m](proj((x,v),k))/
                                 pdfauxkernel[k0][m](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               pdfmixkernel(proj(x,k),m)/
                                 pdfmixkernel(proj(x0,k0),m) *
                               absdetjacinv[n][m](x0,v0)}
           else t0
```

**Listing 5.5:** *Pseudocode for* `mixauxkernel` *and* `mixinvolution`

```
def mixauxkernel[n](x0):
  m = mixkernel(x0)
  v0 = auxkernel[n][m](x0)
  return m + v0

def pdfmixauxkernel[n](x,mixv):
  m = mixv[:l]
  v = mixv[l:]
  return pdfmixkernel(x, m) * pdfauxkernel[n][m](x, v)

def mixinvolution[n](x0,mixv0):
  m = mixv0[:l]
  v0 = mixv0[l:]
  (x,v) = involution[n][m](x0,v0)
  return (x,m + v)

mixindexX = indexX
mixindexY(n) = l + indexY(n)
mixproj((x,v),k) = (x[:mixindexX(k)],v[:mixindexY(k)])
```

### 5.5.2 Direction NP-iMCMC Algorithm

Sometimes it is difficult to specify involutions that explore the model fully. The following technique tells us that bijections are good enough.

Given endofunctions $f^{(n)}$ on $\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$ that are differentiable almost everywhere and bijective for each $n \in \mathbb{N}$ such that the sets $\{f^{(n)}\}_n$ and $\{f^{(n)^{-1}}\}_n$ satisfy the projection commutative property (V4), the ***Direction NP-iMCMC*** algorithm randomly uses either $f^{(n)}$ or $f^{(n)^{-1}}$ to move around the state space and proposes a new sample.

**Pseudocode** This sampler can be expressed in SPCF as the `DirectionNPiMCMC` function in Listing 5.6. (Terms specific to this technique are highlighted.) We assume for each $n \in \mathbb{N}$ and $d \in 2$, there is a SPCF term `bijection[n][d]` where `bijection[n][True]` implements the bijection $f^{(n)}$ and `bijection[n][False]` implements the inverse $f^{(n)^{-1}}$ and SPCF term `absdetjacbij[n][d]` that implements the absolute value of the Jacobian determinant of $f^{(n)}$ if `d = True` and the inverse $f^{(n)^{-1}}$ otherwise.

**Correctness** We show that `DirectionNPiMCMC` can be formulated as an instance of `NPiMCMC` (Listing 5.2) with a specification of `auxkernel[n]` and `involution[n]`.

The SPCF terms `dirauxkernel[n]` and `dirinvolution[n]` in Listing 5.7 would work. The auxiliary space is expanded to include the direction variable `d0` so that the auxiliary variable `dirv0` is in the space $\mathbb{E} \times \mathbb{Y}^{(n)}$ where the Boolean-component `dirv0[0][1]` of its first coordinate gives `d0` and the remaining coordinates `dirv0[1:]` give `v0`. (Note the value of `dirv0[0][0]` is redundant and is only used to make `dirv0[0]` an entropy.) Since the auxiliary space is expanded, the maps `dirindexX` and `dirindexY` and the projection `dirproj` are modified accordingly.

To see how the `NPiMCMC` function with `auxkernel[n]` replaced by `dirauxkernel[n]` and `involution[n]` replaced by `dirinvolution[n]` is equivalent to `DirectionNPiMCMC`, we first consider the density of `dirauxkernel[k0]` at `dirproj((x0,dirv0),k0)`.

```
  pdfdirauxkernel[k0](x0[:dirindexX(k0)], dirv0[:dirindexY(k0)])
= pdfdirauxkernel[k0](x0[:indexX(k0)], dirv0[:1+indexY(k0)])
= pdfcoin(dirv0[0][1]) * pdfnormal(dirv0[0][0]) * pdfauxkernel[k0](x0[:
    indexX(k0)], dirv0[1:1+indexY(k0)])
= 0.5 * pdfnormal(dirv0[0][0]) * pdfauxkernel[k0](proj((x0,v0),k0))
```

where `v0 = dirv0[1:]`. A similar argument can be made for `pdfdirauxkernel[k](dirproj((x,dirv),k))`, which makes the acceptance probability in `NPiMCMC` identical to that in `DirectionNPiMCMC`. Moreover, writing `d0` for `dirv0[0][1]`, the absolute value of the Jacobian determinant of `dirinvolution[n]` at `(x0,dirv0)` is `absdetjacbij[n][d0](x0,v0)`. Most importantly, `dirinvolution[n]` is now involutive. Hence, the resulting `NPiMCMC` algorithm is the same as `DirectionNPiMCMC`.

**Listing 5.6:** *Pseudocode of the Direction NP-iMCMC algorithm*

```
def DirectionNPiMCMC(t0):
  d0 = coin                                          # direction step
  k0 = dim(t0)                                       # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0](x0)                             # stochastic step
  (x,v) = bijection[k0][d0](x0,v0)                   # deterministic step
  n = k0                                             # extend step
  while not intersect(instance(x),support(w)):
    x0 = x0 + [(normal, coin)]*(indexX(n+1)-indexX(n))
    v0 = v0 + [(normal, coin)]*(indexY(n+1)-indexY(n))
    n = n + 1
    (x,v) = bijection[n][d0](x0,v0)
  d = not d0  # not used
  t = intersect(instance(x),support(w))[0]           # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k](proj((x,v),k))/
                                 pdfauxkernel[k0](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               absdetjacbij[n][d0](x0,v0)}
           else t0
```

**Listing 5.7:** *Pseudocode for* `dirauxkernel` *and* `dirinvolution`

```
def dirauxkernel[n](x0):
  d0 = coin
  v0 = auxkernel[n](x0)
  return [(normal, d0)] + v0

def pdfdirauxkernel[n](x0,dirv0):
  d0 = dirv0[0][1]
  v0 = dirv0[1:]
  return pdfcoin(d0) * pdfnormal(dirv0[0][0]) * pdfauxkernel[n](x0,v0)

def dirinvolution[n](x0,dirv0):
  d0 = dirv0[0][1]
  v0 = dirv0[1:]
  (x,v) = bijection[n][d0](x0,v0)
  d = not d0
  return (x, [(dirv0[0][0],d)] + v)

dirindexX = indexX
dirindexY(n) = 1+indexY(n)
dirproj((x,v),k) = (x[:dirindexX(k)], v[:dirindexY(k)])
```

### 5.5.3 Persistent NP-iMCMC Algorithm

It is known that irreversible transition kernels (those that do not satisfy detailed balance) have better mixing times, i.e. converge more quickly to the target distribution, compared to reversible ones. The following technique gives us a method to transform NP-iMCMC algorithms to irreversible ones that still preserve the target distribution. The key is to compose the NP-iMCMC sampler with a transition kernel so that the resulting algorithm does not satisfy detailed balance.

The ***Persistent NP-iMCMC*** algorithm is a MCMC algorithm similar to the Direction NP-iMCMC sampler in which the stochastic step (given by a couple of sets of auxiliary kernels, say $\{K_1^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}\}_n$ and $\{K_2^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}\}_n$ ) and/or the deterministic step (given by the set of bijections $\{f^{(n)} : \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)} \to \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}\}_n$) depend on a direction variable $d_0 \in 2$. The difference is that Persistent NP-iMCMC keeps track of the direction (instead of sampling a fresh one in each iteration) and flips it strategically to make the resulting algorithm irreversible.

**Pseudocode** This sampler can be expressed in SPCF as `PersistentNPiMCMC` in Listing 5.8. (Terms specific to this technique are highlighted.) In addition to the SPCF terms in `DirectionNPiMCMC`, we assume there is a SPCF term `auxkernel[n][d]` such that `auxkernel[n][True]` implements the auxiliary kernel $K_1^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$ and `pdfauxkernel[n][True]` its pdf $\mathrm{pdf}K^{(n)}{}_1$ and similarly for `auxkernel[n][False]` and `pdfauxkernel[n][False]`. Note that `PersistentNPiMCMC` updates samples on the space $\mathbb{X}^{(n)} \times 2$, which can easily be marginalised to $\mathbb{X}^{(n)}$ by taking the first $\iota_{\mathbb{X}}(n)$ components.

**Correctness** We show that `PersistentNPiMCMC` can be formulated as an instance of `NPiMCMC` (Listing 5.2) with a specification of `auxkernel[n]` and `involution[n]`.

The SPCF terms `perauxkernel[n]` and `perinvolution[n]` in Listing 5.9 would work. In this case, the parameter space is expanded to include the direction variable so that a parameter variable `perx` is on the space $\mathbb{E} \times \mathbb{X}^{(n)}$ where `perx[0][1]` gives `d` and `perx[1:]` gives `x`. Since the parameter space is expanded, the maps `perindexX` and `perindexY` and projection `perproj` are modified accordingly.

Again, we first consider the density of `perauxkernel[k0]` at `perproj((perx0,v0),k0)`.

```
  pdfperauxkernel[k0](perx0[:perindexX(k0)], v0[:perindexY(k0)])
= pdfauxkernel[k0][perx[0][1]](perx0[1:1+indexX(k0)], v0[:indexY(k0)])
= pdfauxkernel[k0][d0](x0[:indexX(k0)], v0[:indexY(k0)])
= pdfauxkernel[k0][d0](proj((x0,v0),k0))
```

**Listing 5.8:** *Pseudocode of the Persistent NP-iMCMC algorithm*

```
def PersistentNPiMCMC(t0,d0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0][d0](x0)                       # stochastic step
  (x,v) = bijection[k0][d0](x0,v0)                # deterministic step
  n = k0                                          # extend step
  while not intersect(instance(x),support(w)):
    x0 = x0 + [(normal, coin)]*(indexX(n+1)-indexX(n))
    v0 = v0 + [(normal, coin)]*(indexY(n+1)-indexY(n))
    n = n + 1
    (x,v) = bijection[n][d0](x0,v0)
  d = not d0
  t = intersect(instance(x),support(w))[0]        # accept/reject step
  k = dim(t)
  return (t, not d) if uniform < min{1, w(t)/w(t0) *
                                pdfauxkernel[k][d](proj((x,v),k))/
                                  pdfauxkernel[k0][d0](proj((x0,v0),k0)) *
                                pdfpar[n](x)/pdfpar[n](x0) *
                                pdfaux[n](v)/pdfaux[n](v0) *
                                absdetjacbij[n][d0](x0,v0)}
           else (t0, d)
```

**Listing 5.9:** *Pseudocode for* `perauxkernel` *and* `perinvolution`

```
def perauxkernel[n](perx0):
  d0 = perx0[0][1]
  x0 = perx0[1:]
  v0 = auxkernel[n][d0](x0)
  return v0

def pdfperauxkernel[n](perx0, v0):
  d0 = perx0[0][1]
  x0 = perx0[1:]
  return pdfauxkernel[n][d0](x0, v0)

def perinvolution[n](perx0,v0):
  d0 = perx0[0][1]
  x0 = perx0[1:]
  (x,v) = bijection[n][d0](x0,v0)
  d = not d0
  return ([(perx0[0][0],d)] + x, v)

perindexX(n) = 1+indexX(n)
perindexY = indexY
perproj((x,v),k) = (x[:perindexX(k)],v[:perindexY(k)])
```

where `d0 = perx0[0][1]` and `x0 = perx0[1:]`. A similar argument can be made for `pdfperauxkernel[k](perproj((perx,v),k))`. Moreover, the absolute value of the Jacobian determinant of `perinvolution[n]` at `(perx0,v0)` is `absdetjacbij[n][d0](x0,v0)`. Hence, the acceptance probability in `NPiMCMC` is identical to that in `PersistentNPiMCMC`.

The `NPiMCMC` function with `auxkernel[n]` replaced by `perauxkernel[n]` and `involution[n]` replaced by `perinvolution[n]` is *almost* equivalent to `PersistentNPiMCMC`, except `NPiMCMC` induces a transition kernel on $\mathbb{E} \times \mathbb{X}^{(n)}$ whereas `PersistentNPiMCMC` induces a transition kernel on $\mathbb{2} \times \mathbb{X}^{(n)}$; and when the proposal `t` is accepted, `NPiMCMC` returns `d` whereas `PersistentNPiMCMC` returns *`not`* `d`.

These differences can be reconciled by composing `NPiMCMC` with the transition kernel $t : \mathbb{E} \times \mathbb{X}^{(n)} \rightsquigarrow \mathbb{E} \times \mathbb{X}^{(n)}$ defined as $t([(r, d), \boldsymbol{x}], X) := [[(r, \text{not } d), \boldsymbol{x}] \in X]$. The composition generates a Markov chain on $\mathbb{E} \times \mathbb{X}^{(n)}$ and marginalising it to a Markov chain on $\mathbb{2} \times \mathbb{X}^{(n)}$ gives us the same result as `PersistentNPiMCMC`.

## 5.6 Related Work

The involutive MCMC framework (Neklyudov et al., 2020) can in principle be used for nonparametric models. For instance, Reversible Jump MCMC (Green, 1995) is an instance of iMCMC that works for the infinite GMM model, with the split-merge proposal (Richardson and Green, 1997) specifying when and how states can "jump" across dimensions. However, designing appropriate auxiliary kernels and involutions that enable the extension of an iMCMC sampler to nonparametric models remains challenging and model specific. By contrast, NP-iMCMC only requires the specification of involutions on the finite-dimensional space $\mathbb{X}^n \times \mathbb{Y}^n$; moreover, it provides a general procedure (via Step 4) that drives state movement between dimensions. For designers of nonparametric samplers who do not care to custom build trans-dimensional methods, we contend that NP-iMCMC is their method of choice.

The performance of NP-iMCMC and iMCMC depends on the complexity of the respective auxiliary kernels, involutions and the model in question. Take iGMM for example. RJMCMC with the split-merge proposal which computes the weight, mean, and variance of the new component(s) would be slower than NP-MH, an instance of NP-iMCMC with a computationally light involution (a swap), but more efficient than NP-HMC, an instance of (Multiple Step) NP-iMCMC with the computationally heavy leapfrog integrator as involution.

*Confitebor tibi in cithara, Deus, Deus meus. Quare tristis es, anima mea? et quare conturbas me?*

— Psalm 42:5

# 6

# Correctness of Nonparametric Involutive MCMC

The Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC) algorithm presented in Chapter 5 aims to simulate probabilistic models specified by probabilistic programs. In this chapter, we justify this claim by proving that the Markov chain generated by iterating the NP-iMCMC algorithm (Sec. 5.4) preserves the target distribution, specified by

$$\begin{aligned} \nu : \Sigma_{\mathbb{T}} &\longrightarrow [0, \infty) \\ U &\longmapsto \frac{1}{Z} \int_U w \, \mathrm{d}\mu_{\mathbb{T}} \qquad \text{where } Z \coloneqq \int_{\mathbb{T}} w \, \mathrm{d}\mu_{\mathbb{T}}, \end{aligned}$$

as long as the target density function $w$ (given by the weight function of the probabilistic program) is integrable (V1), almost surely terminating (V2) and each trace in the support of $w$ has a dimension w.r.t. $\iota_{\mathbb{X}}$ (V3); with a probability kernel $K^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$ and an endofunction $\Phi^{(n)}$ on $\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$ that is involutive and differentiable almost everywhere for each $n \in \mathbb{N}$ such that $\{\Phi^{(n)}\}_n$ satisfies the projection commutation property (V4).

Throughout this chapter, we assume the assumptions stated above, and prove the following.

(1) The NP-iMCMC sampler almost surely returns a sample for the simulation (Lem. 2).

(2) The state movement in the NP-iMCMC sampler preserves the state distribution (Lem. 3).

(3) The marginalisation of the state distribution which the state movement of NP-iMCMC preserves coincides with the target distribution (Lem. 4).

## 6.1 Almost Sure Termination

We asserted in Rem. 23 that the almost surely terminating assumption (V 2) on the target density guarantees that the NP-iMCMC algorithm almost surely terminates. We justify this claim here.

Step 4 in the NP-iMCMC algorithm repeats itself if the sample-component $\boldsymbol{x}$ of the new state $(\boldsymbol{x}, \boldsymbol{v})$ (computed by applying the involution $\Phi^{(n)}$ on the extended initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$) does not have an instance in the support of $w$. This loop halts almost surely if the measure of the set

$$\{(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{S} \mid (\boldsymbol{x}, \boldsymbol{v}) = \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0) \text{ and } \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}$$

tends to zero as the dimension $n$ tends to infinity. Since $\Phi^{(n)}$ is invertible and $|\det \nabla \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0)| > 0$ for all $n \in \mathbb{N}$ and $(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{S}$,

$$\begin{aligned}
&\mu_{\mathbb{S}}(\{(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{S} \mid (\boldsymbol{x}, \boldsymbol{v}) = \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0) \text{ and } \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \\
&= \Phi^{(n)}_* \mu_{\mathbb{S}}(\{(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \\
&< \mu_{\mathbb{S}}(\{(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \\
&= \mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}).
\end{aligned}$$

Thus it is enough to show that the measure of a $n$-dimensional parameter variable not having any instances in the support of $w$ tends to zero as the dimension $n$ tends to infinity, i.e.

$$\mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \to 0 \qquad \text{as} \qquad n \to \infty.$$

We start with the following proposition which proves that the chance of a $n$-dimensional parameter variable having some instances in the support of $w$ is the same as the chance of the SPCF program with $w$ as its weight function terminating before $\iota_{\mathbb{X}}(n)$ samples.

**Proposition 16.** $\mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing\}) = \mu_{\mathbb{T}}(\bigcup_{i=1}^{n} \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w))$ *for all* $n \in \mathbb{N}$ *and all tree representable function* $w$ *that satisfy V3.*

*Proof.* Let $n \in \mathbb{N}$ and $w$ be a tree representable function.

For each $i \leq n$, we unpack the set $\{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap A \neq \varnothing\}$ of $i$-dimensional parameter variables that have an instance in the set $A \in \Sigma_{\Omega^{\iota_{\mathbb{X}}(i)}}$ of traces of length $\iota_{\mathbb{X}}(i)$ where $\Omega := \mathbb{R} \cup 2$. Write $\pi : \{1, \ldots, \iota_{\mathbb{X}}(i)\} \to \{\mathbb{R}, 2\}$ for the measurable space $\pi(1) \times \pi(2) \times \cdots \times \pi(\iota_{\mathbb{X}}(i))$ with a probability measure $\mu_\pi := \mu_{\Omega^{\iota_{\mathbb{X}}(i)}}$ on $\pi$; $\pi^{-1}$ for the "inverse" measurable space of $\pi$, i.e. $\pi^{-1}(j) := \Omega \setminus \pi(j)$ for all $j \leq \iota_{\mathbb{X}}(i)$; and

$S$ for the set of all such measurable spaces. Then, for any $i$-dimensional parameter variable $\boldsymbol{x}$, $\boldsymbol{t} \in \mathsf{instance}(\boldsymbol{x}) \cap A$ if and only if there is some $\pi \in S$ where $\boldsymbol{t} \in A \cap \pi$ and $\boldsymbol{x} \in \mathsf{zip}(A \cap \pi, \pi^{-1})$. Hence, $\{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap A \neq \varnothing\}$ can be written as $\bigcup_{\pi \in S} \mathsf{zip}(A \cap \pi, \pi^{-1})$. Moreover $\mu_{\mathbb{X}^{(i)}}(\mathsf{zip}(A \cap \pi, \pi^{-1})) = \mu_{\pi}(A \cap \pi) \cdot \mu_{\pi^{-1}}(\pi^{-1}) = \mu_{\pi}(A \cap \pi)$.

Consider the case where $A := \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w)$. Then, we have

$$\{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \neq \varnothing\} = \bigcup_{\pi \in S} \mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi, \pi^{-1}).$$

We first show that the RHS is a disjoint union, i.e. for all $\pi \in S$, $\mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi, \pi^{-1})$ are disjoint. Let $\boldsymbol{x} \in \mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi_1, {\pi_1}^{-1}) \cap \mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi_2, {\pi_2}^{-1})$ where $\pi_1, \pi_2 \in S$. Then, at least one instance $\boldsymbol{t}_1$ of $\boldsymbol{x}$ is in $\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi_1$ and similarly at least one instance $\boldsymbol{t}_2$ of $\boldsymbol{x}$ is in $\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi_2$. By Prop. 15, $\boldsymbol{t}_1 = \boldsymbol{t}_2$ and hence $\pi_1 = \pi_2$.

Since $\mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi, \pi^{-1})$ are disjoint for all $\pi \in S$, we have

$$\begin{aligned}
&\mu_{\mathbb{X}^{(i)}}(\{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \neq \varnothing\}) \\
&= \mu_{\mathbb{X}^{(i)}}(\bigcup_{\pi \in S} \mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi, \pi^{-1})) \\
&= \sum_{\pi \in S} \mu_{\mathbb{X}^{(i)}}(\mathsf{zip}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi, \pi^{-1})) = \sum_{\pi \in S} \mu_{\pi}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi) \\
&= \sum_{\pi \in S} \mu_{\Omega^{\iota_{\mathbb{X}}(i)}}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \cap \pi) = \mu_{\Omega^{\iota_{\mathbb{X}}(i)}}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w)) = \mu_{\mathbb{T}}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w)).
\end{aligned}$$

Finally, by V 3, $\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing\}$ is equal to $\bigcup_{i=1}^{n} \{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \neq \varnothing\} \times \mathbb{E}^{\iota_{\mathbb{X}}(n) - \iota_{\mathbb{X}}(i)}$ and hence

$$\begin{aligned}
&\mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing\}) \\
&= \mu_{\mathbb{X}^{(n)}}(\bigcup_{i=1}^{n} \{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \neq \varnothing\} \times \mathbb{E}^{\iota_{\mathbb{X}}(n) - \iota_{\mathbb{X}}(i)}) \\
&= \sum_{i=1}^{n} \mu_{\mathbb{X}^{(i)}}(\{\boldsymbol{x} \in \mathbb{X}^{(i)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w) \neq \varnothing\}) \\
&= \sum_{i=1}^{n} \mu_{\mathbb{T}}(\mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w)) \\
&= \mu_{\mathbb{T}}(\bigcup_{i=1}^{n} \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w))
\end{aligned}$$

□

Prop. 16 links the termination of the NP-iMCMC sampler with that of the target density function $w$. Hence by assuming that $w$ terminates almost surely (V 2), we can deduce that the NP-iMCMC algorithm almost surely terminates.

**Lemma 2** (Almost Sure Termination)**.** *Assuming V2 and V3, the NP-iMCMC algorithm (Sec. 5.4) almost surely terminates.*

*Proof.* Since $\Phi^{(n)}$ is invertible for all $n \in \mathbb{N}$, and $w$ almost surely terminates (V 2), i.e. $\lim_{m\to\infty} \mu_{\mathbb{T}}(\bigcup_{j=1}^{m} \mathsf{Supp}^{j}(w)) = 1$ and satisfies (V 3), we deduce from Prop. 16 that

$$
\begin{aligned}
&\mu_{\mathbb{S}}(\{(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{S} \mid (\boldsymbol{x}, \boldsymbol{v}) = \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0) \text{ and } \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \\
&< \mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) = \varnothing\}) \\
&= \mu_{\mathbb{X}^{(n)}}(\mathbb{X}^{(n)} \setminus \{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing\}) \\
&= 1 - \mu_{\mathbb{X}^{(n)}}(\{\boldsymbol{x} \in \mathbb{X}^{(n)} \mid \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing\}) \\
&= 1 - \mu_{\mathbb{T}}(\bigcup_{i=1}^{n} \mathsf{Supp}^{\iota_{\mathbb{X}}(i)}(w)) && \text{(Prop. 16)} \\
&\to 1 - 1 = 0 \quad \text{as } n \to \infty. && \text{(V 2)}
\end{aligned}
$$

So the probability of satisfying the condition of the loop in Step 4 of NP-iMCMC sampler tends to zero as the dimension $n$ tends to infinity, making the NP-iMCMC sampler almost surely terminating. □

## 6.2 Invariant State Distribution

After ensuring the NP-iMCMC sampler almost always returns a sample (Lem. 2), we identify the distribution on the states and show that it is invariant against the movement between states of varying dimensions in NP-iMCMC.

### 6.2.1 State Distribution

Recall a state is an equal dimension parameter-auxiliary pair. We define the ***state distribution*** $\pi$ on the state space $\mathbb{S} := \bigcup_{n\in\mathbb{N}} (\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)})$ to be a distribution with density $\zeta$ (with respect to $\mu_{\mathbb{S}}$) given by

$$
\zeta(\boldsymbol{x}, \boldsymbol{v}) := \frac{1}{Z} \cdot w(\boldsymbol{t}) \cdot \mathsf{pdf} K^{(k)}(\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v})) \tag{6.1}
$$

if $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}^{\text{valid}}$ and $\boldsymbol{t} \in \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w)$ has dimension $k$; and 0 otherwise, where $Z := \int_{\mathbb{T}} w \, \mathrm{d}\mu_{\mathbb{T}}$ (which exists by V 1) and $\mathbb{S}^{\text{valid}}$ is the subset of $\mathbb{S}$ consisting of all *valid states*.

*Remark* 24. If there is some trace in $\mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w)$ for a parameter variable $\boldsymbol{x}$, by Prop. 15 this trace $\boldsymbol{t}$ is unique and hence $\boldsymbol{x}$ represents a sample of the target distribution.

We say a $n$-dimensional state $(\boldsymbol{x}, \boldsymbol{v})$ is ***valid*** if

(i) $\mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing$, and

(ii) $(\boldsymbol{y}, \boldsymbol{u}) = \Phi^{(n)}(\boldsymbol{x}, \boldsymbol{v})$ implies $\mathsf{instance}(\boldsymbol{y}) \cap \mathsf{Supp}(w) \neq \varnothing$, and

(iii) $\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}) \notin \mathbb{S}^{\text{valid}}$ for all $k < n$.

Intuitively, valid states are the states which, when transformed by the involution $\Phi^{(n)}$, the instance of the parameter-component of which does not "fall beyond" the support of $w$.

We write $\mathbb{S}_n^{\text{valid}} := \mathbb{S}^{\text{valid}} \cap (\mathbb{X}^{(n)} \times \mathbb{Y}^{(n)})$ to denote the set of all $n$-dimensional valid states. The following proposition shows that involutions preserve the validity of states.

**Proposition 17.** *Assuming V4, the involution $\Phi^{(n)}$ sends $\mathbb{S}_n^{\text{valid}}$ to $\mathbb{S}_n^{\text{valid}}$ for all $n \in \mathbb{N}$. i.e. If $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}_n^{\text{valid}}$, then $(\boldsymbol{y}, \boldsymbol{u}) = \Phi^{(n)}(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}_n^{\text{valid}}$.*

*Proof.* Let $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}_n^{\text{valid}}$ and $(\boldsymbol{y}, \boldsymbol{u}) = \Phi^{(n)}(\boldsymbol{x}, \boldsymbol{v})$. We prove $(\boldsymbol{y}, \boldsymbol{u}) \in \mathbb{S}_n^{\text{valid}}$ by induction on $n \in \mathbb{N}$.

- Let $n = 1$. As $\Phi^{(1)}$ is involutive and $(\boldsymbol{x}, \boldsymbol{v})$ is a valid state, (i) $\mathsf{instance}(\boldsymbol{y}) \cap \mathsf{Supp}(w) \neq \varnothing$ and (ii) $(\boldsymbol{x}, \boldsymbol{v}) = \Phi^{(1)}(\boldsymbol{y}, \boldsymbol{u})$ and $\mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \neq \varnothing$. (iii) holds trivially and hence $(\boldsymbol{y}, \boldsymbol{u}) \in \mathbb{S}_1^{\text{valid}}$.

- Assume for all $m < n$, $(\boldsymbol{z}, \boldsymbol{w}) \in \mathbb{S}_m^{\text{valid}}$ implies $(\boldsymbol{z}', \boldsymbol{w}') = \Phi^{(m)}(\boldsymbol{z}, \boldsymbol{w}) \in \mathbb{S}_m^{\text{valid}}$. Similar to the base case, (i) and (ii) hold as $\Phi^{(n)}$ is involutive and $(\boldsymbol{x}, \boldsymbol{v})$ is a valid state. Assume for contradiction that (iii) does not hold, i.e. there is $k < n$ where $\mathsf{take}_k(\boldsymbol{y}, \boldsymbol{u}) \in \mathbb{S}_k^{\text{valid}}$. As $\mathsf{instance}(\mathsf{take}_k(\boldsymbol{y})) \cap \mathsf{Supp}(w) \neq \varnothing$, by V4 and the inductive hypothesis,

$$\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}) = \mathsf{take}_k(\Phi^{(n)}(\boldsymbol{y}, \boldsymbol{u})) = \Phi^{(k)}(\mathsf{take}_k(\boldsymbol{y}, \boldsymbol{u})) \in \mathbb{S}_k^{\text{valid}}$$

  which contradicts with the fact that $(\boldsymbol{x}, \boldsymbol{v})$ is a valid state.

□

We can partition the set $\mathbb{S}^{\text{valid}}$ of valid states. Let $(\boldsymbol{x}, \boldsymbol{v})$ be a $m$-dimensional valid state. The parameter variable $\boldsymbol{x}$ can be written as $\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) +\!\!+ \boldsymbol{y}$ where $\boldsymbol{t}_1 \in \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w)$ is of dimension $k_0$, $\boldsymbol{t}_2$ is a trace where $\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) = \mathsf{take}_{k_0}(\boldsymbol{x})$, and $\boldsymbol{y} := \boldsymbol{x}^{\iota_{\mathbb{X}}(k_0)+1 \ldots \iota_{\mathbb{X}}(m)}$ is the remaining suffix of $\boldsymbol{x}$. Similarly, the auxiliary variable $\boldsymbol{v}$ can be written as $\boldsymbol{v}_1 +\!\!+ \boldsymbol{v}_2$ where $\boldsymbol{v}_1 := \mathsf{take}_{k_0}(\boldsymbol{v})$ and $\boldsymbol{v}_2 := \boldsymbol{v}^{\iota_{\mathbb{Y}}(k_0)+1 \ldots \iota_{\mathbb{Y}}(m)}$. Hence, we have

$$\begin{aligned}\mathbb{S}^{\text{valid}} = \bigcup_{k_0=1}^{\infty} \bigcup_{m=1}^{\infty} \{ &(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) +\!\!+ \boldsymbol{y}, \boldsymbol{v}_1 +\!\!+ \boldsymbol{v}_2) \in \mathbb{S}_m^{\text{valid}} \mid \\ &\boldsymbol{t}_1 \in \mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w), \boldsymbol{t}_2 \in \mathbb{T}, \boldsymbol{y} \in \mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}, \boldsymbol{v}_1 \in \mathbb{Y}^{(k_0)}, \boldsymbol{v}_2 \in \mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)} \}\end{aligned}$$

and the state distribution $\pi$ on the measurable set $S \in \Sigma_{\mathbb{S}}$ can be written as

$$\begin{aligned}\pi(S) = \sum_{k_0=1}^{\infty} \sum_{m=1}^{\infty} & \int_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}} \int_{\mathbb{Y}(k_0)} \int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}} \int_{\mathbb{T}} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)} \\ & \left[(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) +\!\!+ \boldsymbol{y}, \boldsymbol{v}_1 +\!\!+ \boldsymbol{v}_2) \in S \cap \mathbb{S}_m^{\text{valid}}\right] \cdot \frac{1}{Z} w(\boldsymbol{t}_1) \cdot \mathsf{pdf} K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2), \boldsymbol{v}_1) \\ & \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\, \mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\, \mu_{\mathbb{Y}(k_0)}(\mathrm{d}\boldsymbol{v}_1)\, \mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{v}_2)\end{aligned}$$

We can now show two simple properties of the state distribution.

**Proposition 18.** *Assuming V1,*

*(i) The state distribution $\pi$ is indeed a probability measure, i.e. $\pi(\mathbb{S}) = 1$.*

*(ii) The set of valid states almost surely covers all states w.r.t. the state distribution, i.e. $\pi(\mathbb{S} \setminus \bigcup_{k=1}^{n} \mathbb{S}_k^{valid}) \to 0$ as $n \to \infty$.*

*Proof.* (i) Consider the set $\mathbb{S}^{\text{valid}}$ with the partition discussed above.

$$\begin{aligned}& \pi(\mathbb{S}) \\ &= \pi(\mathbb{S}^{\text{valid}}) \\ &= \sum_{k_0=1}^{\infty} \sum_{m=1}^{\infty} \int_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}} \int_{\mathbb{Y}(k_0)} \int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}} \int_{\mathbb{T}} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)} \\ &\quad \left[(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) +\!\!+ \boldsymbol{y}, \boldsymbol{v}_1 +\!\!+ \boldsymbol{v}_2) \in \mathbb{S}_m^{\text{valid}}\right] \cdot \frac{1}{Z} w(\boldsymbol{t}_1) \cdot \mathsf{pdf} K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2), \boldsymbol{v}_1) \\ &\quad \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\, \mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\, \mu_{\mathbb{Y}(k_0)}(\mathrm{d}\boldsymbol{v}_1)\, \mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{v}_2) \\ &= \sum_{k_0=1}^{\infty} \int_{\mathbb{Y}(k_0)} \int_{\mathbb{T}} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)} \frac{1}{Z} w(\boldsymbol{t}_1) \cdot \mathsf{pdf} K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2), \boldsymbol{v}_1) \cdot \\ &\quad \Big( \sum_{\ell_1=1}^{\infty} \sum_{\ell_2=1}^{\infty} \int_{\mathbb{E}^{\ell_1}} \int_{\mathbb{E}^{\ell_2}} \left[(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2) +\!\!+ \boldsymbol{y}, \boldsymbol{v}_1 +\!\!+ \boldsymbol{v}_2) \in \mathbb{S}^{\text{valid}}\right] \mu_{\mathbb{E}^{\ell_2}}(\mathrm{d}\boldsymbol{y})\, \mu_{\mathbb{E}^{\ell_1}}(\mathrm{d}\boldsymbol{v}_2) \Big) \\ &\quad \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\, \mu_{\mathbb{Y}(k_0)}(\mathrm{d}\boldsymbol{v}_1) \\ &= \sum_{k_0=1}^{\infty} \int_{\mathbb{T}} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)} \frac{1}{Z} w(\boldsymbol{t}_1) \cdot \Big( \int_{\mathbb{Y}(k_0)} \mathsf{pdf} K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1, \boldsymbol{t}_2), \boldsymbol{v}_1)\, \mu_{\mathbb{Y}(k_0)}(\mathrm{d}\boldsymbol{v}_1) \Big) \\ &\quad \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2) \\ &= \sum_{k_0=1}^{\infty} \int_{\mathbb{T}} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)} \frac{1}{Z} w(\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2) \\ &= \int_{\mathsf{Supp}(w)} \frac{1}{Z} w(\boldsymbol{t}_1)\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1) \;= 1.\end{aligned}$$

(ii) Since $\pi$ is a probability distribution and $\pi(\mathbb{S} \setminus \mathbb{S}^{\text{valid}}) = 0$, the series $\sum_{n=1}^{\infty} \pi(\mathbb{S}_n^{\text{valid}})$

**Listing 6.1:** *Pseudocode of the equivalent program of the NP-iMCMC sampler*

```
def eNPiMCMC(x*,v*):
  t0 = intersect(instance(x*),support(w))[0]    # find a valid state
  k0 = dim(t0)
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0](x0)
  (x,v) = involution[k0](x0,v0)
  n = k0
  while not intersect(instance(x),support(w)):
    x0 = x0 + [(normal, coin)]*(indexX(n+1)-indexX(n))
    v0 = v0 + [(normal, coin)]*(indexY(n+1)-indexY(n))
    n = n + 1
    (x,v) = involution[n](x0,v0)
  (x,v) = involution[n](x0,v0)                  # accept/reject proposed
      state
  t = intersect(instance(x),support(w))[0]
  k = dim(t)
  return (x,v) if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k](proj((x,v),k))/
                                 pdfauxkernel[k0](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               absdetjacinv[n](x0,v0)}
          else (x0,v0)
```

which equals $\pi(\bigcup_{n=1}^{\infty} \mathbb{S}_n^{\text{valid}}) = \pi(\mathbb{S}^{\text{valid}}) = 1$ must converge. Hence $\pi(\mathbb{S} \setminus \bigcup_{k=1}^{n} \mathbb{S}_k^{\text{valid}}) = \pi(\mathbb{S}^{\text{valid}} \setminus \bigcup_{k=1}^{n} \mathbb{S}_k^{\text{valid}}) = \sum_{i=n+1}^{\infty} \pi(\mathbb{S}_i^{\text{valid}}) \to 0$ as $n \to \infty$.

□

### 6.2.2 Equivalent Program

Though the NP-iMCMC algorithm (Sec. 5.4) traverses state, it takes and returns *samples* on the trace space $\mathbb{T}$. Hence instead of asking whether the state distribution $\pi$ is invariant against the NP-iMCMC sampler directly, we consider a program which takes and returns *states* and prove the state distribution $\pi$ is *invariant* w.r.t. this program.

Consider the program `eNPiMCMC` in Listing 6.1. It is similar to `NPiMCMC` (Listing 5.2) syntactically except it takes and returns states instead of traces, and has two additional lines (highlighted). It is easy to deduce from Lem. 2 that `eNPiMCMC` almost surely terminates.

### 6.2.3 Invariant Distribution

After identifying the state distribution and presenting the `eNPiMCMC` program, we now prove that the program preserves the distribution by first defining a transition kernel

of a SPCF term.

Take a SPCF program `M` of type `List(X*Y)-> List(X*Y)` where the SPCF types `X` and `Y` implement the parameter space $\mathbb{X}$ and auxiliary space $\mathbb{Y}$ respectively. We define the ***transition kernel*** of `M` to be the kernel $T_{\texttt{M}} : \mathbb{S} \rightsquigarrow \mathbb{S}$ such that

$$T_{\texttt{M}}(\boldsymbol{s}, S) := \int_{\mathsf{value}_{\texttt{M(s)}}^{-1}(S')} \mathsf{weight}_{\texttt{M(s)}}\, \mathrm{d}\mu_{\mathbb{T}} = \langle\!\langle\, \texttt{M(s)} \,\rangle\!\rangle(S')$$

where `s` implements the state $\boldsymbol{s}$ and $S'$ is the set consisting of SPCF terms that implements states in $S$. Intuitively, $T_{\texttt{M}}(\boldsymbol{s}, S)$ gives the probability that the term `M` returns a state in $S$ given the current state $\boldsymbol{s}$.

**Proposition 19.** *Let* `M` *be a SPCF term of type* `List(X*Y)-> List(X*Y)`*. If* `M(s)` *does not contain any scoring subterm and almost surely terminates for all SPCF terms* `s` *of type* `List(X*Y)` *then its transition kernel* $T_{\texttt{M}}$ *is probabilistic.*

*Proof.* Since the term `M(s)` does not contain $\mathsf{score}(\cdot)$ and terminates almost surely, by Prop. 7 its value measure must be probabilistic. Hence $T_{\texttt{M}}(\texttt{s}, \mathbb{S}) = \langle\!\langle\, \texttt{M(s)} \,\rangle\!\rangle(\mathbb{S}') = \langle\!\langle\, \texttt{M(s)} \,\rangle\!\rangle(\Lambda_v^0) = 1$. □

We say a distribution $\mu$ on states $\mathbb{S}$ is ***invariant*** w.r.t. an almost surely terminating SPCF program `M` of type `List(X*Y)-> List(X*Y)` if $\mu$ is not altered after applying `M`, formally $\int_{\mathbb{S}} T_{\texttt{M}}(\boldsymbol{s}, S)\ \mu(\mathrm{d}\boldsymbol{s}) = \mu(S)$.

We now prove that the program `eNPiMCMC` in Listing 6.1 preserves the state distribution $\pi$ stated in Sec. 6.2.1 by considering the transition kernels given by the two steps in `eNPiMCMC` given in Sec. 6.2.2: find a valid state (Lines 2-12) and accept/reject the computed proposed state (Lines 13-22).

**Finding a Valid Initial State** Assuming the initial state `(x*,v*)` is valid, `eNPiMCMC` (Lines 2-12) aims to construct a *valid* state `(x0,v0)` where `x*` and `x0` share the same instance `t0` that is in the support of the density `w`.

To do this, it first finds the instance `t0` of `x*` which is in the support of `w` (Line 2). Say the dimension of `t0` is `k0` (Line 3). It then forms a `k0`-dimensional state `(x0,v0)` by sampling partners `t` for each value in the trace `t0` to form a `k0`-dimensional parameter variable `x0` (Line 4); and drawing a `k0`-dimensional auxiliary variable from `auxkernel[k0](x0)` (Line 5). Say `v` is the auxiliary value drawn. Then, the `k0`-dimensional state can be written as `(zip(t0,t),v)`.

Note that the `k0`-dimensional state `(zip(t0,t),v)` might *not* be valid. In which case, it repeatedly appends `zip(t0,t)` and `v` with entropies `(normal, coin)` until

the resulting state is valid (Lines 6-12). Say `y` and `u` are the entropy vectors drawn for the parameter and auxiliary variables respectively. Then the resulting state can be written as `(zip(t0,t)+y, v+u)`.

The transition kernel of Lines 2-12 can be expressed as

$$T_1((\boldsymbol{x}^*,\boldsymbol{v}^*),S) := \sum_{n=1}^{\infty} \int_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}} \int_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}} \int_{\mathbb{Y}(k_0)} \int_{\mathbb{T}} \left[(\mathsf{zip}(\boldsymbol{t}_0,\boldsymbol{t}) \dotplus \boldsymbol{y}, \boldsymbol{v} \dotplus \boldsymbol{u}) \in S \cap \mathbb{S}_n^{\mathrm{valid}}\right] \cdot \mathsf{pdf} K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_0,\boldsymbol{t}),\boldsymbol{v}) \\ \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t})\, \mu_{\mathbb{Y}(k_0)}(\mathrm{d}\boldsymbol{v})\, \mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\, \mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{u})$$

if $(\boldsymbol{x}^*,\boldsymbol{v}^*) \in \mathbb{S}^{\mathrm{valid}}$ and $\boldsymbol{t}_0 \in \mathsf{instance}(\boldsymbol{x}^*) \cap \mathsf{Supp}(w)$ has some dimension $k_0 \in \mathbb{N}$; and $0$ otherwise.

*Remark* 25. Recall $\mathsf{zip}(\ell_1,\ell_2) := [(\ell_1{}^1,\ell_2{}^1),(\ell_1{}^2,\ell_2{}^2),\ldots,(\ell_1{}^n,\ell_2{}^n)] \in (L_1 \times L_2)^n$ for any vectors $\ell_1 \in L_1^{n_1}$ and $\ell_2 \in L_2^{n_2}$ with $n := \min\{n_1,n_2\}$. Here we extend the definition to lists $\ell_1,\ell_2 \in (L_1 \cup L_2)^n$ such that either $(\ell_1{}^i,\ell_2{}^i)$ or $(\ell_2{}^i,\ell_1{}^i)$ is in $L_1 \times L_2$ for all $i = 1,\ldots,n$. Then, we write $\mathsf{zip}(\ell_1,\ell_2)$ for the list of pairs in $L_1 \times L_2$.

**Proposition 20.** *Assuming V2 and V3,* $T_1((\boldsymbol{x}_0,\boldsymbol{v}_0),\mathbb{S}^{\mathrm{valid}}) = 1$ *for all* $(\boldsymbol{x}_0,\boldsymbol{v}_0) \in \mathbb{S}^{\mathrm{valid}}$.

*Proof.* Since Lines 2-12 in `eNPiMCMC` can be described by a closed SPCF term that does not contain $\mathsf{score}(\cdot)$ and terminates almost surely. By Prop. 19, its transition kernel is probabilistic. Moreover, as this term always returns a valid state, we have $T_1((\boldsymbol{x}_0,\boldsymbol{v}_0),\mathbb{S}^{\mathrm{valid}}) = T_1((\boldsymbol{x}_0,\boldsymbol{v}_0),\mathbb{S}) = 1$. □

**Proposition 21.** *Assuming V1 to 3, the state distribution* $\pi$ *is invariant against Lines 2-12 in* `eNPiMCMC`.

*Proof.* We aim to show: $\int_{\mathbb{S}} T_1((\boldsymbol{x}^*,\boldsymbol{v}^*),S)\, \pi(\mathrm{d}(\boldsymbol{x}^*,\boldsymbol{v}^*)) = \pi(S)$ for any measurable set $S \in \Sigma_{\mathbb{S}}$. (Changes are highlighted for readability.)

$$\int_{\mathbb{S}} T_1((\boldsymbol{x}^*,\boldsymbol{v}^*),S)\, \pi(\mathrm{d}(\boldsymbol{x}^*,\boldsymbol{v}^*))$$

$=$ { $T_1((\boldsymbol{x}^*,\boldsymbol{v}^*),S) = 0$ for all $(\boldsymbol{x}^*,\boldsymbol{v}^*) \notin \mathbb{S}^{\mathrm{valid}}$ }

$$\int_{\mathbb{S}^{\mathrm{valid}}} T_1((\boldsymbol{x}^*,\boldsymbol{v}^*),S)\, \pi(\mathrm{d}(\boldsymbol{x}^*,\boldsymbol{v}^*))$$

$=$ { Writing $(\boldsymbol{x}^*,\boldsymbol{v}^*)$ as $(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2) \dotplus \boldsymbol{y}, \boldsymbol{v}_1 \dotplus \boldsymbol{v}_2) \in \mathbb{S}_m^{\mathrm{valid}}$ where $\boldsymbol{t}_1 \in \mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w), \boldsymbol{t}_2 \in \mathbb{T}, \boldsymbol{y} \in \mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}, \boldsymbol{v}_1 \in \mathbb{Y}^{(k_0)}, \boldsymbol{v}_2 \in \mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}$, $m,k_0 \in \mathbb{N}$ }

$$\sum_{k_0=1}^{\infty}\sum_{m=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{T}}\int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)}$$

$$T_1((\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2),S)$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2)\in\mathbb{S}_m^{\text{valid}}]\cdot\frac{1}{Z}w(\boldsymbol{t}_1)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2),\boldsymbol{v}_1)$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\;\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}_1)\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{v}_2)$$

$=$ { Definition of $T_1$ on $(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2)\in\mathbb{S}^{\text{valid}}$ where $\boldsymbol{t}_1\in\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)$ }

$$\sum_{k_0=1}^{\infty}\sum_{m=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{T}}\int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)}$$

$$\Bigg(\sum_{n=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{T}}$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}')+\!\!+\,\boldsymbol{y}',\boldsymbol{v}'+\!\!+\,\boldsymbol{u}')\in S\cap\mathbb{S}_n^{\text{valid}}]\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}'),\boldsymbol{v}')$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{u}')\Bigg)$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2)\in\mathbb{S}_m^{\text{valid}}]\cdot\frac{1}{Z}w(\boldsymbol{t}_1)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2),\boldsymbol{v}_1)$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\;\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}_1)\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{v}_2)$$

$=$ { Tonelli's theorem as all measurable functions are non-negative }

$$\sum_{k_0=1}^{\infty}\sum_{n=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{T}}\int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)}$$

$$\Bigg(\sum_{m=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{T}}$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2)\in\mathbb{S}_m^{\text{valid}}]\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2),\boldsymbol{v}_1)$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_2)\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}_1)\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y})\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{v}_2)\Bigg)$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}')+\!\!+\,\boldsymbol{y}',\boldsymbol{v}'+\!\!+\,\boldsymbol{u}')\in S\cap\mathbb{S}_n^{\text{valid}}]\cdot\frac{1}{Z}w(\boldsymbol{t}_1)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}'),\boldsymbol{v}')$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\;\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y}')\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{u}')$$

$=$ { Definition of $T_1$ on $(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2)\in\mathbb{S}^{\text{valid}}$ where $\boldsymbol{t}_1\in\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)$ }

$$\sum_{k_0=1}^{\infty}\sum_{n=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{T}}\int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)}$$

$$T_1((\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2),\mathbb{S}^{\text{valid}})$$

$$[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}')+\!\!+\,\boldsymbol{y}',\boldsymbol{v}'+\!\!+\,\boldsymbol{u}')\in S\cap\mathbb{S}_n^{\text{valid}}]\cdot\frac{1}{Z}w(\boldsymbol{t}_1)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}'),\boldsymbol{v}')$$

$$\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\;\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y}')\;\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}')\;\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{u}')$$

$=$ { By Prop. 20, $T_1((\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}_2)+\!\!+\,\boldsymbol{y},\boldsymbol{v}_1+\!\!+\,\boldsymbol{v}_2),\mathbb{S}^{\text{valid}})=1$ }

$$
\begin{aligned}
&\sum_{k_0=1}^{\infty}\sum_{n=1}^{\infty}\int_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}\int_{\mathbb{Y}^{(k_0)}}\int_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}\int_{\mathbb{T}}\int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w)}\\
&[(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}')+\boldsymbol{y}',\boldsymbol{v}'+\boldsymbol{u}')\in S\cap\mathbb{S}_n^{\text{valid}}]\cdot\frac{1}{Z}w(\boldsymbol{t}_1)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}'),\boldsymbol{v}')\\
&\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}_1)\,\mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\,\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(k_0)}}(\mathrm{d}\boldsymbol{y}')\,\mu_{\mathbb{Y}^{(k_0)}}(\mathrm{d}\boldsymbol{v}')\,\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(k_0)}}(\mathrm{d}\boldsymbol{u}')\\
=\quad&\{\ \text{Writing } (\boldsymbol{x}^*,\boldsymbol{v}^*)\in S\cap\mathbb{S}_n^{\text{valid}} \text{ as } (\mathsf{zip}(\boldsymbol{t}_1,\boldsymbol{t}')+\boldsymbol{y}',\boldsymbol{v}'+\boldsymbol{u}') \text{ where}\\
&\quad\boldsymbol{t}_1\in\mathsf{Supp}^{\iota_{\mathbb{X}}(k_0)}(w),\boldsymbol{t}'\in\mathbb{T},\boldsymbol{y}'\in\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(k_0)},\boldsymbol{v}'\in\mathbb{Y}^{(k_0)},\boldsymbol{u}'\in\mathbb{E}^{\iota_{\mathbb{Y}}(m)-\iota_{\mathbb{Y}}(k_0)},\\
&\quad n,k_0\in\mathbb{N}\ \ \}\\
&\pi(S)
\end{aligned}
$$

□

**Accept/Reject Proposed State** After constructing a valid state `(x0,v0)`, say of dimension `n`, `eNPiMCMC` traverses the state space via `involution[n]` to obtain a proposal state `(x,v)` (Line 13). By Prop. 17, `(x,v)` must also be a `n`-dimensional valid state. Say it has an instance `t` of dimension `k` in the support of `w`, then (Line 14-22) `(x,v)` is accepted with probability

$$
\begin{aligned}
\alpha(\boldsymbol{x}_0,\boldsymbol{v}_0) &:= \min\Big\{1,\frac{w(\boldsymbol{t})\cdot\mathsf{pdf}K^{(k)}(\mathsf{take}_k(\boldsymbol{x},\boldsymbol{v}))\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x})\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v})}{w(\boldsymbol{t}_0)\cdot\mathsf{pdf}K^{(k_0)}(\mathsf{take}_{k_0}(\boldsymbol{x}_0,\boldsymbol{v}_0))\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0)\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)}\cdot\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad|\det\left(\nabla\Phi^{(n)}(\boldsymbol{x}_0,\boldsymbol{v}_0)\right)|\Big\}\\
&= \min\Big\{1,\frac{\zeta(\boldsymbol{x},\boldsymbol{v})\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x})\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v})}{\zeta(\boldsymbol{x}_0,\boldsymbol{v}_0)\cdot\varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0)\cdot\varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)}\cdot|\det\left(\nabla\Phi^{(n)}(\boldsymbol{x}_0,\boldsymbol{v}_0)\right)|\Big\}.
\end{aligned}
$$

The transition kernel for Line 13-22 can be expressed as

$$T_2(\boldsymbol{s},S) := \alpha(\boldsymbol{s})\cdot[\Phi^{(n)}(\boldsymbol{s})\in S]+(1-\alpha(\boldsymbol{s}))\cdot[\boldsymbol{s}\in S]$$

if $\boldsymbol{s}\in\mathbb{S}_n^{\text{valid}}$ for some $n\in\mathbb{N}$; and 0 otherwise.

To show that the state distribution $\pi$ is invariant against $T_2$, we consider a partition of the set of valid states. Let $S_{ij}^{(n)}$ be the set of $n$-dimensional valid states where $\boldsymbol{i}$ is the list of boolean values in all states in $S_{ij}^{(n)}$ and $\Phi^{(n)}$ maps any $\boldsymbol{s}\in S_{ij}^{(n)}$ to a (valid) state with boolean values given by the list $\boldsymbol{j}$. Note that both lists $\boldsymbol{i},\boldsymbol{j}$ of booleans must be of length $\tilde{n} := \iota_{\mathbb{X}}(n)+\iota_{\mathbb{Y}}(n)$. Formally,

$$S_{ij}^{(n)} := \{\boldsymbol{s}\in\mathbb{S}_n^{\text{valid}}\mid \boldsymbol{s}=\mathsf{zip}(\boldsymbol{r},\boldsymbol{i}) \text{ and } \Phi^{(n)}(\boldsymbol{s})=\boldsymbol{s}'=\mathsf{zip}(\boldsymbol{q},\boldsymbol{j}) \text{ for some } \boldsymbol{r},\boldsymbol{q}\in\mathbb{R}^{\tilde{n}}\}.$$

Then, the set $\mathbb{S}^{\text{valid}}$ of valid states can written as $\bigcup\{S_{ij}^{(n)}\mid \boldsymbol{i},\boldsymbol{j}\in 2^{\tilde{n}} \text{ and } n\in\mathbb{N}\}$.

**Proposition 22.** *Assuming V1 to 4, for $n\in\mathbb{N}$, $\boldsymbol{s}\in\mathbb{S}_n^{valid}$ and $\boldsymbol{s}'=\Phi^{(n)}(\boldsymbol{s})$, we have*

$$\alpha(\boldsymbol{s}')\cdot\zeta'(\boldsymbol{s}')\cdot|\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| = \alpha(\boldsymbol{s})\cdot\zeta'(\boldsymbol{s})$$

*where* $\zeta'(\boldsymbol{z},\boldsymbol{w}) := \zeta(\boldsymbol{z},\boldsymbol{w}) \cdot \varphi_{\mathbb{X}^{(m)}}(\boldsymbol{z}) \cdot \varphi_{\mathbb{Y}^{(m)}}(\boldsymbol{w})$ *for any* $(\boldsymbol{z},\boldsymbol{w}) \in \mathbb{S}_m$.

*Proof.* Let $\boldsymbol{s} \in S^{(n)}_{\boldsymbol{ij}}$ where there are $\boldsymbol{r},\boldsymbol{q} \in \mathbb{R}^{\tilde{n}}$, $\boldsymbol{i},\boldsymbol{j} \in 2^{\tilde{n}}$ such that $\boldsymbol{s} = \mathsf{zip}(\boldsymbol{r},\boldsymbol{i})$ and $\boldsymbol{s}' := \Phi^{(n)}(\boldsymbol{s}) = \mathsf{zip}(\boldsymbol{q},\boldsymbol{j})$.[1] Hence, taking the Jacobian determinant on both sides of the equation $\Phi^{(n)}_{\boldsymbol{j}\to\boldsymbol{i}} \circ \Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}} = \mathsf{id}$ gives us

$$|\det\left(\nabla\Phi^{(n)}(\boldsymbol{s}')\right)| = |\det\left(\nabla\Phi^{(n)}_{\boldsymbol{j}\to\boldsymbol{i}}(\boldsymbol{q})\right)| = \frac{1}{|\det\left(\nabla\Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{r})\right)|} = \frac{1}{|\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)|}. \quad (6.2)$$

Moreover we can write the acceptance ratio in terms of $\zeta'$ as

$$\alpha(\boldsymbol{s}'') = \min\{1, \frac{\zeta'(\Phi^{(n)}(\boldsymbol{s}''))}{\zeta'(\boldsymbol{s}'')} \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s}'')\right)|\} \text{ for any } \boldsymbol{s}'' \in \mathbb{S}_m.$$

Hence given $\boldsymbol{s}' = \Phi^{(n)}(\boldsymbol{s})$, we have

$$\begin{aligned}
&\alpha(\boldsymbol{s}') \cdot \zeta'(\boldsymbol{s}') \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| \\
&= \begin{cases} \dfrac{\zeta'(\boldsymbol{s})}{\zeta'(\boldsymbol{s}')} \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s}')\right)| \cdot \zeta'(\boldsymbol{s}') \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| & \\ \qquad\qquad \text{if } \dfrac{\zeta'(\boldsymbol{s})}{\zeta'(\boldsymbol{s}')} \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s}')\right)| < 1 & (\boldsymbol{s} = \Phi^{(n)}(\boldsymbol{s}')) \\ \zeta'(\boldsymbol{s}') \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| \quad \text{otherwise} & \end{cases} \\
&= \begin{cases} \zeta'(\boldsymbol{s}) & \text{if } \dfrac{\zeta'(\boldsymbol{s}')}{\zeta'(\boldsymbol{s})} \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| > 1 \\ \dfrac{\zeta'(\boldsymbol{s}')}{\zeta'(\boldsymbol{s})} \cdot |\det\left(\nabla\Phi^{(n)}(\boldsymbol{s})\right)| \cdot \zeta'(\boldsymbol{s}) & \text{otherwise} \end{cases} \quad \text{(By Eq. (6.2))} \\
&= \alpha(\boldsymbol{s}) \cdot \zeta'(\boldsymbol{s})
\end{aligned}$$

□

**Proposition 23.** *Assuming V1 to 4, the state distribution* $\pi$ *is invariant against Line 13-22 in* `eNPiMCMC`.

*Proof.* We aim to show: $\int_{\mathbb{S}} T_2(\boldsymbol{s},S)\ \pi(\mathrm{d}\boldsymbol{s}) = \pi(S)$ for all $S \in \Sigma_{\mathbb{S}}$.

Let $\boldsymbol{s}$ be a $n$-dimensional valid state and $S \in \Sigma_{\mathbb{S}}$. Then we can write $T_2(\boldsymbol{s},S)$ as $[\boldsymbol{s} \in S] + [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s}) - [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s})$. Hence, it is enough to show that the integral of the second and third terms over all valid states are the same, i.e.

$$\int_{\mathbb{S}^{\text{valid}}} [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s})\ \pi(\mathrm{d}\boldsymbol{s}) = \int_{\mathbb{S}^{\text{valid}}} [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s})\ \pi(\mathrm{d}\boldsymbol{s})$$

First we consider the valid states in $S^{(n)}_{\boldsymbol{ij}}$ where $n \in \mathbb{N}$, $\boldsymbol{i},\boldsymbol{j} \in 2^{\tilde{n}}$ and $\tilde{n} := \iota_{\mathbb{X}}(n) + \iota_{\mathbb{Y}}(n)$. These are $n$-dimensional valid states with boolean values given by $\boldsymbol{i}$ and are mapped by

[1] We write $\mathsf{zip}(\boldsymbol{r},\boldsymbol{i})$ for $(\mathsf{zip}(\boldsymbol{r}^{1\ldots\iota_{\mathbb{X}}(n)},\boldsymbol{i}^{1\ldots\iota_{\mathbb{X}}(n)}),\mathsf{zip}(\boldsymbol{r}^{\iota_{\mathbb{X}}(n)+1\ldots\iota_{\mathbb{X}}(n)+\iota_{\mathbb{Y}}(n)},\boldsymbol{i}^{\iota_{\mathbb{X}}(n)+1\ldots\iota_{\mathbb{X}}(n)+\iota_{\mathbb{Y}}(n)}))$ when the context is clear.

$\Phi^{(n)}$ to valid states with boolean values given by $\boldsymbol{j}$. Then we have $\mathsf{zip}(\cdot,\boldsymbol{j})^{-1}(S^{(n)}_{\boldsymbol{ji}}) = \Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}\big(\mathsf{zip}(\cdot,\boldsymbol{i})^{-1}(S^{(n)}_{\boldsymbol{ij}})\big)$ where $\mathsf{zip}(\cdot,\boldsymbol{j}) : \mathbb{R}^{\tilde{n}} \to \mathbb{E}^{\tilde{n}}$ is a measurable function. Writing $\zeta'(\boldsymbol{z},\boldsymbol{w})$ for $\zeta(\boldsymbol{z},\boldsymbol{w}) \cdot \varphi_{\mathbb{X}^{(m)}}(\boldsymbol{z}) \cdot \varphi_{\mathbb{Y}^{(m)}}(\boldsymbol{w})$ for any $(\boldsymbol{z},\boldsymbol{w}) \in \mathbb{S}_m$, we have

$$
\begin{aligned}
&\int_{S^{(n)}_{\boldsymbol{ji}}} [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s}) \\
&= \int_{S^{(n)}_{\boldsymbol{ji}}} [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s}) \cdot \zeta'(\boldsymbol{s})\, \mu_{\mathbb{E}^{\tilde{n}}}(\mathrm{d}\boldsymbol{s}) && \text{(Definition of } \pi\text{)} \\
&= \int_{\mathsf{zip}(\cdot,\boldsymbol{j})^{-1}(S^{(n)}_{\boldsymbol{ji}})} [\mathsf{zip}(\boldsymbol{r},\boldsymbol{j}) \in S] \cdot \alpha(\mathsf{zip}(\boldsymbol{r},\boldsymbol{j})) \cdot \zeta'(\mathsf{zip}(\boldsymbol{r},\boldsymbol{j}))\, \mu_{\mathbb{R}^{\tilde{n}}}(\mathrm{d}\boldsymbol{r}) \\
&\qquad\qquad (\mathsf{zip}(\cdot,\boldsymbol{j})_* \mu_{\mathbb{R}^{\tilde{n}}} = \mu_{\mathbb{E}^{\tilde{n}}} \text{ on } S^{(n)}_{\boldsymbol{ji}}) \\
&= \int_{\mathsf{zip}(\cdot,\boldsymbol{i})^{-1}(S^{(n)}_{\boldsymbol{ij}})} [\mathsf{zip}(\Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q}),\boldsymbol{j}) \in S] \cdot \\
&\qquad\qquad \alpha(\mathsf{zip}(\Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q}),\boldsymbol{j})) \cdot \zeta'(\mathsf{zip}(\Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q}),\boldsymbol{j})) \cdot |\det \nabla \Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q})|\, \mu_{\mathbb{R}^{\tilde{n}}}(\mathrm{d}\boldsymbol{q}) \\
&\qquad\qquad \text{(Change of variable where } \boldsymbol{r} = \Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q})\text{)} \\
&= \int_{\mathsf{zip}(\cdot,\boldsymbol{i})^{-1}(S^{(n)}_{\boldsymbol{ij}})} [\Phi^{(n)}(\mathsf{zip}(\boldsymbol{q},\boldsymbol{i})) \in S] \cdot \alpha(\mathsf{zip}(\boldsymbol{q},\boldsymbol{i})) \cdot \zeta'(\mathsf{zip}(\boldsymbol{q},\boldsymbol{i}))\, \mu_{\mathbb{R}^{\tilde{n}}}(\mathrm{d}\boldsymbol{q}) \\
&\qquad\qquad \text{( Prop. 22 as } \Phi^{(n)}(\mathsf{zip}(\boldsymbol{q},\boldsymbol{i})) = \mathsf{zip}(\Phi^{(n)}_{\boldsymbol{i}\to\boldsymbol{j}}(\boldsymbol{q}),\boldsymbol{j}) \text{ for } (\mathsf{zip}(\boldsymbol{q},\boldsymbol{i})) \in S^{(n)}_{\boldsymbol{ij}}\text{)} \\
&= \int_{S^{(n)}_{\boldsymbol{ij}}} [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s}) \cdot \zeta'(\boldsymbol{s})\, \mu_{\mathbb{E}^{\tilde{n}}}(\mathrm{d}\boldsymbol{s}) && (\mathsf{zip}(\cdot,\boldsymbol{i})_* \mu_{\mathbb{R}^{\tilde{n}}} = \mu_{\mathbb{E}^{\tilde{n}}} \text{ on } S^{(n)}_{\boldsymbol{ij}}) \\
&= \int_{S^{(n)}_{\boldsymbol{ij}}} [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s})
\end{aligned}
$$

Recall the set $\mathbb{S}^{\text{valid}}$ of all valid states can be written as $\bigcup\{S^{(n)}_{\boldsymbol{ij}} \mid \boldsymbol{i},\boldsymbol{j} \in 2^{\tilde{n}} \text{ and } n \in \mathbb{N}\}$. Hence, we conclude our proof with

$$
\begin{aligned}
&\int_{\mathbb{S}^{\text{valid}}} [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s}) = \sum_{n=1}^{\infty} \sum_{\boldsymbol{i},\boldsymbol{j} \in 2^{\tilde{n}}} \int_{S^{(n)}_{\boldsymbol{ij}}} [\Phi^{(n)}(\boldsymbol{s}) \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s}) \\
&= \sum_{n=1}^{\infty} \sum_{\boldsymbol{i},\boldsymbol{j} \in 2^{\tilde{n}}} \int_{S^{(n)}_{\boldsymbol{ji}}} [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s}) = \int_{\mathbb{S}^{\text{valid}}} [\boldsymbol{s} \in S] \cdot \alpha(\boldsymbol{s})\, \pi(\mathrm{d}\boldsymbol{s}).
\end{aligned}
$$

□

Since the transition kernel of `eNPiMCMC` is the composition of $T_1$ and $T_2$ and both $T_1$ and $T_2$ are invariant against $\pi$ (Propositions 21 and 23), with Prop. 10 we deduce that `eNPiMCMC` preserves the state distribution $\pi$.

**Lemma 3** (State Invariant)**.** *$\pi$ is the invariant distribution of the Markov chain generated by iterating* `eNPiMCMC`.

## 6.3 Marginalised Markov Chains

As discussed above, the Markov chain $\{\boldsymbol{s}_i\}_{i\in\mathbb{N}}$ generated by iterating `eNPiMCMC` (which has invariant distribution $\pi$ (Lem. 3)) has elements on the state space $\mathbb{S}$ and *not* the trace space $\mathbb{T}$. The chain we are in fact interested in is the *marginalised* chain $\{\mathfrak{m}(\boldsymbol{s}_i)\}_{i\in\mathbb{N}}$ where the measurable function $\mathfrak{m} : \mathbb{S}^{\text{valid}} \to \mathbb{T}$ takes a valid state $\boldsymbol{s} = (\boldsymbol{x}, \boldsymbol{v})$ and returns the instance of the parameter variable $\boldsymbol{x}$ that is in the support of the target density function $w$.

In this section we show that this marginalised chain simulates the target distribution $\nu$. Let $T_{\texttt{NPiMCMC}} : \mathbb{T} \rightsquigarrow \mathbb{T}$ be a kernel such that

$$T_{\texttt{NPiMCMC}}(\boldsymbol{t}, A) := \begin{cases} T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A)) & \text{if } \boldsymbol{t} \in \mathsf{Supp}(w) \text{ and } \boldsymbol{s} \in \mathfrak{m}^{-1}(\{\boldsymbol{t}\}) \\ 0 & \text{otherwise.} \end{cases}$$

Comparing the commands of `NPiMCMC` and `eNPiMCMC` in Listings 5.2 and 6.1, we claim that $T_{\texttt{NPiMCMC}}$ is *the* transition kernel of `NPiMCMC`.

**Proposition 24.** *We consider some basic properties of* $T_{\texttt{NPiMCMC}}$.

*(i)* $T_{\texttt{NPiMCMC}}$ *is well-defined.*

*(ii)* $T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A)) = T_{\texttt{NPiMCMC}}(\mathfrak{m}(\boldsymbol{s}), A)$ *for all* $\boldsymbol{s} \in \mathbb{S}^{\text{valid}}$ *and* $A \in \Sigma_{\mathbb{T}}$.

*Proof.* (i) Let $\boldsymbol{t} \in \mathsf{Supp}(w)$ and $A \in \Sigma_{\mathbb{T}}$. Say $\boldsymbol{s}, \boldsymbol{s}' \in \mathfrak{m}^{-1}(\{\boldsymbol{t}\})$. Since only the instance of the input state matters in `eNPiMCMC` (Listing 6.1), the value of $T_{\texttt{NPiMCMC}}(\boldsymbol{t}, A)$ given by $\boldsymbol{s}$ and $\boldsymbol{s}'$ are the same, i.e.

$$T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A)) = T_{\texttt{eNPiMCMC}}(\boldsymbol{s}', \mathfrak{m}^{-1}(A)).$$

(ii) Let $\boldsymbol{s} \in \mathbb{S}^{\text{valid}}$ and $A \in \Sigma_{\mathbb{T}}$. Then, $T_{\texttt{NPiMCMC}}(\mathfrak{m}(\boldsymbol{s}), A) = T_{\texttt{eNPiMCMC}}(\boldsymbol{s}', \mathfrak{m}^{-1}(A))$ for some $\boldsymbol{s}' \in \mathfrak{m}^{-1}(\{\mathfrak{m}(\boldsymbol{s})\})$. Since $\boldsymbol{s} \in \mathfrak{m}^{-1}(\{\mathfrak{m}(\boldsymbol{s})\})$, we have

$$T_{\texttt{eNPiMCMC}}(\boldsymbol{s}', \mathfrak{m}^{-1}(A)) = T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A)).$$

□

To show $T_{\texttt{NPiMCMC}}$ preserves the target distribution, we consider a distribution $\pi_n$ on each of the $n$-dimensional state space $\mathbb{S}^{(n)} := \mathbb{X}^{(n)} \times \mathbb{Y}^{(n)}$ with density $\zeta_n$ (w.r.t. $\mu_{\mathbb{S}^{(n)}}$) given by

$$\zeta_n(\boldsymbol{x}, \boldsymbol{v}) := \begin{cases} \dfrac{1}{Z_n} \cdot w(\boldsymbol{t}) \cdot \mathsf{pdf} K^{(k)}(\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v})) \\ \qquad \text{if } \boldsymbol{t} \in \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \text{ has dimension } k \leq n \\ 0 \qquad \text{otherwise} \end{cases}$$

where $Z_n \coloneqq \int_{\mathbb{T}} [|\boldsymbol{t}| \leq \iota_{\mathbb{X}}(n)] \cdot w(\boldsymbol{t})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t})$. Notice that $Z_n \cdot \zeta_n = Z \cdot \zeta$ on valid states $\mathbb{S}_n^{\text{valid}}$. The following proposition shows how the state distribution $\pi$ can be represented using $\pi_n$.

**Proposition 25.** *Let $n \in \mathbb{N}$.*

*(i) $\pi_n$ is a probability measure.*

*(ii) For $k \leq n$, $Z_k \cdot \pi_k = Z_n \cdot \mathsf{take}_{k*}\pi_n$ on $\mathbb{S}_k^{valid}$.*

*(iii) Let $g^{(n)} : \mathbb{S}^{(n)} \rightharpoonup \bigcup_{k=1}^{n} \mathbb{S}_k^{valid}$ be the partial measurable function that returns the projection of the input state that is valid, if it exists. Formally, $g^{(n)}(\boldsymbol{s}) = \mathsf{take}_k(\boldsymbol{s})$ if $\mathsf{take}_k(\boldsymbol{s}) \in \mathbb{S}_k^{valid}$. Then $Z \cdot \pi = Z_n \cdot g_*^{(n)}\pi_n$ on $\bigcup_{k=1}^{n} \mathbb{S}_k^{valid}$.*

*Proof.* (i) Consider $\pi_n(\mathbb{S}^{(n)})$,

$$
\begin{aligned}
\pi_n(\mathbb{S}^{(n)}) &= \sum_{k=1}^{n} \int_{\mathbb{S}^{(n)}} [\boldsymbol{t} \in \mathsf{instance}(\boldsymbol{x})] \cdot [|\boldsymbol{t}| = \iota_{\mathbb{X}}(k)] \cdot \frac{1}{Z_n} \cdot w(\boldsymbol{t}) \cdot \\
&\quad \mathsf{pdf} K^{(k)}(\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}))\, \mu_{\mathbb{S}^{(n)}}(\mathrm{d}(\boldsymbol{x}, \boldsymbol{v})) \\
&= \sum_{k=1}^{n} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k)}(w)} \int_{\mathbb{T}} \int_{\mathbb{Y}^{(k)}} \frac{1}{Z_n} \cdot w(\boldsymbol{t}) \cdot \mathsf{pdf} K^{(k)}(\mathsf{zip}(\boldsymbol{t}, \boldsymbol{t}'), \boldsymbol{v}') \\
&\quad \mu_{\mathbb{Y}^{(k)}}(\mathrm{d}\boldsymbol{v}')\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}) \\
&= \sum_{k=1}^{n} \int_{\mathsf{Supp}^{\iota_{\mathbb{X}}(k)}(w)} \int_{\mathbb{T}} \frac{1}{Z_n} \cdot w(\boldsymbol{t})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}')\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}) \\
&\qquad\qquad (K^{(k)} \text{ is a probability kernel}) \\
&= \int_{\mathbb{T}} [|\boldsymbol{t}| \leq \iota_{\mathbb{X}}(n)] \cdot \frac{1}{Z_n} \cdot w(\boldsymbol{t})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}) = 1
\end{aligned}
$$

(ii) Let $S \in \Sigma_{\mathbb{S}}$ where $S \subseteq \mathbb{S}_k^{\text{valid}}$. Hence $Z_k \cdot \zeta_k(\boldsymbol{s}) = Z_k \cdot \zeta_k(\boldsymbol{s}')$ if $\boldsymbol{s} \in S$ and $\boldsymbol{s} = \mathsf{take}_k(\boldsymbol{s}')$. Then,

$$
\begin{aligned}
& Z_n \cdot (\mathsf{take}_{k*}\pi_n)(S) \\
&= Z_n \cdot \pi_n(\mathsf{take}_k{}^{-1}(S)) \\
&= \int_{\mathbb{S}^{(n)}} [\mathsf{take}_k(\boldsymbol{s}') \in S] \cdot Z_n \cdot \zeta_n(\boldsymbol{s}')\, \mu_{\mathbb{S}^{(n)}}(\mathrm{d}(\boldsymbol{s}')) \\
&= \int_{\mathbb{S}^{(k)}} [\boldsymbol{s} \in S] \cdot Z_k \cdot \zeta_k(\boldsymbol{s}) \cdot\ \mu_{\mathbb{S}^{(k)}}(\mathrm{d}(\boldsymbol{s})) \\
&= Z_k \cdot \pi_k(S)
\end{aligned}
$$

(iii) Let $S \in \Sigma_{\mathbb{S}}$ where $S \subseteq \bigcup_{k=1}^{n} \mathbb{S}_k^{\text{valid}}$. Then, $Z \cdot \zeta(\boldsymbol{s}) = Z_k \cdot \zeta_k(\boldsymbol{s})$ for all $\boldsymbol{s} \in S \cap \mathbb{S}_k^{\text{valid}}$.

$$
\begin{aligned}
Z \cdot \pi(S) &= \int_S \left[\boldsymbol{s} \in \mathbb{S}^{\text{valid}}\right] \cdot Z \cdot \zeta(\boldsymbol{s})\, \mu_{\mathbb{S}}(\mathrm{d}\boldsymbol{s}) \\
&= \sum_{k=1}^{n} \int_S \left[\boldsymbol{s} \in \mathbb{S}_k^{\text{valid}}\right] \cdot Z \cdot \zeta(\boldsymbol{s})\, \mu_{\mathbb{S}^{(k)}}(\mathrm{d}\boldsymbol{s}) \\
&= \sum_{k=1}^{n} \int_S \left[\boldsymbol{s} \in \mathbb{S}_k^{\text{valid}}\right] \cdot Z_k \cdot \zeta_k(\boldsymbol{s})\, \mu_{\mathbb{S}^{(k)}}(\mathrm{d}\boldsymbol{s}) \\
&= \sum_{k=1}^{n} Z_k \cdot \pi_k(S \cap \mathbb{S}_k^{\text{valid}}) \\
&= Z_n \sum_{k=1}^{n} \mathsf{take}_{k*}\pi_n(S \cap \mathbb{S}_k^{\text{valid}}) \qquad \text{(i)} \\
&= Z_n \cdot \pi_n(\bigcup_{k=1}^{n} \{\boldsymbol{s} \in \mathbb{S}^{(n)} \mid \mathsf{take}_k(\boldsymbol{s}) \in S \cap \mathbb{S}_k^{\text{valid}}\}) \\
&= Z_n \cdot g_*^{(n)} \pi_n(S).
\end{aligned}
$$

□

**Lemma 4** (Invariant)**.** *Assuming V1 to 4, $\nu$ is the invariant distribution of the Markov chain generated by iterating the NP-iMCMC algorithm (Sec. 5.4).*

*Proof.* Assuming (1) $\nu = \mathfrak{m}_*\pi$ on $\mathbb{T}$ and (2) $\mu_{\mathbb{T}} = \mathfrak{m}_*\mu_{\mathbb{S}}$ on $\mathsf{Supp}(w)$, we have for any $A \in \Sigma_{\mathbb{T}}$,

$$
\begin{aligned}
\nu(A) &= \mathfrak{m}_*\pi(A) && (1) \\
&= \int_{\mathbb{S}} T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A))\, \pi(\mathrm{d}\boldsymbol{s}) && \text{(Lem. 3)} \\
&= \int_{\mathbb{S}^{\text{valid}}} T_{\texttt{eNPiMCMC}}(\boldsymbol{s}, \mathfrak{m}^{-1}(A))\, \pi(\mathrm{d}\boldsymbol{s}) \\
&= \int_{\mathbb{S}^{\text{valid}}} T_{\texttt{NPiMCMC}}(\mathfrak{m}(\boldsymbol{s}), A)\, \pi(\mathrm{d}\boldsymbol{s}) && \text{(Prop. 24.ii)} \\
&= \int_{\mathsf{Supp}(w)} T_{\texttt{NPiMCMC}}(\boldsymbol{t}, A)\, \mathfrak{m}_*\pi(\mathrm{d}\boldsymbol{t}) \\
&= \int_{\mathsf{Supp}(w)} T_{\texttt{NPiMCMC}}(\boldsymbol{t}, A)\, \nu(\mathrm{d}\boldsymbol{t}) && (2) \\
&= \int_{\mathbb{T}} T_{\texttt{NPiMCMC}}(\boldsymbol{t}, A)\, \nu(\mathrm{d}\boldsymbol{t}).
\end{aligned}
$$

It is enough to show (1) and (2).

(1) Let $A \in \Sigma_{\mathbb{T}}$ where $A \subseteq \mathsf{Supp}^{\iota\times(n)}(w)$ and $\delta > 0$. Then partitioning $\mathfrak{m}^{-1}(A)$ using

$\mathbb{S}_k^{\text{valid}}$, we have for sufficiently large $m$,

$$
\begin{aligned}
&\mathfrak{m}_*\pi(A)\\
&= \pi\left(\bigcup_{k=1}^{m} \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\right) + \pi\left(\bigcup_{k=m+1}^{\infty} \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\right)\\
&< \frac{Z_m}{Z} \cdot g_*^{(m)} \pi_m\left(\bigcup_{k=1}^{m} \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\right) + \delta \qquad \text{(Prop. 24.iii, Prop. 25.ii)}\\
&\leq \frac{Z_m}{Z} \cdot \pi_m(\{(\mathsf{zip}(\boldsymbol{t}, \boldsymbol{t}') + \boldsymbol{y}, \boldsymbol{v}) \mid \boldsymbol{t} \in A, \boldsymbol{t}' \in \mathbb{T}, \boldsymbol{y} \in \mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(n)}, \boldsymbol{v} \in \mathbb{Y}^{(m)}\}) + \delta\\
&= \frac{1}{Z} \int_A \int_{\mathbb{T}} \int_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(n)}} w(\boldsymbol{t}) \cdot\\
&\qquad \Big( \int_{\mathbb{Y}^{(m)}} \mathsf{pdf} K^{(n)}(\mathsf{take}_n(\mathsf{zip}(\boldsymbol{t}, \boldsymbol{t}') + \boldsymbol{y}, \boldsymbol{v}))\, \mu_{\mathbb{Y}^{(m)}}(\mathrm{d}\boldsymbol{v}) \Big)\\
&\quad \mu_{\mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(n)}}(\mathrm{d}\boldsymbol{y}) \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}') \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}) + \delta\\
&= \frac{1}{Z} \int_A w(\boldsymbol{t})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{t}) + \delta \qquad (K^{(n)} \text{ is a probability kernel})\\
&= \nu(A) + \delta.
\end{aligned}
$$

For any measurable set $A \in \Sigma_{\mathbb{T}}$, we have $\mathfrak{m}_*\pi(A) = \mathfrak{m}_*\pi(A \cap \mathsf{Supp}(w)) = \sum_{n=1}^{\infty} \mathfrak{m}_*\pi(A \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(n)}(w)) \leq \sum_{n=1}^{\infty} \nu(A \cap \mathsf{Supp}^{\iota_{\mathbb{X}}(n)}(w)) = \nu(A \cap \mathsf{Supp}(w)) = \nu(A)$. Since both $\nu$ and $\pi$ are probability distributions, we also have $\nu(A) = 1 - \nu(\mathbb{T} \setminus A) \leq 1 - \mathfrak{m}_*\pi(\mathbb{T} \setminus A) = 1 - (1 - \mathfrak{m}_*\pi(A)) = \mathfrak{m}_*\pi(A)$. Hence $\mathfrak{m}_*\pi = \nu$ on $\mathbb{T}$.

(2) Similarly, let $A \in \Sigma_{\mathbb{T}}$ where $A \subseteq \mathsf{Supp}^{\iota_{\mathbb{X}}(n)}(w)$ and $\delta > 0$. Then by Prop. 24.iii, for sufficiently large $m$, we must have $\mu_{\mathbb{S}}(\bigcup_{k=m+1}^{\infty} \mathbb{S}_k^{\text{valid}}) = \mu_{\mathbb{S}}(\mathbb{S}^{\text{valid}} \setminus \mathbb{S}_{\leq m}^{\text{valid}}) < \delta$. Hence,

$$
\begin{aligned}
&\mathfrak{m}_*\mu_{\mathbb{S}}(A)\\
&= \mu_{\mathbb{S}}\left(\bigcup_{k=1}^{m} \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\right) + \mu_{\mathbb{S}}\left(\bigcup_{k=m+1}^{\infty} \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\right)\\
&< \sum_{k=1}^{m} \mu_{\mathbb{S}^{(k)}}(\mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}) + \delta\\
&= \sum_{k=1}^{m} \mu_{\mathbb{S}^{(m)}}(\{(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}^{(m)} \mid \mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}) \in \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\}) + \delta\\
&= \mu_{\mathbb{S}^{(m)}}(\bigcup_{k=1}^{m} \{(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}^{(m)} \mid \mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v}) \in \mathfrak{m}^{-1}(A) \cap \mathbb{S}_k^{\text{valid}}\}) + \delta\\
&\leq \mu_{\mathbb{S}^{(m)}}(\{(\mathsf{zip}(\boldsymbol{t}, \boldsymbol{t}') + \boldsymbol{y}, \boldsymbol{v}) \mid \boldsymbol{t} \in A, \boldsymbol{t}' \in \mathbb{T}, \boldsymbol{y} \in \mathbb{E}^{\iota_{\mathbb{X}}(m)-\iota_{\mathbb{X}}(n)}, \boldsymbol{v} \in \mathbb{Y}^{(m)}\}) + \delta\\
&= \mu_{\mathbb{T}}(A) + \delta.
\end{aligned}
$$

Then the proof proceeds as in (1). Note that since $w$ almost surely terminates (V2), $\mathfrak{m}_*\mu_{\mathbb{S}}(\mathsf{Supp}(w)) = \mu_{\mathbb{T}}(\mathsf{Supp}(w)) = 1$

□

*Spera in Deo, quoniam adhuc confitebor illi, salutare vultus mei, et Deus meus.*

— Psalm 42:6

# 7 Nonparametric Hamiltonian Monte Carlo

In this chapter, we study the nonparametric Hamiltonian Monte Carlo (NP-HMC) inference algorithm and its variants. We first show that NP-HMC is an instance of a variant of the NP-iMCMC sampler, where the involution is applied multiple times to generate a proposed state and then we apply some techniques to NP-HMC to form nonparametric variants of Generalised HMC and Look Ahead HMC.

## 7.1 Motivation

Recall the Hamiltonian Monte Carlo (HMC) algorithm can be seen as an instance of the `DirectioniMCMC` algorithm with the leapfrog function $\mathbf{L}$ as its bijection (Sec. 4.4.1) What happens when we extend HMC using the NP-iMCMC framework?

A direct nonparametric extension is actually very inefficient! This is because whenever the dimension of the state is changed, the NP-iMCMC algorithm (Sec. 5.4) has to "re-run" the involution in the extend step (Step 4.ii). In the example of extending HMC, this means the number of leapfrog steps performed in one iteration is unbounded. To remedy this problem, we introduce two new concepts:

- The *slice function* which can make "re-runs" (Step 4.ii) quicker.
- The *Multiple Step NP-iMCMC* sampler, a variant of NP-iMCMC, which uses a list of bijections to move around the state space.

## 7.2 Slice function

For each dimension $n \in \mathbb{N}$, we call the measurable function $s^{(n)} : \mathbb{S}^{(n)} \to \mathbb{E}^{\iota_{\mathbb{X}}(n)-\iota_{\mathbb{X}}(n-1)} \times \mathbb{E}^{\iota_{\mathbb{Y}}(n)-\iota_{\mathbb{Y}}(n-1)}$ a ***slice*** of the endofunction $\Phi^{(n)}$ on $\mathbb{S}^{(n)}$ if given a state $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{S}^{(n)}$ with an instance $\boldsymbol{t}$ of $\boldsymbol{x}$ in the support of $w$ and has a lower dimension than $\boldsymbol{x}$, $s^{(n)}(\boldsymbol{x}, \boldsymbol{v})$ is equal to the $n$-th dimension of the result of $\Phi^{(n)}(\boldsymbol{x}, \boldsymbol{v})$. Formally, this means

$$s^{(n)}(\boldsymbol{x}, \boldsymbol{v}) = (\mathsf{drop}_{n-1} \circ \Phi^{(n)})(\boldsymbol{x}, \boldsymbol{v}) \quad \text{if } \boldsymbol{t} \in \mathsf{instance}(\boldsymbol{x}) \cap \mathsf{Supp}(w) \text{ and } |\boldsymbol{t}| < \iota_{\mathbb{X}}(n).$$

Note we can always define a slice of $\Phi^{(n)}$ by setting $s^{(n)} := \mathsf{drop}_{n-1} \circ \Phi^{(n)}$.

With the slice function $s^{(n)}$ defined for each involution $\Phi^{(n)}$, Step 4.ii in the NP-iMCMC algorithm (Sec. 5.4):

(Step 4.ii) Move around the $n+1$-dimensional state space $\mathbb{X}^{(n+1)} \times \mathbb{Y}^{(n+1)}$ and compute the new state by applying the involution $\Phi^{(n+1)}$ to the initial state $(\boldsymbol{x}_0 +\!\!+\, \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+\, \boldsymbol{u}_0)$;

can be replaced by the following Step 4.ii':

(Step 4.ii') Replace and extend the $n$-dimensional new state from $(\boldsymbol{x}, \boldsymbol{v})$ to a state $(\boldsymbol{x} +\!\!+ \boldsymbol{y}, \boldsymbol{v} +\!\!+ \boldsymbol{u})$ of dimension $n+1$ where $(\boldsymbol{y}, \boldsymbol{u})$ is the result of $s^{(n+1)}(\boldsymbol{x}_0 +\!\!+\, \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+\, \boldsymbol{u}_0)$.

By V 4, the first $n$ components of the new $n+1$-dimensional state $\Phi^{(n+1)}(\boldsymbol{x}_0 +\!\!+\, \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+\, \boldsymbol{u}_0)$ is

$$\mathsf{take}_n(\Phi^{(n+1)}(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)) = \Phi^{(n)}(\mathsf{take}_n(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)) = \Phi^{(n)}(\boldsymbol{x}_0, \boldsymbol{v}_0) = (\boldsymbol{x}, \boldsymbol{v})$$

and by the definition of slice the $n+1$ component of the new state is

$$\mathsf{drop}_n(\Phi^{(n+1)}(\boldsymbol{x}_0 +\!\!+\, \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+\, \boldsymbol{u}_0)) = s^{(n+1)}(\boldsymbol{x}_0 +\!\!+\, \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+\, \boldsymbol{u}_0).$$

Hence the new states computed by Step 4.ii and Step 4.ii' are the same.

The slice function $s^{(n)}$ is useful when the involution is computationally expensive but its slice function is cheap. After replacing Step 4.ii with Step 4.ii', the NP-iMCMC sampler needs only to run the involution once (Step 3) and any subsequent "re-runs" (Step 4) can be performed by the slice function.

If the slice function $s^{(n)}$ is implemented as `slice[n]` in SPCF, Line 11 in `NPiMCMC` can be changed from `(x,v) = involution[n](x0,v0)` to

```
(x',v') = slice[n](x0,v0); (x,v) = (x + x', v + v')
```

### 7.2.1 Example (HMC)

The momentum update where the gradient of the density function is calculated is the most computationally heavy component in the HMC sampler. Hence it would be useful if it had a lightweight slice function.

In the setting of NP-iMCMC, we assume the trace space $\mathbb{T}$ is a list of the Real measurable space $\mathbb{R}$. Then, the $n$-dimensional momentum update $\phi_k^M$ is an endofunction on $\mathbb{R}^n \times \mathbb{R}^n$ defined as

$$\phi_k^M(\boldsymbol{q}, \boldsymbol{p}) \coloneqq (\boldsymbol{q}, \boldsymbol{p} - k\nabla U(\boldsymbol{q}))$$

where $U(\boldsymbol{q}) \coloneqq -\log \max\{w(\boldsymbol{t}) \mid \boldsymbol{t} \in \mathsf{instance}(\boldsymbol{q})\}$ is the $n$-dimensional potential energy.

Given a $n$-dimensional state $(\boldsymbol{q}, \boldsymbol{p})$ where $\boldsymbol{t} \in \mathsf{instance}(\boldsymbol{q}) \cap \mathsf{Supp}(w)$ has dimension lower than $n$, the gradient of the potential energy $U$ at $\boldsymbol{q}$ w.r.t. the $n$-th coordinate is zero. Hence,

$$(\mathsf{drop}_{n-1} \circ \phi_k^M)(\boldsymbol{q}, \boldsymbol{p}) = \mathsf{drop}_{n-1}(\boldsymbol{q}, \boldsymbol{p} - k\nabla U(\boldsymbol{q})) = (\boldsymbol{q}^n, \boldsymbol{p}^n),$$

and the slice of the momentum update $\phi_k^M$ is simply the projection $\mathsf{drop}_{n-1}(\boldsymbol{q}, \boldsymbol{p}) \coloneqq (\boldsymbol{q}^n, \boldsymbol{p}^n)$.

However, not *all* momentum updates in the re-runs of the leapfrog function $\mathbf{L}$ can be replaced by its slice $\mathsf{drop}_{n-1}$. This is because when the dimension increments to, say $n+1$, only the extended initial state $(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)$ has the property that it has an instance with dimension lower than $n+1$ and not the intermediate states. Hence we introduce another idea.

## 7.3 Multiple Step NP-iMCMC

Say the involution of a NP-iMCMC sampler is comprised of a list of bijective endofunctions on $\mathbb{S}^{(n)}$, namely $\Phi^{(n)} \coloneqq f_L^{(n)} \circ \cdots \circ f_2^{(n)} \circ f_1^{(n)}$. To compute the new state, we can either

- apply the involution $\Phi^{(n)}$ to the initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ in one go and check whether the result $(\boldsymbol{x}, \boldsymbol{v})$ has an instance in the support of $w$, or
- for each $\ell = 1, \ldots, L$, apply the endofunction $f_\ell^{(n)}$ to $(\boldsymbol{x}_{\ell-1}, \boldsymbol{v}_{\ell-1})$ and (immediately) check whether the intermediate state $(\boldsymbol{x}_\ell, \boldsymbol{v}_\ell)$ has an instance in the support of $w$.

The NP-iMCMC sampler presented in Sec. 5.4 takes the first option as it is conceptually simple. However, the second option is just as valid and more importantly gives us the requirements needed to replace each endofunction by its slice in any subsequent "re-runs".

### 7.3.1 The Multiple Step NP-iMCMC Algorithm

Assume the target density $w$ satisfies V 1 to 3; and for each $n \in \mathbb{N}$, there are a probability kernel $K^{(n)}$ and a list of $L$ bijective endofunctions $\{f_\ell^{(n)} : \mathbb{S}^{(n)} \to \mathbb{S}^{(n)} \mid \ell = 1, \ldots, L\}_n$ such that for each $\ell$, $\{f_\ell^{(n)}\}_n$ satisfies the projection commutation property (V 4) and for each $n \in \mathbb{N}$, their composition $f_L^{(n)} \circ \cdots \circ f_1^{(n)}$ is involutive.

Let $s_\ell^{(n)}$ be a slice of the endofunction $f_\ell^{(n)}$. Given a SPCF program $M$ with weight function $w$ on the trace space, the ***Multiple Step NP-iMCMC*** sampler generates a Markov chain as follows. Given a current sample $\boldsymbol{t}_0$ of dimension $k_0$,

1. (Initialisation Step) Form a $k_0$-dimensional parameter variable $\boldsymbol{x}_0 \in \mathbb{X}^{(k_0)}$ by pairing each value ${\boldsymbol{t}_0}^i$ in $\boldsymbol{t}_0$ with a randomly drawn value $t$ of the other type to make a pair $({\boldsymbol{t}_0}^i, t)$ or $(t, {\boldsymbol{t}_0}^i)$ in the entropy space $\mathbb{E}$.

2. (Stochastic Step) Introduce randomness to the sampler by drawing a $k_0$-dimensional value $\boldsymbol{v}_0 \in \mathbb{Y}^{(k_0)}$ from the probability measure $K^{(k_0)}(\boldsymbol{x}_0, \cdot)$.

3. (Multiple Step) Initialise $\ell = 1$. If $\ell = L$, proceed to Step 4 with $\boldsymbol{t}$ as the proposed sample; otherwise

   3.1. (Deterministic Step) Compute the $\ell$-th state $(\boldsymbol{x}_\ell, \boldsymbol{v}_\ell)$ by applying the endofunction $f_\ell^{(n)}$ to $(\boldsymbol{x}_{\ell-1}, \boldsymbol{v}_{\ell-1})$ where $n = \dim(\boldsymbol{x}_{\ell-1})$.

   3.2. (Extend Step) Test whether any instance $\boldsymbol{t}$ of $\boldsymbol{x}_\ell$ is in the support of $w$. If so, go to Step 3 with an incremented $\ell$; otherwise (none of the instances of $\boldsymbol{x}_\ell$ is in the support of $w$),

   3.2.i. Extend and replace the $n$-dimensional initial state from $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ to a state $(\boldsymbol{x}_0 +\!\!+ \boldsymbol{y}_0, \boldsymbol{v}_0 +\!\!+ \boldsymbol{u}_0)$ of dimension $n+1$ where $\boldsymbol{y}_0$ and $\boldsymbol{u}_0$ are values drawn randomly from $\mu_{\mathbb{E}^{\iota_{\mathbb{X}}(n+1)-\iota_{\mathbb{X}}(n)}}$ and $\mu_{\mathbb{E}^{\iota_{\mathbb{Y}}(n+1)-\iota_{\mathbb{Y}}(n)}}$ respectively.

   3.2.ii. For each $i = 1, \ldots, \ell$, extend and replace the $n$-dimensional $i$-th intermediate state from $(\boldsymbol{x}_i, \boldsymbol{v}_i)$ to a state $(\boldsymbol{x}_i +\!\!+ \boldsymbol{y}_i, \boldsymbol{v}_i +\!\!+ \boldsymbol{u}_i)$ of dimension $n+1$ where $(\boldsymbol{y}_i, \boldsymbol{u}_i)$ is the result of $s_i^{(n+1)}(\boldsymbol{x}_{i-1}, \boldsymbol{v}_{i-1})$.

   3.2.iii. Go to Step 3.2 with the extended $n+1$-dimensional states $(\boldsymbol{x}_i, \boldsymbol{v}_i)$ for $i = 0, \ldots, \ell$.

4. (Accept/reject Step) Accept the proposed sample $\boldsymbol{t}$ as the next sample with probability

$$\min\left\{1;\ \frac{w(\boldsymbol{t}) \cdot \mathsf{pdf}K^{(k)}(\mathsf{take}_k(\boldsymbol{x}_L, \boldsymbol{v}_L)) \cdot \varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_L) \cdot \varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_L)}{w(\boldsymbol{t}_0) \cdot \mathsf{pdf}K^{(k_0)}(\mathsf{take}_{k_0}(\boldsymbol{x}_0, \boldsymbol{v}_0)) \cdot \varphi_{\mathbb{X}^{(n)}}(\boldsymbol{x}_0) \cdot \varphi_{\mathbb{Y}^{(n)}}(\boldsymbol{v}_0)} \cdot \prod_{\ell=1}^{L} \left|\det(\nabla f_\ell^{(n)}(\boldsymbol{x}_{\ell-1}, \boldsymbol{v}_{\ell-1}))\right|\right\}$$

where $n = \dim(\boldsymbol{x}_0) = \dim(\boldsymbol{v}_0)$, $k$ is the dimension of $\boldsymbol{t}$ and $k_0$ is the dimension of $\boldsymbol{t}_0$; otherwise reject the proposal and repeat $\boldsymbol{t}_0$.

Unlike in NP-iMCMC, the Multiple Step NP-iMCMC sampler computes the intermediate states $\{(\boldsymbol{x}_\ell, \boldsymbol{v}_\ell)\}_{\ell=1,\dots,L}$ one-by-one, making sure in Step 3.2 that each of these state $(\boldsymbol{x}_\ell, \boldsymbol{v}_\ell)$ has an instance in the support of $w$. Hence when the dimension is incremented from $n$ to $n+1$, the slice functions can be used to extend intermediate states to states of dimension $n+1$.

*Remark* 26. The Multiple Step NP-iMCMC sampler can be seen as a generalisation of NP-iMCMC as we can recover NP-iMCMC by setting $L$ to one and take the involution $\Phi^{(n)}$ as the only endofunction in Multiple Step NP-iMCMC.

### 7.3.2 Pseudocode of Multiple Step NP-iMCMC Algorithm

Listing 7.1 gives a SPCF implementation of Multiple Step NP-iMCMC as the function `MultistepNPiMCMC` with target density `w`; auxiliary kernel `auxkernel[n]` and its density `pdfauxkernel[n]` and `L` number of endofunctions `f[n][l]` (`l` ranges from `1` to `L`) for each dimension `n` with slice `slice[n][l]` and the absolute value of its Jacobian determinant `absdetjacf[n][l]`; parameter and auxiliary index maps `indexX` and `indexY` and projection `proj`.

### 7.3.3 Correctness of Multiple Step NP-iMCMC Algorithm

The Multiple Step NP-iMCMC sampler cannot be formulated as an instance of NP-iMCMC and requires a separate proof. Nonetheless, the arguments are similar.

**Almost Sure Termination** Since the target density $w$ satisfies V 3, Prop. 16 tells us that as long as $w$ almost surely terminates (V 2), the measure of a $n$-dimensional parameter variable not having any instances in the support of $w$ tends to zero as the dimension $n$ tends to infinity. As $f_\ell^{(n)}$ is bijective (and hence invertible), the Multiple Step NP-iMCMC sampler almost surely satisfies the condition in the loop in Step 3.2 and hence almost surely terminates.

**Invariant State Distribution** Next, we identify the state distribution of Multiple Step NP-iMCMC. We say a $n$-dimensional state $(\boldsymbol{x}, \boldsymbol{v})$ is ***valid*** if

(i) For all $\ell = 1, \dots, L$, $\mathsf{instance}(\boldsymbol{x}_\ell) \cap \mathsf{Supp}(w) \neq \varnothing$ where $(\boldsymbol{x}_0, \boldsymbol{v}_0) := (\boldsymbol{x}, \boldsymbol{v})$ and $(\boldsymbol{x}_\ell, \boldsymbol{v}_\ell) := f_\ell^{(n)}(\boldsymbol{x}_{\ell-1}, \boldsymbol{v}_{\ell-1})$; and

**Listing 7.1:** *Pseudocode of the Multiple Step NP-iMCMC algorithm*

```
def MultistepNPiMCMC(t0):
  k0 = dim(t0)                                          # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0](x0)                                # stochastic step

  # start of multiple step
  n = k0
  (x[0],v[0]) = (x0,v0)
  for l in range(1,L+1):
    (x[l],v[l]) = f[n][l](x[l-1],v[l-1])                # deterministic step
    while not intersect(instance(x[l]),support(w)):  # extend step
      x[0] = x[0] + [(normal, coin)]*(indexX(n+1)-indexX(n))
      v[0] = v[0] + [(normal, coin)]*(indexY(n+1)-indexY(n))
      for i in range(1,l+1):
        (y,u) = slice[n+1][i](x[i-1],v[i-1])
        (x[i],v[i]) = (x[i]+y, v[i]+u)
      n = n + 1
  (x0,v0) = (x[0],v[0])
  (x,v) = (x[L],v[L])
  # end of multiple step

  t = intersect(instance(x),support(w))[0]              # accept/reject
      step
  k = dim(t)
  return t if uniform < min{1,w(t)/w(t0) *
                        pdfauxkernel[k](proj((x,v),k))/
                          pdfauxkernel[k0](proj((x0,v0),k0)) *
                        pdfpar[n](x)/pdfpar[n](x0) *
                        pdfaux[n](v)/pdfaux[n](v0) *
                        product([absdetjacf[n][l](x[l-1],v[l-1]) for l in
                            range(1,L+1)])}
         else t0
```

(ii) For all $\ell = 1, \ldots, L$, $\mathsf{instance}(\boldsymbol{y}_\ell) \cap \mathsf{Supp}(w) \neq \varnothing$ where $(\boldsymbol{y}_0, \boldsymbol{u}_0) := (\boldsymbol{x}, \boldsymbol{v})$ and $(\boldsymbol{y}_{L-\ell+1}, \boldsymbol{u}_{L-\ell+1}) := f_\ell^{(n)^{-1}}(\boldsymbol{y}_{L-\ell}, \boldsymbol{u}_{L-\ell})$; and

(iii) For all $k < n$, $\mathsf{take}_k(\boldsymbol{x}, \boldsymbol{v})$ is not a valid state, i.e. not satisfying (i) and (ii).

Then, we can define the state distribution and show that the state movement in Multiple Step NP-iMCMC is invariant against this distribution.

**Marginalised Markov Chain** Finally, we conclude that the marginalised chain of the state movement in the Multiple Step NP-iMCMC sampler is invariant against the target distribution.

**Listing 7.2:** *Pseudocode of the State-dependent Multiple Step NP-iMCMC Mixture algorithm*

```
def MixtureMSNPiMCMC(t0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  m = mixkernel(x0)                               # mixture step
  v0 = auxkernel[k0][m](x0)                       # stochastic step

  # start of multiple step
  n = k0
  (x[0],v[0]) = (x0,v0)
  for l in range(1,L+1):
    (x[l],v[l]) = f[n][l][m](x[l-1],v[l-1])       # deterministic step
    while not intersect(instance(x[l]),support(w)):     # extend step
      x[0] = x[0] + [(normal, coin)]*(indexX(n+1)-indexX(n))
      v[0] = v[0] + [(normal, coin)]*(indexY(n+1)-indexY(n))
      for i in range(1,l+1):
        (y,u) = slice[n+1][i][m](x[i-1],v[i-1])
        (x[i],v[i]) = (x[i]+y, v[i]+u)
      n = n + 1
  (x0,v0) = (x[0],v[0])
  (x,v) = (x[L],v[L])
  # end of multiple step

  t = intersect(instance(x),support(w))[0]      # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k][m](proj((x,v),k))/
                                 pdfauxkernel[k0][m](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               pdfmixkernel(proj(x,k),m)/
                                 pdfmixkernel(proj(x0,k0),m) *
                               product([absdetjacf[n][m](x[l-1],v[l-1]) for
                                    l in range(1,L+1)])}
         else t0
```

### 7.3.4 Techniques on Multiple Step NP-iMCMC

Recall we discussed three techniques in Sec. 5.5, each when applied to the NP-iMCMC sampler, improving its flexibility and/or efficiency. We now see how these techniques can be applied to Multiple Step NP-iMCMC.

#### State-dependent Multiple Step NP-iMCMC Mixture

This technique allows us to 'mix' Multiple Step NP-iMCMC samplers in such a way that the resulting sampler still preserves the posterior. Given a collection of Multiple Step NP-iMCMC samplers, indexed by $m \in \mathbb{E}^\ell$ for some $\ell \in \mathbb{N}$, the *State-dependent*

**Listing 7.3:** *Pseudocode for the correctness of State-dependent Multiple Step NP-iMCMC Mixture*

```
def mixauxkernel[n](x0):
  m = mixkernel(x0)
  v0 = auxkernel[n][m](x0)
  return m + v0

def pdfmixauxkernel[n](x,mixv):
  m = mixv[:l] # l is the dimension of the variable m
  v = mixv[l:]
  return pdfmixkernel(x, m) * pdfauxkernel[n][m](x, v)

def mixf[n][l](x0,mixv0):
  m = mixv0[:l]
  v0 = mixv0[l:]
  (x,v) = f[n][l][m](x0,v0)
  return (x,m + v)

def absdetjacmixf[n][l](x0,mixv0):
  m = mixv0[:l]
  v0 = mixv0[l:]
  return absdetjacf[n][l][m](x0,v0)

def mixslice[n][l](x0,mixv0):
  m = mixv0[:l]
  v0 = mixv0[l:]
  (y,u) = slice[n][l][m](x0,v0)
  return (y,u)

mixindexX = indexX
mixindexY(n) = l + indexY(n)
mixproj((x,v),k) = (x[:mixindexX(k)],v[:mixindexY(k)])
```

*Multiple Step NP-iMCMC Mixture* sampler draws an indicator $m \in \mathbb{E}^\ell$ from a probability measure $K_M(\boldsymbol{x}_0, \cdot)$ on $\mathbb{E}^\ell$ where $K_M : \bigcup_{n\in\mathbb{N}} \mathbb{X}^{(n)} \rightsquigarrow \mathbb{E}^\ell$ is a probability kernel and $\boldsymbol{x}_0$ is the parameter variable constructed from the current sample $\boldsymbol{t}_0$ in Step 1. A proposal $\boldsymbol{t}$ is then generated by running Steps 2 and 3 of the $m$-indexed Multiple Step NP-iMCMC sampler, and is accepted with a modified probability that includes the probability of picking $m$.

**Pseudocode** Listing 7.2 gives the SPCF implementation of this sampler as the `MixtureMSNPiMCMC` function. (Terms specific to this technique are highlighted.) We assume the SPCF term `mixkernel` implements the mixture kernel $K_M$; `pdfmixkernel` implements the probability density function $\mathrm{pdf}K_M$; and for each $m \in \mathbb{E}^\ell$ and $n \in \mathbb{N}$, `auxkernel[n][m]` implements the auxiliary kernel and `pdfauxkernel[n][m]` implements its density; `f[n][l][m]` implements the endofunction, `slice[n][l][m]` imple-

ments its slice and `absdetjacf[n][l][m]` implements the absolute value of the Jacobian determinant of the endofunction of the `m`-indexed Multiple Step NP-iMCMC sampler.

**Correctness** `MixtureMSNPiMCMC` can be formed as an instance of `MultistepNPiMCMC` with auxiliary kernel `mixauxkernel[n]` and its density `mixpdfauxkernel[n]` and `L` number of endofunctions `mixf[n][l]` (`l` ranges from `1` to `L`) for each dimension `n` with slice `mixslice[n][l]` and the absolute value of its Jacobian determinant `absdetjacmixf[n][l]`; parameter and auxiliary index maps `mixindexX` and `mixindexY` and projection `mixproj` given in Listing 7.3.

### Direction Multiple Step NP-iMCMC

This technique allows us to relax the assumption that the composition $f_L^{(n)} \circ \cdots \circ f_2^{(n)} \circ f_1^{(n)}$ is involutive. Assuming for $\ell = 1, \ldots, L$, both sets $\{f_\ell^{(n)}\}_n$ and $\{f_\ell^{(n)^{-1}}\}_n$ satisfy the projection commutation property (V4), the ***Direction Multiple Step NP-iMCMC*** sampler randomly employs either $f_L^{(n)} \circ \cdots \circ f_2^{(n)} \circ f_1^{(n)}$ or $f_1^{(n)^{-1}} \circ f_2^{(n)^{-1}} \circ \cdots \circ f_L^{(n)^{-1}}$ to move around the $n$-dimensional state space and proposes a new sample.

**Pseudocode** Listing 7.4 gives the SPCF implementation of this sampler as `DirectionMSNPiMCMC` function. (Terms specific to this technique are highlighted.) We assume for each $n \in \mathbb{N}$ and $d \in 2$, the SPCF term `f[n][l][True]` implements the endofunction $f_\ell^{(n)}$ and `f[n][l][False]` implements the inverse $f_{L-\ell+1}^{(n)^{-1}}$; `slice[n][l][True]` implements the slice of $f_\ell^{(n)}$ and `slice[n][l][False]` implements the slice of $f_{L-\ell+1}^{(n)^{-1}}$; and `absdetjacf[n][l][True]` implements the absolute value of the Jacobian determinant of $f_\ell^{(n)}$ and `absdetjacf[n][l][False]` implements that of $f_{L-\ell+1}^{(n)}$.

**Correctness** `DirectionMSNPiMCMC` can be formed as an instance of `MultistepNPiMCMC` with auxiliary kernel `dirauxkernel[n]` and its density `pdfdirauxkernel[n]` and `dirL` number of endofunctions `dirf[n][l]` (`l` ranges from `1` to `dirL`) for each dimension `n` with slice `dirslice[n][l]` and the absolute value of its Jacobian determinant `absdetjacf[n][l]`; parameter and auxiliary index maps `dirindexX` and `dirindexY` and projection `dirproj` given in Listing 7.5. Note the `dirf[n]` function denotes the composition that flips the direction after applying the endofunctions $f_\ell^{(n)}$ for $\ell = 1, \ldots, L$ with an inverse that flips the direction and then applies the endofunctions $f_{L-\ell+1}^{(n)}$ for $\ell = 1, \ldots, L$.

**Listing 7.4:** *Pseudocode of the Direction Multiple Step NP-iMCMC algorithm*

```
def DirectionMSNPiMCMC(t0):
  d0 = coin                                          # direction step
  k0 = dim(t0)                                       # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0](x0)                             # stochastic step

  # start of multiple step
  n = k0
  (x[0],v[0]) = (x0,v0)
  for l in range(1,L+1):
    (x[l],v[l]) = f[n][l][d0](x[l-1],v[l-1])              # deterministic
        step
    while not intersect(instance(x[l]),support(w)):     # extend step
      x[0] = x[0] + [(normal, coin)]*(indexX(n+1)-indexX(n))
      v[0] = v[0] + [(normal, coin)]*(indexY(n+1)-indexY(n))
      for i in range(1,l+1):
        (y,u) = slice[n+1][i][d0](x[i-1],v[i-1])
        (x[i],v[i]) = (x[i]+y, v[i]+u)
      n = n + 1
  (x0,v0) = (x[0],v[0])
  (x,v) = (x[L],v[L])
  d = not d0                                         # flip direction (not used)
  # end of multiple step

  t = intersect(instance(x),support(w))[0]      # accept/reject step
  k = dim(t)
  return t if uniform < min{1, w(t)/w(t0) *
                               pdfauxkernel[k](proj((x,v),k))/
                                 pdfauxkernel[k0](proj((x0,v0),k0)) *
                               pdfpar[n](x)/pdfpar[n](x0) *
                               pdfaux[n](v)/pdfaux[n](v0) *
                               product([absdetjacf[n][l][d0](x[l-1],v[l-1])
                                    for l in range(1,L+1)])}
          else t0
```

### Persistent Multiple Step NP-iMCMC Algorithm

This technique gives us a method to construct irreversible Multiple Step NP-iMCMC samplers. The key is to persist the direction from a previous iteration.

The ***Persistent Multiple Step NP-iMCMC*** sampler keeps track of a direction variable $d_0 \in 2$ (instead of sampling a fresh one at the start) and uses it to determine the auxiliary kernel ($K_{\mathsf{T}}^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$ or $K_{\mathsf{F}}^{(n)} : \mathbb{X}^{(n)} \rightsquigarrow \mathbb{Y}^{(n)}$) and list of endofunctions ($f_L^{(n)} \circ \cdots \circ f_1^{(n)}$ or ${f_1^{(n)}}^{-1} \circ \cdots \circ {f_L^{(n)}}^{-1}$) employed. This direction variable is flipped strategically to make the resulting algorithm irreversible.

**Listing 7.5:** *Pseudocode for the correctness of Direction Multiple Step NP-iMCMC*

```
dirL = L+1

def dirauxkernel[n](x0):
  d0 = coin
  v0 = auxkernel[n](x0)
  return [(normal, d0)] + v0

def pdfdirauxkernel[n](x0,dirv0):
  d0 = dirv0[0][1]
  v0 = dirv0[1:]
  return pdfcoin(d0) * pdfnormal(dirv0[0][0]) * pdfauxkernel[n](x0,v0)

def dirf[n][l](x,dirv):
  d = dirv[0][1]
  v = dirv[1:]
  if l == dirL:
    return (x,[(dirv[0][0],not d)] + v)
  else:
    (x,v) = f[n][l][d](x,v)
    return (x, [(dirv[0][0],d)] + v)

def absdetjacdirf[n][l](x,dirv):
  d = dirv[0][1]
  v = dirv[1:]
  if l == dirL: return 1
  else: return absdetjacf[n][l][d](x, v)

def dirslice[n][l](x,dirv):
  d = dirv[0][1]
  v = dirv[1:]
  (y,u) = slice[n][l][d](x,v)
  return (y,u)

dirindexX = indexX
dirindexY(n) = 1+indexY(n)
dirproj((x,v),k) = (x[:dirindexX(k)], v[:dirindexY(k)])
```

**Pseudocode** Listing 7.6 gives the SPCF implementation of this sampler as the function `PersistentMSNPiMCMC`. (Terms specific to this technique are highlighted.) In addition to the SPCF terms in `DirectionMSNPiMCMC`, the SPCF term `auxkernel[n][True]` implements the auxiliary kernel $K_{\mathsf{T}}^{(n)}$ and `pdfauxkernel[n][True]` implements its density $\mathrm{pdf}K^{(n)}{}_{\mathsf{T}}$ and `auxkernel[n][False]` implements the auxiliary kernel $K_{\mathsf{F}}^{(n)}$ and `pdfauxkernel[n][False]` implements its density $\mathrm{pdf}K^{(n)}{}_{\mathsf{F}}$. Note that `PersistentMSNPiMCMC` updates samples on the space $\mathbb{X}^{(n)} \times 2$, which can easily be marginalised to $\mathbb{X}^{(n)}$ by taking the first $\iota_{\mathbb{X}}(n)$ components.

**Listing 7.6:** *Pseudocode of the Persistent Multiple Step NP-iMCMC algorithm*

```
def PersistentMSNPiMCMC(t0,d0):
  k0 = dim(t0)                                    # initialisation step
  x0 = [(e, coin) if Type(e) in R else (normal, e) for e in t0]
  v0 = auxkernel[k0][d0](x0)                       # stochastic step

  # start of multiple step
  n = k0
  (x[0],v[0]) = (x0,v0)
  for l in range(1,L+1):
    (x[l],v[l]) = f[n][l][d0](x[l-1],v[l-1])              # deterministic
        step
    while not intersect(instance(x[l]),support(w)):     # extend step
      x[0] = x[0] + [(normal, coin)]*(indexX(n+1)-indexX(n))
      v[0] = v[0] + [(normal, coin)]*(indexY(n+1)-indexY(n))
      for i in range(1,l+1):
        (y,u) = slice[n+1][i][d0](x[i-1],v[i-1])
        (x[i],v[i]) = (x[i]+y, v[i]+u)
      n = n + 1
  (x0,v0) = (x[0],v[0])
  (x,v) = (x[L],v[L])
  d = not d0                                      # flip direction
  # end of multiple step

  t = intersect(instance(x),support(w))[0]      # accept/reject step
  k = dim(t)
  return (t, not d) if uniform < min{1, w(t)/w(t0) *
                              pdfauxkernel[k][d](proj((x,v),k))/
                                pdfauxkernel[k0][d0](proj((x0,v0),k0)) *
                              pdfpar[n](x)/pdfpar[n](x0) *
                              pdfaux[n](v)/pdfaux[n](v0) *
                              product([absdetjacf[n][l][d0](x[l-1],v[l-1])
                                   for l in range(1,L+1)])}
          else (t0, d)
```

**Correctness** Consider the `MultistepNPiMCMC` function with auxiliary kernel `perauxkernel[n]` and its density `pdfperauxkernel[n]` and `perL` number of endo-functions `perf[n][l]` (`l` ranges from `1` to `perL`) for each dimension `n` with slice `perslice[n][l]` and the absolute value of its Jacobian determinant `absdetjacperf[n][l]`; parameter and auxiliary index maps `perindexX` and `perindexY` and projection `perproj` given in Listing 7.7.

The `MultistepNPiMCMC` function with the primitives indicated in Listing 7.7 is *almost* equivalent to `PersistentMSNPiMCMC`, except `MultistepNPiMCMC` induces a transition kernel on $\mathbb{E} \times \mathbb{X}^{(n)}$ whereas `PersistentMSNPiMCMC` induces a transition kernel on $\mathbb{2} \times \mathbb{X}^{(n)}$; and when the proposal `t` is accepted, `MultistepNPiMCMC` returns `d` whereas `PersistentMSNPiMCMC` returns `not d`.

**Listing 7.7:** *Pseudocode for the correctness of Persistent Multiple Step NP-iMCMC*

```
perL = L+1

def perauxkernel[n](perx0)
  d0 = perx0[0][1]
  x0 = perx0[1:]
  v0 = auxkernel[n][d0](x0)
  return v0

def pdfperauxkernel[n](perx0, v0):
  d0 = perx0[0][1]
  x0 = perx0[1:]
  return pdfauxkernel[n][d0](x0, v0)

def perf[n][l](perx,v)
  d = perx[0][1]
  x = perx[1:]
  if l == perL:
    return ([(perx[0][0],not d)] + x, v)
  else:
    (x,v) = f[n][l][d](x,v)
    return ([(perx[0][0],d)] + x, v)

def absdetjacperf[n][l](perx,v)
  d = perx[0][1]
  x = perx[1:]
  if l == perL: return 1
  else: return absdetjacf[n][l][d](x, v)

def perslice[n][l](perx,v)
  d = perx[0][1]
  x = perx[1:]
  (y,u) = slice[n][l][d](x,v)
  return (y,u)

perindexX(n) = 1+indexX(n)
perindexY = indexY
perproj((x,v),k) = (x[:perindexX(k)],v[:perindexY(k)])

def flipdir(perx0,v0):
  perx0[0][1] = not perx0[0][1]
  return (perx0,v0)
```

By composing `MultistepNPiMCMC` with `flipdir` which flips the direction and marginalising the Markov chain generated by the composition from $\mathbb{E} \times \mathbb{X}^{(n)}$ to $\mathbb{2} \times \mathbb{X}^{(n)}$, we get `PersistentMSNPiMCMC`.

## 7.4 Nonparametric Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo (also known as Hybrid Monte Carlo) (HMC) algorithm is a popular MCMC inference that makes use of Hamiltonian dynamics to simulate a target distribution on the measure space $(\mathbb{R}^n, \mathcal{B}_n, \mathcal{N}_n)$.

In this section, we explore how *Nonparametric Hamiltonian Monte Carlo (NP-HMC)* as presented in (Mak et al., 2021b) can be seen as an instance of Direction Multiple Step NP-iMCMC. Furthermore, we introduce *new* variants of NP-HMC using the techniques introduced in Sec. 7.3.4, namely NP-Discontinuous HMC (an extension of Discontinuous HMC (Nishimura et al., 2020)), Generalised NP-DHMC (an extension of Generalised HMC (Horowitz, 1991)) and Look Ahead NP-HMC (an extension of the Look Ahead HMC (Sohl-Dickstein et al., 2014), equivalent to the Extra Chance Generalised HMC (Campos and Sanz-Serna, 2015)).

### 7.4.1 Nonparametric HMC

Nonparametric Hamiltonian Monte Carlo (NP-HMC) is a MCMC sampler introduced by (Mak et al., 2021b) for probabilistic programming. Here we show that it is an instance of the Direction Multiple Step NP-iMCMC sampler (Sec. 7.3.4).

Typically, the HMC sampler proposes a new state by simulating $L$ leapfrog steps:

$$\mathbf{L} := (\phi^M_{\epsilon/2} \circ \phi^P_\epsilon \circ \phi^M_{\epsilon/2})^L$$

where $\phi^M_\epsilon(\boldsymbol{x}, \boldsymbol{v}) := (\boldsymbol{x}, \boldsymbol{v} - \epsilon \nabla U(\boldsymbol{x}))$ and $\phi^P_\epsilon(\boldsymbol{x}, \boldsymbol{v}) := (\boldsymbol{x} + \epsilon \boldsymbol{v}, \boldsymbol{v})$ are the momentum and position updates with step size $\epsilon$ respectively. Notice that the momentum and position updates satisfy projection commutation property (V 4), have inverses $(\phi^M_\epsilon)^{-1} = M \circ \phi^M_\epsilon \circ M$ and $(\phi^P_\epsilon)^{-1} = M \circ \phi^P_\epsilon \circ M$ where $M(\boldsymbol{x}, \boldsymbol{v}) := (\boldsymbol{x}, -\boldsymbol{v})$ and slices $\mathsf{drop}_{n-1}$ (for $\phi^M_{\epsilon/2}$, see Sec. 7.2.1 for more details) and $(\boldsymbol{x}, \boldsymbol{v}) \mapsto (\boldsymbol{x}^n + \epsilon \boldsymbol{v}^n, \boldsymbol{v}^n)$ (for $\phi^P_\epsilon$) respectively. Moreover, the absolute values of the Jacobian determinants of both updates are $|\det \nabla \phi^M_\epsilon(\boldsymbol{x}, \boldsymbol{v})| = |\det \nabla \phi^P_{\epsilon/2}(\boldsymbol{x}, \boldsymbol{v})| = 1$.

Hence, the HMC sampler can be extended using the Direction Multiple Step NP-iMCMC sampler as follows. Given an input sample $\boldsymbol{t}_0 \in \mathbb{R}^{k_0}$, a $k_0$-dimensional initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ is formed where $\boldsymbol{x}_0 := \boldsymbol{t}_0$ and $\boldsymbol{v}_0$ drawn from $K^{(n)}(\boldsymbol{x}, \cdot) := \mathcal{N}_n$. A direction variable $d_0$ is drawn to determine whether $\mathbf{L}$ or its inverse $\mathbf{L}^{-1}$ is performed on the

**Listing 7.8:** *Pseudocode for the NP-HMC algorithm*

```
def NPHMC(t0):
  d0 = coin                                        # direction step
  k0 = dim(t0)                                     # initialisation step
  x0 = t0
  v0 = [normal]*k0                                 # stochastic step
  # start of multiple step
  n = k0
  (x,v) = (x0,v0)
  for m in range(1,L+1):
    (x,v) = (x, v-ep/2*grad(U)(x))          # half momentum update
    (x,v) = (x+ep*v, v)                     # full position update
    while not intersect(instance(x),support(w)): # extend step
      x0 = x0 + [normal]
      v0 = v0 + [normal]
      (y,u) = (x0[n+1] + m*ep*v0[n+1], v0[n+1])    # slice of leapfrog step
      (x,v) = (x+y, v+u)
      n = n + 1
    (x, v) = (x, v-ep/2*grad(U)(x))          # half momentum update
  d = d0
  # end of multiple step
  t = intersect(instance(x),support(w))[0]       # accept/reject step
  k = dim(t)
  return t if uniform < min{1,w(t)/w(t0)*pdfnormal[n](x)/pdfnormal[n](x0)*
                                         pdfnormal[n](v)/pdfnormal[n](v0)}
        else t0
```

initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$, one update at a time, extending the dimension as required. Say the initial state is extended to a $n$-dimensional state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ and is traversed to the $n$-dimensional new state $(\boldsymbol{x}^*, \boldsymbol{v}^*)$ which has an instance $\boldsymbol{t}$ in the support of $w$. $\boldsymbol{t}$ is returned with probability

$$\min\left\{1; \frac{w(\boldsymbol{t}) \cdot \varphi_n(\boldsymbol{x}^*) \cdot \varphi_n(\boldsymbol{v}^*)}{w(\boldsymbol{t}_0) \cdot \varphi_n(\boldsymbol{x}_0) \cdot \varphi_n(\boldsymbol{v}_0)}\right\}.$$

**Pseudocode of NP-HMC** Listing 7.8 gives the SPCF implementation `NPHMC` of the NP-HMC sampler as an instance of the Direction Multiple Step NP-iMCMC sampler. (Terms that differ from HMC are highlighted.) Note that we do not need to perform the extend step if the previous deterministic step does not change the sample-component $\boldsymbol{x}$ of the state. This is the case for the half momentum step in HMC, hence we only do the extend step after the full position step in Listing 7.8. Moreover, we can compose the slice functions to become $(\boldsymbol{x}, \boldsymbol{v}) \mapsto (\boldsymbol{x}^n + t \cdot \boldsymbol{v}^n, \boldsymbol{v}^n)$ for time $t$.

**Correctness** Since both $\phi^M_{\epsilon/2}$ and $\phi^P_\epsilon$ are bijective and satisfy the projection commutation property (V 4), the correctness of NP-HMC is implied by the correctness of Direction Multiple Step NP-iMCMC sampler.

### 7.4.2 Nonparametric Discontinuous HMC (NP-DHMC)

Similar to Discontinuous HMC discussed in Sec. 4.4.2, we can form a discontinuous variant of the NP-HMC sampler using a different bijection for the discontinuous variables.

Recall the DHMC algorithm samples a new proposal by simulating the leapfrog steps for the continuous variables $C$ and the discontinuous steps for the discontinuous variables $D$, resulting with the bijection

$$\mathbf{Int} := (\phi^M_{\epsilon/2} \circ \phi^P_{\epsilon/2} \circ \chi^D_\epsilon \circ \phi^P_{\epsilon/2} \circ \phi^M_{\epsilon/2})^L$$

on $\mathbb{R}^n \times \mathbb{R}^n$ where $\chi^D_\epsilon(\boldsymbol{x}, \boldsymbol{v})$ is the result of applying $\psi^j_\epsilon$ defined by

$$\psi^j_\epsilon(\boldsymbol{x}, \boldsymbol{v}) := \begin{cases} (\boldsymbol{x} + \epsilon \mathrm{sign}(\boldsymbol{v}^j)\boldsymbol{e}_j, \boldsymbol{v} - \mathrm{sign}(\boldsymbol{v}^j)(\Delta U)\boldsymbol{e}_j) & \text{if } |\boldsymbol{v}^j| > \Delta U \\ (\boldsymbol{x}, R_j \cdot \boldsymbol{v}) & \text{otherwise.} \end{cases}$$

for each $j \in D$, where $\Delta U := U(\boldsymbol{x} + \epsilon \mathrm{sign}(\boldsymbol{v}^j)\boldsymbol{e}_j) - U(\boldsymbol{x})$, $\boldsymbol{e}_j$ is the $j$-th standard basis vector, $R_j := \mathsf{diag}(1, \ldots, 1, -1, 1, \ldots, 1)$ is the diagonal matrix with diagonal entries 1 everywhere except in the $j$-th position, where it is -1 and $\epsilon$ is the time step.

Similar to its continuous counterparts $\phi^M$ and $\phi^P$, the discontinuous step $\psi^j$ also satisfies projection commutation property (V 4), has inverse $(\psi^j_\epsilon)^{-1} = M \circ \psi^j_\epsilon \circ M$ where $M(\boldsymbol{x}, \boldsymbol{v}) := (\boldsymbol{x}, -\boldsymbol{v})$ (Prop. 13). Moreover, the absolute value of the Jacobian determinant of the update is $|\det \nabla \psi^j_\epsilon(\boldsymbol{x}, \boldsymbol{v})| = 1$.

Consider the slices of $\psi^j_\epsilon$. Let $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{R}^n \times \mathbb{R}^n$ be a state with an instance $\boldsymbol{t}$ of $\boldsymbol{x}$ where $|\boldsymbol{t}| < |\boldsymbol{x}|$. For $j \neq n$,

$$(\mathsf{drop}_{n-1} \circ \psi^j_\epsilon)(\boldsymbol{x}, \boldsymbol{v}) = \mathsf{drop}_{n-1}(\boldsymbol{x}, \boldsymbol{v}),$$

whereas for $j = n$, the difference in potential energy before and after the discontinuous step for dimension $j = n$ would be the same, hence $\Delta U = 0$ and

$$\begin{aligned} (\mathsf{drop}_{n-1} \circ \psi^j_\epsilon)(\boldsymbol{x}, \boldsymbol{v}) &= \mathsf{drop}_{n-1}(\boldsymbol{x} + \epsilon \mathrm{sign}(\boldsymbol{v}^n)\boldsymbol{e}_n, \boldsymbol{v} - \mathrm{sign}(\boldsymbol{v}^n)(\Delta U)\boldsymbol{e}_n) \\ &= \mathsf{drop}_{n-1}(\boldsymbol{x} + \epsilon \mathrm{sign}(\boldsymbol{v}^n)\boldsymbol{e}_n, \boldsymbol{v}) \\ &= (\boldsymbol{x}^n + \epsilon \mathrm{sign}(\boldsymbol{v}^n), \boldsymbol{v}^n). \end{aligned}$$

Hence, similar to the NP-HMC sampler, DHMC can be extended using the Direction Multiple Step NP-iMCMC sampler to work on nonparametric models.

**Listing 7.9:** *Integrator for NP-DHMC*

```
def NPintegrator(x0,v0):
  (x,v) = (x0,v0)
  n = len(x0)
  for i in range(L):
    v[C] = v[C] - ep/2 * grad(U)(x[C])
    x[C] = x[C] + ep/2 * v[C]
    for j in D:
      x* = x
      x*[j] = x[j] + ep*sign(v[j])
      if abs(v[j]) > U(x*)-U(x):          # refract
        x = x*
        v[j] = v[j] - sign(v[j])*(U(x*)-U(x))
      else:                               # reflect
        v[j] = -v[j]
      # extend step
      while not intersect(instance(x),support(w)):
        n = n + 1
        x0.append(normal); v0.append(normal)
        (y,u) = NPintegratorslice((x0,v0),n,ep*m)
        x.append(y); v.append(u)
    x[C] = x[C] + ep/2 * v[C]
    v[C] = v[C] - ep/2 * grad(U)(x[C])
  return ((x0,v0),(x,v))

def NPintegratorslice((x0,v0),n,t):
  (y,u) = (x0[n] + t*v0[n], v0[n]) if n in C    # slice of leapfrog
          else (x0[n] + t*sign(v0[n]),v0[n])    # slice of disstep
  return (y,u)

def inverse_NPintegrator(x0,v0):
  (x0,v0) = momflip(x0,v0)
  ((x0,v0),(x,v)) = NPintegrator(x0,v0)
  return (momflip(x0,v0), momflip(x,v))
```

**Pseudocode for NP-DHMC** Listing 7.10 gives the SPCF implementation `NPDHMC` of the NP-DHMC sampler as an instance of the Direction Multiple Step NP-iMCMC sampler. (Terms that differ from DHMC is highlighted.) First notice that we only need to perform the extend step in the integrator implemented by `NPintegrator` in Listing 7.9 after the discontinuous step because only the discontinuous variables can fall out of the current dimension. Moreover, say the target density is continuous w.r.t. the extended dimension $n+1$, the state is extended as in NP-HMC. Otherwise, the slice of $\psi_t^{n+1}$ is performed instead.

**Listing 7.10:** *Pseudocode for the NP-DHMC algorithm*

```
def NPDHMC(t0):
  d0 = coin                                        # direction step
  k0 = dim(t0)                                     # initialisation step
  x0 = t0
  v0 = [normal]*k0                                 # stochastic step
  ((x0,v0),(x,v)) = NPintegrator(x0,v0) if d0
                    else inverse_NPintegrator(x0,v0)
  d = not d0
  n = len(x0)
  t = intersect(instance(x),support(w))[0]         # accept/reject step
  return t if uniform < min{1,w(t)/w(t0)*pdfnormal[n](x)/pdfnormal[n](x0)*
                                        pdfnormal[n](v)/pdfnormal[n](v0)}
         else t0
```

**Listing 7.11:** *Pseudocode for the Generalised NP-DHMC algorithm*

```
GenNPHMC((x0,v0),d0) = PersistentHMC(CorruptMom((x0,v0),d0))

def CorruptMom((x0,v0),d0):
  u0 = [normal(v0[i]*sqrt(1-alpha^2), alpha^2) for i in range(len(v0))]
  (((x,v),d),u) = (((x0,u0),d0),v0)
  return ((x,v),d)

def PersistentNPDHMC((x0,v0),d0):
  k0 = dim(t0)                                       # initialisation step
  (x0,v0),(x,v) = NPintegrator(x0,v0) if d0          # multiple step
                  else inverse_NPintegrator(x0,v0)
  d = not d0                                         # flip direction
  (x*,v*) = intersect(instance(x,v),support(HMCw))[0]   # accept/reject
  return ((x*,v*), not d)
           if uniform < min{1, HMCw(x*,v*)/HMCw(x0,v0) *
                               pdfnormal[n](x)/pdfnormal[n](x0) *
                               pdfnormal[n](v)/pdfnormal[n](v0) }
           else ((x0,v0), d)
```

### 7.4.3 Generalised NP-DHMC

With the catalogue of techniques explored in Sec. 7.3.4, different irreversible variants of the NP-DHMC algorithm can be formed. Here we focus on the Generalised NP-DHMC algorithm which can be seen as a nonparametric and discontinuous extension of the Generalised HMC algorithm (Horowitz, 1991).

**Generalised HMC** As discussed in Sec. 4.4.3, two changes are made to the conventional HMC algorithm in Generalised HMC to generate an irreversible Markov chain on

$\mathbb{R}^n \times \mathbb{R}^n$, namely

1. a "corrupted" momentum is used to move round the state space; and
2. the direction is "persisted" if the proposal is accepted; otherwise it is negated.

Moreover, it can be presented as a composition of an iMCMC algorithm that "corrupts" the momentum and a Persistent iMCMC algorithm that uses Hamiltonian dynamics to find a new state with a persisting direction. We consider a similar approach in our construction of a nonparametric and discontinuous extension of Generalised HMC.

**State Density** Let the state $(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{R}^n \times \mathbb{R}^n$ have density $w'(\boldsymbol{x}, \boldsymbol{v}) := w(\boldsymbol{x})$ w.r.t. the normal distribution $\mathcal{N}_{2n}$. It is clear that this density $w'$ is integrable (V1) and almost surely terminating (V2). By setting the parameter index map to $\iota_{\mathbb{X}}(n) := 2n$ and parameter space $\mathbb{X}^{(n)} := \mathbb{R}^n \times \mathbb{R}^n$, the state $(\boldsymbol{x}, \boldsymbol{v})$ of length $2n$ is a $n$-dimensional parameter variable. `HMCw` in Listing 4.11 is a SPCF implementation of $w'$.

**Corrupt Momentum** Given the current state $(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{R}^n \times \mathbb{R}^n$ with direction $d_0 \in 2$, a new momentum is drawn from the distribution $\mathcal{N}_n(\boldsymbol{v}_0\sqrt{1-\alpha^2}, \alpha^2)$ for a hyper-parameter $\alpha \in [0, 1]$. Note that $\alpha$ is the degree of momentum persistence in the algorithm and can be fine-tuned to deal with different models. $\alpha = 1$ means a fresh momentum will be sampled and hence making the resulting Generalised NP-DHMC identical to NP-DHMC.

This can be presented in the NP-iMCMC format with the auxiliary variable $\boldsymbol{u}$ sampled from $\mathcal{N}_n(\boldsymbol{v}_0\sqrt{1-\alpha^2}, \alpha^2)$ and the swap $(((\boldsymbol{x}_0, \boldsymbol{v}_0), d_0), \boldsymbol{u}) \mapsto (((\boldsymbol{x}_0, \boldsymbol{u}), d_0), \boldsymbol{v}_0)$ as the involution. Since the new state $(\boldsymbol{x}_0, \boldsymbol{u})$ always has an instance in the support of $w'$, and the acceptance ratio is

$$\min\left\{1, \frac{w'(\boldsymbol{x}_0, \boldsymbol{u}) \cdot \varphi_{2n}(\boldsymbol{x}_0, \boldsymbol{u}) \cdot \mathsf{pdf}_2(d_0) \cdot \varphi_n(\boldsymbol{v}_0 \mid \boldsymbol{u}\sqrt{1-\alpha^2}, \alpha^2)}{w'(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot \varphi_{2n}(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot \mathsf{pdf}_2(d_0) \cdot \varphi_n(\boldsymbol{u} \mid \boldsymbol{v}_0\sqrt{1-\alpha^2}, \alpha^2)}\right\} = 1,$$

the extend step (Step 4) and the accept/reject step (Step 5) of the NP-iMCMC sampler can both be skipped. This results in a sampler that has the SPCF implementation `CorruptMom` in Listing 7.11. (This is identical to the one given in Listing 4.11 for `GenHMC`.)

**Persistent NP-DHMC** We introduce the *Persistent NP-DHMC* sampler using the Persistent Multiple Step NP-iMCMC algorithm (Sec. 7.3.4) with the target density $w'$ as follows.

Given a $k_0$-dimensional parameter $(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{X}^{(n)} := \mathbb{R}^n \times \mathbb{R}^n$ and direction $d_0 \in 2$, a *dummy* auxiliary variable $\boldsymbol{u} \in \mathbb{Y}^{(n)} := \mathbb{R}^n$ is sampled from $K^{(n)}((\boldsymbol{x}_0, \boldsymbol{v}_0), \cdot) := \mathcal{N}_n$ to form an initial state $((\boldsymbol{x}_0, \boldsymbol{v}_0), \boldsymbol{u})$. Depending on the direction $d_0$, either $\mathbf{Int} \times$

$\mathsf{id}_{\mathbb{R}^n}$ or its inverse is performed on $((\boldsymbol{x}_0, \boldsymbol{v}_0), \boldsymbol{u})$, one update at a time, extending the dimension as required. Say the initial state is extended to a $n$-dimensional $((\boldsymbol{x}_0^*, \boldsymbol{v}_0^*), \boldsymbol{u}^*)$ and is traversed to the $n$-dimensional new state $((\boldsymbol{x}^*, \boldsymbol{v}^*), \boldsymbol{u}^*)$. Then, the instance $(\boldsymbol{x}, \boldsymbol{v}) \in \mathsf{Supp}(w')$ of the $n$-dimensional parameter $(\boldsymbol{x}^*, \boldsymbol{v}^*)$ is returned alongside the direction variable $d_0$ with probability

$$\min\left\{1; \frac{w'(\boldsymbol{x}, \boldsymbol{v}) \cdot \varphi_n(\boldsymbol{x}^*) \cdot \varphi_n(\boldsymbol{v}^*)}{w'(\boldsymbol{x}_0, \boldsymbol{v}_0) \cdot \varphi_n(\boldsymbol{x}_0^*) \cdot \varphi_n(\boldsymbol{v}_0^*)}\right\}.$$

Note that the auxiliary variable $\boldsymbol{u}$ has no effect on the sampler. Hence, Listing 7.11 gives a SPCF implementation `PersistentNPDHMC` where the stochastic step (Step 2) is skipped.

**Generalised NP-DHMC** Composing the sampler which "corrupts" the momentum with the Persistent NP-DHMC sampler gives us the ***Generalised NP-DHMC*** algorithm, which is a nonparametric extension of Generalised HMC. Listing 7.11 gives the SPCF implementation `GenNPHMC` by composing `CorruptMom` and `PersistentNPDHMC`.

### 7.4.4 Look Ahead NP-HMC

Last but not least, we extend the Look Ahead HMC algorithm (Sohl-Dickstein et al., 2014; Campos and Sanz-Serna, 2015) discussed in Sec. 4.4.3.

**Look Ahead HMC** Recall, the Look Ahead HMC sampler modifies the Generalised HMC algorithm by performing *extra* leapfrog steps when the proposal state is rejected, resulting in a high acceptance rate. There are two different ways of formulating the Look Ahead HMC sampler. We can either use the *Multiple HMC* sampler, which draws a random number in the stochastic step and then perform said number of steps; or the *Extra Chance HMC* sampler, which after each set of leapfrog steps, determines whether to perform an extra set of leapfrog steps in the acceptance step. To form a nonparametric and discontinuous variant of Look Ahead HMC, we consider the Extra Chance HMC sampler.

**Extra Chance NP-DHMC** Similar to Generalised NP-DHMC, we formulate the *Extra Chance NP-DHMC* sampler as a composition where we first corrupt the momentum of the current parameter.

Then, given the corrupted $k_0$-dimensional parameter $(\boldsymbol{x}_0, \boldsymbol{v}_0) \in \mathbb{R}^n \times \mathbb{R}^n$ and direction $d_0 \in 2$, we use the Persistent Multiple Step iMCMC algorithm (Sec. 7.3.4) to sample from the target density $w'(\boldsymbol{x}, \boldsymbol{v}) := w(\boldsymbol{x})$. In the stochastic step, a random variable $u \in [0, 1)$ is sampled from the uniform distribution $\mathcal{U}(0, 1)$ to form an initial state $((\boldsymbol{x}_0, \boldsymbol{v}_0), u, d_0)$.

**Listing 7.12:** *Pseudocode for the Look Ahead NP-DHMC algorithm*

```
LookAheadNPDHMC((x0,v0),d0) = ExtraChanceNPDHMC(CorruptMom((x0,v0),d0))

def ExtraChanceNPDHMC((x0,v0),d0):
  k0 = dim(x0)                                    # initialisation step
  u = uniform                                     # stochastic step
  # start of multiple step
  (x,v) = (x0,v0)
  stop = False
  j = 1
  while not stop:
    ((xl,vl),(x,v)) = NPintegrator(x,v) if d0
                      else inverse_NPintegrator(x0,v0)
    n = len(x)
    # update the initial state (x0,v0) using (xl,vl)
    for i in range(len(x0),n):
      (y,u) = momflip(NPintegratorslice(momflip(xl,vl),i+1,ep*(j-1)*L)) if d0
              else NPintegratorslice((xl,vl),i+1,ep*(j-1)*L)
      x0.append(y); v0.append(u)
    if u > min{1,HMCw(x,v)/HMCw(x0,v0) *
                 pdfnormal[n](x)/pdfnormal[n](x0) *
                 pdfnormal[n](v)/pdfnormal[n](v0) }:
      if j <= J:  # perform an extra set of leapfrog steps
        j = j + 1
      else:       # no leapfrog steps is performed
        (x,v) = (x0,v0)
        stop = True
        d = d0
    else:         # enough leapfrog steps are performed
      stop = True
      d = not d0
  return ((x,v), not d)
```

Similar to Extra Chance HMC, the first set **Int** of integrator steps is performed on $(\boldsymbol{x}_0, \boldsymbol{v}_0)$, extending the dimension as required. The acceptance ratio is then computed and compared to the value of $u$. If $u$ is lower than the acceptance ratio, the proposal state is returned. Otherwise, an extra set **Int** is performed. This is repeated for at most $J$ times. If $u$ is still higher than the acceptance ratio after $J$ sets of integrator steps, the initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ is returned. This can be encoded as a set of bijections similar to the one given for Extra Chance HMC.

Notice that if the dimension is extended in the $j$-th set of integrator steps, the algorithm needs to back track and update the initial state $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ using the slice of the involution (highlighted).

Combining `ExtraChanceNPDHMC` with `CorruptMom`, the `LookAheadNPDHMC` function implements the Look Ahead NP-DHMC sampler in Listing 7.12.

**Table 7.1:** *Geometric distribution example: total variation difference from the ground truth, averaged over 10 runs, and standard deviation. Each run:* $10^3$ *samples with* $L \in \{2, 5\}$ *leapfrog steps of size* $\epsilon = 0.1$. *with corruption parameter* $\alpha \in \{0.1, 0.5\}$ *and look-ahead* $K \in \{1, 2\}$.

| | Corruption | TVD from ground truth |
|---|---|---|
| $L = 2$ | $\alpha = 0.1$ | $0.0534 \pm 0.0058$ |
| $L = 2$ | $\alpha = 0.5$ | $0.0570 \pm 0.0115$ |
| $L = 2$ | $\alpha = 1$ | $0.0768 \pm 0.0181$ |
| $L = 5$ | $\alpha = 0.1$ | $0.0461 \pm 0.0083$ |
| $L = 5$ | $\alpha = 0.5$ | $0.0464 \pm 0.0074$ |
| $L = 5$ | $\alpha = 1$ | $0.0524 \pm 0.0069$ |

## 7.5 Experiments

These NP-HMC variants are evaluated in (Mak et al., 2022) on the following benchmarks: a model for the geometric distribution, a model involving a random walk, an unbounded Gaussian mixture model, and a Dirichlet process mixture model. They are all nonparametric models that can be defined in the SPCF language. We ran the NP-DHMC, Generalised NP-DHMC parametrised by $\alpha \in [0, 1]$, Lookahead NP-DHMC, parametrised by $\alpha \in [0, 1]$ and $K \geq 0$, which is the number of extra set of leapfrog steps to try before rejecting a proposed sample.

These algorithms are implemented by Fabian Zaiser in Python [1] and are published in (Mak et al., 2022). He has kindly agreed to allow this thesis to use his code to demonstrate the usefulness of these algorithms.

### 7.5.1 Geometric distribution

A classic example for the use of recursion in probabilistic programming is the construction of the geometric distribution with parameter $p$. It can be specified by flipping a biased coin with probability $p$. The pseudocode is given in Ex. 18 (ii).

We ran Generalised NP-DHMC on this problem with $p = 0.2$, $L \in \{2, 5\}$ sets of leapfrog steps and different degree of corruption $\alpha \in [0, 1]$. The algorithm has no problem jumping between the traces of different length (Fig. 7.1). As can be seen in Table 7.1, Generalised NP-DHMC usually performs better with a high level of momentum corruption (low $\alpha$). In fact, the configuration $L = 2, \alpha = 0.1$ is almost as good as $L = 5$ without corruption, despite taking 2.5 times less computing time.

[1] The source code is available at `https://github.com/fzaiser/nonparametric-hmc`.

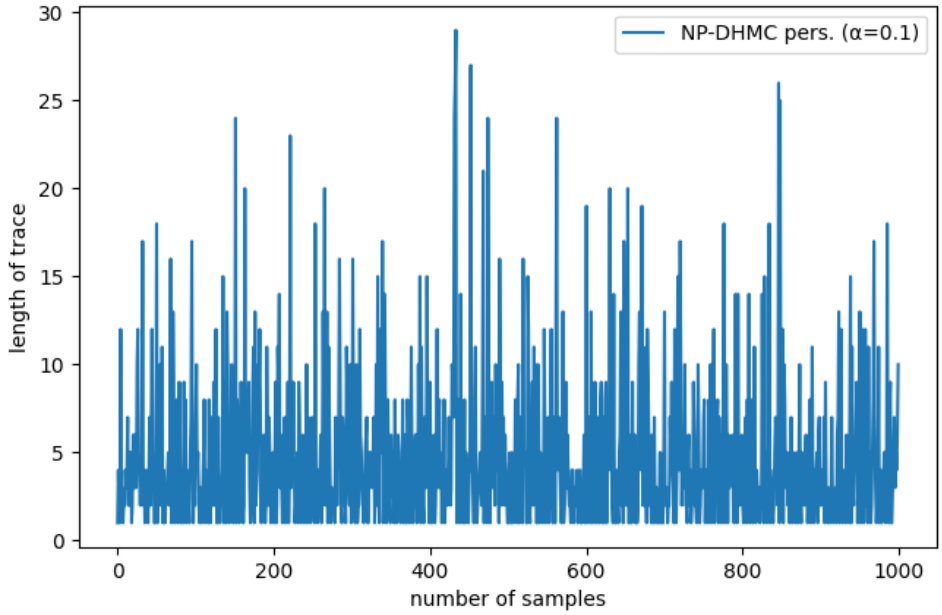


**Figure 7.1:** *Geometric distribution: Length of traces of the persistent NP-DHMC algorithm with corruption parameter* $\alpha = 0.1$.

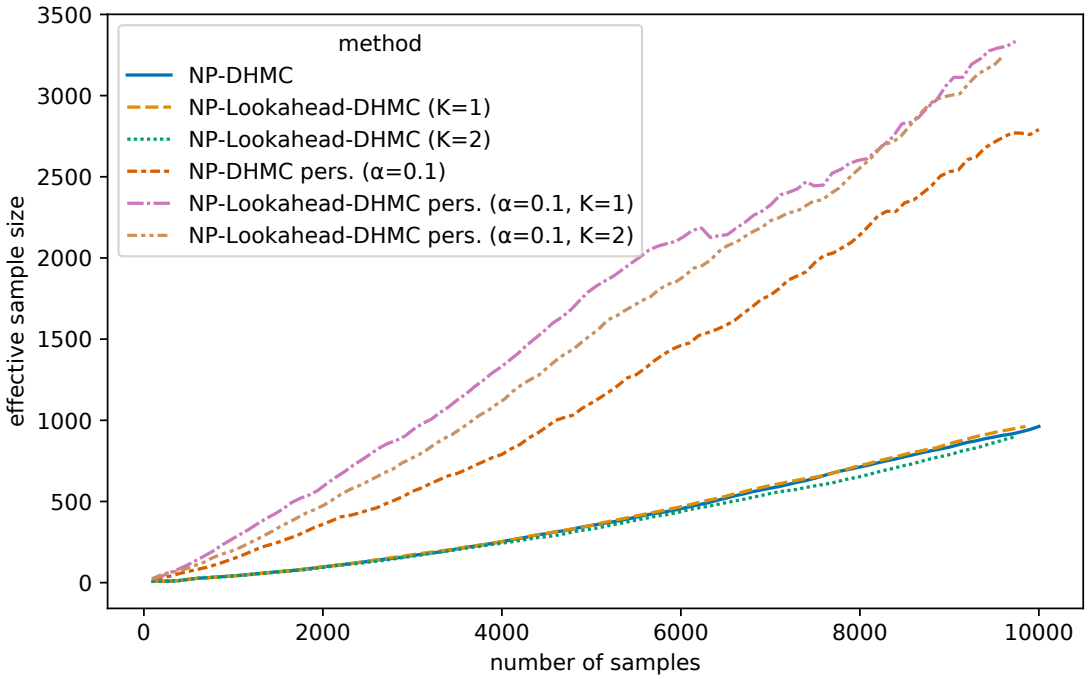


**Figure 7.2:** *Random walk example: ESS in terms of number of samples, computed from 10 runs. Each run:* $10^3$ *samples with* $L = 5$ *leapfrog steps of size* $\epsilon = 0.1$ *with corruption parameter* $\alpha \in \{0.1, 0.5\}$ *and look-ahead* $K \in \{1, 2\}$.

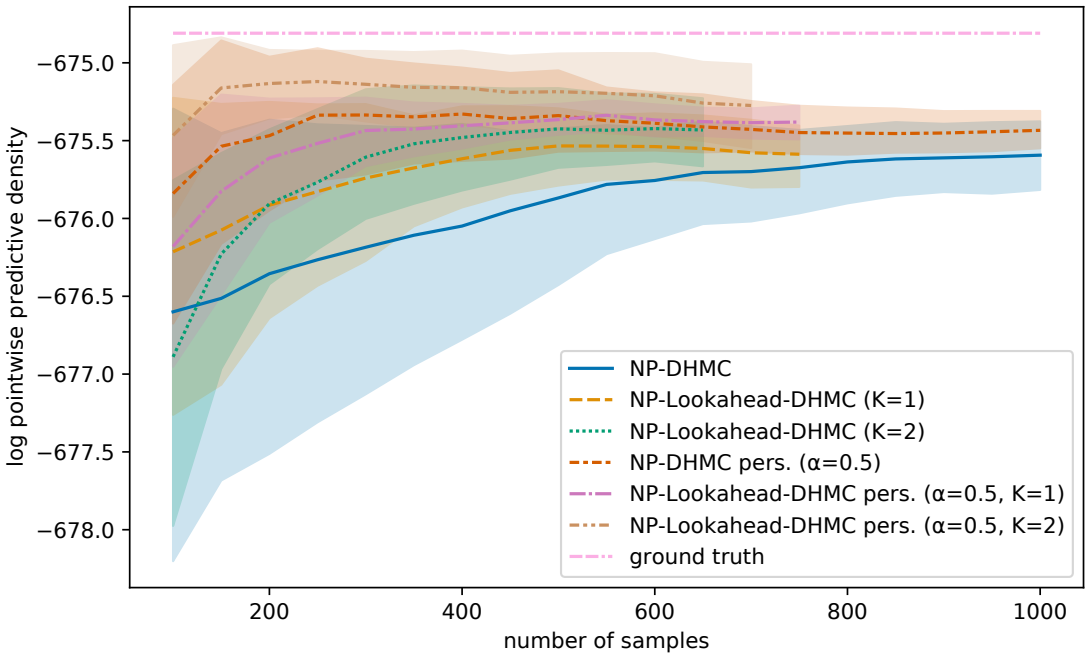


**Figure 7.3:** *Gaussian mixture with Poisson prior: LPPD in terms of number of samples, averaged over 10 runs. The shaded area is one standard deviation. Each run:* $10^3$ *samples with* $L = 25$ *leapfrog steps of size* $\epsilon = 0.05$ *with corruption parameter* $\alpha = 0.5$ *and look-ahead* $K \in \{1, 2\}$.

### 7.5.2 Random walk

Another typical nonparametric model is the random walk. Here we consider a one-sided random walk model with the distance travelled as the observed data. The story goes as follows. A pedestrian starts from a random point in [0, 3] and walks a uniformly random distance of at most 1 in either direction, until they pass 0. Given a (noisily) measured total distance of 1.1 travelled, what is the posterior distribution of the starting point? The pseudocode is given in Sec. 3.2.2.

Fig. 7.2 shows the effective sample size (ESS) in terms of the number of samples drawn, comparing NP-DHMC, Generalised NP-DHMC ($\alpha = 0.1$) and Look-ahead NP-DHMC ($J \in \{1, 2\}$). We can see again that irreversible MCMC methods are clearly advantageous. Look-ahead ($J \in \{1, 2\}$) seems to give an additional boost on top. We ran all these versions with the same computation time budget, which is why the lines for $J = 1, 2$ are cut off before the others.

### 7.5.3 Infinite Gaussian mixture model

We also consider the following infinite Gaussian mixture model where the number of Gaussian distributions is drawn from a Poisson prior. The pseudocode is given by

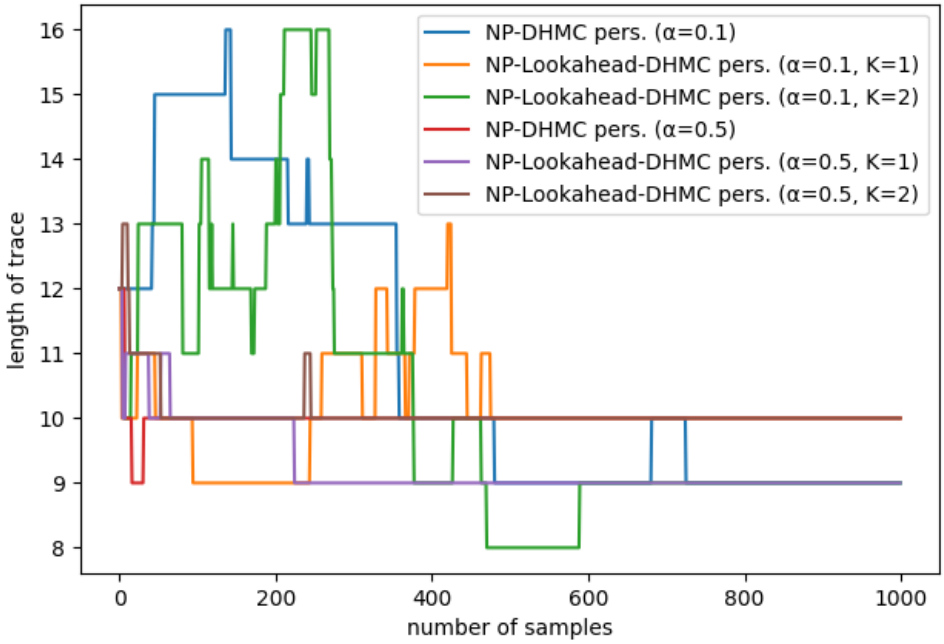


**Figure 7.4:** *Gaussian mixture with Poisson prior: Length of traces.*

```
def iGMM():
  K = Poisson(10)+1
  for k in range(K):
    mean[k] = [uniform(0,100)]*3
  for d in Data:
    score(pdfnormal(mean[k], [100]*3)(d))
```

A training data set is generated from a mixture of 9 components (the ground truth). We ran NP-DHMC, Generalised NP-DHMC ($\alpha = 0.5$) and Look Ahead NP-DHMC ($\alpha = 0.5$ and $J \in \{1, 2\}$). These algorithms can clearly jump between dimensions (Fig. 7.4). Furthermore, we compute the log pointwise predictive density (LPPD) on a test data set drawn from the same distribution as the training data. It is given in Fig. 7.3 in terms of the number of samples. Note that the experiments with Look Ahead NP-DHMC ($K \in \{1, 2\}$) converge more quickly than the others.

### 7.5.4 Dirichlet process mixture model

Last but not least, we consider a Gaussian mixture whose weights are drawn from a Dirichlet process.

The setup is the same as for the Poisson prior, and the results are shown in Fig. 7.5. The version with corruption is worse at the start but obtains a better LPPD at the end.

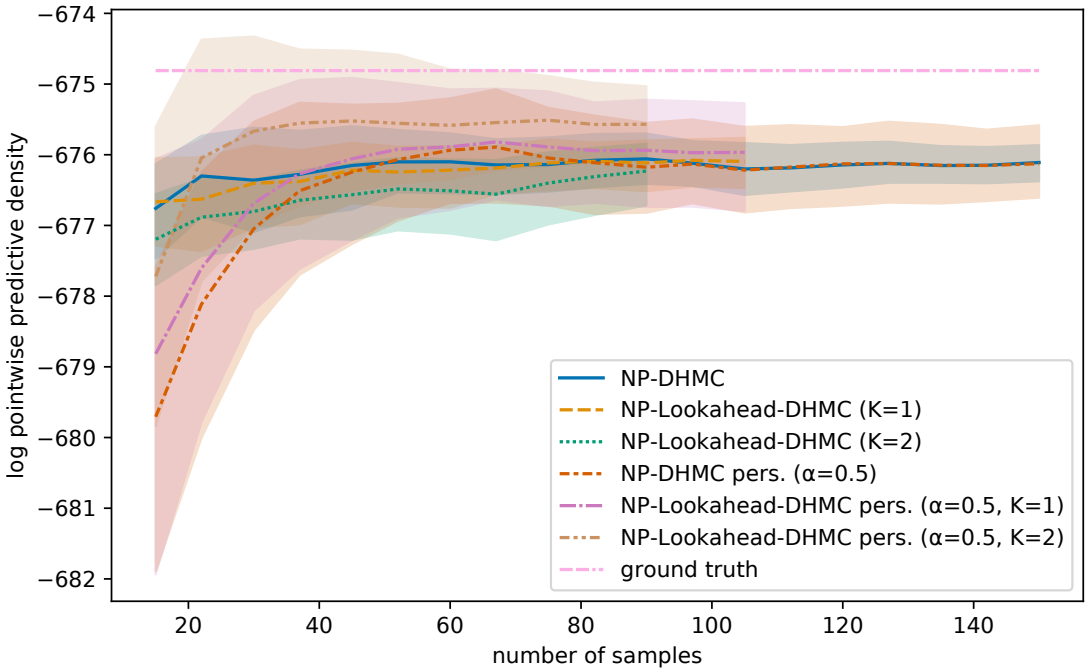


**Figure 7.5:** *Dirichlet process mixture: LPPD in terms of number of samples, averaged over 10 runs. The shaded area is one standard deviation. Each run: 150 samples with* $L = 20$ *leapfrog steps of size* $\epsilon = 0.05$*, with corruption parameter* $\alpha = 0.5$ *and look-ahead* $K \in \{1, 2\}$.

Look-ahead ($K \in \{1, 2\}$) yields a small additional boost in the LPPD. It should be noted that the variance over the 10 runs is larger in this example than in the previous benchmarks, so the conclusion of this benchmark is less clear-cut.

*Dominus illuminatio mea.*

— Psalm 26

# 8 Conclusion

## 8.1 Summary

In Chapter 3, we introduced probabilistic programming: the idea of constructing probabilistic models for Bayesian inference as computer programs where their posterior distributions are computed automatically. To gain a theoretical understanding of this programming paradigm, we discussed the Statistical Programming Computable Functions (SPCF), a probabilistic variant of the infamous functional PCF language, where all computable probabilistic models can be specified. We gave a simple small-step reduction system for SPCF programs, and defined for each SPCF program its density function and value measure. This theoretical understanding of SPCF gave us the means to discuss features of the probabilistic programming paradigm. In particular, we identified a key characteristic of the densities, namely that they must be *tree representable* (Prop. 6).

We then turned our attention to the computation of the posterior distributions of SPCF programs in Chapter 4. Since calculating the exact posterior distribution for complex models is often intractable in practice, it is necessary for probabilistic programming languages to implement approximations like the Markov Chain Monte Carlo (MCMC) inference methods, where the posterior is simulated via a Markov chain of samples. We studied the recently suggested involutive Markov Chain Monte Carlo (iMCMC) framework, and described the popular Hamiltonian Monte Carlo (HMC) and Reversible Jump MCMC (RJMCMC) samplers as instances of iMCMC. Furthermore, we discussed how discontinuous and irreversible extensions of HMC can be formed by applying generic iMCMC techniques on it.

The HMC and RJMCMC samplers both have their advantages and limitations. While HMC is easily extensible and works out-of-the-box, it can only compute the posterior of probabilistic models with a finite number of random variables and is not suitable for the nonparametric models that can be described in SPCF. On the other hand, if we were to use the RJMCMC method on a nonparametric model, we are required to define trans-dimensional (between-model) mappings which makes the resulting inference algorithm model-specific.

This thesis proposed the Nonparametric Involutive Markov chain Monte Carlo (NP-iMCMC) algorithm in Chapter 5, a general framework to design MCMC algorithms for nonparametric probabilistic models specified in the SPCF language. Relying on the tree representable structure of their density functions, the NP-iMCMC algorithm automates the trans-dimensional movement in the sampling process and only requires the specification of proposal distributions and mappings on fixed dimensional spaces which are provided by iMCMC methods like HMC. As SPCF can specify all computable probabilistic models, NP-iMCMC is applicable to virtually all useful probabilistic models. Furthermore, techniques identified for iMCMC can also be applied on the NP-iMCMC sampler to facilitates powerful extensions. With some minor assumptions, we in Chapter 6 justified the NP-iMCMC algorithm and proved that the generated Markov chain preserves the target distribution of the given SPCF program.

Finally, we designed some MCMC algorithms for nonparametric models by extending the conventional HMC algorithm, an instance of iMCMC, via the Multiple Step NP-iMCMC framework described in Chapter 7. The resulting algorithm which we called the Nonparametric HMC (NP-HMC) works out-of-the-box and can be applied to virtually all useful probabilistic models. Furthermore, we applied some techniques to NP-HMC to form discontinuous and nonparametric variants of Generalised HMC and Look Ahead HMC with minimal effort.

## 8.2 Evaluation

Probabilistic programming makes Bayesian inference more accessible by embedding the computation of the posterior inference into its analysis process. This frees up resource for model designing and analysis. However, no single inference technique fits all scenarios. In fact, the art of designing inference algorithms for probabilistic programming mostly resides in an adequate trade-off amongst reliability, efficiency, applicability, adaptability, simplicity of implementation, easiness to extend, and automation. Here, we provide an assessment of how the proposed NP-HMC algorithm and its variants in Chapter 7 dealt with these desirable qualities.

**Theoretically reliable** Using the operational semantics of SPCF, we proved the correctness of the general NP-iMCMC framework (Lem. 4) and hence NP-HMC and its variants. This gives a theoretical guarantee that these inference algorithms work in the way we expect, i.e. the generated Markov chain indeed simulates the posterior distribution.

**Efficiency depends on automatic differentiation** It is not a coincidence that the HMC algorithm becomes more popular after computer systems implement the efficient automatic differentiation method for computing gradients. The main computation of HMC rests in the calculation of the gradient of the density function in each leapfrog step. NP-HMC and its variants inherit this property, and hence their efficiencies depend on that of computational differentiation. Note that the high acceptance ratio of NP-HMC and variants make it more efficient than simple MCMC inferences.

**Widely applicable** The NP-HMC algorithm and variants are applicable to all almost surely terminating and integrable probabilistic models, making it a generic option for virtually all useful programs defined in a probabilistic programming language.

**Fairly Adaptable** NP-HMC and variants do not explicitly incorporate important characteristics of specific target distribution. For instance, NP-HMC does not make the most out of the modality of the distribution (unlike Reversible Jump MCMC (Green, 1995) and Divide, Conquer and Combine (Zhou et al., 2020)) or the type of random variables being sampled (unlike Lightweight Metropolis-Hastings (Wingate et al., 2011)). However, given a smooth target density, the Hamiltonian dynamics used by NP-HMC traverses the target density in such a way that areas with high density are visited more often and hence sampled more often.

**Simple to implement** Given that the host language has an efficient way to compute gradients and can compile the density function of a probabilistic program, the implementation of the NP-HMC algorithm and variants are surprisingly straightforward.

**Simple to extend** Due to the success of the conventional HMC algorithm, it has been extended for different tasks. Similarly the NP-HMC algorithm can also be extended. In this thesis, we presented the NP-DHMC, a discontinuous variant of NP-HMC, the Generalised NP-DHMC and Look Ahead NP-DHMC.

**Almost automatic** Once the involutions and auxiliary kernels in NP-iMCMC are set, the resulting inference algorithm is fully automatic for probabilistic programs that might have an unbounded number of random variables. For example after the hyperparameters are tuned, NP-HMC and its variants are fully automatic and can jump between states of varying dimensions.

## 8.3 Future Direction

This thesis has taken a step towards designing automatic inference algorithms for probabilistic programming that can serve as a foundation for further development on computation of posterior.

One can improve the theoretical reliability of NP-iMCMC by providing an ergodic theory which discusses the conditions on which the Markov chain generated by NP-iMCMC indeed converges to the target distribution almost surely. Here we have pursued this task to some extent by proving that the stationary distribution of the Markov chain is the target distribution, but it is evident that the ergodic theory needs further investigation.

To design a *fully* automatic trans-dimensional inference for probabilistic programs, it would be beneficial to develop a nonparametric extension of the Non-U-Turn Sampler (NUTS) (Hoffman and Gelman, 2014), which embeds the tuning of the hyperparameters in the HMC algorithm. It would be interesting to examine NUTS as a iMCMC algorithm and extend it using the NP-iMCMC framework.

One limitation of NP-iMCMC is its lack of adaptability to the characteristics of probabilistic models. A future direction would be to consider correspondence between the states when their dimensions are being changed in the extend step. This technique should make the algorithm better adapt to the shape of the target distribution, resulting in higher quality samples.

Finally, it would be interesting to implement the NP-iMCMC algorithm in a host language that supports both probabilistic and differentiable constructs. This would make designing trans-dimensional MCMC inferences more accessible.

# Appendices

# Bibliography


Hadi Mohasel Afshar and Justin Domke. Reflection, refraction, and hamiltonian monte carlo. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3007–3015, 2015. URL http://papers.nips.cc/paper/5801-reflection-refraction-and-hamiltonian-monte-carlo.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 72(3):269–342, 2010. ISSN 13697412, 14679868. URL http://www.jstor.org/stable/40802151.

Robert J. Aumann. Borel structures for function spaces. *Illinois J. Math.*, 5(4):614–630, 12 1961. URL https://projecteuclid.org:443/euclid.ijm/1255631584.

John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, Irving Ziller, Robert A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In Morton M. Astrahan, editor, *Papers presented at the 1957 western joint computer conference: Techniques for reliability, IRE-AIEE-ACM 1957 (Western), Los Angeles, California, USA, February 26-28, 1957*, pages 188–198. ACM, 1957. doi: 10.1145/1455567.1455599. URL https://doi.org/10.1145/1455567.1455599.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019. URL http://jmlr.org/papers/v20/18-403.html.

David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 33–46, 2016.

Mónica F. Bugallo, Víctor Elvira, Luca Martino, David Luengo, Joaquín Míguez, and Petar M. Djurić. Adaptive importance sampling: The past, the present, and the future. *IEEE Signal Processing Magazine*, 34:60–79, 2017.

Cédric M. Campos and J.M. Sanz-Serna. Extra chance generalized hybrid monte carlo. *Journal of Computational Physics*, 281:365–374, 2015. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2014.09.037. URL https://www.sciencedirect.com/science/article/pii/S0021999114006731.

Eric Cances, Frédéric Legoll, and Gabriel Stoltz. Theoretical and numerical comparison of some sampling methods for molecular dynamics. *ESAIM: Mathematical Modelling and Numerical Analysis*, 41:351–389, 03 2007. doi: 10.1051/m2an:2007014.

Bradley P. Carlin and Siddhartha Chib. Bayesian model choice via markov chain monte carlo methods. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(3):473–484, 1995. ISSN 00359246. URL http://www.jstor.org/stable/2346151.

John Chavis, Amy Cochran, and Christopher Earls. Cu-msdsp: A flexible parallelized reversible jump markov chain monte carlo method. *SoftwareX*, 14:100664, 06 2021. doi: 10.1016/j.softx.2021.100664.

Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 368–392. Springer, 2017. doi: 10.1007/978-3-662-54434-1_14. URL https://doi.org/10.1007/978-3-662-54434-1_14.

Marco Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. Automating involutive mcmc using probabilistic and differentiable programming, 2020.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 221–236. ACM, 2019.

Vincent Danos and Thomas Ehrhard. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Inf. Comput.*, 209(6):966–991, 2011. doi: 10.1016/j.ic.2011.02.001. URL https://doi.org/10.1016/j.ic.2011.02.001.

Luc Devroye. *Discrete Univariate Distributions*, pages 485–553. Springer-Verlag, New York, NJ, USA, 1986.

Arnaud Doucet, Nando de Freitas, and Neil Gordon. *An Introduction to Sequential Monte Carlo Methods*, pages 3–14. Springer New York, New York, NY, 2001.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. The computational meaning of probabilistic coherence spaces. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 87–96, 2011. doi: 10.1109/LICS.2011.29. URL https://doi.org/10.1109/LICS.2011.29.

Thomas Ehrhard, Christine Tasson, and Michele Pagani. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 309–320. ACM, 2014. doi: 10.1145/2535838.2535865. URL https://doi.org/10.1145/2535838.2535865.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. Full abstraction for probabilistic PCF. *CoRR*, abs/1511.01272, 2015. URL http://arxiv.org/abs/1511.01272.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *PACMPL*, 2(POPL):59:1–59:28, 2018.

Víctor Elvira, Luca Martino, David Luengo, and Mónica F. Bugallo. Generalized Multiple Importance Sampling. *Statistical Science*, 34(1):129 – 155, 2019. doi: 10.1214/18-STS668. URL https://doi.org/10.1214/18-STS668.

D. H Fremlin. *Measure theory*, volume two. Torres Fremlin, Colchester, 2010. ISBN 978-0-9538129-7-4.

Philippe Gagnon and Arnaud Doucet. Nonreversible jump algorithms for bayesian nested model selection. *Journal of Computational and Graphical Statistics*, 30:1–12, 11 2020. doi: 10.1080/10618600.2020.1826955.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 09–11 Apr 2018.

Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan: A probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015. doi: 10.3102/1076998615606113.

Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, pages 68–85, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg. ISBN 978-3-540-39041-1.

Simon J. Godsill. On the relationship between markov chain monte carlo methods for model uncertainty. *Journal of Computational and Graphical Statistics*, 10(2):230–248, 2001. ISSN 10618600. URL http://www.jstor.org/stable/1391010.

Simon J. Godsill. Proposal densities and product-space models. In Nils Lid Hjort Peter J. Green and Sylvia Richardson, editors, *Highly Structured Stochastic Systems*, volume 27 of *Oxford Statistical Science Series*, chapter 6A, pages 199–203. Oxford University Press, 2 edition, 2003.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2021-2-4.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In David A. McAllester and Petri Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229. AUAI Press, 2008. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 167–181, 2014. doi: 10.1145/2593882.2593900. URL http://doi.acm.org/10.1145/2593882.2593900.

N.J. Gordon. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)*, 140:107–113(6), April 1993. ISSN 0956-375X.

Peter J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, 12 1995. ISSN 0006-3444. doi: 10.1093/biomet/82.4.711. URL https://doi.org/10.1093/biomet/82.4.711.

Peter J. Green. Trans-dimensional markov chain monte carlo. In Nils Lid Hjort Peter J. Green and Sylvia Richardson, editors, *Highly Structured Stochastic Systems*, volume 27 of *Oxford Statistical Science Series*, chapter 6, pages 179–198. Oxford University Press, 2 edition, 2003.

Peter J. Green and David I. Hastie. Reversible jump mcmc. 2009.

Ulf Grenander and Michael I. Miller. Representations of knowledge in complex systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 56(4):549–581, 1994. doi: https://doi.org/10.1111/j.2517-6161.1994.tb02000.x. URL https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1994.tb02000.x.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, April 1970.

Daniel Heck, Antony Overstall, Quentin Gronau, and Eric-Jan Wagenmakers. Quantifying uncertainty in transdimensional markov chain monte carlo using discrete markov models. *Statistics and Computing*, 29, 07 2019. doi: 10.1007/s11222-018-9828-0.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A Convenient Category for Higher-Order Probability Theory. 2017. URL http://arxiv.org/abs/1701.02547.

Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014. URL http://dl.acm.org/citation.cfm?id=2638586.

Alan M. Horowitz. A generalized guided monte carlo algorithm. *Physics Letters B*, 268(2):247–252, 1991. ISSN 0370-2693. doi: https://doi.org/10.1016/0370-2693(91)90812-5. URL https://www.sciencedirect.com/science/article/pii/0370269391908125.

Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 101–114, 1979.

Thomas Leventis. Probabilistic böhm trees and probabilistic separation. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 649–658, 2018. doi: 10.1145/3209108.3209126. URL http://doi.acm.org/10.1145/3209108.3209126.

Yicheng Luo, Antonio Filieri, and Yuan Zhou. Symbolic parallel adaptive importance sampling for probabilistic program analysis. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1166–1177. ACM, 2021. doi: 10.1145/3468264.3468593. URL https://doi.org/10.1145/3468264.3468593.

Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg,*

*March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 432–461. Springer, 2021a. doi: 10.1007/978-3-030-72019-3_16. URL https://doi.org/10.1007/978-3-030-72019-3_16.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric hamiltonian monte carlo. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7336–7347. PMLR, 2021b. URL http://proceedings.mlr.press/v139/mak21a.html.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric involutive markov chain monte carlo. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 14802–14859. PMLR, 2022. URL https://proceedings.mlr.press/v162/mak22a.html.

Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014. URL http://arxiv.org/abs/1404.0099.

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. doi: 10.1063/1.1699114. URL https://doi.org/10.1063/1.1699114.

T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

Praveen Narayanan and Chung-chieh Shan. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42 (2):1–60, 2020.

Radford M. Neal. Philosophy of bayesian inference. http://www.cs.toronto.edu/~radford/res-bayes-ex.html, 1998. Accessed: 2021-09-23.

Radford M. Neal. Mcmc using hamiltonian dynamics. In Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng, editors, *Handbook of Markov Chain Monte Carlo*, chapter 5. Chapman & Hall CRC Press, 2011.

Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry P. Vetrov. Involutive MCMC: a unifying framework. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 7273–7282. PMLR, 2020. URL http://proceedings.mlr.press/v119/neklyudov20a.html.

Akihiko Nishimura, David B Dunson, and Jianfeng Lu. Discontinuous hamiltonian monte carlo for discrete parameters and discontinuous likelihoods. *Biometrika*, 107 (2):365–380, Mar 2020. ISSN 1464-3510. doi: 10.1093/biomet/asz083. URL http://dx.doi.org/10.1093/biomet/asz083.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 171–182, 2005. doi: 10.1145/1040305.1040320. URL http://doi.acm.org/10.1145/1040305.1040320.

Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5 (3):223–255, 1977. doi: 10.1016/0304-3975(77)90044-5. URL https://doi.org/10.1016/0304-3975(77)90044-5.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 154–165, 2002. doi: 10.1145/503272.503288. URL http://doi.acm.org/10.1145/503272.503288.

Rajesh Ranganath, Sean Gerrish, and David M. Blei. Black box variational inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 814–822. JMLR.org, 2014. URL http://proceedings.mlr.press/v33/ranganath14.html.

Sylvia. Richardson and Peter J. Green. On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 59(4):731–792, 1997. doi: https://doi.org/10.1111/1467-9868.00095. URL https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00095.

Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. C3: lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, volume 51 of *JMLR Workshop and Conference Proceedings*, pages 28–37. JMLR.org, 2016. URL http://proceedings.mlr.press/v51/ritchie16.html.

Nasser Saheb-Djahromi. Probabilistic lcf. In *International Symposium on Mathematical Foundations of Computer Science*, pages 442–451. Springer, 1978.

John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Comput. Sci.*, 2:e55, 2016.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60, 2017.

Dana S. Scott. A type-theoretical alternative to iswim, cuch, OWHY. *Theor. Comput. Sci.*, 121(1&2):411–440, 1993.

Kurt Sieber. Relating full abstraction results for different programming languages. In *Foundations of Software Technology and Theoretical Computer Science, Tenth Conference, Bangalore, India, December 17-19, 1990, Proceedings*, pages 373–387, 1990.

Jascha Sohl-Dickstein, Mayur Mudigonda, and Michael Robert DeWeese. Hamiltonian monte carlo without detailed balance. In *ICML*, 2014.

Sam Staton. Commutative semantics for probabilistic programming. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 855–879. Springer, 2017.

Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 525–534. ACM, 2016. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2935313. URL https://doi.org/10.1145/2933575.2935313.

Matthew Stephens. Bayesian analysis of mixture models with an unknown number of components- an alternative to reversible jump methods. *Annals of Statistics*, 28:40–74, 2000.

Luke Tierney. Markov chains for exploring posterior distributions. *Ann. Statist.*, 22 (4):1701–1728, 12 1994. doi: 10.1214/aos/1176325750. URL https://doi.org/10.1214/aos/1176325750.

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank D. Wood. Output-sensitive adaptive metropolis-hastings for probabilistic programs. In Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, João Gama, Alípio Jorge, and Carlos Soares, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II*, volume 9285 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2015. doi: 10.1007/978-3-319-23525-7_19. URL https://doi.org/10.1007/978-3-319-23525-7_19.

Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL):36:1–36:29, 2019. doi: 10.1145/3290349. URL https://doi.org/10.1145/3290349.

Matthijs Vákár and Luke Ong. On s-finite measures and kernels, 2018.

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *PACMPL*, 2(ICFP):87:1–87:30, 2018.

David Williams. *Probability with Martingales*. Cambridge University Press, 1991. doi: 10.1017/CBO9780511813658.

David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 770–778. JMLR.org, 2011.

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 1024–1032. JMLR.org, 2014. URL http://proceedings.mlr.press/v33/wood14.html.

Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating efficient MCMC kernels from probabilistic programs. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 1068–1076. JMLR.org, 2014. URL http://proceedings.mlr.press/v33/yang14d.html.

Guangyao Zhou. Mixed hamiltonian monte carlo for mixed discrete and continuous variables. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c6a01432c8138d46ba39957a8250e027-Abstract.html.

Yuan Zhou, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. LF-PPL: A low-level first order probabilistic programming language for non-differentiable models. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pages 148–157. PMLR, 2019.

Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 11534–11545. PMLR, 2020. URL http://proceedings.mlr.press/v119/zhou20e.html.