Automating Inference for Non–Standard Models

Yuan Zhou

Keble College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2020

Abstract

Probabilistic models enable us to infer the underlying relationships within data and make decisions based on this information. Certain models are more commonly used not because they are more appropriate to imitate a particular system, but because they are simple enough to analyze given the current computational resources and available inference algorithms. However, there are a broad range of non-standard models that we *could* use to describe real-world tasks better; they may have complex dependency structures, variables following non-common distributions, and/or a stochastic number of variables. But we typically *avoid* using them as these features substantially complicate the inference procedure such that many conventional inference algorithms would fail when dealing with these models.

To alleviate the computational concerns of data scientists or domain experts and free them to develop non-standard, complex models as needed, more sophisticated inference methods need to be more easily accessible. Additionally, because it is usually difficult to extract helpful information of non-standard models ahead of inference, like where the modes might be, such inference techniques should be expected to work reasonably well without too much such information. In an ideal world, domain experts would be able to specify any model of interest, and no longer need to worry about the technical details of inference algorithms or how to implement them. Moreover, these algorithms are provided as generic engines and can be used to reason the models in an automated manner. Achieving so is the ambitious, long-term goal of the emerging field called ***probabilistic programming***.

The aim of this thesis is to make inroads to this ultimate goal: we develop and automate advanced inference methods that are applicable for a broad range of non-standard probabilistic models. In particular, we introduce a novel class of adaptive inference algorithms which can be implemented in a black-box manner. They are especially useful for the models with multiple, separated modes where many off-the-shelf options struggle. Furthermore, we investigate both a *restricted* class of probabilistic programming systems (PPSs) that impose strong constraints on the model class to ensure inference efficiency, and *universal* PPSs which are designed around the long-term goal and aim to support any possible model but substantially complicate the inference procedures. For the former, we propose a principled way to extend them to incorporate a broader range of models and inference engines that are not supported before. For the latter, we develop a general framework to handle one class of the most challenging models where the support of a model can be stochastic.

Acknowledgements

I would first like to give myself a thumbs up for getting through this journey. Doing a DPhil at Oxford is challenging, sometime tough but fulfilling, and full of inspirations. I am grateful for my persistence and courage no matter what is in front of me. The credit must go to my parents and my entire family: thank you for your undoubted support, and I am what I am because of your unwavering love.

I would like to thank my supervisor, Tom Rainforth, for being encouraging and supportive all the time. This journey would be a lot less enjoyable without your accompany. Thank you for setting a high standard for all-academic related matters, encouraging me when I occasionally felt down or lost directions, and supporting me to pursue what I want to achieve. I feel so lucky to have you as my friend and colleague. I would like to thank my supervisor Hongseok Yang. Thank you for introducing me to the world of probabilistic programming and the journey of my DPhil. I am grateful that you have always been accessible and helpful no matter at Oxford or remotely. I admire you for your extraordinary passion and attitude of science, which have also inspired me a lot. I would like to thank my supervisor Yee Whye Teh. Your wide and deep understanding of various subjects and critical suggestions of the directions of projects have been very useful, and your exceptional ability to manage different kinds of work productively and to balance life and work has always been the magic that I want to gain. I feel so proud to be part of OxCSML and thank you for making this possible. I would like to thank my supervisor Frank Wood. Thank you for pushing me to go out of my comfort zone and encouraging me to pursue a better myself. Our original Wood group at Oxford is amazing. I would like to thank my supervisor Sam Staton. Thank you for always taking care of me and being there for whenever I need.

My full gratitude also goes to my additional collaborators: Antonio Filieri, Bradley Gram-Hansen, Tobias Kohn, Xiaoyu Lu, Yicheng Luo, Jan Willem van de Meent, and David Tolpin. Thank you for all the hard work you have put into our projects, and this thesis would not be possible without your efforts. I am also grateful to have friends and colleagues from the OxCSML group at Stats, Frank's lab in Engineering, the group with Luke Ong and Sam at CS, and other friends at Oxford or abroad. Thank you for making this four-year so enjoyable and unforgettable. Finally, I would like to thank Anthony Caterini, Adam Goliński, Tuan Anh Le, Tom Rainforth, David Tolpin, Jean-Francois Ton and Hongseok Yang for proofreading earlier drafts of this thesis.

Contents

1 Introduction **1**

1.1 Thesis Outline 5

2 Probabilistic Machine Learning **7**

2.1 Bayesian Modeling: Assumptions and Examples 8

2.1.1 Example: Gaussian with Unknown Mean 10

2.1.2 Example: State Space Model 11

2.2 Bayesian Inference and Monte Carlo Estimation 13

2.3 Basic Monte Carlo Inference Methods 15

2.3.1 Importance Sampling 15

2.3.2 Sequential Monte Carlo 17

2.3.3 Markov Chain Monte Carlo 19

2.4 Inference Methods beyond the Basic Setup 21

2.4.1 Adaptive Importance Sampling Methods 21

2.4.2 Hamiltonian Monte Carlo 24

2.4.3 Variational Inference 28

3 Inference Trees **32**

3.1 Background and Related Work 34

3.1.1 Adaptive Monte Carlo Inference 34

3.1.2 Multi-Armed Bandits 35

3.1.3 Monte Carlo Tree Search 36

3.2 Algorithm Overview 37

3.3 The Inference Tree Estimator 39

3.3.1 Combining Estimates from Disjoint Partitions 39

3.3.2 Propagation of Estimates 40

3.3.3 Consistency 41

3.4 Partitioning the Target Space 42

3.5 Traversal Strategy 44

3.5.1 Exploitation Target 45

3.5.2 Exploration Target 46
3.6 Refinement Strategy 47
3.7 Theoretical Justification 49
3.8 Experiments 53
3.8.1 Gaussian Mixture Model 53
3.8.2 Chaotic Dynamics Model 55
3.9 Conclusion 59

4 Restricted Probabilistic Programming 60
4.1 A High-Level Introduction to Probabilistic Programming Systems 61
4.2 Model Specification as Program Code 63
4.2.1 An FOPPL Example 64
4.3 The Abstraction Layer 66
4.3.1 Internal Representation of Graphical Models 67
4.4 Inference Methods as Automated Engines for the Restricted Setup 68
4.4.1 Gibbs Sampler 69
4.4.2 Hamiltonian Monte Carlo Engine 70
4.5 Interpretation of Branching Statements in Probabilistic Programs 71

5 LF-PPL: A new Low-level First-order Probabilistic Programming Language 74
5.1 Motivation 75
5.2 Background and Related Work 77
5.3 The Language 78
5.4 Compilation Scheme 80
5.5 Mathematical Foundation and Compilation Details 82
5.5.1 Piecewise Smooth Functions 82
5.5.2 Translation Rules 85
5.5.3 A Compilation Example 89
5.6 Example Inference Engine: Discontinuous HMC 91
5.6.1 Gaussian Mixture Model 91
5.6.2 Heavy Tail Piecewise Model 92
5.7 Conclusion 93

6 Universal Probabilistic Programming 95
6.1 Language Expressivity 96
6.1.1 Open-universe Model 97
6.1.2 Inverting Simulators 98
6.1.3 Mutually Recursive Problem 100

6.2 Execution-based Formalization . . . 102
6.2.1 Addressing Scheme . . . 102
6.2.2 Execution Traces . . . 103
6.2.3 Probability Distribution of Execution Traces . . . 104
6.3 Universal Inference Engines . . . 105
6.3.1 Separation between Program Code and Inference Engine . . . 106
6.3.2 Importance Sampling as a Universal Engine . . . 107
6.3.3 A Single-site Metropolis Hastings . . . 108

7 Divide, Conquer, and Combine 110
7.1 Motivation . . . 110
7.2 Background . . . 113
7.3 Shortcomings of Existing Inference Engines . . . 114
7.3.1 Why is MCMC so Hard with Stochastic Support? . . . 116
7.4 Divide, Conquer, and Combine . . . 118
7.4.1 Divide . . . 118
7.4.2 Conquer . . . 120
7.4.3 Combine . . . 121
7.5 Theoretical Correctness . . . 121
7.6 DCC in Anglican . . . 124
7.6.1 Local Estimators . . . 125
7.6.2 Discovering SLPs . . . 127
7.6.3 Allocating Resources Between SLPs . . . 130
7.7 Experiments . . . 133
7.7.1 Gaussian Mixture Model (GMM) . . . 133
7.7.2 GMM with Misspecified Prior . . . 134
7.7.3 Function Induction . . . 135
7.8 Conclusion . . . 138

8 Conclusion and Future Directions 139

Appendices

A Appendix for Inference Trees 143
A.1 Discrete Variables . . . 143
A.2 Additional Details on Theory . . . 144
A.3 Estimates for Empirical Variance and Effective Sample Size . . . 146
A.4 Derivation of the Pure-Exploitation Target . . . 147

A.5 Additional Density Estimation Details 150
A.5.1 Additional Heuristics 150
A.5.2 Additional Intuition and Parameters 151
A.6 Additional Details on Refinement Strategy 151

B Appendix for LF-PPL **153**
B.1 Discontinuous Hamiltonian Monte Carlo 153
B.2 Program code 156

C Appendix for DCC **157**
C.1 Existing PPSs and Inference Engines for Probabilistic Programs with Stochastic Support 157
C.2 Details on Experiments 161
C.2.1 Gaussian Mixture Model 161
C.2.2 Function Induction 161

Bibliography **164**

1 Introduction

Imagine a world where machines are "smart": they can "understand" the outside world and make decisions like human beings. For instance, given an image, a computer is able to tell whether it contains a cat or a dog. Or, placed at the entrance of a room, a robot can find its way to the exit without crashing into any obstruction. For simple tasks, one may write an algorithm with explicit instructions, in which case, the machine just needs to follow the exact command and has nothing to "learn" by itself. However, for most tasks, like the examples mentioned above, it is difficult or infeasible for a human to write an explicit program to instruct a machine to accomplish so. Can we instead let the machine learn to accomplish a task by using data (with some aids from humans if needed)? *Machine learning* is the study trying to answer this question. It addresses this problem via building mathematical models which describes how a system works, and develops computer algorithms to improve the understanding automatically based on some available sample data, or training data. Akin to how humans learn and gain knowledge, we hope the machine can learn from what it has seen before and use the gained information for future tasks. The concept of *learning from experience* is at the core of machine learning.

One class of widely-used methods focus on learning the input-output relationship. They are designed primarily for the classification or regression tasks, and are known as *discriminative* approaches. They work by constructing models to figure out the mapping functions from the input to the output where parameters of a model are usually trained

on a large dataset with input data and their corresponding true labels (output). Many classical deep learning approaches following this framework have achieved extraordinary success; popular examples include computer vision [Krizhevsky et al., 2012], natural language processing [Manning and Schutze, 1999], speech recognition [Graves et al., 2013], recommendation systems [Gong and Zhang, 2016] and many other applied areas like autonomous vehicles [Huval et al., 2015; Lefèvre et al., 2014]. Such models usually contain (deep) artificial neural networks with multiple layers, which are expressive to describe complex mappings between the input and output, and the parameters of the neural networks are fitted using the training data.

Though this approach has achieved extraordinary results in certain tasks, it is not sufficient to allow machine to gain human-like intelligence: it is infeasible to collect large enough, well-labeled training datasets that cover all possible cases for every task, and the input-output mapping relationship is not adequate to succeed in all real world learning tasks. Thinking of humans, we do not learn the world by remembering every object and their corresponding labels [LeCun et al., 2015]. We seem to be able to understand the world with only small amount of data and generalize the learned knowledge to other unseen objects. For example, a child can tell a cat is an animal with his or her existing knowledge of a dog, even if the child has never seen a cat before.

This brings out another popular form of machine learning referred as *generative* approaches. Unlike the previously mentioned discriminative modeling approach which directly learns the input-output relationship, the generative approach imposes assumptions of the data generation process, and specifies a joint model of the inputs and outputs. Existing knowledge is able to be encoded in the model, and model is also more likely to work well with limited amount of data making use of the joint relationship. Moreover, the probabilistic nature of this approach results in an important fact that the uncertainty of the model is naturally considered whereas it sometimes requires separate estimation in the discriminative counterparts. It is essential since models are always approximations of the real world; when a model is used to make decisions, knowing how confident a certain action is taken is necessary for many tasks [Ghahramani, 2015]. For instance, how likely a vaccine may have adverse effects makes a huge difference in its approval.

In particular, the *Bayesian paradigm* has been a popular and principled framework for machine learning following the generative setting [Ghahramani, 2015]. Such approach works by encoding subjective beliefs as a prior in a model, conditioning on the actual data we have observed, and updating our beliefs using the information from data. The process of obtaining the updated model using Bayes' rule is called *Bayesian inference*. In doing so, we start with the domain expertise when constructing a model and gradually refine it using data following rigorous statistical principles.

To realize the Bayesian framework in a real-world application, traditionally people would design a model according to the problem and then implement the corresponding inference algorithm by hand. An immediate challenge, however, is that the inference procedure usually contains the computation of complex integrals which are intractable. Most problems cannot be solved analytically and would need mathematical approximations to a certain extent. It is also non-trivial to implement the inference algorithm in an efficient way, and to diagnose the statistical properties of the inference results to ensure the correctness. On the other hand, this also restricts the way how one specifies a model as there might not be any inference algorithm able to reason about the model. Certain assumptions of a model are made usually not because they are more appropriate to describe the specific system, but because of the requirements of available inference options. Beyond these difficulties, statistical expertise is highly requested, not only in understanding and implementing an inference method, but also in defining a domain-specific problem into a probabilistic model. Consequently, although the Bayesian approach is intuitive and nicely formulated, it is unfortunately difficult to apply the idea in many real world tasks.

Probabilistic programming systems (PPSs) provide a powerful platform to alleviate these issues by separating model specification and inference procedure [Goodman et al., 2008a; Gordon et al., 2014]. In particular, probabilistic models are defined as program codes by the user, and inference algorithms are supplied as built-in engines which operate in an automated manner. This substantially decreases the burden from the domain experts as they can focus on developing novel models with their domain expertise, and no longer need to worry about the technical foundations or the implementations of inference algorithms. It also benefits the algorithm developers since once an inference algorithm is developed

and properly implemented in a PPS, it is then widely applicable to a broad range of models specified in the system.

Although we wish to implement a fully automated, generic pipeline as a long-term, ambitious goal, many widely-used PPSs are *restricted*, that is, they have imposed many restrictions to the models they can support. This is due to the requirements of their provided inference engines, which are usually efficient because they can exploit certain structure information of the model. However, these restrictions have excluded a large group of models that domain experts might wish to use beyond the standard setup. As a result, the constraint on the model class highly limits the wide usability of these PPSs.

To overcome this restriction, many *universal* PPSs focusing on the expressiveness of models are emerging in recent years. They are *universal* in the sense that they do not impose any restriction on the models allowed in the system. Compared to the restricted options, these PPSs largely expand the range of models that one might be interested in, but meanwhile, the flexibility also substantially complicates inference procedure. It becomes extremely difficult to ensure the performance of the inference engines; many conventional methods in the restricted PPS are no longer applicable in the universal setting.

This thesis is about developing and automating inference algorithms for non-standard models both in a probabilistic programming context and in general. For the automation of inference in a general setup, we make inroads into adaptive inference methods and introduce a novel class referred as Inference Trees [Rainforth et al., 2018]. It overcomes key pathologies of existing baselines for complicated models with multiple and separated modes, and the self-adaptive nature also enables our method to operate in a black-box manner, rather than requiring heavy manual efforts. From the aspect of automating inference using a PPS, it first involves improving the design of a PPS such that it is able to extract more information of a model to allow more sophisticated inference methods to be automated. To this end, we have proposed a novel way of formalizing the underlying modeling language of a PPS called LF-PPL [Zhou et al., 2019a]. It extends the standard setup of the class of restricted PPSs to incorporate a wider class of models as well as more advanced inference choices. Furthermore, we investigate the universal PPSs and aim to actually support more powerful, non–standard models in the sense that they can be reasoned about by built-in inference

engines. In order to do so, we have studied a class of commonly used but challenging models in universal PPSs which may have varying dimensions. We have developed a new inference scheme, Divide, Conquer and Combine [Zhou et al., 2019b], to deal with them and demonstrated how it can be implemented as a PPS engine.

1.1 Thesis Outline

The structure of the thesis is as follows. At a high level, Chapter 2, 4 and 6 provide the overview of essential related work, from the basics of probabilistic machine learning to different approaches of probabilistic programming. These chapters serve as the stepping stones to explain the challenges of automating inference for non-standard models from different aspects. Chapter 3, 5 and 7 outline our main contributions to address the problem, which include developing a statistically novel class of inference methods, optimizing the design of restricted PPSs, and proposing a new inference scheme for a typical class of models in universal PPSs.

Specifically, Chapter 2 presents the key fundamentals of probabilistic machine learning, or Bayesian machine learning to be more precise, and discusses a few problems in the scenario of non-standard models. It is followed by Chapter 3 which introduces a novel class of adaptive inference methods called Inference Trees, that are suitable for complicated models and can be implemented in a black-box manner without the requirement of much manual adjustment. Chapter 4 reviews key aspects of probabilistic programming which provides a powerful platform to automate Bayesian inference, and focuses mainly on restricted, inference-driven PPS which impose heavy restrictions on the supported models. To overcome this limitation, in Chapter 5, we propose LF-PPL, which shows how to extend these PPSs in a principled way for a broader range of models and inference engines. After that, Chapter 6 turns to universal PPSs and discusses their differences compared with the restricted counterparts. Their key challenge is that although many non–standard models can be specified in the system, they cannot be reasoned about properly due to the lack of available inference engines. Chapter 7 then provides a novel solution, Divide, Conquer and Combine, to one class of the most challenging models where the very existences of the random variables in the model can be stochastic. Part of the idea about resource allocation

builds upon Chapter 3, and we will also demonstrate how it can be included in a universal PPS to operate in an automated fashion. We finish the thesis with conclusions in Chapter 8 where we will also discuss some open remaining questions.

2

Probabilistic Machine Learning

In this chapter, we will review some key aspects of the probabilistic machine learning. This framework offers a principled way of representing the learning process as inferring plausible probabilistic models to explain observed data [Ghahramani, 2015]. We will focus mostly on the probabilistic generative approach, i.e. the *Bayesian paradigm* to be more specific, where we make explicit assumptions and encode subjective beliefs into models, and "learn" from the observed data via updating prior beliefs using Bayesian inference. We will explain how Bayesian models are specified as well as how inference is conducted in the model before we get on to a few basic inference algorithms. Although these algorithms on their own might be limited in practical settings, they are the essential building blocks for more sophisticated options. We will finish the chapter with the introduction of some more advanced and nowadays more commonly used inference techniques, and the discussion of their benefits and restrictions.

The *Bayesian paradigm* implements the learning process via updating subjective beliefs. Suppose the aim is to query about some unknown parameters or variables x [1]. People firstly encode their subjective beliefs of the latent variables as a *prior* distribution $p(x)$. Provided some observed data, we can then define the likelihood function $p(y|x)$ which describes the probability that data y are generated when the latent variables are fixed at specific values

[1] Note that we will not distinguish the terms "parameter" and "variable" unless emphasizing separately as they function the same way in this context.

x. Both quantities together form a joint model of x and y, or a *Bayesian model* in this context, using the product rule as $p(x, y) = p(x)p(y|x)$. We can then obtain the *posterior* distribution $p(x|y)$, which encapsulates our updated beliefs, by applying Bayes' rule

$$p(x|y) = \frac{p(x)p(y|x)}{p(y)}. \tag{2.1}$$

This process of obtaining $p(x|y)$ using Bayes' rule is the heart of *Bayesian inference*. The normalizing constant $p(y)$ in (2.1) is defined as

$$p(y) = \int_x p(x)p(y|x)dx, \tag{2.2}$$

and it is also called the *marginal likelihood* or the *model evidence*. Intuitively, it describes the plausibility of a model in terms of explaining the observed data with all latent variables marginalized out, and it can be used as a metric to compare and select different models. Since $p(y)$ has a constant value, one can also express the posterior in a convenient way that $p(x|y) \propto p(x)p(y|x)$. We now have a look of some key aspects of both components of this paradigm–Bayesian modeling and Bayesian inference–in detail.

2.1 Bayesian Modeling: Assumptions and Examples

As we have mentioned, a Bayesian model encodes all assumptions and hypotheses of the corresponding problem. This is achieved via defining $p(x)$ and $p(y|x)$ for each latent and observable variable, and their explicit forms contain all information about our hypotheses. Although it sounds simple and straightforward, how close a model is compared to the actual system makes a huge difference in terms of the final result. One normally needs to start with some assumptions and leverage domain knowledge, if it exists, to construct a model.

Perhaps the most essential and widely applied assumption among Bayesian modeling is the *conditional independence* (CI) property for the observable variables given the latent variables. That is, we assume that each observed datapoint does not depend on each other provided that the latent variables are fixed at specific values. In other words, the observed data are only affected by the latent variables of the model but not other observed ones. The overall likelihood function can therefore be decomposed as

$$p(y|x) = \prod_{i=1}^{N_y} p(y_i|x) \tag{2.3}$$

for a given dataset $\{y_n\}_{n=1}^{N_y}$. This assumption stems from a fundamentally important result known as *de Finetti's Theorem* [De Finetti, 1937]. Details of this theorem are beyond the scope of this thesis, but at a high-level, it shows that if our observed variables y are exchangeable and our finite sequence can be assumed to be part of an infinite sequence of random variables, then there exist some latent variables x that the observed ones y are conditionally independent with respect to. In theory, it is possible to construct a model where the model variables x encapsulate all necessary information to generate the observed values.

Independence assumptions are also essential in defining model dependency structures. A *graphical model* (GM) is a convenient way to represent a joint model with explicit dependency relationships among variables [Bishop, 2006; Murphy, 2012]. A graph representation $G = (V, A)$ consists a set of nodes or vertices, $V = \{v_1, \ldots, v_{N_V}\}$, which includes all the variables (both the latent variables x and observable ones y), and a set of edges or arcs, $A = \{(v_i, v_j)|v_i, v_j \in V\}$, which represents a connection between the node v_i and node v_j. The joint probability distribution of a graph G, $p_G(v)$ is then defined by specifying the probability distribution of each node v.

Though various GMs are available, our main focus is *directed acyclic graphical models* (DAGs) or Bayesian networks as they are sometimes known. In the context of DAGs, we can write the elements of the edge set A as $v_i \rightarrow v_j$ meaning that the graph is directed and node v_i is the parent of the node v_j. A nice property of DAGs is that we can order the vertices according to the "parent-child" relationship, and define the ordered Markov property which assumes that a node in G only depends on its immediate parents but not any upper stream ancestor node above its parents. Therefore, we have the joint probability distribution defined by a DAG as

$$p_G(V) = \prod_{v \in V} p(v|PA(v)), \tag{2.4}$$

where $PA(v)$ represent the direct parent nodes of v, and $p(v|PA(v))$ defines the probability density function or probability mass function for node v. It is convenient to specify a model in the form of DAGs, as it naturally breaks the overall problem into small modules, where one can define the distribution of each variable and its directly related ones, and combine the distributions together via factorization. For example, to define a Bayesian model with

the joint distribution of $p(x, y)$, we can specify each variable as a node in a DAG, and the prior and likelihood are constructed via defining the probability density or mass function of each node. DAGs also offer benefits in extracting the model dependencies in a generic way, which provides essential information for some inference algorithms and enables their automatic implementation. We will demonstrate this in Section 4.2.

2.1.1 Example: Gaussian with Unknown Mean

We now consider the simple, one-dimensional Gaussian with unknown mean model as an example. It can be used to estimate quantity such as the location of an object, the volume of a sound, or the speed of an aircraft, where one only has access to its noisy observation. Such a quantity is represented by a latent variable x, and before seeing any data, we might guess the x is around some value μ_0 with a small variance σ_0^2. Thus we define the prior distribution of x to be

$$p(x) = \mathcal{N}(x; \mu_0, \sigma_0^2).$$

The noisy measurement of x is denoted by y, which is observable, and we assume the observation noise is drawn from a Gaussian distribution with zero mean and a standard deviation of σ_y. We therefore define the likelihood function to be

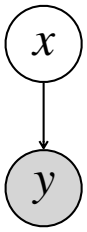
$$p(y|x) = \mathcal{N}(y; x, \sigma_y^2).$$

Putting both together, we have the joint distribution of the model as

$$p(x, y) = \mathcal{N}(x; \mu_0, \sigma_0^2)\mathcal{N}(y; x, \sigma_y^2).$$

The DAG representation of the model is shown in Figure 2.1 (left), where we shaded the observable variable y. There is one directed edge pointed from x to y, indicating that x is the "parent" of y and the distribution of y depends on x. A more common statistical notation is shown in Figure 2.1 (right).

In order to update our prior belief on x, we want to obtain the posterior distribution $p(x|y)$, which describes the probability distribution of the unknown x after observing y. Because both the prior and the likelihood are Gaussian distributed, the posterior is a *conjugate* distribution with the prior and therefore is analytically tractable. The closed



$$x \sim \mathcal{N}(\mu_0, \sigma_0^2)$$
$$y|x \sim \mathcal{N}(x, \sigma_y^2)$$

Figure 2.1: The DAG representation (left) and the statistical denotation (right) for the Gaussian with unknown mean model.

form of the posterior is as follows,

$$p(x|y) = \mathcal{N}(x; \mu', \sigma'^2), \quad \text{where } \mu' = \sigma'^2 \left(\frac{\mu_0}{\sigma_0^2} + \frac{x}{\sigma_y^2} \right), \ \sigma'^2 = \frac{1}{\left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma_y^2} \right)}.$$

However, a conjugate prior is not available in most real-world cases and one would need to carry out Bayesian inference (to be introduced in Section 2.2) to approximate $p(x|y)$.

2.1.2 Example: State Space Model

The dependency structure for the Gaussian with unknown mean example is straightforward since it only has two variables. We now consider a more advanced model, the State Space Model (SSM), where the independence relationships among variables are more complicated. At a high-level, SSMs are a popular class of probabilistic models for sequential data, and they are widely applied for tasks such as object tracking and time series forecasting.

For an SSM, the hidden state of interest is denoted by latent variable x_t whereas the noisy observation for each state is represented by y_t. Suppose we have obtained a sequence of observed data $y = y_{1:T}$. In a general set up, we have the joint density to be

$$p(x, y) = p(x_1)p(y_1|x_1) \prod_{t=2}^{T} p(x_t|x_{1:t-1})p(y_t|x_t), \tag{2.5}$$

where $p(x_t|x_{1:t-1})$ is the transition distribution and $p(y_t|x_t)$ is the observation distribution. Its DAG representation is shown in Figure 2.2. A general SSM is powerful since later states are able to leverage all the information of previous states, but it is sometimes too computationally expensive to reason about.

In a simpler set up, we assume the transition process to be Markovian, and therefore the current hidden state x_t depends only on the previous state x_{t-1} (rather than all previous states $x_{1:t-1}$), as shown in Figure 2.3. The transition distribution $p(x_t|x_{1:t-1})$ is simplified to

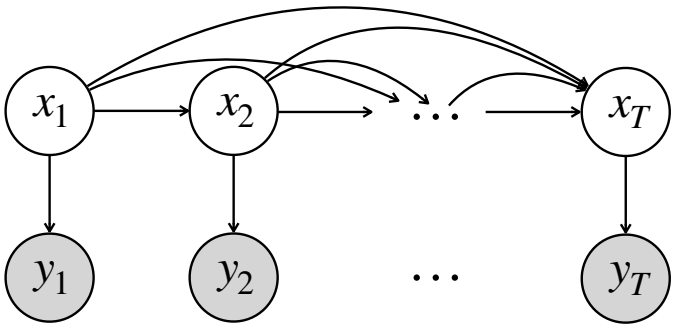


Figure 2.2: The DAG representation of a general state space model.

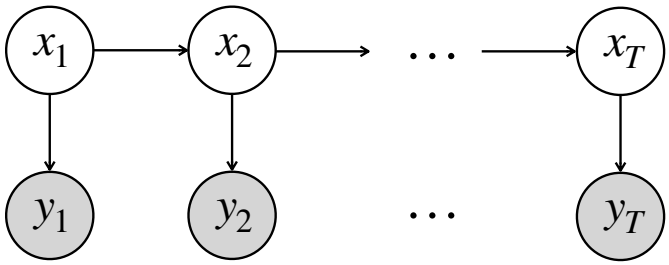


Figure 2.3: The DAG representation of a Markov state space model.

$p(x_t|x_{t-1})$, and we shall call this model *Markovian SSM*. In particular, if the hidden state x_t is normally distributed and the mean is a linear function of the previous state x_{t-1}, and each observable state y_t is normally distributed as well with the mean being a linear function of x_t, then this model is further known as a linear-Gaussian SSM (LG-SSM) or a linear dynamical system (LDS). The transition and observation distribution can be written as

$$x_t \sim \mathcal{N}(A_t x_{t-1} + B_t, Q_t) \tag{2.6}$$

$$y_t \sim \mathcal{N}(C_t x_t + D_t, R_t) \tag{2.7}$$

where A_t, B_t, C_t, D_t, Q_t, and R_t are hyper-parameters.

The quantity of interest is the posterior distribution $p(x_{1:T}|y_{1:T})$, which can be used, for example, to predict the hidden states for future time-steps. However, it is much more difficult in SSM, compared to the previous model, to calculate the posterior analytically and to obtain samples from it. Some inference algorithms have been proposed to approximate $p(x_{1:T}|y_{1:T})$ by exploiting the sequential model structure and we will see one shortly in Section 2.3.2.

2.2 Bayesian Inference and Monte Carlo Estimation

In the previous section, we have seen how one can specify a Bayesian joint model $p(x, y)$ using a prior distribution and a likelihood function. We now investigate the other main component of Bayesian machine learning–Bayesian inference–that obtains the posterior $p(x|y)$ from the joint model by applying Bayes' rule. We will outline some key challenges of such a calculation, and focus mainly on *Monte Carlo* methods which form the foundation for the specific inference algorithms that we shall study in the rest of this thesis.

The goal of Bayesian inference is to obtain either the exact form, or an approximate representation of, the posterior $p(x|y)$ as per (2.1). We shall define our *target* distribution $\pi(x)$ to be the posterior, i.e. $\pi(x) = \gamma(x)/Z$, also refer the joint density $p(x, y)$ as the *unnormalized* target density $\gamma(x)$ and Z the normalizing constant $p(y)$. An immediate challenge we encounter is that computing the closed form for π is extremely difficult. This is because (2.1) involves the calculation of complex integral for Z, which is usually intractable. Moreover, one is typically not interested in π itself, but rather using it to compute the expectation of some target function f as

$$I := \mathbb{E}_{\pi(x)}[f(x)] = \int f(x)\pi(x)\mathrm{d}x. \tag{2.8}$$

Therefore, even if one could have the exact expression of π, one might not be able to compute the integration analytically in (2.8). Can we instead obtain an approximation of π in a way that we are able to use it to calculate other quantities such as I? *Monte Carlo* [Metropolis and Ulam, 1949] provides us an important way of achieving this.

Monte Carlo (MC) methods are widely used in many mathematical and physical problems including optimization, numerical integration and inverse problems. The key underlying idea is to solve complex computation problems through *repeated random sampling*. For instance, it can be used to form a characterization of a probability distribution, such as our target $\pi(x)$ and to approximate an integral such as the marginal likelihood Z and the expectation I. Explicitly, we have the simple MC estimator $\hat{I}^{MC}$ of I as

$$I \approx \hat{I}^{MC} := \frac{1}{N}\sum_{n=1}^{N} f(\hat{x}_n), \tag{2.9}$$

where $\hat{x}_{1:N}$ are N independently and identically distributed (i.i.d.) samples drawn from the target distribution $\pi(x)$. Before we introduce specific MC methods, we first review a few key properties of MC methods.

Unbiasedness To start with, the MC estimator $\hat{I}^{MC}$ as per (2.9) is unbiased. Because $\hat{x}_{1:N}$ are i.i.d. and all follow the target distribution $\pi(x)$, we have $\mathbb{E}[f(\hat{x}_1)] = \cdots = \mathbb{E}[f(\hat{x}_N)]$. By definition, we have $\mathbb{E}[f(\hat{x}_1)] = I$. Therefore, together with linearity of the expectation, we can demonstrate that MC estimator is *unbiased* by showing that the bias of the MC estimator as per (2.9) is zero, with

$$\begin{aligned}\mathbb{E}[\hat{I}^{MC}] - I &= \mathbb{E}\left[\frac{1}{N}\sum_{n=1}^{N} f(\hat{x}_n)\right] - I = \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}[f(\hat{x}_n)] - I \\ &= \frac{1}{N}\sum_{n=1}^{N}\int f(\hat{x}_n)\pi(\hat{x}_n)\mathrm{d}\hat{x}_n - I = \int f(x)\pi(x)\mathrm{d}x - I = 0.\end{aligned} \tag{2.10}$$

This demonstrates that MC estimation does not induce any systematic error in the approximation process. Let's have a look at the following example. Suppose we want to estimate the average income of the people in the city of London. Due to the large population, it is infeasible to survey every individual. However, what we can do is to randomly sample a finite number of people (say 1000), record the income of each person, and then take the empirical average. This forms an estimator of the average income of London. Because our estimator is unbiased, the expectation of these estimates should be the true average income no matter how many samples (eg. 1000 or $10,000$) we take as long as they are i.i.d..

Variance The second important property of an estimator is the variance. Formally, the variance of our MC estimator is

$$Var[\hat{I}^{MC}] := \mathbb{E}\left[(\hat{I}^{MC} - I)^2\right] = \mathbb{E}\left[\left(\frac{1}{N}\sum_{n=1}^{N} f(\hat{x}_n) - I\right)^2\right] = \frac{\sigma^2}{N} \tag{2.11}$$

with σ^2 being the variance of the $f(x)$, i.e. $\sigma^2 = Var[f(x)] = \mathbb{E}[(f(x_1) - I)^2]$. The variance of the MC estimator goes to zero as $N \to \infty$. * In the previous average income estimation example, though sample size will not affect the expectation of estimates because of the unbiasedness, it affects the variability of estimates if you repeat the estimation multiple times: the more samples we take, the less our estimates are spread out from the

true average. Ideally, when choosing an estimator, one would prefer an estimator with lower variance, but it might not always be the case as we shall see shortly.

Consistency The last but not least property is the consistency. We say a sequence of estimators $\{\hat{t}_n; n \geq 0\}$ of a parameter θ is a consistent estimator if and only if

$$\lim_{n\to\infty} P(|\hat{t}_n - \theta| > \epsilon) = 0, \ \forall \epsilon > 0 \tag{2.12}$$

According to the weak *Law of Large Numbers* (LLN), our MC estimator $\hat{I}^{MC}$ in (2.9) converges in probability to I as per (2.8). It means that our estimator gets more and more accurate as we increase the sample size N and it becomes arbitrarily close to the true value with probability one as N goes to infinity.

If all other properties were the same, an unbiased estimator is usually preferable to a biased one. However, it might not be the case in practice because either an unbiased estimator does not exist, or is difficult to compute, or may have a much higher variance. Especially in the cases where the efficiency of an estimator is the bottleneck but its accuracy is comparatively less important, a biased but consistent estimator might be a good choice as it would still converge to the true value when $N \to \infty$.

The target distribution π in (2.9) is the posterior distribution in the Bayesian inference context, and up to this point, we have assumed that we could obtain i.i.d. samples from it upfront. However, this prerequisite does not hold in most situations; it is part of what we want to achieve in Bayesian inference. Therefore, having reviewed the fundamental properties of MC estimators, we are ready to take a closer look at a few popular MC methods in the next section and see how they obtain an approximation of π as well as the expectation I as per (2.9) in different ways.

2.3 Basic Monte Carlo Inference Methods

2.3.1 Importance Sampling

Perhaps the most basic MC method is Importance Sampling (IS) and it is a key stepping stone of many other sophisticated methods. The foremost basic IS assumes that π is available and can be evaluated pointwise (but does not need to be sampled from). It

obtains the posterior samples by sampling from a proposal distribution $q(x)$ and assigning an importance weight to each sample. The target expectation I is then estimated by the weighted average of the samples drawn from the proposal distribution. Explicitly, we can construct the MC estimator as

$$\begin{aligned} I &= \int f(x)\pi(x)\mathrm{d}x = \int f(x)\frac{\pi(x)}{q(x)}q(x)\mathrm{d}x \\ I &\approx \hat{I}^{IS} := \frac{1}{N}\sum_{n=1}^{N}\frac{\pi(\hat{x}_n)}{q(\hat{x}_n)}f(\hat{x}_n) \quad \text{where} \quad \hat{x}_n \sim q(x) \end{aligned} \tag{2.13}$$

where the *importance weight* w_n for each sample $\hat{x}_n$ is defined as $w_n := \pi(\hat{x}_n)/q(\hat{x}_n)$.

A nice property of IS is that the estimator $\hat{I}^{IS}$ is unbiased, which can be shown as,

$$\begin{aligned} \mathbb{E}\left[\hat{I}^{IS}\right] &= \mathbb{E}\left[\frac{1}{N}\sum_{n=1}^{N}\frac{\pi(\hat{x}_n)}{q(\hat{x}_n)}f(\hat{x}_n)\right] = \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}\left[\frac{\pi(\hat{x}_n)}{q(\hat{x}_n)}f(\hat{x}_n)\right] \\ &= \frac{1}{N}\sum_{n=1}^{N}\int q(x)\frac{\pi(x)}{q(x)}f(x)\mathrm{d}x = \frac{1}{N}\sum_{n=1}^{N}\int \pi(x)f(x)\mathrm{d}x = I \end{aligned} \tag{2.14}$$

Furthermore, the IS estimator is also consistent. This can be shown with the unbiasedness result along with L^2 convergence. The latter is because the mean squared error $\mathbb{E}\left[\left(\hat{I}^{IS} - I\right)^2\right] = \sigma^2/N$ where $\sigma^2 := \mathbb{E}\left[\left(\frac{\pi(\hat{x}_1)}{q(\hat{x}_1)}f(\hat{x}_1) - I\right)^2\right] = \mathrm{Var}\left[\frac{\pi(x)}{q(x)}f(x)\right]$ [Owen, 2013]. Given that the squared L^2-norm of $\hat{I}^{IS}$, $||\hat{I}^{IS} - I||_2^2$, equals to the mean squared error, we can show the L^2 convergence as

$$\lim_{N\to\infty}||\hat{I}^{IS} - I||_2 = \lim_{N\to\infty}\frac{\sigma}{\sqrt{N}} = 0$$

which implies the convergence in probability of the IS estimator.

2.3.1.1 Self-Normalized Importance Sampling

Typically, we do not have direct access to the posterior π directly due to the intractable normalizing constant Z; instead, we can only evaluate its unnormalized version $\gamma(x)$. In this context, one can still estimate I with some small changes and we call this variant of IS the Self-Normalized Importance Sampling (SNIS).

We replace $\pi(x)$ by $\gamma(x)/Z$ and rewrite I slightly as

$$I = \int f(x)\frac{\gamma(x)}{Z}\mathrm{d}x = \frac{1}{Z}\int f(x)\frac{\gamma(x)}{q(x)}q(x)\mathrm{d}x = \frac{1}{Z}\,\mathbb{E}_{\,q(x)}\left[f(x)\frac{\gamma(x)}{q(x)}\right]$$

$$\text{and} \quad Z = \int \gamma(x)dx = \int \frac{\gamma(x)}{q(x)}q(x)dx = \mathbb{E}_{\,q(x)}\left[\frac{\gamma(x)}{q(x)}\right]$$

Therefore, we can separately estimate the expectation I w.r.t. $\gamma(x)$ and the marginal likelihood Z, and use Z to *normalize* the expectation. The estimator for Z is

$$\hat{Z}^{SNIS} := \frac{1}{N}\sum_{n=1}^{N} w_n \quad \text{where} \quad w_n := \frac{\gamma(\hat{x}_n)}{q(\hat{x}_n)},\ \hat{x}_n \sim q(x) \tag{2.15}$$

and therefore we have the estimator for I being

$$\hat{I}^{SNIS} := \frac{1}{\hat{Z}^{SNIS}} \cdot \frac{1}{N}\sum_{n=1}^{N} w_n f(\hat{x}_n) = \frac{\sum_{n=1}^{N} w_n f(\hat{x}_n)}{\sum_{n=1}^{N} w_n}. \tag{2.16}$$

Note that $\hat{Z}^{SNIS}$ in (2.15) is an unbiased and consistent estimator of Z but I^{SNIS} in (2.16) is a *biased* but consistent estimator of I. This is due to the fact that though $\hat{Z}^{SNIS}$ is an unbiased estimator of Z itself, $1/\hat{Z}^{SNIS}$ is not an unbiased estimator of $1/Z$ which can be shown by Jensen's inequality.

2.3.2 Sequential Monte Carlo

Sequential Monte Carlo (SMC) [Gordon et al., 1993], or particle filtering as it is sometimes known, builds on sequential importance sampling with resampling. It is particularly effective in highly structured problems which naturally lend themselves to a particular series of targets, such as state-space models as introduced in Section 2.1.2. SMC can exploit the structure by approximating the intermediate target distributions which get incrementally closer to the full target.

The posterior distribution of such models can usually be written as,

$$p(x|y) = \frac{1}{p(y)} f_1(x_1) g_1(y_1|x_1) \prod_{t=2}^{T} f_t(x_t|x_{1:t-1}) g_t(y_t|x_{1:t}) \tag{2.17}$$

where f_t is the transition function, g_t defines the observation density or the likelihood function and T is the total dimension of the model. At each step t, SMC generates a population of weighted samples known as a particle set $(\tilde{x}_{1:t}^k, w_t^k)_{k=1}^K$, where $k = 1, \ldots, K$ is the index of particles. In doing so, SMC firstly resamples the ancestors for the next step; it samples the index of the ancestor a_{t-1}^k for each particle from a Categorical distribution $\mathcal{F}(\cdot|\bar{w}_{t-1}^1, \ldots, \bar{w}_{t-1}^K)$, where $\bar{w}_{t-1}^k = w_{t-1}^k / \sum_l w_{t-1}^l$ is the normalized weight. Then it draws random samples from a given proposal distribution $x_t \sim q_t(x_t|\tilde{x}_{1:t-1}^{a_{t-1}^k})$, and calculates the corresponding weights as

$$w_t^k = \frac{g_t(y_t|\tilde{x}_{1:t}^k) f_t(x_t^k|\tilde{x}_{1:t-1}^{a_{t-1}^k})}{q_t(x_t^k|\tilde{x}_{1:t-1}^{a_{t-1}^k})}. \tag{2.18}$$

Algorithm 2.1 Sequential Monte Carlo

Inputs: Observed data $y_{1:T}$, number of particles K, proposal distributions q_t
Outputs: Weighted particles $(\tilde{x}^k_{1:T}, w^k_T)^K_{k=1}$

for k = 1, . . . , K **do**
  $x^k_1 \sim q_1(x_1)$, Set $\tilde{x}^k_1 \leftarrow x^k_1$
  $w^k_1 = \frac{g_1(y_1|x^k_1)f(x^k_1)}{q_1(x^k_1)}$
  Obtain initial particle set $(\tilde{x}^k_1, w^k_1)^K_{k=1}$
end for
for t = 2, . . . , T **do**
  for k = 1, . . . , K **do**
    $a^k_{t-1} \sim \mathcal{F}(\cdot|\bar{w}^1_{t-1}, \ldots, \bar{w}^K_{t-1})$
    $x^k_t \sim q_t(x_t|\tilde{x}^{a^k_{t-1}}_{1:t-1})$
    Set $\tilde{x}^k_{1:t} \leftarrow (\tilde{x}^{a^k_{t-1}}_{1:t-1}, x^k_t)$
    $w^k_t = \frac{g_t(y_t|\tilde{x}^k_{1:t})f_t(x^k_t|\tilde{x}^{a^k_{t-1}}_{1:t-1})}{q_t(x^k_t|\tilde{x}^{a^k_{t-1}}_{1:t-1})}$
  end for
end for

The particle set for step t is $(\tilde{x}^k_{1:t}, w^k_t)^K_{k=1}$, where $\tilde{x}^k_{1:t} = (\tilde{x}^{a^k_{t-1}}_{1:t-1}, x^k_t)$, and it is propagated forward in a similar way for the remaining iterations. A simple summary is given in Algorithm 2.1.

Given the particle systems returned by SMC $(\tilde{x}^k_{1:T}, w^k_T)^K_{k=1}$, the expectation in Equation 2.8 can be estimated as,

$$I \approx \hat{I}^{SMC} = \frac{1}{K}\sum_{k=1}^{K} \bar{w}^k_T f(\tilde{x}^k_{1:T}) \tag{2.19}$$

One typical problem SMC struggles is handling models with long-range dependencies. For example, if x_1 in (2.17) represents a global parameter, it could influence every transition and/or emission distribution. In such scenarios, the marginal distribution given the full observations $p(x_1|y_{1:T})$, which is targeted by the full SMC sweep, is typically highly peaked and significantly different from the intermediate target $p(x_1|y_1) \propto f_1(x_1)g(y_1|x_1)$ used for sampling of x_1. Consequently, very few of these samples for x_1 will survive the numerous resampling steps carried out by SMC, giving a very poor estimate of $p(x_1|y_{1:T})$ at the end of the SMC run. We will propose one way to address this problem in Chapter 3.

2.3.3 Markov Chain Monte Carlo

All previously mentioned MC methods characterize the posterior distribution by the collection of weighted samples. We will now have a look at another class of versatile MC methods which are known as Markov chain Monte Carlo (MCMC) methods [Metropolis et al., 1953]. They approximate the posterior distribution by constructing a Markov chain which has an equilibrium distribution being the target distribution π, where one can obtain samples from this distribution by recording the values of each state.

Let $\{X_n\}_{n=0}^{\infty}$ be a homogeneous Markov chain with the initial state distribution $X_0 \sim p^{(0)}(\cdot)$ and the transition probability distribution $p(X_{n+1} = x_{n+1}|X_n = x_n)$, where the next state X_{n+1} only depends on the current state X_n due to the Markovian hypothesis. The transition probability $p(X_{n+1} = x_{n+1}|X_n = x_n)$ is also known as the *transition kernel*. We want the stationary distribution $X_n \sim p(x)$ of this chain to be our target distribution $\pi(x)$, which is true if we have

$$\pi(x'') = \int p(x''|x')\pi(x')dx'. \tag{2.20}$$

That is, if $p(X_n = x') = \pi(x')$, then $p(X_{n+1} = x'') = \pi(x'')$, where the target distribution $\pi(x)$ is stationary under repeatedly applying the transition kernel. A sufficient (but not necessary) condition for (2.20) to hold is that the chain satisfies the *detailed balance* condition, which is defined as,

$$\pi(x')p(x''|x') = \pi(x'')p(x'|x''). \tag{2.21}$$

Chains satisfying detailed balance are also known to be *reversible*.

To show that the Markov chain converges to the target distribution $\pi(x)$, beyond the property that the chain admits $\pi(x)$ as the stationary distribution, we would also require the stationary distribution to be *unique*. This is ensured by the ergodicity property, which requires the Markov chain to be *irreducible* and *aperiodic*. Intuitively, it means that no matter what the "start state" X_0 the Markov chain has, it always converges to $\pi(x)$. We refer the reader to Norris [1998] for a detailed proof of the convergence theory and present one of the most widely-used methods, Metropolis-Hasting [Hastings, 1970] algorithm in the following.

Metropolis-Hastings The Metropolis-Hastings (MH) algorithm draws samples from the target distribution $\pi(x)$, provided that $\pi(x)$ or its unnormalized version $\gamma(x)$ can be evaluated pointwisely. Choosing an arbitrary starting state x_0, it works by repeatedly proposing a new sample x' from a proposal distribution and accepting or rejecting this new sample according to some probability. More explicitly, at iteration n, we firstly propose a new sample x' from a proposal distribution $x' \sim q(x'|x_n)$, and then accept this new sample with probability

$$\mathcal{A}(x', x_n) = \min\left(1, \frac{\pi(x')q(x_n|x')}{\pi(x_n)q(x'|x_n)}.\right) \tag{2.22}$$

If x' is accepted, we set the new state $x_{n+1} = x'$. Otherwise, we reject the new sample and then copy the value from the old state as $x_{n+1} = x_n$. The details of the algorithm can be seen in Algorithm 2.2.

As we can see from (2.22), when only $\gamma(x) = Z\pi(x)$ is available, we can replace $\pi(x)$ with $\gamma(x)$ in both the numerator and the denominator, as the normalizing constant Z cancels out. The transition probability can be written as $p(x'|x_n) = q(x'|x_n)\mathcal{A}(x', x_n)$. We can show that the Markov chain we construct following the MH algorithm converges to the target distribution since the transition distribution satisfies the detailed balance condition as

$$\begin{aligned}\pi(x_n)p(x'|x_n) &= \pi(x_n)q(x'|x_n)\mathcal{A}(x', x_n) = \pi(x_n)q(x'|x_n)\min\left(1, \frac{\pi(x')q(x_n|x')}{\pi(x_n)q(x'|x_n)}\right) \\ &= \pi(x')q(x_n|x')\min\left(1, \frac{\pi(x_n)q(x'|x_n)}{\pi(x')q(x_n|x')}\right) = \pi(x')q(x_n|x')\mathcal{A}(x_n, x) = \pi(x')p(x_n|x').\end{aligned} \tag{2.23}$$

Though it is flexible to choose the form of the proposal distribution q, what we choose will substantially affect the performance. For instance, it is viable to have $q(x'|x_n) = q(x')$, where q is completely independent from the current state when proposing a new sample. But it is not suggested in most cases as this proposal stops the information from being passed to the next iteration along the chain. Alternatively, one, in general, would prefer to undertake a local movement making use of the current state. The intuition is that, if x_n is a good sample, we hope x' that is close to x_n might also be good samples. This enables the MH algorithm (as well as many other MCMC methods) to exhibit a hill-climbing behavior, which is usually desirable when we explore the target space.

Algorithm 2.2 Metropolis-Hastings Algorithm

Inputs: target distribution $\pi(x)$, proposal distributions $q(x)$
Outputs: accepted samples $x_{1:N}$

Initialize $\mathbf{x}_0$
for n = 0, . . . , N-1 **do**
    Sample $x' \sim q(\cdot|x_n)$
    Compute acceptance rate $\mathcal{A}(x', x_n)$ as per (2.22)
    Sample $u \sim \text{Uniform}(0, 1)$
    if $u < \mathcal{A}(x', x_n)$ **then**
        Accept x' and set $x_{n+1} = x'$
    else
        Reject x' and set $x_{n+1} = x_n$
    end if
end for

2.4 Inference Methods beyond the Basic Setup

Thus far, we have introduced some key underpinnings of Monte Carlo estimation and reviewed three basic Monte Carlo methods for Bayesian inference. These methods might seem elementary on their own, but they are especially important as they are the building blocks for more sophisticated techniques. In this section, we will cover a few more advanced inference methods beyond the basic setup: adaptive sampling methods in Section 2.4.1 are mainly around one central question in Monte Carlo – how to construct a better proposal distribution to achieve better efficiency in a self-adaptive way; Hamiltonian Monte Carlo in Section 2.4.2 shows how people have leveraged ideas from physics to improve the scalability of MCMC methods; we finish this section by introducing variational inference (Section 2.4.3), which works differently from previously discussed MC methods, and also composes an important branch of Bayesian inference literature. These advanced inference algorithms are particularly important for complex models where the basic options fail.

2.4.1 Adaptive Importance Sampling Methods

A critical question of MC methods is how to construct a good proposal distribution $q(x)$. It is in general difficult to come up with such q from scratch, so one possible way is to adapt the proposal automatically as part of the inference algorithm. Though there are a range of different options of what to undertake exactly when adapting $q(x)$, most methods share a common framework: they alternate between sampling using the current

proposal, and adapting the proposal using existing samples for the future, with the latter often taking the form of a (potentially implicit) density estimation. This idea is applied within both an IS based scheme and an MCMC sampler. For the latter, a common choice is to design the transition probability distribution such that the current state depends on all or multiple previous states, rather than only the last one, and therefore (multiple) previous states can be used to adapt the parameters of the transition kernel. We will not go further into adaptive MCMC directions but want to note that it is easy to cause problems in the convergence and ergodicity of the algorithm, and so they require careful design. For the rest of this section, we will focus on the adaptive IS approaches, the shortcoming of which motivates our contribution in Chapter 3.

Generally speaking, AIS is about iteratively updating one or multiple proposal probability densities such that they approximate the posterior better. Ideally, we want the proposal to have adequately high probability wherever the target density is high. A general framework proposed by [Bugallo et al., 2017] contains three core steps: sampling from proposal distribution(s), weighting all samples, and adapting the parameters of the proposal(s). For clarity, we distinguish a few *weighting strategies* commonly used in IS and AIS when presented with multiple proposals. Instead of using a single proposal distribution q to aid approximating the target distribution π as introduced in Section 2.3.1, one can also employ a collection of proposals $\{q_k\}_{k=1}^{K_q}$; this is called the multiple IS (MIS) method. Suppose we draw one sample from each proposal, ie. $\hat{x}_k \sim q_k$ for $k = 1 : K_q$. We have three options when calculating the corresponding weight of each sample. The simplest choice is called standard MIS (s-MIS) where each sample is weighted using its own proposal only, as

$$w_k^{s-MIS} := \pi(\hat{x}_k)/q_k(\hat{x}_k). \tag{2.24}$$

Deterministic mixture MIS (DM-MIS), as a more advanced variation, averages each weight among all proposals as

$$w_k^{DM-MIS} := \frac{\pi(\hat{x}_k)}{\frac{1}{K_q}\sum_{i=1}^{K_q} q_i(\hat{x}_k)}. \tag{2.25}$$

Though it tends to have better performance, it requires K_q times more evaluation of each proposal per sample. A trade-off between these two is called Partial DM-MIS, where one

only chooses a subset, $L_q \leq K_q$, of the total proposals randomly to average over:

$$w_k^{P-DM-MIS} := \frac{\pi(\hat{x}_k)}{\frac{1}{L_q}\sum_{i=1}^{L_q} q_i(\hat{x}_k)}. \tag{2.26}$$

One can classify AIS methods according to different features of each algorithm, such as the number of the proposal distributions being used, what parameters to adapt in each proposal (e.g. the location and/or the scale of a proposal), different weighting schemes (e.g. standard weighting using one sample or normalizing the importance weights among a mixture of proposals), and various adaptation strategies [Bugallo et al., 2017]; we will have a look at the last one in detail.

The first common adaptation strategy is *resampling*. In this category, different AIS methods usually resample the drawn samples at last iteration and use the updated set to adapt the parameters of the proposal distributions for the next iteration. Popular AIS methods sharing this feature are population Monte Carlo (PMC) [Cappé et al., 2004] and its variants. In particular, PMC maintains a population of K_q proposal distributions and updates the proposal distribution via resampling. At each iteration, it generates one sample from each proposal, calculates each importance weight independently as in (2.15), and resamples the set of samples where the new set becomes the parameters that are used to construct the K_q proposals for the next iteration. The key concept behind PMC is that it exploits the good samples so far and uses this information to adapt the proposals. However, PMC can be unstable due to the simple weighting scheme, that is, each sample is weighted by its own proposal only. Mixture-PMC (M-PMC) [Cappé et al., 2008] improves PMC by using a mixture of proposals and adapting the parameters as well as the weights of proposals by minimizing the KL-divergence between the target and the mixture of proposals. Deterministic mixture weighting PMC (DM-PMC) [Elvira et al., 2017] extends PMC by increasing the number of samples drawn per proposal and employing the DM-MIS weighting scheme before resampling. Both methods have improved the performance of the standard PMC significantly.

Another popular adaptation scheme is via *moment matching*. One updates the proposal distribution via estimating the first few moments (usually the mean and variance) of the target distribution. Take the Adaptive MIS (AMIS) [Cornuet et al., 2012] as an example. At

each iteration, it updates the location and scale parameters of the proposal distribution using the empirical mean and covariance of all previously drawn, weighted samples. Alternatively, the recently developed Adaptive population IS (APIS) [Martino et al., 2015] uses a mixture of proposals per iteration with DM weighting. Unlike in DM-PMC where one resamples the last set of drawn samples, APIS updates the location parameter of the proposal by its empirical mean. It inherits the stability of AMIS but also maintains computational tractability as the number of iterations increases.

The last strategy of adaptation is to use independent adaptation processes. In contrast to all methods above which use previously drawn samples to adapt future proposals, methods falling in this category establish an independent process to update the proposal. For example, Gradient APIS (GAPIS) [Elvira et al., 2015] uses the gradient information of the target distribution to update the location parameters and the Hessian matrix of $-\log \pi$ to update the scale parameters. Compared to vanilla APIS, GAPIS performs better in high-dimensional space with the aid of the gradient information. Layered AIS (LAIS) [Martino et al., 2017], as another example, separates the proposal adaptation and target estimation into two layers: it constructs the proposal distribution via one or more MCMC chains, and generates importance samples from the proposal to approximate the target distribution. The proposal adaptation happens fully within the MCMC process, which is independent from previously drawn importance samples. LAIS is in particular relevant to this thesis due to its simple and generic design and its good performance, especially when multimodality is present: it can be starting point to incorporate adaptive methods into existing or new probabilistic programming systems.

2.4.2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC), also known as Hybrid Monte Carlo, [Duane et al., 1987] is an instance of the MH algorithm that is able to perform inference effectively in high dimensional spaces. This ability results from the physical process that governs the dynamics of the algorithm [Neal, 2011; Brooks et al., 2011; Girolami and Calderhead, 2011; Betancourt, 2017]: the fundamental Hamiltonian dynamics constrains HMC, but also enables HMC to propose distant proposals with high acceptance probabilities.

2.4.2.1 Hamiltonian dynamics and the use in MCMC

The Hamiltonian of a physical system is defined in terms of the set of points (x, ρ), where x is the position and ρ is the momentum variable. The total energy of the system is given by the Hamiltonian

$$H(x, \rho) = K(\rho) + U(x) \tag{2.27}$$

where $U(x)$ is the potential energy and $K(\rho)$ represents the kinetic energy, and we have the equations of the motion being

$$\frac{dx}{dt} = \frac{\partial H}{\partial \rho}, \quad \frac{d\rho}{dt} = -\frac{\partial H}{\partial x}. \tag{2.28}$$

Hamiltonian dynamics can be visualized as a frictionless puck sliding over a surface with various heights [Neal, 2011]. The state of the system consists of the position and momentum of the puck respectively, given by x and ρ. The potential energy of the puck $U(x)$ is proportional to the height of the surface, and the kinetic energy $K(\rho)$ is equal to $|\rho|^2/2M$ where the M is the mass of the puck. Both the potential energy and the kinetic energy together construct the total energy of the system.

For a physical system with the energy function $E(x)$, we can define a canonical distribution over states to have the probability density function

$$P(x) = \frac{1}{C} \exp(-\frac{E(x)}{T}) \tag{2.29}$$

where T is the temperature of the system and C is the normalizing constant. We can substitute $E(x)$ with the Hamiltonian $H(x, \rho)$ and obtain a joint probability distribution of both x and ρ as

$$P_H(x, \rho) = \frac{1}{C} \exp(-\frac{H(x, \rho)}{T}) = \frac{1}{C} \exp(-\frac{U(x)}{T}) \exp(-\frac{K(\rho)}{T}). \tag{2.30}$$

As we can see x and ρ are independent and each have canonical distributions with energy functions being $U(x)$ and $K(\rho)$.

Remember our goal is to obtain samples from the target distribution $\pi(x)$. If we write π in the form of canonical distribution ($T = 1$), we have $U(x)$ being its energy function, i.e. $\pi(x) = \gamma(x)/Z = \exp(-U(x))/Z$, and $U(x) := -\log \gamma(x)$. The momentum ρ, though not directly relevant to our goal, is key here to simulate the Hamiltonian dynamics. It also has a canonical distribution (setting $T = 1$) with the energy function being $K(\rho) = \rho^T M^{-1} \rho/2$,

where M is a symmetric, positive-definite "mass matrix". Because of the similar form as the probability density function of Multivariate Gaussian distribution with zero mean, M covariance matrix, ρ here is also called Gaussian momentum [2]. The HMC sampler generates both states, x and ρ, from the joint probability distribution $P_H(x, \rho)$, where x is the latent variable of interest, and ρ is an auxiliary variable. According to Hamiltonian Equations, x and ρ evolve following the dynamics as

$$\begin{aligned} \frac{dx}{dt} &= \frac{\partial H}{\partial \rho} = \frac{\partial K}{\partial \rho} = M^{-1}\rho \\ \frac{d\rho}{dt} &= -\frac{\partial H}{\partial x} = -\frac{\partial U}{\partial x}. \end{aligned} \tag{2.31}$$

2.4.2.2 The Leapfrog Integrator

We approximate Hamilton's equations (2.31) by discretizing time for computer implementation, using some small time stepsize, ϵ. For the following discussion, we will assume the Hamiltonian maintains the form in (2.31) and the kinetic energy would have the simple form $K(\rho) = \rho^T M^{-1}\rho/2$, where M is diagonal with diagonal elements $m_1, \ldots, m_d$ and d as the dimension. The Hamiltonian can be written as

$$H(x, \rho) = K(\rho) + U(x) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i} - \log \gamma(x), \tag{2.32}$$

and the Leapfrog integrator solves the Hamiltonian Equation as follows:

$$\rho_i(t + \frac{\epsilon}{2}) = \rho_i(t) - \frac{\epsilon}{2} \cdot \frac{\partial U}{\partial x_i}(x(t)) \tag{2.33}$$

$$x_i(t + \epsilon) = x_i(t) + \epsilon \cdot \frac{\rho_i(t + \epsilon/2)}{m_i} \tag{2.34}$$

$$\rho_i(t + \epsilon) = \rho_i(t + \frac{\epsilon}{2}) - \frac{\epsilon}{2} \cdot \frac{\partial U}{\partial x_i}(x(t + \epsilon)). \tag{2.35}$$

At each iteration, the position and momentum variables (x, ρ) are updated by the leapfrog method for a trajectory of length L, and then the updated pair is put through a Metropolis accept/reject step. The new position variable either accepts the updated one or remains unchanged, which will become the starting point of the position variable in the next iteration. The auxiliary momentum variable will be discarded at the end of each iteration and will be re-generated at the start of next one. Details of HMC can be seen in Algorithm 2.3.

[2] In theory, K could take any form but empirically non-Gaussian momentums seem to have poor performance, especially in high dimensional space. [Betancourt, 2017]

Algorithm 2.3 Hamiltonian Monte Carlo

Inputs Number of iterations N, step size ϵ, trajectory length L, unnormalized density γ, the Hamiltonian H.

Outputs $x_{1:N}$

Initialize x_0
for $n = 0, \ldots, N-1$ **do**
  $x^{(0)} = x_i$
  $\rho^{(0)} \sim \mathcal{N}(\mathbf{0}, I)$
  for $\ell = 1, \ldots, L$ **do**
    $\rho^{(\ell-1)} = \rho^{(\ell-1)} + \frac{\epsilon}{2} \nabla_{x^{(\ell-1)}} \log \gamma(x^{(\ell-1)})$
    $x^{(\ell)} = x^{(\ell-1)} + \epsilon\, \rho^{(\ell-1)}$
    $\rho^{(\ell)} = \rho^{(\ell-1)} + \frac{\epsilon}{2} \nabla_{x^{(\ell)}} \log \gamma(x^{(\ell)})$
  end for

  $u \sim U(0, 1)$
  $\mathcal{A} = \min\{1, \alpha\}$, where $\alpha = \exp(H(x^{(L)}, \rho^{(L)}) - H(x^{(0)}, \rho^{(0)}))$
  if $u < \mathcal{A}$ **then**
    $x_{n+1} = x^{(L)}$
  else
    $x_{n+1} = x^{(i)}$
  end if
end for

2.4.2.3 Extending HMC to complex models

However, standard HMC with leapfrog integrator is restricted since one needs to calculate the derivatives of the joint density $\gamma(x)$ w.r.t. the variables of interest, i.e. the position variables x. This results in the fact that only the models with a finite number of continuous parameters fall in this scope. It is highly restricted as it is natural to have generative models with discrete random variables or branching statements, where the target density is not necessarily fully differentiable.

In order to extend HMC to a larger class of probabilistic models, a number of approaches have been explored. Zhang et al. [2012] uses continuous relaxations to embed discrete parameters into a multivariate joint normal; Pakman and Paninski [2014] solves Hamilton's equation of motions exactly for a piecewise Gaussian distribution, although in practice this is typically challenging to do; Afshar and Domke [2015] performs standard HMC with boundary detection between non-differentiable regions and energy changes at the boundaries; Nishimura et al. [2017] takes a different route and designed a coordinate-wise

integrator using both Gaussian and Laplace momentums depending on the type of each variable that in theory enables one to sample from non-differentiable parameters; they call this discontinuous HMC (DHMC). We will demonstrate the feasibility of extending and automating HMC-based inference methods to a broader class of models by incorporating its variants later in this thesis (see Chapter 5).

2.4.3 Variational Inference

Both particle based methods and MCMC methods that we have introduced earlier in this chapter approximate the posterior distribution via sampling. As an alternative, variational methods achieve this goal via *optimization*, and they compose another important family of Bayesian inference methods. For completeness, we briefly review how it works to finish this Chapter.

As introduced in Section 2.2, we are interested in the posterior distribution $p(x|y)$ of the latent variables x, which can further be used to calculate the expectation of a test function $f(x)$ w.r.t. the posterior, i.e. $\mathbb{E}_{p(x|y)}[f(x)]$. At a high-level, variational inference (VI) approximates the posterior distribution by specifying a family of variational distributions $q_\phi(x) := q(x;\phi)$ with parameters ϕ, and finding the optimal candidate $q^*(x)$ that matches the posterior the best w.r.t. some divergence metric $\mathcal{D}$ by optimizing the parameters, i.e.

$$q^*(x) = \arg\min_{\phi^*} \mathcal{D}\left(q_\phi(x), p(x|y)\right). \tag{2.36}$$

We call q the variational distribution or variational approximation.

A common choice for the divergence $\mathcal{D}$ is the Kullback-Leibler (KL) divergence which is defined as

$$\mathcal{D}_{KL}\left(q_\phi(x)||p(x|y)\right) = \int q_\phi(x) \log \frac{q_\phi(x)}{p(x|y)} dx = \mathbb{E}_{q_\phi(x)}\left[\log \frac{q_\phi(x)}{p(x|y)}\right]. \tag{2.37}$$

It measures the similarity between q and p, and it is asymmetric, i.e. $\mathcal{D}_{KL}(q||p) \neq \mathcal{D}_{KL}(p||q)$ unless they are all zero meaning that q and p are identical. One can also choose the forward KL, i.e. $\mathcal{D}_{KL}(p||q)$, but we choose this way round so that we can directly construct a Monte Carlo estimate by drawing from the reference distribution of the expectation. Note that (2.37) is usually difficult to compute exactly because it involves the evaluation of the

typically intractable posterior $p(x|y)$. However, we can rewrite (2.37) as

$$\begin{aligned}\mathcal{D}_{KL}\left(q_\phi(x)||p(x|y)\right) &= \int q_\phi(x)\log\frac{q_\phi(x)}{p(x|y)}dx = \int q_\phi(x)\log\frac{q_\phi(x)p(y)}{p(x,y)}dx \\ &= \int q_\phi(x)\log p(y)dx + \int q_\phi(x)\log\frac{q_\phi(x)}{p(x,y)}dx \\ &= \log p(y) + \int q_\phi(x)\log\frac{q_\phi(x)}{p(x,y)}dx \;\geq 0.\end{aligned}$$

Since the marginal likelihood $p(y)$ is a constant, we can instead minimize the other term on the right, which is equivalent to minimizing $\mathcal{D}_{KL}\left(q_\phi(x)||p(x|y)\right)$.

We define the evidence lower bound (ELBO) of the log marginal likelihood as

$$\begin{aligned}\log p(y) &\geq -\int q_\phi(x)\log\frac{q_\phi(x)}{p(x,y)}dx \\ &= \mathbb{E}_{q_\phi}[\log p(x,y)] - \mathbb{E}_{q_\phi}[\log q_\phi(x)] \;=: \text{ELBO}(\phi).\end{aligned} \tag{2.38}$$

Therefore, minimizing $\mathcal{D}_{KL}\left(q_\phi(x)||p(x|y)\right)$ can be achieved equivalently via maximizing the lower bound. Note that the ELBO is sometimes used to evaluate how well the model fits the data or model selection. The ELBO equals to $\log p(y)$ only when q_ϕ is identical to $p(x|y)$ such that the KL divergence between the variational distribution and the posterior becomes zero. With the ELBO in (2.38), we can then specify the family of the variational distribution q and the optimization algorithm to maximize the ELBO and to find the optimal q^* that approximate the posterior $p(x|y)$ the best within the family.

A common choice of the variational family q has been a *mean field*, fully factorized distribution; that is, for $x = \{x_{1:D}\}$,

$$q(x;\phi) = \prod_{d=1}^{D} q_d(x_d;\phi_d), \tag{2.39}$$

where we assume the latent variables are independent and each factor q_d has its own parameters ϕ_d. Substituting the factorized q in the ELBO as per (2.38), we can use a coordinate ascent optimization technique to maximize the ELBO and update the variational parameters ϕ until reaching convergence (local maximum). On the one hand, the mean field assumption makes the approximation tractable and eases the calculation. Especially for models of the exponential family, the optimal q^* can be computed analytically. On the other hand, the assumption of independence among latent variables also highly restricts

the distribution it can characterize; only a small number of models can be approximated via the decomposition used in the mean field VI.

2.4.3.1 Black box variational inference

For a broader range of models where the conditionally conjugate prior is not available, the ELBO contains an intractable integration which might not be analytical calculable. In such cases, one no longer has the closed form of the optimal variational distribution q^*. Black box variational inference (BBVI) [Ranganath et al., 2014] as an alternative, provides a more generic way to achieve this. At its core, BBVI specifies the gradient of the optimizing target, ELBO, as an expectation, and carries out MC methods to estimate such quantity.

Specifically, the gradient of the ELBO in (2.38) can be re-written in the form of an expectation w.r.t. q_ϕ as

$$\begin{aligned}\nabla_\phi \text{ELBO}(\phi) &= \nabla_\phi \int q_\phi(x) \log \frac{p(x,y)}{q_\phi(x)} dx = \int \nabla_\phi\, q_\phi(x) \log \frac{p(x,y)}{q_\phi(x)} dx \\ &= \int (\nabla_\phi\, q_\phi(x)) \log \frac{p(x,y)}{q_\phi(x)} dx + \int q_\phi(x) \nabla_\phi \cancelto{0}{\left(\log \frac{p(x,y)}{q_\phi(x)} \right)} dx \\ &= \int q_\phi(x) \left(\nabla_\phi \log q_\phi(x)\right) \log \frac{p(x,y)}{q_\phi(x)} dx \;\; = \mathbb{E}_{q_\phi(x)} \left[\Big(\nabla_\phi \log q_\phi(x)\Big) \log \frac{p(x,y)}{q_\phi(x)} \right].\end{aligned} \tag{2.40}$$

Therefore, it is straightforward to estimate $\nabla_\phi \text{ELBO}(\phi)$ using standard MC methods as

$$\begin{aligned}\nabla_\phi \text{ELBO}(\phi) \approx \hat{\mathcal{G}}_{REINF} &= \frac{1}{N} \sum_{n=1}^{N} \zeta(\hat{x}_n), \\ \text{where } \hat{x}_n \sim q_\phi(x),\; \zeta(x) &:= \Big(\nabla_\phi \log q_\phi(x)\Big) \log \frac{p(x,y)}{q_\phi(x)}\end{aligned} \tag{2.41}$$

and the MC estimator $\hat{\mathcal{G}}_{REINF}$ is also known as *REINFORCE* estimator, score function estimator or likelihood ratio estimator. As one can see from (2.41), the REINFORCE estimator only requires the unnormalized density $p(x, y)$ to be available pointwise and also the gradient of the variational distribution q, some quantity chosen by us. We can show that $\hat{\mathcal{G}}_{REINF}$ is an unbiased estimator of the ELBO but unfortunately it usually has too high variance to be useful in most scenarios. Some techniques including control variates [Ranganath et al., 2014] have been proposed to alleviate this issue for certain cases.

In contrast, another popular technique to obtain the gradient of the ELBO is the *reparameterization trick* [Kingma and Welling, 2014]. This works by reparameterizing the

random variable $x \sim q(x)$ by a deterministic function over a random source, i.e. $x = g_\phi(\varepsilon)$, $\varepsilon \sim s(\cdot)$. For example, if $q_\phi(x) = \mathcal{N}(\mu, \sigma^2)$, we could have $s(\varepsilon)$ being $\mathcal{N}(0, 1)$ and map ε to x via the function $g_\phi(\varepsilon) = \mu + \sigma\varepsilon$. We then derive the gradient of the ELBO as

$$\begin{aligned} \nabla_\phi \,\text{ELBO}(\phi) &= \nabla_\phi \,\mathbb{E}_{\,q_\phi(x)} \left[\log \frac{p(x,y)}{q_\phi(x)}\right] = \nabla_\phi \,\mathbb{E}_{\,q(\varepsilon)} \left[\log \frac{p(g_\phi(\varepsilon),y)}{q_\phi(g_\phi(\varepsilon))}\right] \\ &= \mathbb{E}_{\,q(\varepsilon)} \Big[\nabla_\phi \log \frac{p(g_\phi(\varepsilon),y)}{q_\phi(g_\phi(\varepsilon))}\Big] \end{aligned} \tag{2.42}$$

which can be estimated by a MC estimator,

$$\nabla_\phi \text{ELBO}(\phi) \approx \hat{\mathcal{G}}_{repa} = \frac{1}{N}\sum_{n=1}^{N} \nabla_\phi \log \frac{p(g_\phi(\hat{\varepsilon}_n),y)}{q_\phi(g_\phi(\hat{\varepsilon}_n))}, \quad \text{where } \hat{\varepsilon}_n \sim s(\varepsilon). \tag{2.43}$$

The reparameterization gradient is unbiased and generally has lower variance than the REINFORCE gradient. However, the reparameterization gradient estimator requires both the unnormalized density $p(x, y)$ and the variational proposal $q_\phi(x)$ to be differentiable w.r.t. x, as well as $q_\phi(x)$ differentiable w.r.t. ϕ. This greatly restricts the class of the models for which the reparameterization gradient is applicable, especially for models with discrete latent variables.

3
Inference Trees

As we have pointed out in Section 2.4.1, the choice of proposal distribution is a key factor in the performance of Monte Carlo (MC) methods. Unfortunately, it is typically difficult to know what constitutes a good proposal prior to performing inference. For this reason, many methods use past samples to adapt the proposal at future iterations [Cappé et al., 2004; Cornebise et al., 2008; Cornuet et al., 2012; Gu et al., 2015; Liang et al., 2011], for example by minimizing the KL divergence between the empirical distribution over samples and the proposal. These strategies implicitly assume that preceding samples are representative of the true posterior. This leads to the somewhat undesirable characteristic that we already need good samples to have effective adaptation, which is presumably difficult to achieve given our need to adapt in the first place. Adaptive methods can consequently exhibit pathologies, such as collapsing to a single mode or even adapting to invalid proposals [Andrieu and Thoms, 2008; Cappé et al., 2008].

To address these issues, we propose that adaptive methods should not only carry out *exploitation*, that is sample in regions where we believe the posterior mass is high, but also *exploration*, that is explicitly invest computational resources to sample in regions where our current uncertainty about the posterior mass is high. In other words, we should recognize that the utility derived from the generated samples originates not only from their direct contribution to the estimator, but also the degree to which they inform future sampling.

We introduce inference trees (ITs) [Rainforth et al., 2018], a new class of adaptive methods that build on ideas from Monte Carlo tree search (MCTS) [Browne et al., 2012; Kocsis and Szepesvári, 2006]. ITs hierarchically partition the parameter space into disjoint regions in an online manner, resulting in more fine-grained partitions for regions where the posterior density is large. This transforms the problem of inference on the full parameter space to a set of constrained inference problems, which we can combine in a manner akin to stratified sampling [Carpentier et al., 2015; Neufeld et al., 2014]. By adaptively choosing regions in which to refine our estimates, we can explicitly control the exploration-exploitation trade-off. This results in an algorithm that can expend computational resources to investigate whether the proposal can be improved, for example by searching for missing modes, rather than just greedily exploiting the best proposal learned so far. We emphasize that ITs are explicitly *not* an adaptive importance sampling method; breaking the standard independent sampling framework is critical to their performance. For example, they can force sampling in under-explored areas and quickly exploit new modes when they are found.

ITs can be thought of as a meta-algorithm that controls the allocation of computational resources of a base inference algorithm. We show that, under mild assumptions, ITs define a consistent estimator whenever the base algorithm itself provides a consistent estimator. This property is independent of the methods for learning the partitioning and allocation of computational resources between the partitions. In addition to the theoretical guarantees this provides, the resulting flexibility proves critical to the empirical performance of ITs. For example, we exploit this flexibility to introduce a novel allocation scheme that uses *targeted* exploration. Namely, rather than just relying on an optimism boost [Auer et al., 2002] to ensure a minimum level of exploration for all regions, ITs use explicit uncertainty estimates for the true marginal posterior masses of regions to identify important areas to explore, such as those likely to contain a missing mode. Underlying this approach is a novel estimator in its own right. This estimator performs density estimation on the sample weights to predict the probability of the true marginal posterior mass of a region is above a certain threshold. Remarkably, it can remain robust even if the MC estimate of the marginal is thousands of orders of magnitude smaller than the truth.

We find that the gains that ITs provide are particularly pronounced when they are combined with sequential Monte Carlo (SMC) [Doucet et al., 2001]. Here they offer a means of capturing long-range dependencies, that is variables having influence many steps after they are sampled. They are able to do this because they provide a means for samples to be "forced through" the resampling steps, thereby permitting performance improvements beyond what can be achieved by the so-called (one-step) optimal SMC proposal.

3.1 Background and Related Work

Recall that our aim is to approximate a target density $\pi(x) = \gamma(x)/Z$, for which it is possible to evaluate the unnormalized density $\gamma(x)$ pointwise, but computation of the normalization constant Z is intractable. We will assume that we have a base MC algorithm that returns weighted samples and makes use of some form of proposal distribution $q(x)$. Though we will later consider other approaches (see §3.8.2), for exposition, it will be easiest to think of this base algorithm as being self-normalized importance sampling [Owen, 2013], which defines an estimated measure based on weighted samples from q,

$$\hat{\pi}(\cdot) := \sum_{n=1}^{N} \bar{w}_n \delta_{\hat{x}^n}(\cdot) \quad \text{where } \hat{x}^n \sim q(x),\ \bar{w}_n := \frac{w_n}{\sum_{n=1}^{N} w_n},\ \text{and } w_n := \frac{\gamma(\hat{x}^n)}{q(\hat{x}^n)}. \tag{3.1}$$

We now discuss some of the key underlying themes motivating ITs.

3.1.1 Adaptive Monte Carlo Inference

The performance of MC inference methods is, in general, critically dependent on the proposal $q(x)$. Consequently many methods look to adapt $q(x)$, using information from previous samples to improve the performance at future iterations. The key pathologies of these strategies stem from the fact that they are *greedy*, using the best estimated proposal at each iteration and maintaining no notion that these might be poor. One of the key motivations for ITs is to introduce an explicit *exploration* component to adaptation, recognizing that the utility from sampling stems not only from the direct contribution of those samples to the estimator, but also the information they provide for future adaptation.

Though the need for exploration has previously had little consideration in adaptive importance sampling contexts, it has been considered more extensively in MCMC settings [Swendsen and Wang, 1986; Bornn et al., 2013]. Here the need for exploration

is much more explicit, due to the tendency of MCMC samplers to become stuck in a particular mode of the target density for long periods of time, which can thus easily lead to modes being missed. A number of different methods try to alleviate this by targeting an adjusted density where transitions between modes can be made more easily. Two common themes for this are parallel tempering algorithms [Swendsen and Wang, 1986; Geyer, 1991; Miasojedow et al., 2013] and methods based around the Wang-Landau algorithm [Wang and Landau, 2001; Bornn et al., 2013].

The latter of these shares some interesting similarities to ITs, because they also impose a partitioning of the target space (typically imposed indirectly from partitions of the values of the log-density) and then looks to control the amount of samples drawn from each. In particular, [Bornn et al., 2013] also adaptively adjust the partitioning used. However, there is still a multitude of differences between the two approaches. To give a few examples: ITs are not an MCMC algorithm and return weighted samples, ITs can be used in conjunction with various different base algorithms like SMC, asymptotically ITs produces samples from the true target distribution rather than a biased approximation, ITs impose a hierarchical (rather than flat) partitioning and learn this in a very distinct way, and ITs use uncertain estimates to target their exploration, rather than relying solely on conventional Monte Carlo estimates.

3.1.2 Multi-Armed Bandits

In multi-armed bandit problems, an agent sequentially chooses between multiple actions, known as arms, each of which returns a stochastic reward. The agent's goal is to maximize the long-term cumulative reward [Agrawal and Goyal, 2012; Berry and Fristedt, 1985]. One common strategy is upper confidence bounding (UCB) [Auer et al., 2002], which chooses the arm j that maximizes the utility

$$u_j = \hat{r}_j + \frac{\beta \log \sum_i M_i}{\sqrt{M_j}}. \tag{3.2}$$

In this definition, $\hat{r}_j \in [0, 1]$ is the current estimate of the expected reward for each arm, M_j is the number of times arm j was previously pulled, and β is a parameter that controls the level of exploration. Here $\hat{r}_j$ is an *exploitation* term that ensures we pull arms with high expected reward more frequently, while $(\beta \log \sum_i M_i)/\sqrt{M_j}$ is an *exploration*

term, sometimes known as an optimism boost, which encourages us to pull arms which have been pulled infrequently so far.

Of particular relevance to our work is the study of bandits in the stratified sampling setting [Carpentier and Munos, 2011; Carpentier et al., 2015; Etoré and Jourdain, 2010; Etore et al., 2011; Kawai, 2010; Neufeld, 2016]. Here one splits a target integral into a number of strata, then looks to minimize the overall error by allocating samples to the MC estimators associated with each strata. The focus is typically on the MC integration setting, where the optimal strategy can be shown to sample each strata in proportion to the standard deviation of its evaluations [Carpentier and Munos, 2011]. Because one now needs to asymptotically sample from each arm infinitely often, rather than just identify the best arm, the strategy is adjusted to

$$u_j = \frac{1}{M_j}\left(\hat{r}_j + \frac{\beta \log \sum_i M_i}{\sqrt{M_j}}\right) \tag{3.3}$$

where $\hat{r}_j$ is typically set to the empirical standard deviation.

Auer et al. [2002] and Carpentier and Munos [2011] respectively showed that (3.2) and (3.3) have a *cumulative regret*, that is the shortfall in the cumulative reward compared to an optimal oracle, with impressive asymptotic behavior. However, these analyses require strong assumptions on the distribution of the reward $\hat{r}_j$, namely that it does not have heavier-than-Gaussian tails (see e.g. Bubeck et al. [2012] for more details). This assumption will be comprehensively violated for most practical inference problems,[1] which can in turn lead to very poor practical performance as demonstrated in our empirical results. To alleviate this, we will introduce a targeted exploration component to (3.3) that allows for more careful consideration of where to allocate resources.

3.1.3 Monte Carlo Tree Search

MCTS [Kocsis and Szepesvári, 2006; Browne et al., 2012] uses a hierarchy of arms where one traverses the tree by sequentially choosing child nodes using (3.2) (or a variation thereof) until a leaf is reached. Once a leaf is reached, it is then *refined*, either by directly evaluating

[1] Strictly speaking, this assumption typically actually holds on the technicality that $\hat{r}_j$ will usually have a finite maximum possible value. However, the distribution is usually extremely heavily tailed away from this value, such that the associated constant factors in the analysis explode and, for all practical purposes, the assumption is catastrophically violated.

it or splitting it to form new child nodes and evaluating those. The updated estimates are *propagated* up through the tree in order to improve future traversal strategies, with the average reward of a non-leaf node comprising of the average of its children, while M_j is taken as the number of times a node has been traversed.

MCTS has traditionally been used for planning [Silver et al., 2016] and in discrete decision settings. While there has previously been some limited application of MCTS in continuous settings [Van den Broeck and Driessens, 2011] and for MAP estimation [Tolpin and Wood, 2015], we believe that our work is the first to consider MCTS in the context of inference or integration, as opposed to optimization. ITs also vary from the standard MCTS setting in how rewards are calculated and propagated; rewards can be evaluated at any node in the tree and do not require a full roll out. Thus our reward estimate at any node is a combination of a local estimate and estimates from its children.

3.2 Algorithm Overview

ITs hierarchically partition the target space, run inference separately on the resulting disjoint regions to obtain local estimates, and then combine these into one overall estimate. Each node in the tree corresponds to a region of target space, A_j, such that the region of a parent node is the union of its children, $A_j = A_{\ell_j} \cup A_{r_j}$ where ℓ_j and r_j are the two children indices, and the union of all leaf nodes is the full space. Note that only binary split are undertaken throughout the paper. We further assume that we are able to sample from the proposal restricted to a node, $q(x|x \in A_j)$, and evaluate this re-normalized truncated density pointwisely. How this is achieved is discussed in §3.4. The IT learning process can now be broken down into three components as follows (see also Figure 3.1).

Traversal: The traversal step adaptively allocates computational resources to areas of the target space, balancing exploration and exploitation to minimize the error of our final overall estimate. Following similar lines to MCTS, it starts at the root node and then recursively choosing a child node until a leaf is reached. To choose between children, we use the stratified sampling UCB formulation given in (3.3). However, as we explain in detail in §3.5, our reward $\hat{r}_j$ will vary from standard settings: ours must be adapted to reflect the fact

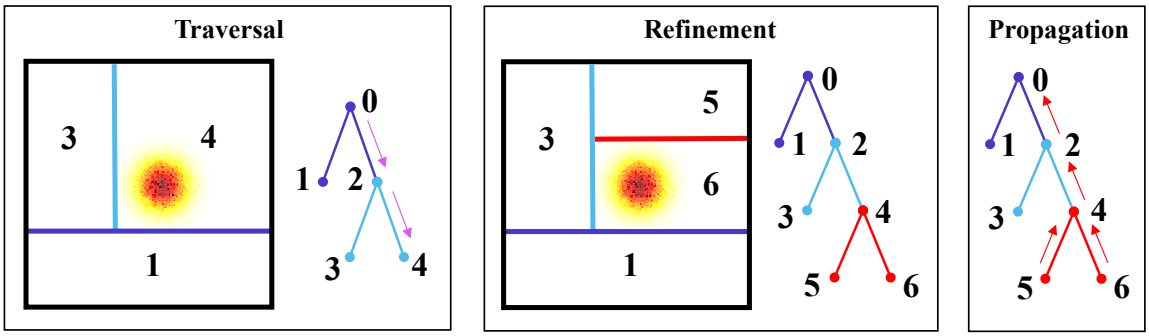


Figure 3.1: Characterization of key algorithm steps of ITs. See text for details.

we are doing inference and rather than relying solely on the optimism boost for exploration; we will incorporate a targeted exploration term to enhance more principled exploration.

Refinement: In the refinement step we improve the estimate at the chosen node, either by running inference directly and updating the local estimate, or expanding the tree by splitting the node and running inference at each of the generated child nodes. For both cases, the inference itself is performed using the base algorithm and the truncated proposal $q\left(x \mid x \in A_j\right)$ which ensures the generated samples are restricted to the target region. The two considerations for refinement are whether to split and how to split. They are discussed in §3.6.

Propagation: In the propagation step, we recursively update the tree with the new estimates produced by the refinement step, starting with the refined node(s) and then updating all their ancestors. This improves our posterior representation and guides the future traversal strategy. Along with a small number of additional terms required for the traversal, two key quantities are propagated up through the tree: a marginal likelihood estimate $\hat{Z}_j$ and an unnormalized empirical measure $\hat{\gamma}_j(\cdot)$. The truncated posterior approximation at any node in the tree is then given by the self-normalized estimated measure $\hat{\pi}_j(\cdot) = \hat{\gamma}_j(\cdot)/Z_j$, with the root note estimate $\hat{\pi}_0(\cdot)$ representing our overall approximation. The specifics of the propagation are discussed in §3.3.

Putting these components together leads to an adaptive online inference algorithm as summarized in Algorithm 3.1. We now discuss the individual elements of ITs in more detail. We note that, while the method for propagation is tightly coupled with the IT estimator itself, the consistency of this estimator is independent of the traversal and refinement strategies.

Algorithm 3.1 Inference Tree Training

Inputs: Unnormalized target density $\gamma(x)$, "truncatable" proposal $q(x)$, base inference algorithm $\mathcal{F}$, complete target space A_0, number of iterations to run T, batch size b, existing tree $\mathcal{T}$ (optional)

Outputs: Inference Tree $\mathcal{T}$, posterior estimate $\hat{\pi}_0(\cdot)$

Initialize tree $\mathcal{T}$ by running inference on full space $\{\hat{x}_0^n, w_0^n\}_{n=1}^b \leftarrow \mathcal{F}\left(\gamma(x), q\left(x|A_0\right), b\right)$
for $t = 1 : T$ **do**
  Traverse tree by recursively selecting child with highest u_j (3.12) until a leaf (j) is reached
  if decide to split node j **then** ▷ See §3.6
    Use existing samples to split node $A_{\ell_j}, A_{r_j} \leftarrow A_j$ where $A_j = A_{\ell_j} \cup A_{r_j}$ ▷ §3.4, §3.6
    $\{\hat{x}_{\ell_j}^n, w_{\ell_j}^n\}_{n=1}^b \leftarrow \mathcal{F}\left(\gamma(x), q\left(x|A_{\ell_j}\right), b\right)$, $\{\hat{x}_{r_j}^n, w_{r_j}^n\}_{n=1}^b \leftarrow \mathcal{F}\left(\gamma(x), q\left(x|A_{r_j}\right), b\right)$
  else Run further inference on current node
    $\left\{\hat{x}_j^n, w_j^n\right\}_{n=N_j+1}^{N_j+b} \leftarrow \mathcal{F}\left(\gamma(x), q\left(x|A_j\right), b\right)$
  end if
  Update $\hat{\gamma}_j(\cdot)$ and $\hat{Z}_j$ for refined node(s) and all ancestors using (3.5a) and (3.5c)
end for
Return tree $\mathcal{T}$ and empirical measure $\hat{\pi}_0(\cdot) \leftarrow \hat{\gamma}_0(\cdot)/\hat{Z}_0$

Consequently, a wide range of possible approaches fall under the general IT framework we have just introduced. To this end, we see ITs as a general approach to performing inference with multiple changeable components, rather than a very specific algorithm. We thus envisage a lot of opportunities for adaptation or refinement, within the general IT framework.

3.3 The Inference Tree Estimator

In this section we will explain how, for a given tree, inference can be run on separate partitions and the results can be combined to form the consistent inference tree estimator. This is effectively equivalent to the propagation component of ITs introduced in the previous section, for which we just perform online updates of the estimators introduced here. To demonstrate the IT estimator, we first consider combining estimates from disjoint regions. as this is what we be required for combining estimates from child nodes.

3.3.1 Combining Estimates from Disjoint Partitions

Assume we are trying to estimate the expectation of a measurable function $f(x)$ with respect to the target measure $\pi(x)$. For any set of disjoint regions $\{A_i\}_{i\in\mathcal{I}}$ covering the full target space, then, noting $\sum_{i\in\mathcal{I}} \mathbb{1}(x \in A_i) = 1\,\forall x$, we have

$$\mathbb{E}_{\pi(x)}[f(x)] = \frac{\int \gamma(x) f(x)\, dx}{\int \gamma(x)\, dx} = \frac{\int \gamma(x) f(x) \sum_{i\in\mathcal{I}} \mathbb{1}(x \in A_i)\, dx}{\int \gamma(x) \sum_{i\in\mathcal{I}} \mathbb{1}(x \in A_i)\, dx} = \frac{\sum_{i\in\mathcal{I}} \mathbb{E}_{q(x|A_i)}\left[\frac{\gamma(x)f(x)}{q(x|A_i)}\right]}{\sum_{i\in\mathcal{I}} \mathbb{E}_{q(x|A_i)}\left[\frac{\gamma(x)}{q(x|A_i)}\right]}.$$

Approximating both the numerator and the denominator with a Monte Carlo estimator now gives

$$\mathbb{E}_{\pi(x)}[f(x)] \approx \frac{\sum_{i\in\mathcal{I}} \frac{1}{N_i} \sum_{n=1}^{N_i} w_i^n f(\hat{x}_i^n)}{\sum_{i\in\mathcal{I}} \frac{1}{N_i} \sum_{n=1}^{N_i} w_i^n} \text{ where } \hat{x}_i^n \sim q(x|A_i) \text{ and } w_i^n := \frac{\gamma(\hat{x}_i^n)}{q(\hat{x}_i^n|A_i)}. \tag{3.4}$$

We thus see that we can calculate estimates separately for each region A_i and then combine these in an unweighted manner – there are no correction factors for the strategy used to assign computational resources. However, we emphasize that there are two key reasons that we are able to do this. Firstly, rather than locally self-normalizing, we are separately combining unnormalized target estimates and an estimate for the normalization constant, and then *globally self-normalizing* the estimate. Secondly, the truncated proposals $q(x|A_i)$ are *correctly normalized* such that $\int_{x\in A_i} q(x|A_i)dx = 1$.

In practice, we often do not know $f(x)$ at inference time. However, we can always compute empirical measures based on weighted samples $\frac{1}{N_i} \sum_{n=1}^{N_i} w_i^n \delta_{\hat{x}_i^n}(\cdot)$, which can then later be used to evaluate any target function as and when required.

3.3.2 Propagation of Estimates

Though the leaves of an inference tree form a suitable disjoint partitioning of the target space, we also have access to local estimates from non-leaf nodes, left over from when those nodes were previously leaves themselves. The IT estimator is therefore constructed recursively, such that the estimate at any node is a combination of its child estimates and this local estimate; the propagation step of the algorithm corresponds to online updates of these estimates. To combine estimates from parents with the children, we introduce a preference factor to the estimator from the child nodes, $c_j \in [0, 1]$, and define the IT estimator for a non-leaf node j recursively using

$$\hat{\pi}_j(\cdot) := \frac{\hat{\gamma}_j(\cdot)}{\hat{Z}_j}, \quad \text{where} \tag{3.5a}$$

$$\hat{\gamma}_j(\cdot) := \frac{(1-c_j)}{N_j} \sum_{n=1}^{N_j} w_j^n \delta_{\hat{x}_j^n}(\cdot) + c_j \left(\hat{\gamma}_{l_j}(\cdot) + \hat{\gamma}_{r_j}(\cdot)\right), \tag{3.5b}$$

$$\hat{Z}_j := \frac{(1-c_j)}{N_j} \sum_{n=1}^{N_j} w_j^n + c_j \left(\hat{Z}_{l_j} + \hat{Z}_{r_j}\right), \tag{3.5c}$$

ℓ_j and r_j refer to the child node indices, and our overall estimate is given by that of the root node $\hat{\pi}_0(\cdot)$. For the leaf nodes, we only have the local estimate (i.e. $c_j = 0$).

The child preference factors c_j represent a relative weight given to the estimate from the child nodes in our combined estimator. In the absence of other information, it would thus be natural to set $c_j = \frac{M_j - N_j}{M_j}$ where M_j is the total number of traversals (including running inference at the parent) and N_j is the number of times inference has been run at the parent node, such that the estimates are weighted in proportion to the number of component samples. However, we also expect the *per-sample efficiency* of the child estimate to be better than the parent because of the adaptation provided by the inference tree. Therefore, we want to give more preference to the child estimates. To encode this, we set c_j as

$$c_j = \frac{\lambda^{(\mathbb{E}[d_{\text{ch}}] - d_j)}(M_j - N_j)}{N_j + \lambda^{(\mathbb{E}[d_{\text{ch}}] - d_j)}(M_j - N_j)} \tag{3.6}$$

where d_j is the depth of node j in the tree and $\mathbb{E}[d_{\text{ch}}]$ is the average depth of nodes in the child subtrees. $\lambda \in [1, \infty)$ is preference parameters and can be roughly interpreted as how many times more efficient we expect the d_{j+1}-th layer of the tree to be than the d_j-th layer. We use $\lambda = 1.2$ as a default in our experiments.

We note a critical property that $c_j \to 1$ as $M_j \to \infty$ for a fixed N_j. This ensures that any estimates with only finite number of samples do not effect the overall estimate in the limit of inference tree iterations. This is discussed in more depth in §3.7.

3.3.3 Consistency

As shown in §3.7, then regardless of the partitioning, traversal and refinement strategies, subject to some very mild assumptions, each $\hat{\pi}_j(\cdot)$ as defined by (3.5) converges weakly to $\pi(x|x \in A_j)$ and, in particular, $\hat{\pi}_0(\cdot)$ converges weakly to $\pi(x)$. Inevitably, however, these strategies will affect the practical performance, as will be the focus for much of the rest of the paper.

Despite this strong consistency result, it is important to note that, for a finite number of iterations, the estimates produced by ITs are biased, even when the base estimation scheme is not. This is because the N_j are random variables that are correlated with the estimate such that $\mathbb{E}[\hat{Z}_j \mid N_j = n_1] \neq \mathbb{E}[\hat{Z}_j \mid N_j = n_2]$ for $n_1 \neq n_2$ in general. Intuitively, if a node is initially underestimated, the adaptation strategy will allocate fewer additional samples and the underestimation persists, inducing a bias. Overestimates, on the other hand, are readily corrected, leading to a negative bias for most allocation strategies.

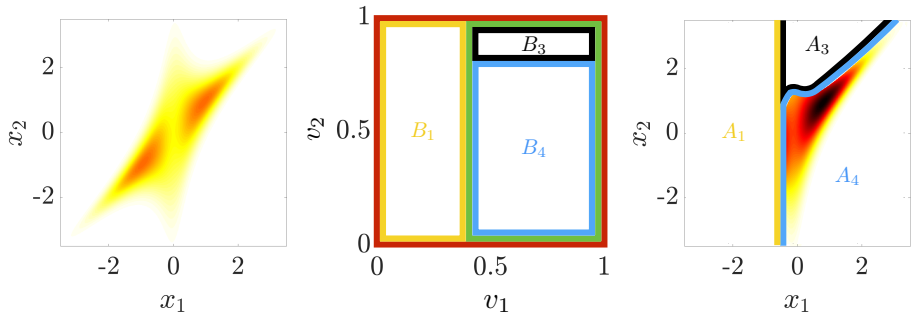


Figure 3.2: Truncation of a proposal $q(x_{1:2})$. Numbering left to right, [1] shows the original proposal and [2] the hierarchical partitioning of $v_{1:2}$ imposed by the tree. [3] shows the partitioning implied by $q(x_{1:2})$ and the leaf nodes on the target space $x_{1:2}$, where we note that the partition between A_3 and A_4 is nonlinear. It further shows the proposal truncated to A_4 and renormalized.

3.4 Partitioning the Target Space

Directly partitioning in the space of x can be challenging. Typically it will not be desirable for the partitions to be axis-aligned or even linear. Conversely, it is in general not possible to evaluate the partitioned proposal $q(x|A_j)$ for an arbitrary A_j. To address this, ITs use a reparameterization of the *proposal*, such that $x = g(v_{1:D})$, where $v_{1:D}$ is uniformly distributed on the unit hypercube $[0,1]^D$, and we choose partitions which are axis-aligned in the space of $v_{1:D}$. The partitions can thus be expressed as

$$B_j := \zeta_1^j \times \zeta_2^j \times \cdots \times \zeta_D^j \tag{3.7}$$

where each $\zeta_d^j \subseteq [0,1]$ is a partition for the corresponding dimension of v_d. This then in turn defines (typically nonlinear) partitions on x, namely we have

$$A_j := \{g(v_{1:D}) : v_{1:D} \in B_j\} = \{g(v_{1:D}) : v_1 \in \zeta_1^j \cap v_2 \in \zeta_2^j \cap \cdots \cap v_D \in \zeta_D^j\}. \tag{3.8}$$

A graphical representation for this process is shown in Figure 3.2.

In general, g can be thought of as the inverse cumulative distribution function of q. Namely, if we presume that x is also D dimensional and our proposal factorizes as

$$q(x_{1:D}) = q(x_1)q(x_2|x_1)\ldots q(x_D|x_{1:D-1})$$

then v_d is defined by the series of cumulative distribution mappings

$$v_d := \eta_d(x_d; x_{1:d-1}) = \int_{-\infty}^{x_d} q_d(x'_d|x_{1:d-1})dx'_d, \tag{3.9}$$

which in turn implicitly defines $g = \eta^{-1}$. As we are free to choose the form of the proposal, we can always ensure that g can be calculated. In some scenarios, it might even be helpful to define $q(x)$ implicitly through g.

The benefits of this partitioning approach is threefold. Firstly, splitting in the space of v can be axis-aligned because the distribution is uniform over $v_{1:D}$. This forms an equivalent partition in the space of x eliminating the problem of trying to choose splits that align well with the contours of q. Secondly, it means that we can easily evaluate the truncated proposal for x on a partition A_j via v. Because the distribution $\rho(\cdot)$ over $v_{1:D}$ is a uniform hypercube, the probability of generating an x whose pre-image is in B_j is just the hypervolume of B_j (which is in turn given by the product of the lengths of ζ_d^j). We have the properly normalized $q(x|A_j)$ as

$$q(x|A_j) = \frac{q(x)\mathbb{1}(x \in A_j)}{\int q(x)\mathbb{1}(x \in A_j)dx} = \frac{q(x)\mathbb{1}(x \in A_j)}{\int \rho(v)\mathbb{1}(v \in B_j)dx} = \frac{1}{\|B_j\|}q(x) \tag{3.10}$$

Last but not the least, to draw from the truncated proposal $q(x|x \in A_j)$, we first sample $\hat{v}_{d,j}^n \sim \textsc{Uniform}(\zeta_d^j)$ for $d = 1 : D$ and then set $\hat{x}_j^n = g(\hat{v}_{1:D,j}^n)$. before evaluating the corresponding weights as

$$w_j^n := \frac{\pi(\hat{x}_j^n)}{q(\hat{x}_j^n|\hat{x}_j^n \in A_j)} = \frac{\pi(\hat{x}_j^n)}{q(\hat{x}_j^n|\hat{v}_{1:D,j}^n \in B_j)} = \frac{\pi(\hat{x}_j^n)}{q(\hat{x}_j^n)}\|B_j\| \tag{3.11}$$

where $\|B_j\|$ is just the (known) area of B_j.

An important point of note is that the mapping function g has to be surjective (but not necessarily bijective). In other words, one may choose a different form of g other than the inverse CDF of q as we have seen earlier, in which case x may have a different dimension compared to v; it is perfectly permissible for g to map multiple different vs to one x.[2] This is necessary, for example, when x is discrete. In this scenario, the A_j may no longer be disjoint,[3] but here we can instead rely on the law of the unconscious statistician: we can think in terms of performing inference on $v_{1:D}$ (for which the partitions are disjoint) and then taking the pushforward distribution that induces on x. Note that this

[2]Intuitively, one can view v as raw uniform random variables, and x as variables of interest which are defined as the pushforward of v.

[3]From a practical perspective, we postulate that it may sometimes be preferable to not perform the reparameterization for discrete variables and instead directly split these in the space of x.

does not require any algorithmic changes, but it does require special consideration in the theoretical justification as discussed in Appendix A.1.

We finish our discussion on partitioning the target space by noting that it should, in principle, be possible to also adapt proposals within individual regions, in addition to the adaptation already provided by inference trees. This can be done by sampling $v_{1:D}|B_j$ from a non-uniform distribution which is learned adaptively, and adjusting (3.11) accordingly.

3.5 Traversal Strategy

As explained in §3.2, the traversal strategy starts at the root node and then recursively chooses the child node with the higher utility u_j until a leaf node is reached. Though we will use a utility of the UCB form given in (3.3), our reward estimate $\hat{r}_j$ will reflect both the need for exploitation and exploration, unlike in standard approaches where it represents only exploitation.

We start by quoting our final choice for the utility, before explaining each of the component terms in detail. Using $\text{pa}(j)$ and $\text{si}(j)$ to denote the parent and sister of node j respectively, we have

$$u_j = \frac{1}{M_j}\left(\underbrace{(1-\delta)\left(\frac{\hat{\tau}_j}{\hat{\tau}_{\text{pa}(j)}}\right)^{(1-\alpha)}}_{\substack{\text{Exploitation}\\ \text{Target}}} + \underbrace{\delta\frac{\hat{p}^s_j}{\hat{p}^s_j + \hat{p}^s_{\text{si}(j)}}}_{\substack{\text{Exploration}\\ \text{Target}}} + \underbrace{\beta\frac{\|B_j\|}{\|B_{\text{pa}(j)}\|}\frac{\log M_{\text{pa}(j)}}{\sqrt{M_j}}}_{\substack{\text{Optimism}\\ \text{Boost}}}\right) \tag{3.12}$$

where each term is defined as follows

- M_j is the number of times the node has been traversed.
- $\delta \in [0, 1]$ is a parameter that controls the relative emphasis on exploitation and exploration. We will typically reduce δ over time to encourage more exploitation.
- $\hat{\tau}_j$ is an estimate for the exploitation target, τ_j, derived in §3.5.1. Intuitively, τ_j shows how large the posterior mass node j may contain, and we desire to draw samples in proportion to τ_j asymptotically. The reason that $\hat{\tau}_j$ is normalized by $\hat{\tau}_{\text{pa}(j)}$ is so that the exploitation target is kept (roughly) in the region $[0, 1]$ (as is typical for UCB methods), thereby ensuring the scaling of the three terms matches.

- $\alpha \in [0, 1]$ is an annealing parameter that encourages sampling of the tails. It will reduced during the course of the IT training.

- $\hat{p}_j^s$ is a subjective probability estimate for the node containing significant posterior mass that is calculated using a density estimation of the log sample weights. It will be derived in §3.5.2. The different normalizations for $\hat{\tau}_j$ and $\hat{p}_j^s$ originate from the fact that we want the exploration term to dominate whenever $\hat{\tau}_{\text{pa}(j)} \gg \hat{\tau}_j + \hat{\tau}_{\text{si}(j)}$, implying that the children have underestimated the exploitation target compared to the parent.

- β is a parameter controlling the level of optimism boost, as per standard UCB approaches.

- $\|B_j\|/\|B_{\text{pa}(j)}\|$ is the ratio of the volume of the child node to its parent in the reparameterized space. This term ensures that the optimism boost is larger for larger nodes.

- $(\log M_{\text{pa}(j)})/\sqrt{M_j}$ is a standard UCB term for basic exploration. It ensures a minimum level of sampling from each child that increases logarithmically with the number of times the parent node has been traversed.

3.5.1 Exploitation Target

To derive our exploitation target, τ_j, we ask the question: what is the asymptotically optimal rate for allocating samples to regions? In other words, if all our node estimates were perfect, how should we allocate our samples? One might intuitively expect that the answer to this would be to allocate samples in proportion to the marginal probability mass of a region. However, it turns out that this is not the case: the variance on the weights is different for different regions and so we also need to sample more from regions where this variance is high. In fact, as we show in Appendix A.4, the optimal allocation strategy is to sample according to

$$\tau_j = \sqrt{Z_j^2 + (1 + \kappa)\sigma_j^2} \tag{3.13}$$

where σ_j^2 is the variance of the weights (as produced by single traversal) and Z_j is the marginal posterior mass of the region as before. Here $\kappa \in [0, \infty]$ is a "smoothness" parameter, which dictates the relative importance of the two terms when using the generated samples

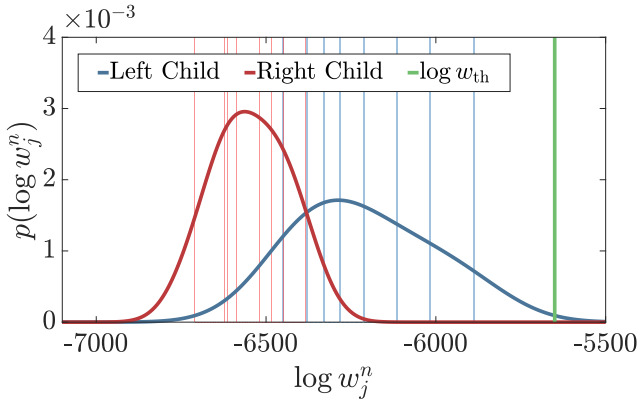


Figure 3.3: Density estimation for log weights.

to estimate a particular expectation as per (3.4). For example, $\kappa \to \infty$ corresponds to the optimal setup for estimating the marginal likelihood, for which $f(x) = 1$ is completely flat.

To estimate τ_j, we use $\hat{\tau}_j = \sqrt{\hat{Z}_j^2 + (1+\kappa)\hat{\sigma}_j^2}$ with propagated estimates $\hat{Z}_j$ and $\hat{\sigma}_j^2$. The former is given by (3.5c), while the latter requires a distinct propagation scheme as discussed in Appendix A.3.

3.5.2 Exploration Target

Relying only on the optimism boost for exploration, as done by standard UCB schemes, can be chronically inefficient in practice as it only encourages a uniform exploration. We, therefore, introduce a exploration term into our utility, $\hat{p}_j^s$, which provides a subjective probability estimate for the event that the region contains significant posterior mass that we have thus far missed.

Providing such a reliable estimate is a challenging problem. Our global proposal $q(x)$ is often very poor meaning standard MC estimates can be woefully inadequate: we will consider experiments where we regularly underestimate the marginal likelihood (ML) by factors in excess of 10^{1000}.

Our insight is that, even when the ML is substantially underestimated, the raw log weights still convey useful information about what the true value *could* be. We exploit this insight by carrying out *density estimation of the log weights* and using this as a basis for constructing $\hat{p}_j^s$. Consider the demonstrative example shown in Figure 3.3 where we want to predict whether the true log ML of each child is above some threshold $\log w_{\text{th}}$. Here we see that there is a high chance that the left child has a true ML above the threshold, but we can be

reasonably confident the right does not. Critically, we can make this assertion even though our MC estimates for the ML are underestimated by hundreds of orders of magnitude.

To formalize this intuition, let $\psi(\log w_j^n)$ denote a density estimator for a nodes local weights, with associated cumulative density $\Psi(\log w_j^n)$. The key idea is to use this density estimator to predict the probability that one more samples will exceed a target threshold $\log w_{\text{th}}$ if we were to generate another T "lookahead" samples, where T is some large, but finite, number. When $\log w_j^n$ varies over a large range, the MC estimate for the ML is effectively equal to the maximum weight, and so we have

$$P\left(\hat{Z}_j(T) > w_{\text{th}}\right) \approx P\left(\max(w_j^{1:T}) > w_{\text{th}}\right) \approx 1 - \left(\Psi(\log w_{\text{th}})^T\right)$$

where $\hat{Z}_j(T)$ is MC estimate for the ML after taking T samples. Though we could now use this estimate to construct $\hat{p}_j^s$ directly, ie. $\hat{p}_j^s := 1 - (\Psi(\log w_{\text{th}})^T)$, we further apply a heuristic of scaling by the effective sample size (ESS) [Owen, 2013] of the node (see Appendix A.3) on the basis that a high ESS suggests that we have already a reasonable ML estimate and thus do not need to explore further.

To complete the picture, we define the propagation strategy for these probability estimates by assuming that the $\hat{p}_j^s$ are independent for sibling nodes, finally yielding the recursive definition [4]

$$\hat{p}_j^s := (1 - c_j)\,\frac{1 - \left(\Psi(\log w_{\text{th}})^T\right)}{\text{ESS}_j} + c_j\left(\hat{p}_{\ell_j}^s + \hat{p}_{r_j}^s - \hat{p}_{\ell_j}^s \hat{p}_{r_j}^s\right) \tag{3.14}$$

analogous to that of $\hat{Z}_j$ in (3.5c). In our experiments, we found $\log w_j^n$ was typically well approximated by a Gaussian (there is also theoretical evidence this is appropriate when SMC is used as the base algorithm [Bérard et al., 2014; Doucet et al., 2015; Pitt et al., 2012]) and so this simple choice was taken for ψ. In cases where this gives a poor fit, one could instead use a kernel density estimator. Setting T and w_{th} is detailed in Appendix A.5.2.

3.6 Refinement Strategy

Once a leaf is chosen by the traversal, there are two ways we can refine the tree: update the local estimate or split the node. The two considerations here are whether to split and how to split.

[4] In practice, we also use some additional heuristics, giving a slightly different estimator. See Appendix A.5.1.

At a high-level, a good partitioning structure is one in which the posterior mass is concentrated in a small number of regions. In essence, we gain most from being able to "eliminate regions" from consideration, reducing the proportion of the target space that needs to be actively considered. When we propose to split a node, we thus want to find the split that best concentrates the posterior mass. Conveniently, we can use the samples already generated at the node to try and predict what will be a good split. Namely, we can hypothesize a number of splits and then evaluate how well each split will concentrate the mass, based on the existing samples. Though we do not directly use them in this way, ITs indirectly parameterize an importance sampling proposal, whereby we traverse the tree, recursively sampling a child with probability proportional to $\hat{\tau}_j$. We can, therefore, measure the concentration of mass through the entropy of this implied proposal.

Recall from §3.4 that ITs use axis-aligned partitions in the reparameterized space $v_{1:D}$ and that our proposal for a leaf node is uniform in this space. We can therefore analytically calculate the entropy of a hypothetical split (see Appendix A.6) and use this as loss criterion for choosing a split:

$$\textsc{Loss}(\text{split}) = \hat{Z}_\ell \log \frac{\|B_\ell\|}{\hat{Z}_\ell} + \hat{Z}_r \log \frac{\|B_r\|}{\hat{Z}_r} \tag{3.15}$$

where the child volumes and marginal probability estimates are implied for any hypothetical split. The lower this loss, the more information our split conveys about where the posterior mass is concentrated. As hypothetical splits can be quickly tested – there is no need to run inference – we can efficiently test out a relatively large number (~ 100) of random splits and then choose the one that minimizes (3.15). We then initialize the newly generated nodes by running inference separately on each of them.

We further introduce heuristics for whether to split in order to avoid unnecessary over-splitting. Firstly, we only attempt to split once N_j reaches a certain threshold and if the ratio ESS_j/N_j falls below a certain threshold: we want to stop splitting once a node represents a near perfect sampler. Secondly, whenever we split a node, we check that split passes a usefulness test, namely a significance test that the distributions of the $\log w_j$ are different, rejecting the split if this test fails.

3.7 Theoretical Justification

In this section, we demonstrate the correctness of the IT algorithm. We first demonstrate that for any partitioning $\{A_i\}_{i\in\mathcal{I}}$ and set of consistent estimators for each partition, then the combination strategies given in §3.3 similarly lead to consistent estimators. Moreover, we demonstrate that this convergence holds when we combine multiple sets of estimators, each with their own partitioning, for example the parent estimator and children estimator in (3.5). At a high-level we make three assumptions: each constituent estimator is consistent in isolation, each set of estimators only has finite combination weight in the limit of large overall computational budget if each of constituent region estimators receives a finite proportion of that overall computational budget, and the number of each regions is finite for each estimator set. For exposition, we will, for now, assume that the A_i are disjoint (in Assumption 1), but we show in Appendix A.1 how that this assumption can be relaxed to any proposal constructed from the form given in Section 3.4.

Assumption 1. *Let $\mathbb{X}$ denote the support of x. For every independent estimator set $\ell \in \{1,\ldots,L\}$, we are given a) a disjoint partitioning $\{A_{\ell,i}\}_{i\in\mathcal{I}_\ell}$ of the $\mathbb{X}$ such that $A_{\ell,i}\cap A_{\ell,j} = \emptyset$ for $i \neq j$ and $\bigcup_{i\in\mathcal{I}_\ell} A_{\ell,i} = \mathbb{X}$, and b) a family $\{\hat{\gamma}_{\ell,i}^{N_{\ell,i}}\}_{i\in\mathcal{I}_\ell}$ of estimated measures on $\mathbb{X}$*

$$\hat{\gamma}_{\ell,i}^{N_{\ell,i}}(\cdot) := \frac{1}{N_{\ell,i}}\sum_{n=1}^{N_{\ell,i}} w_{\ell,i}^n\,\delta_{\hat{x}_{\ell,i}^n}(\cdot)$$

for some random variables $w_{\ell,i}^n$ and $\hat{x}_{\ell,i}^n$ such that each $\hat{\gamma}_{\ell,i}^{N_{\ell,i}}(\cdot)$ converges weakly to the following measure on $\mathbb{X}$ as $N_{\ell,i}\to\infty$

$$\gamma(x)\mathbb{1}(x\in A_{\ell,i}).$$

Further each marginal probability estimate converges in probability as follows

$$\hat{Z}_{\ell,i}^{N_{\ell,i}} := \frac{1}{N_{\ell,i}}\sum_{n=1}^{N_{\ell,i}} w_{\ell,i}^n \xrightarrow{p} \int_{\mathbb{X}} \mathbb{1}(x\in A_{\ell,i})\gamma(dx).$$

Assumption 2. *Let $k_\ell : \mathbb{R}_{\geq 0}\to\mathbb{R}_{\geq 0}$ be combination weight functions which produce unnormalized combination weights $k_\ell(N_\ell)$ when provided with the total number of samples used for the corresponding estimator set $N_\ell = \sum_{i\in\mathcal{I}_\ell} N_{\ell,i}$ such that $\lim_{N_\ell\to\infty} k_\ell(N_\ell) = \infty$ for each ℓ, each $k_\ell(N_\ell)$ is finite for any finite N_ℓ, and $\sum_{\ell=1}^L k_\ell(N_\ell) > 0$ whenever $R = \sum_{\ell=1}^L N_\ell > 0$. We further assume that for each estimator set ℓ, either all of the $N_{\ell,i}$*

tend to infinity or none of them. More precisely, we assume there is a non-empty subset $\mathcal{L}_0 \subseteq \{1, \dots, L\}$ *such that for all* $\ell \in \mathcal{L}_0$ *and* $i \in \mathcal{I}_\ell$*,*

$$\lim_{R\to\infty} N_{\ell,i} = \infty$$

and for all $\ell \notin \mathcal{L}_0$*,*

$$\lim_{R\to\infty} N_\ell < \infty$$

almost surely.

Assumption 3. *Each* $\mathcal{I}_\ell$ *is a finite set.*

The last of these assumptions can probably be relaxed to $\mathcal{I}_\ell$ being a countable set, but as it will be algorithmically beneficial to ensure that the depth of the tree remains bounded, this case is of little interest anyway. The need for the second assumption is to ensure that any individual estimator which only has finite computational budget in the limit of large overall budget is given zero weight after normalization.

We are now ready to demonstrate the consistency of our estimator combination.

Lemma 1. *If Assumptions 1, 2 and 3 hold, then*

$$\hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}} := \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i\in\mathcal{I}_\ell} \hat{\gamma}_{\ell,i}^{N_{\ell,i}} \tag{3.16}$$

converges weakly to the measure $\gamma(x)$ *on* $\mathbb{X}$ *as* $R \to \infty$*.*

Proof. By assumption we have that each $\hat{\gamma}_{\ell,i}^{N_{\ell,i}}$ converges weakly to the measure $\gamma(x)\mathbb{1}(x \in A_{\ell,i})$ as $N_{\ell,i}$ tends to ∞. We note that the estimates for $\ell \notin \mathcal{L}_0$ need not converge as $R \to \infty$ as they do not have $N_{\ell,i} \to \infty$ here, but do not affect the final estimate as

$$\lim_{R\to\infty} \frac{k_\ell(N_\ell)}{\sum_{\ell=1}^{L} k_\ell(N_\ell)} = 0 \quad \forall \ell \notin \mathcal{L}_0.$$

To show the claim of this theorem, we now consider an arbitrary bounded continuous function $f : \mathbb{X} \to \mathbb{R}$ for which we have

$$\int f(x)\, \hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}}(dx) = \int f(x) \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i\in\mathcal{I}_\ell} \hat{\gamma}_{\ell,i}^{N_{\ell,i}}(dx) \tag{3.17}$$

$$= \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i\in\mathcal{I}_\ell} \int f(x) \hat{\gamma}_{\ell,i}^{N_{\ell,i}}(dx) \tag{3.18}$$

which using Assumptions 1 and 2 converges as $R \to \infty$ to

$$\frac{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \int f(x) \mathbb{1}\left(x \in A_{\ell,i}\right) \gamma(dx)}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)}$$
$$= \frac{1}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)} \sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \int f(x)\, \gamma(dx) \; = \int f(x)\, \gamma(dx)$$

and thus the expectation taken with respect to $\hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}}$ converges to the true expectation $\int f(x)\, \gamma(dx)$. Now as this holds for an arbitrary f, this implies weak convergence as required. See Appendix A.2 for more derivations. □

Corollary 1. *Let*

$$\hat{Z}^{\{N_{\ell,i}\}_{\ell,i}} := \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \hat{Z}_{\ell,i}^{N_{\ell,i}}. \tag{3.19}$$

If the assumptions of Lemma 1 hold, then

$$\hat{Z}^{\{N_{\ell,i}\}_{\ell,i}} \xrightarrow{p} Z. \tag{3.20}$$

and

$$\hat{\pi}_{\ell,i}^{\{N_{\ell,i}\}} := \frac{\hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}}}{\hat{Z}^{\{N_{\ell,i}\}_{\ell,i}}} \tag{3.21}$$

converges weakly to the measure $\pi(x)$ on $\mathbb{X}$ as $R \to \infty$.

Proof. Using the same arguments as Lemma 1 with $f(x) = 1$ gives

$$\hat{Z}^{\{N_{\ell,i}\}_{\ell,i}} \xrightarrow{p} \frac{1}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)} \sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \int_{\mathbb{X}} \mathbb{1}(x \in A_{\ell,i}) \gamma(dx) \tag{3.22}$$
$$= \frac{1}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)} \sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \int_{\mathbb{X}} \gamma(dx) \tag{3.23}$$
$$= Z. \tag{3.24}$$

The second result now follows directly from Slutsky's Theorem and Lemma 1. □

These results firstly convey that if we combine convergent estimators for the partitioned parts of the overall target, we get a convergent estimator for the target. Secondly, it implies that we can similarly combine a number of estimates for the target, which come from different partitionings. For example, we can combine a estimate $\hat{\gamma}(x)\mathbb{1}(x \in A_j)$ for the trivial partition $\{A_j\}$ of A_j, with that given by combining $\hat{\gamma}(x)\mathbb{1}(x \in A_{\ell_j})$ and $\hat{\gamma}(x)\mathbb{1}(x \in A_{r_j})$ for the partitioned parts A_{ℓ_j} and A_{r_j} where $A_j = A_{\ell_j} \cup A_{r_j}$, in a manner that

preserves consistency, i.e. we can consistently combine parents estimates with their children. These results hold independently of how the k_ℓ are chosen, provided Assumption 2 holds. However, the variances of the associated estimates are likely to depend heavily on the choice of k_ℓ – we wish to place more weight on the partitionings with lower variance estimates.

A critical point is that the combination of estimators does not require any correction factor for the number of times that an estimator and a partition were "proposed" – i.e. we do not need to correct for the fact that more computational resources are provided for some estimates than others or because some partitions of the space are potentially larger than others. All such potential factors either cancel out, or are dealt with by the correct normalization of the truncated proposal. As such, any strategy on deciding the partitions or how often a partition is proposed only need satisfy the stated assumptions to ensure consistency. We are now thus ready to prove Theorem 1 as follows.

Theorem 1. *If the following hold as the number of IT iterations becomes infinitely large*

- *The total number of leaf nodes remains bounded and each is visited infinitely often;*
- *When provided with an infinite sample budget and an arbitrary subregion A_j generated by the node splitting procedure, the base inference algorithm produces an empirical measure $\hat{\gamma}_j(\cdot)$ and normalization constant estimate $\hat{Z}_j$ which respectively converge weakly to $\gamma(x)\mathbb{1}(x \in A_j)$ and converge in probability to $\int_{x \in A_j} \gamma(x)dx$;*

then each $\hat{\pi}_j(\cdot)$ as defined by (3.5) *converges weakly to $\pi(x|x \in A_j)$ and, in particular, $\hat{\pi}_0(\cdot)$ converges weakly to $\pi(x)$.*

Proof. The proof follows using a combination of showing that Assumptions 1, 2 and 3 are satisfied and a recursive application of Lemma 1 and Corollary 1.

We start by considering $\hat{\gamma}_j(\cdot)$ and $\hat{Z}_j$ for a node j whose children are both leaf nodes. Here Assumption 3 is trivially satisfied as we have two estimates: the local parent estimate and the combined child estimate. By construction, the combination of a parent node and child node estimates satisfies the partitioning requirements of Assumption 1, while by the final assumption in the theorem, we have the required consistency of each of the child and parent node estimates in isolation. Thus Assumption 1 is also satisfied. Assumption 2 is satisfied through the assumption that each leaf node is visited infinitely often as the budget

becomes arbitrarily large and the fact that, by construction, $c_j \to 1$ for the parent node as this happens unless the number of samples used to construct the local parent estimator also becomes infinitely large, in which case both estimates converge anyway.

Lemma 1 now tells us that $\hat{\gamma}_j(\cdot) \to \gamma(x)\mathbb{1}(x \in A_j)$ and Corollary 1 tells us that $\hat{Z}_j \to \int_{x \in A_j} \gamma(x)dx$ and $\hat{\pi}_j(\cdot) \to \pi(\cdot|x \in A_j)$. We thus have the Theorem holds for leaf nodes and all nodes whose children a both leaves.

We can now recursively apply the same logic to show that the Theorem holds for all nodes in the tree. Specifically, we have that a node also converges if both its children nodes convergence, and so by induction all the nodes in the tree must converge. □

Remark 1. *This result can be trivially extended to convergence in probability, $\mathcal{L}^P$ convergence, and almost sure convergence of the expectation estimates, given the assumption that both the $\hat{Z}_j$ and the corresponding unnormalized local expectation estimates*

$$\hat{\varrho}_j := \frac{1}{N_j} \sum_{n=1}^{N_j} w_j^n f\left(\hat{x}_j^n\right)$$

provide the required convergence. This follows by simply noting that the arguments in each proof remain equally valid for $\hat{\varrho}_j$ and for the different forms of convergence.

3.8 Experiments

3.8.1 Gaussian Mixture Model

Our first experiment is to infer the cluster means in a Gaussian mixture model (GMM). Specifically,

$$z_n \sim \text{Categorical}(\{1/K, \ldots, 1/K\}),$$
$$\mu_k \sim \mathcal{N}(0, \Sigma_\mu), \quad y_n \Big| z_n = k, \mu_k \sim \mathcal{N}(\mu_k, \Sigma_y),$$

where we set $\Sigma_\mu = I$, $\Sigma_y = 0.2I$, and $K = 4$. We generated a two-dimensional synthetic dataset $y_{1:200}$ using the generative model and then ran ITs with importance sampling as the base algorithm to conduct inference on μ_k, with the z_n marginalized out by summation. We use the prior on μ_k as our base proposal. Though simple, this constitutes a surprisingly challenging inference problem, as symmetries in the model mean that the posterior is concentrated in 24 well-separated modes, each of which occupy less than 10^{-10} of the overall eight-dimensional parameter space.

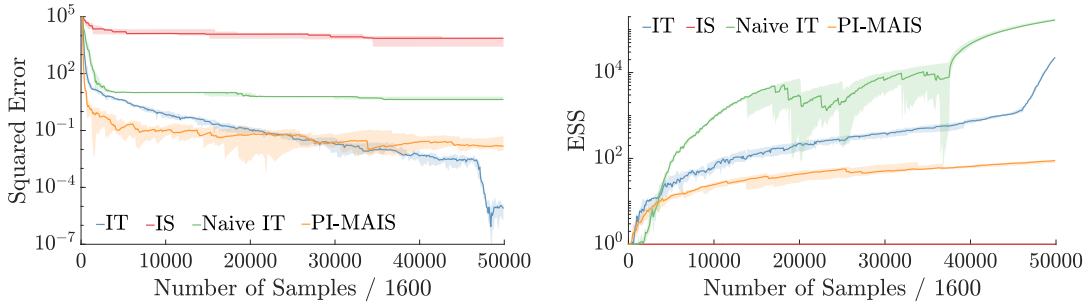
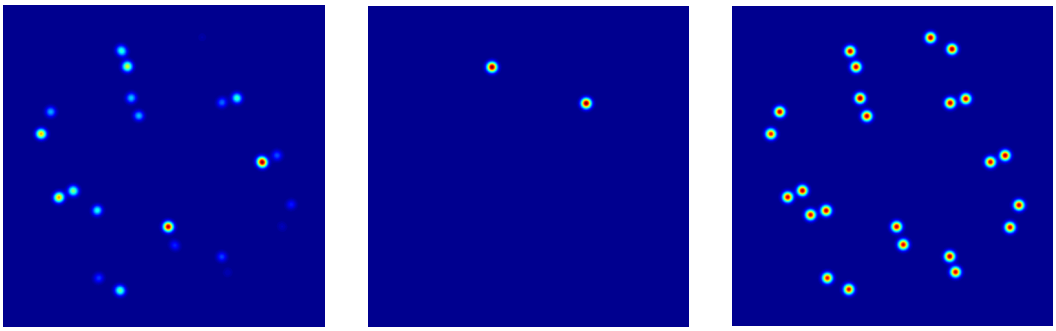


Figure 3.4: Convergence for the GMM in terms of the log ML estimate (left) and the ESS (right). The ground truth log marginal was estimated using a very large number of samples and a manually adapted proposal. Solid lines represent median over 10 runs and shading the 25%-75% quantiles. The reason for the "just-in-time" style convergence of the IT stems from the fact that the parameter annealing schedules start to kick in and encourage far more exploitation near the end of the runs.



(a) PI-MAIS (b) Naïve IT (c) IT

Figure 3.5: Kernel density estimate of projected posterior estimates for the GMM. We use a linear projection of the original 8-dimensional spaces and exaggerate the variance of the modes for visualization purposes. The IT has successfully recovered all modes and inferred that all the modes have equal mass. Though the naïve IT implementation produced good estimates for the modes it found, it missed modes for both problems. The baseline PI-MAIS, however, found a number of modes but still missed some and misestimated their relative masses.

For computational efficiency, we fixed "one run" of the base inference algorithm to be comprised of drawing 100 importance samples and we undertook 16 runs of this base algorithm for each refinement step (with each counting as a separate traversal). We further took the convention in, for example, log weight density estimation that each "run" returns a single amalgamated w_i^n, which might itself contain multiple samples (similarly w_i^n becomes the SMC ML estimate in the next experiment). We compared to the following baselines given the same total budget of target density evaluations: i) non-adaptive importance sampling; ii) a naïve IT implementation where we set $\delta = 0$, $\alpha = 0$, and $\beta = 0.5$, which means that our target ignores the $\hat{p}_j^s$ terms and relies solely

on the optimism boost for exploration; and iii) PI-MAIS [Martino et al., 2017], a state-of-the-art adaptive importance sampler based on simulating a large number of Markov chains to construct the proposal. Each algorithm was given a budget of 8×10^7 target evaluations. The IT parameters were set as $\kappa = 1$ and $\beta = 0.1$. Denoting ρ as the proportion of total iterations run thus far, the annealing parameters were given schedules of $\delta(\rho) = \frac{1}{2}\left(1 + \tanh\left(20\left(0.9 - \rho\right)\right)\right)$ and $\alpha(\rho) = \frac{5}{8}\left(1 + \tanh\left(25\left(0.95 - \rho\right)\right)\right)$. We further fixed $\beta = 0$ for the last 25% of the iterations to reflect the fact that, because we are carrying out inference rather than optimization, we want to spend part of our sample budget more directly exploiting the learned tree.

Our main baseline, PI-MAIS [Martino et al., 2017] is a state-of-the-art adaptive importance sampling algorithm that runs a number of independent MCMC chains targeting the joint distribution and then uses the locations of these chains to, at each iteration, construct a mixture of Gaussian proposal distribution, with each component centered on the location of one of the chains. We used $N = 100$ such chains and proposed $M = 15$ samples from each chain at each iteration, noting that the algorithm requires $N(M + 1)$ target evaluations. We further used an random walk kernel with covariance $0.0001I$ for each of the MCMC chains, while each proposal component is taken as an isotropic Gaussian with covariance $0.01I$.

For comparison, we examined the convergence of the ML estimate and ESS (Figure 3.4) and a kernel density estimator of the final output (Figure 3.5). The results show that ITs outperformed the alternatives. Unsurprisingly, vanilla importance sampling performed poorly throughout, ending with an ESS of effectively 1. The naïve IT implementation managed to generate a very high ESS, but typically only found two or three modes leading to a substantial error in the ML estimate. PI-MAIS did better at finding modes, though still substantially worse than IT. Further, it ended with a low ESS and produced poor estimates for the relative sizes of the modes, in turn giving an inferior log ML estimate.

3.8.2 Chaotic Dynamics Model

Dealing with long-range dependencies, i.e. variables that have influence many steps after they are sampled, can be challenging in SMC as variables are often fixed before all dependent terms are incorporated, leading to sample degeneracy. Viewing this in another light, the intermediate target distributions can vary substantially from the target marginal distribution

on the relevant variables. Naïve strategies for dealing with this tend to be futile – the resampling step always corrects to the intermediate target and thus incorporating lookahead information in proposals often reduces the effective sample size. In some cases, auxiliary weighting schemes provide a degree of lookahead [Doucet et al., 2006; Lin et al., 2013], but these typically entail a substantial increase in computational cost while providing only a short-range lookahead. Moreover, problems with degeneracy can be compounded in the context of adaptation as information is only received for particles that survive the resampling.

We now show that ITs can address these challenges by running inference on separate regions. Namely, the IT process allows information to be gathered even in the face of degeneracy. Constraining different sweeps to different regions allows samples to be "forced through" the resampling steps, hereby dealing with long-range dependencies. This is done without losing the key benefits of SMC, as gains from resampling are still seen when running inference within a particular region. Note that ITs only require an unbiased estimate for the weights in a manner akin to pseudo-marginal methods [Andrieu and Roberts, 2009], such that we can run SMC when there are some latent variables not directly controlled by the IT.

To test ITs in this setting, we consider an adaptation of the chaotic dynamical system tracking problem introduced by Rainforth et al. [2016a]. This model comprises of an extended Kalman filter defined as

$$\begin{aligned} x_0 &\sim \mathcal{N}(0, I) \\ f_t(x_t|x_{t-1}) &= A(x_{t-1}, \theta) + \upsilon_{t-1}, \quad \upsilon_{t-1} \sim \mathcal{N}(0, 0.01I) \\ g_t(y_t|x_t) &= Cx_t + \varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0, 0.2I). \end{aligned}$$

where C is a known $K \times 3$ matrix. The transition function $A(\cdot, \theta)$ dictates the underlying dynamics with parameters θ. We will assume that the form of A is known but not the parameters. Namely, we consider the example where the dynamics correspond to the Pickover attractor defined as

$$\begin{aligned} x_{t,1} &= \sin(bx_{t-1,2}) - \cos(ax_{t-1,1})\, x_{t-1,3} \\ x_{t,2} &= \sin(dx_{t-1,1})\, x_{t-1,3} - \cos(cx_{t-1,2}) \\ x_{t,3} &= \sin(x_{t-1,1}) \end{aligned}$$

where $\theta = (a, b, c, d)$. We finish the model by defining the prior on each dynamics parameter to be a uniform over $[-\pi, \pi]$. A synthetic dataset $y_{1:200}$ was generated by fixing $b = -2.3$, $a = 2.5$, $d = -1.5$, $c = 1.25$, $K = 20$, and drawing each column of C from a symmetric Dirichlet distribution with concentration 0.1.

We desire to conduct inference over both the dynamics parameters θ and the latent variables $x_{1:T}$, but will only use ITs to control the sampling of the former. This model contains long-range dependencies because the dynamics parameters affect each transition and so the smoothing marginal $p(\theta|y_{1:T})$ is very different to the filtering marginal $p(\theta|y_1)$. In fact, the two are so different that using the so-called one-step-optimal proposal, the target for most methods of SMC proposal adaptation [Gu et al., 2015], provides no noticeable performance improvement over simply sampling from the prior. Furthermore, there are again symmetries in the model, causing $p(\theta|y_{1:T})$ to have four dominant modes.

Because PI-MAIS requires an MCMC sampler to be run on the target $p(\theta|y_{1:T})$, it is inappropriate for this problem. We instead compare to using i) SMC without adaptation, ii) SMC with 1000 times more particles, iii) the naïve IT implementation, and iv) PMMH [Andrieu et al., 2010], a method explicitly designed for dealing with global parameters in SMC. We allowed a budget of 1×10^7 target evaluations and used 8 SMC sweeps of 500 particles per refinement step for the IT approaches. Our main baseline was PMMH, where one runs an MCMC sampler targeting $p(\theta|y_{1:T})$ but with the likelihood evaluation in the MH acceptance step replaced with the unbiased ML estimate produced by an SMC sweep. For this, we use isotropic random walk proposal with a covariance of $0.0004I$. For the SMC sweeps, we used 500 particles and the bootstrap proposal. The same IT parameters were used as the GMM experiment, with the exception that we changed the annealing schedules to match the lower number of iterations, setting $\delta(\rho) = \frac{1}{2}\left(1 + \tanh\left(4\left(0.7 - \rho\right)\right)\right)$ and $\alpha(\rho) = \frac{5}{8}\left(1 + \tanh\left(10\left(0.8 - \rho\right)\right)\right)$. We used the same comparison metrics as for the GMM, with results shown in Figures 3.6 and 3.7. We see that ITs again outperformed the other methods.

To further demonstrate that PMMH fails to move between modes in any of the runs, we now plot the individual sample paths as shown in Figure 3.8. We see that for the parameters

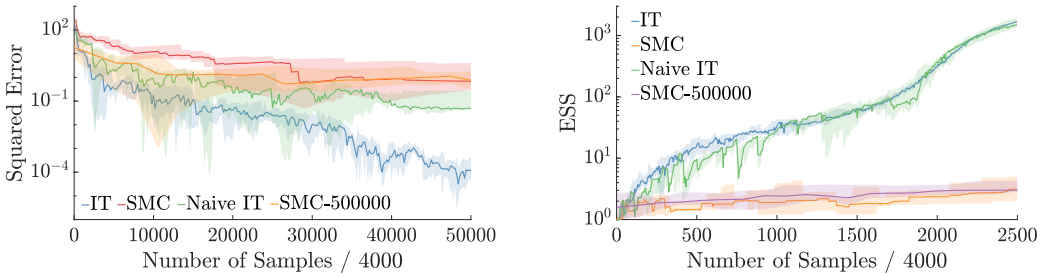
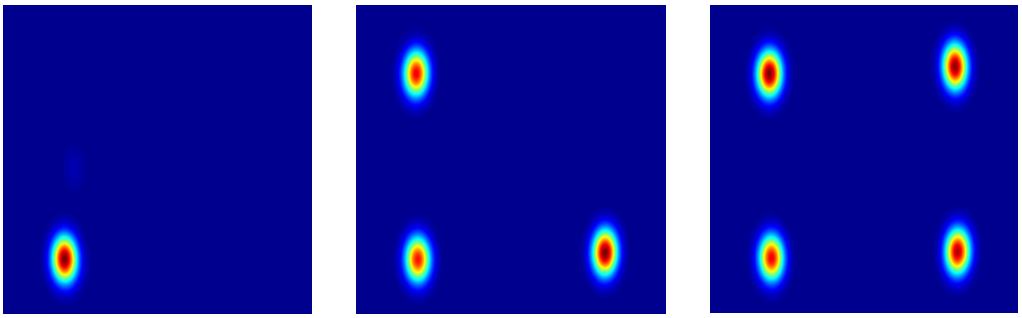


Figure 3.6: Convergence of log ML and ESS for chaos model, conventions as per Figure 3.4. PMMH is not shown as it returns unweighted samples and no ML estimate.



(a) PMMH (b) Naïve IT (c) IT

Figure 3.7: Kernel density estimate of projected posterior estimates for chaos model with exaggerated mode variances. We see that PMMH successfully exploits one of the modes, but it does not move been modes. Meanwhile, the naïve IT finds three of the modes and the main IT algorithm all four.

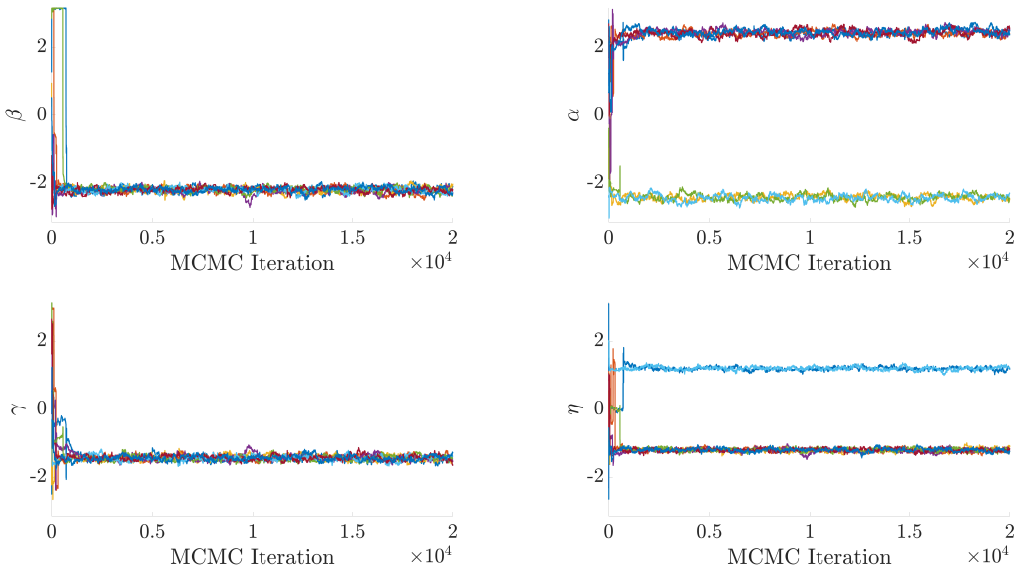


Figure 3.8: Sample paths of PMMH.

with multiple modes, α and η, the PMMH sampler never moves between the modes. Thus in all runs we see PMMH was only able to pick up a single mode.

3.9 Conclusion

We have introduced inference trees (ITs), a new adaptive inference algorithm drawing on ideas from Monte Carlo tree search. We have shown that, by carrying out explicit *exploration* in the adaptation process, ITs can avoid common pathologies with other adaptive schemes and reliably uncover multiple modes. We have consequently found that, for the tested models, ITs outperformed previous state-of-the-art adaptive importance sampling and particle MCMC methods. In addition to the immediate utility of the proposed approach, we believe that the general IT framework opens up many opportunities for new research, due to the separation between their consistency and the specifics of the learning algorithm.

4

Restricted Probabilistic Programming

In the previous two chapters, we have introduced what Bayesian machine learning approaches are and how to solve non–standard models using a well–designed inference algorithm. Practically, implementing such process involves heavy manual work. Experts from different fields firstly need to formalize a problem as a probabilistic model, and then they need to code up the required inference method(s) to reason about such model. Depending on the proficiency of their programming skills, these two steps are usually implemented in an intertwined, end-to-end way. This is problematic due to the need of iterating the model design; it becomes extremely inefficient as any small change in the model may require a re-implementation of the inference scheme associated with the current model. Quoting from Mansinghka et al. [2014], "even relatively simple probabilistic models and their associated inference schemes can be difficult and time-consuming to design, specify, analyze, implement, and debug". Therefore, a common wish among various fields is that inference methods can be more easily accessible, or even better, can be used automatically without much extra learning or engineering cost for new models. Such wish stimulates an emergent field called *probabilistic programming*.

In this chapter, we will first introduce what probabilistic programming is at a high-level, and how to design a probabilistic programming system (PPS) in general, before settling on the *restricted* class of PPSs for the rest of this chapter. In particular, we will introduce how model are specified as program in these systems, how to design an abstraction layer

between the model and the inference procedure, and how inference algorithms can be implemented as automated engines. Some key ideas will be presented using a specific PPL, FOPPL [van de Meent et al., 2018], as a running example but they can also be applied to other restricted PPSs as well.

4.1 A High-Level Introduction to Probabilistic Programming Systems

Probabilistic programming provides a powerful framework to build automated, principled systems that separate model specification and inference [Gordon et al., 2014]. In general, models are defined as program code whereas inference algorithms are provided as automated engines. On the one hand, it has substantially reduced the burden of the domain experts by allowing them to access various inference methods in a convenient way, rather than requiring them to learn and implement every needed inference option by hand. This enables the model development to iterate in a much more time-saving manner, and also frees the modelers to focus on coming up with good models (instead of wasting time on repetitive engineering work). On the other hand, the separation and automation also benefits the statistical or machine learning community who focuses on developing novel inference algorithms. Once an inference method is coded in a system and is properly engineered and optimized, it is ready to be applied to a broad range of models which satisfy its prerequisites, if there are any.

As pointed out by van de Meent et al. [2018], deep learning is able to achieve rapid exploration nowadays largely because there are well-equipped toolchains available to perform gradient-based optimizations. However, such a tool did not commonly exist for probabilistic machine learning to perform inference-based reasoning. *Probabilistic programming systems* (PPSs) have the potential to become such toolkits and boost the adoption of probabilistic machine learning methods. In particular, we mainly focus on the PPSs for Bayesian inference algorithms, and highlight that achieving a fully automated, generally applicable, and widely efficient pipeline for all probabilistic machine learning problems is an ambitious, long-term goal.

To design a PPS, we normally require two things: a modeling language to support model specification, and one or multiple inference engines to automatically reason about

the provided models. The richness of the modeling language decides the class of models that a PPS can support, and we say a PPS is *restricted* or *universal* depending on the expressivity of the modeling language. From a user's perspective, such language should be easy to learn and straightforward to use for people without extensive programming skills. To enable inference algorithm to work in an automated manner, one can separate the modeling language and the inference procedure by an *abstraction layer* — it extracts all the information from the model into a convenient representation which can be used afterwards at inference stage. Inference algorithms can be then implemented in a generic way that they only interact with the internal representation and no longer care how every single model is written by the user. They can even be implemented in a language different from the modeling language, such as a lower-level language where the code is actually run, for the purpose of efficiency. Of particular note is, the choice of the inference algorithm does affect the type of models can be supported by a system (since each algorithm has different prerequisites and focuses). How to balance the trade-off between model expressiveness and inference efficiency has been the key question to be considered when designing a PPS.

Existing PPSs can be roughly classified into two classes: restricted PPSs and universal PPSs. By "***restricted*** class", we refer to PPSs which are mainly inference-driven, such as BUGS [Spiegelhalter et al., 1996], Infer.NET [Minka et al., 2010], LibBi [Murray, 2013], Stan [Stan Development Team, 2014], PyMC3 [Salvatier et al., 2016], and Edward [Tran et al., 2016]. Such PPSs are mainly designed to automate one or two efficient inference algorithms. For this reason, they usually impose strict restrictions on the class of models which can be denoted in the system, and only allow models that can be handled by the corresponding inference engines (e.g. only supporting DAG models or models with differentiable densities). Though being restrictive, some PPSs of this type (eg. BUGS in particular) are widely used to solve many statistical models and have played an important role in the (early-stage) development of probabilistic programming.

As an alternative, the other class of PPSs are ***universal*** and aim to support any possible model one might be interested in, where examples of this type include Church [Goodman et al., 2008a], Venture [Mansinghka et al., 2014], WebPPL [Goodman and Stuhlmüller, 2014], Anglican [Wood et al., 2014], Birch [Murray and Schön, 2018], Turing.jl [Ge

et al., 2018a], and Pyro [Uber, 2017]. By design, such systems target the richest possible models; their underlying languages are usually universal, and therefore they can be used to specify any computable distribution. Their expressivity at the same time substantially complicates the automated inference procedure. How to automate inference for any possible model as well as maintain the efficiency is the key challenge for the universal PPSs. We will go back to this point in Chapter 6, and for the rest of this chapter, we will focus on the restricted class of PPSs.

4.2 Model Specification as Program Code

A probabilistic model is defined as a piece of program code using the specific modeling language in PPS, which is known as a *probabilistic program*, and the modeling language is further referred as the probabilistic programming language (PPL). Compared to a standard, non-probabilistic program, a probabilistic program has two distinct features: the ability to draw from distributions, and the ability to condition the value of a random variable through observation [Gordon et al., 2014].

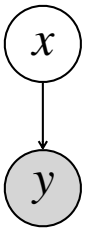
To fulfill these two properties, it requires some special constructs in the language. The first feature, drawing from distributions, is not too surprising since many standard, deterministic languages already have packages for generating pseudo-random numbers. A common way taken by many existing PPLs is to define a special `sample` statement[1] to generate samples from *elementary random procedures* (ERPs) wrapping around existing packages. The second feature, conditioning, however, is much more complicated. It includes specifying which variables are observable and what their observed values are, and closely interacts with the inference algorithm to obtain the conditional, or the posterior, distribution. Such operation is uncommon in normal programing languages (unless one manually defines a function which performs the aforementioned conditioning steps explicitly). Many PPLs implement conditioning by defining a special form `observe`, or over-loading `sample` with extra arguments. Users can then indicate the observable variables together with the observed value.

[1] Throughout this thesis, we will sometimes demonstrate the key aspects of PPL borrowing the Lisp dialect syntax partly used in the existing PPL Anglican [Wood et al., 2014; Tolpin et al., 2016], FOPPL [van de Meent et al., 2018], and LF-PPL [Zhou et al., 2019a].

For restricted PPSs, the supported models are limited in order to satisfy the requirements of a certain inference algorithm. Their underlying PPLs are usually *first-order*; that is the functions of such language must be first-order; functions cannot be passed as arguments to other functions or return other functions as results. Also, unbounded recursion and stochastic loop are not allowed in such languages. Models defined in such a restricted PPL have a finite number of random variables and their conditional dependencies can be determined at compile time. Therefore, there is a fixed number of `sample` and `observe` statements in models in restricted PPSs. Note that apart from these common features in the modeling language, each PPS normally has a few extra requirements, depending on the inference engine it is built for. For example, in order to support the Gibbs sampler [Geman and Geman, 1984], BUGS [Spiegelhalter et al., 1996] requires the access to the Markov blanket of each latent variable. To support HMC [Neal, 2011] and its variants, Stan [Stan Development Team, 2014] only allows continuous latent variables and tries to make sure the joint densities of its models are differentiable. We will use a recent example of a first-order restricted PPL, **FOPPL**, introduced by van de Meent et al. [2018] as a running example to demonstrate some key aspects of probabilistic programming but note that many ideas of FOPPL appear in other restricted PPSs as well.

4.2.1 An FOPPL Example

FOPPL is a carefully customized first-order PPL without recursions or stochastic loops and is based on the functional programming language Clojure [Hickey, 2008]. The language by design is particularly simple with only eight expressions, including two special forms, `sample` and `observe`, for the probabilistic procedures. All functions in FOPPL are first-order by definition, and models written in FOPPL can only have a finite number of random variables. Though `if-else` branching statement is allowed in FOPPL, we will ignore it for now and come back to this point at the end of this chapter. It might sound cumbersome to use the language to specify models directly, it is easy to reason about the language due to its simplicity. The expressivity of FOPPL is actually equivalent to the predominantly used PPS, BUGS [Spiegelhalter et al., 1996], and it serves as a perfect example for the pedagogical purposes.



$$x \sim \mathcal{N}(\mu_0, \sigma_0^2)$$
$$y|x \sim \mathcal{N}(x, \sigma_y^2)$$

```
(let [x (sample (normal 0 2))]
   (observe (normal x 1) y)
   x)
```

Figure 4.1: Gaussian with unknown mean model from Section 2.1.1: the DAG representation (left), the statistical denotation (middle), and the probabilistic program in FOPPL (right).

There are two special forms, **`sample`** and **`observe`**, which work with the probabilistic procedures closely. Specifically, **`sample`** statement takes a distribution object, `dist`, as an input whereas the **`observe`** takes a distribution `dist` and a constant value. Suppose there are N_x latent variables and N_y observable variables in the model. A probabilistic program written in such PPL denotes a generative model with a joint probability density $p(x, y)$ as,

$$\gamma(x) := p(x, y) = \prod_{i=1}^{N_x} f_i\left(x_i \mid \eta_i\right) \prod_{j=1}^{N_y} g_j\left(y_j \mid \phi_j\right) \tag{4.1}$$

where f and g are the probability density or mass functions defined by the **`sample`** and **`observe`** statements respectively, and η_i and ϕ_j include all the parameters needed to evaluate f_i and g_i. Note that data y are presumed to not appear in either η or ϕ for this restricted setup. As we can see, the product of **`sample`** terms, $\prod_{i=1}^{N_x} f_i\left(x_i \mid \eta_i\right)$, is exactly the prior $p(x)$ whereas the product of the **`observe`** probability densities, $\prod_{j=1}^{N_y} g_j\left(y_j \mid \phi_j\right)$, corresponds to the overall likelihood functions $p(y|x)$. Therefore a FOPPL program essentially defines a finite, static DAG.

Furthermore, the returning expression of a FOPPL program is usually a deterministic function over the latent variables $\Omega(x)$. For instance, it may directly return the variables x, in which case we return the generated samples for the corresponding latent variables. In particular, once inference has been carried out, these returned samples can be assumed to follow the target posterior distribution $\pi(x)$. One could then *query* the model with some test function $f(\cdot)$ manipulating the posterior samples and estimate the expectation of such function w.r.t. the posterior using the MC estimator following (2.9).

To have an idea about what a probabilistic program might look like, we consider the Gaussian with unknown mean example from Section 2.1.1 again with all parameters μ_0, σ_0 and σ_y instantiated. The program snippet on the right of Figure 4.1 is the corresponding

program of this model written in FOPPL. The latent variable x is declared using the **sample** statement followed by the prior distribution object (**normal** 0 2). After that, the likelihood function is defined via the **observe** statement, with a distribution object (**normal** x 1) and data y. With this program, one can construct the joint distribution $\gamma(x)$ of this model according to (4.1) with all **sample** and **observe** terms factored into γ. If one has the test function $f(\cdot)$ of interest beforehand, it can be incorporated into the returning statement of the program to estimate its expectation; otherwise, one could just return the latent variable x and manipulate it afterwards when needed.

4.3 The Abstraction Layer

For the inference algorithm to work in a generic way, we need to establish an abstraction layer to separate the modeling and inference procedure. Behind the scenes, the inference engine usually does not interact directly with the exact program code a user writes; instead, it interacts with an intermediate representation or internal data structure that stores necessary information of the model that is needed at inference time; this forms an *abstraction layer* between the model code that users write and the underlying inference engine(s). How to construct such internal representation depends on the core inference choice and the type of models, and therefore varies among different restricted PPSs.

The most widely used PPS, BUGS [Spiegelhalter et al., 1996], including its descendants WinBUGs [Lunn et al., 2000] and Jags [Plummer et al., 2003], focuses on Gibbs sampling [Geman and Geman, 1984], and models allowed in such systems are predominantly graphical models, DAGs in particular. Models written in the BUGS modeling language will be compiled to an internal data structure storing the information as a directed graphical model, and at inference time, for any given latent variable, it is easy to extract its Markov blanket using such data structure. The modern competitor, Stan [Stan Development Team, 2014] alternatively focuses on more scalable methods based on gradient information such as HMC and its variants. It does not care about the conditional dependencies among variables as needed in BUGS, but instead needs to evaluate the joint probability density of the model and to automatically calculate its derivatives. Therefore, models written in Stan will compile to an internal presentation storing these two functions, which will be called repeatedly

during HMC update. In the following section, we will present the internal representation of a probabilistic program in FOPPL, which has a well-established semantics and thorough description compared to other restricted PPSs. The way it constructs the intermediate data structure also shares many key ideas with other existing PPSs.

4.3.1 Internal Representation of Graphical Models

As introduced in Section 4.2.1, models allowed in FOPPL by design are all graphical models. This is an important property to exploit for the inference procedure, and is ensured by the careful design of the language and can be shown by its semantics. We refer the readers to van de Meent et al. [2018] for more details of the proof, and instead introduce how a FOPPL program is transformed to a graphical model data structure.

For the standard setup, a FOPPL program e can be compiled to a graphical model with the data structure G and a returning expression E. As we have introduced in Section 2.1, a graphical model contains key information about the vertices V, which correspond to all the variables of the model, and the arcs A telling the dependency relationships among the variables. Apart from V and A, the graphical model representation G in FOPPL also includes two more components in order to incorporate the density information. Specifically, it also contains a map $\mathcal{P}$ about the probability density (or mass) function for each variable in V, and a partial map $\mathcal{Y}$ about the observed values. Putting all together, we have the internal data structure for the graphical model G as a tuple $(V, A, \mathcal{P}, \mathcal{Y})$.

We illustrate this data structure using the FOPPL program for the Gaussian with unknown mean model in Figure 4.1 as an example. The vertex set V include variables x and y, and the arc set $A = \{(x, y)\}$. The map $\mathcal{P}$ contains information about the densities related to these variables as $[x \to \mathcal{N}(x; 0, 2),\ y \to \mathcal{N}(y; x, 2)]$, whereas $\mathcal{Y}$ specifies the value of the observed variable as $[y \to 9]$. The returning expression E of the program is the random variable x.

Extracting information from the model and transforming to the graphical model representation G is important because it is much more convenient for inference engines to interact with G than to the actual program code. Models defining the same joint distribution might be written in various ways, but they can often be transformed to similar Gs. Intuitively, G is an abstraction of a FOPPL model with all key information, and inference engines

can access necessary information. For instance, by constructing the data structure G, we can obtain the conditional dependence graph of the random variables, from which we can extract the Markov blanket of each random variable. When performing Gibbs style updates, the sampler can make use of this information to construct the conditional distribution of each latent variable.

Furthermore, we can define the density of a FOPPL program using its transformed graphical model representation G. Following the GM formalization as per (2.4), the distribution specified by G can be constructed by factorizing the density or mass function of each variable in V as,

$$\gamma(x) = p_G(V) = \prod_{v \in V} p(v|PA(v)), \tag{4.2}$$

where the density function $p(v|PA(v))$ is stored in $\mathcal{P}$ and can be evaluated by calling $\mathcal{P}(v)$ with all parameters evaluated. One can also show that it is equivalent to break $p_G(V)$ into a prior $p(x)$ and a likelihood $p(y|x)$ term where we first separate V into observable nodes Y and the unobservable or latent ones X. Suppose we use Y to denote all observable nodes in V that also appear in $\mathcal{Y}$. Then let X be the rest of the nodes in V, i.e., $X = V \backslash Y$. We can then rewrite $p_G(V)$ using X and Y as,

$$\begin{aligned} &\gamma(x) = p_G(V) = p(x)p(y|x), \\ &\quad \text{where} \quad p(x) = \prod_{x \in X} p(x|PA(x)),\ p(y|x) = \prod_{y \in Y} p(y|PA(y)). \end{aligned} \tag{4.3}$$

We can see that a FOPPL program also corresponds to a joint generative model $p(x, y)$, and the target distribution of interest $\pi(x)$ is the posterior $p(x|y)$. In the next section, we will discuss how to approximate π using Bayesian inference methods. In particular, we will highlight why G is convenient for the inference procedure and how inference engines interact with G in different ways.

4.4 Inference Methods as Automated Engines for the Restricted Setup

Recall that a central goal of a PPS is to characterize the target posterior distribution $\pi(x)$ of interest in an automated manner. Bayesian inference has offered a principled way to approximate $\pi(x)$ when we can only access to the unnormalized version $\gamma(x)$

defined by the joint model. Inference algorithms, such as the MC methods introduced in Section 2.2, can be incorporated as automatic engines in a PPS. Once implemented, they can be used repetitively to analyze any model written in the modeling language of the system. Different inference methods, however, would impose different requirements on the type of the models they can deal with. Therefore, it is essential to ensure that a PPS is designed carefully such that all constraints of the inference engines can be satisfied for any model written in the underlying PPL.

Restricted PPSs make an explicit trade-off about this: they are usually built around one or two inference engines, and restricts the PPL accordingly to constrain the class of supported models. In this section, we will present some key ideas of automating inference in restricted PPSs by showing two inference engines for FOPPL. In particular, we highlight the difference between *graph-based* and *evaluation-based* inference pointed out by van de Meent et al. [2018]. At a high-level, the former refers to the scenarios where a probabilistic program can be compiled to a static graphical model representation G and inference techniques interact with G. By contrast, the latter is usually used in the universal setup where compiling a probabilistic program to a graph model is infeasible. Therefore inference is normally undertaken via evaluating the execution of a probabilistic program. We will focus mainly on the *graph-based* inference in this section and investigate the other scenario in Chapter 6. Note that a *graph-based* implementation normally only applies to the restricted setup whereas evaluation-based options in theory are applicable to both.

4.4.1 Gibbs Sampler

Gibbs sampling [Geman and Geman, 1984] is a representative inference algorithm that exploits the conditional dependency relationship. It is a special instance of the MH algorithm introduced in Section 2.3.3 and it works by updating one or a subset of variables at a time. Suppose the total number of the latent variables is D. At each iteration, the Gibbs sampler updates one variable x_d using a proposal distribution $p(x_d|x_{\neg d}, y)$ with all the rest variables being fixed, before iterating over other latent variables, $x_{\neg d}$, in a similar manner. Standard Gibbs sampler works well in many problems in practice, especially when the full conditional distribution $p(x_d|x_{\neg d}, y)$ for all ds are easy to obtain and to sample from. However, this is

only feasible for very specific cases, such as models containing discrete variables only or models with densities of all variables falling in the exponential family.

When the conditional is not analytically calculable, an alternative option is to take an *Metropolis-within-Gibbs* (MwG) approach. Similar to the original Gibbs algorithm, MwG sampler also cycles through all variables and update one variable or a block at a time. What is different is that when updating variable x_d, instead of directly sampling from the full conditional, we generate a sample x'_d from a user-defined proposal $q(\cdot|x_d)$. This gives us the updated x' with all the rest unchanged apart from x_d, i.e. $x' = (x \backslash \{x_d\}) \cup \{x'_d\}$. We then perform an MH accept/reject step to decide whether to accept x' with probability as

$$\mathcal{A}(x', x) = \min\left(1, \frac{\gamma(x')q(x_d|x')}{\gamma(x)q(x'_d|x)}\right) \tag{4.4}$$

To evaluate (4.4), the joint distribution γ can be evaluated as per (4.2). Note that many terms of the ratio $\gamma(x)/\gamma(x')$ can be naturally canceled out since γ is the product of individual density of each node, and the change of value for x_d would only affect the calculation of its own density and the ones of its direct children nodes. Such simplification can be carried out conveniently when we are provided the internal graphical model representation G.

Metropolis-within-Gibbs method has demonstrated superior performance for certain problems, but its performance can be quite sensitive to the proposal choices. In the following section, we have a look at how HMC, which is an efficient inference algorithm for high-dimensional models, can be implemented as an automated engine.

4.4.2 Hamiltonian Monte Carlo Engine

As we have introduced in Section 2.4.2, HMC works by leveraging gradient information to evolve efficiently in high-dimensional space. Apart from latent variables x, we also introduce auxiliary momentum variables ρ to simulate from the Hamiltonian dynamics. In particular, it requires the evaluation of the potential energy $U(x) := -\log\gamma(x)$ and the kinetic energy $K(\rho)$, and their derivatives.

The calculations of K and ∇K are usually not problematic as K is chosen by the user and one can choose a form for which the derivative is easy to compute (e.g. Gaussian). U and ∇U are defined by the unnormalized density of the model and its derivative, and hence storing the information about them in a convenient way in G is essential. Evaluating U is

straightforward as γ can be obtained from G using (4.2) akin to Gibbs sampler. In terms of ∇U, one would require a mechanism to compute derivatives of given functions in an automated manner; FOPPL uses reverse-mode automatic differentiation (AD), which is also widely applied in many machine learning systems such as TensorFlow [Abadi et al., 2016] and PyTorch [Paszke et al., 2017].

With both in place, one is then ready to implement HMC shown in Algorithm 2.3 as an automated engine. Specifically, at each iteration, the sampler first draws a fresh sample of the momentum variable ρ according to its corresponding distribution. Then it carries out L leapfrog steps where it updates x and ρ iteratively using gradient information following (2.33)–(2.35). At the end of this iteration, it performs a standard MH accept/reject step, including the evaluation of U and K, to decide whether to accept this movement.

We highlight that one critical question of implementing an HMC engine is ensuring the differentiability of γ. Throughout the discussion, we have assumed this holds but it is usually not the case if a PPL allows discrete ERPs or `if-else` branching statement. For instance, even if we limit all variables to have a continuous prior, the joint distribution γ may not be differentiable as one can easily break the continuity by using `if-else` statements to alternate likelihood functions. We will further the discussion on the effect of branching statement in the following section, and will propose a principled way of working with it when incorporating gradient-based inference methods in a PPS in Chapter 5.

4.5 Interpretation of Branching Statements in Probabilistic Programs

Branching statement is an important programing language construct to express different consequent computations depending on the condition of specific parameters. In a deterministic language, its semantics is usually clear and straightforward. Intuitively, the computation is split into two branches after the **`if`** predicate expression, and we normally take one branch where the predicate holds. When it comes to a PPL, `if-else` branching statement requires some extra considerations. This is because unlike standard execution of a deterministic program, a probabilistic program also defines a probabilistic model with

an unnormalized density γ. How `if-else` is interpreted affects the understanding of the density construction and so the correctness of the inference results.

As introduced in Section 4.2, a PPL usually has two special constructs, **`sample`** and **`observe`** statements, compared to a deterministic programming language. The first scenario of interest is when we allow **`sample`** statements to appear in downstream branches. Normally, latent variables are declared through **`sample`** statements, which determines the support of a model. Most mainstream programming languages evaluate **`if`** expression in a *lazy* manner, that is, only one branch will be evaluated depending on the value of the predicate. This might cause a problem of *varying support* if we have different **`sample`** statements on different downstream branches, and therefore some variables only exist on certain paths.

As an alternative, one may take an *eager* approach to evaluate **`if`** expression, that is, we evaluate both branches but only return the value from the one that the predicate is satisfied. For the variables declared in the branches that are not actually used (even though), one can treat them as auxiliary variables and they, in fact, will not affect the marginal posterior distribution of other variables. It might sound wasteful by carrying around many "inactive" variables, but it makes the support of a model static and explicit, which is important to apply many inference methods such HMC. However, such eager evaluation is only achievable in the scenarios where the number of variables is fixed and the computation graph described by the model is also static. FOPPL, in the standard setup, is able to take the eager approach due to the restrictions in the language where its programs correspond to graphical models. As a result, a FOPPL program will still have a static support even if it has **`sample`** statements in `if-else` branches: the local variables are essentially "lifted" to global, inactive ones in the internal representations G, without affecting the overall joint distribution.

It seems that eager evaluation works well for FOPPL when presented with only downstream **`sample`** statements. Does it hold when we have **`observe`** statements occurring in the `if-else` branches as well? Unlike **`sample`** terms, **`observe`** statements do not generate new variables, but instead factor the joint density with likelihood functions. If one eagerly evaluates both branches with **`observe`** statements and incorporate both likelihood terms, they might take unnecessary likelihood terms into account wrongly and construct

a *wrong* target distribution. How FOPPL resolves this is to assign probability 1 to the observed variables that are not on the active paths. Intuitively, we will only incorporate the likelihood term from the branch where the predicate holds, and only factor 1 to the joint probability when evaluating the other branch (which will not change the value of the joint). By doing so, even if we "evaluate" `observe` statements in an eager way, they won't affect the correctness of the overall distribution.

There is an explicit trade-off between increasing expressivity by allowing branching statements and extra considerations to ensure it works properly. First of all, it can be infeasible to eagerly evaluate all possible branches for a universal probabilistic program as we shall see in Chapter 6 because the number of possible branches may be infinite. In such scenario, the program will have a problem of stochastic support where many conventional inference algorithms cannot be applied anymore. We will investigate this case and discuss corresponding inference options in Chapters 6 and 7. Furthermore, branching statement can break the continuity of the joint density even though we only allow continuous distributions, and it is difficult to conclude whether it does or does not. It is problematic when gradient-based inference methods are used in PPSs since the algorithm may provide a wrong answer or never converge. Such problem exists in FOPPL [van de Meent et al., 2018], Stan [Stan Development Team, 2014] and PyMC3 [Salvatier et al., 2016], where they do not ensure the differentiability of the joint density strictly from the PPL perspective, but rely on the user to be self-cautious. This is undesirable and violates the automation purpose of a PPS at the first place. We will make inroads in resolving this issue briefly in Chapter 5.

5

LF-PPL: A new Low-level First-order Probabilistic Programming Language

Systems introduced in Chapter 4 are somewhat restrictive in the model expressiveness in order to ensure the performance of inference engines. One popular design choice is to exploit gradient-based inference methods, HMC and its variants in particular, and to restrict the models in the system to have a differentiable density. Such restrictions have excluded many non–standard models of interest in real world applications, such as models with discrete variables and if-else branching statements, and therefore, substantially limited the usage of many restricted PPSs. Can we somehow extend such PPSs in a way that they can support a broader range of models while the performance of inference engines is still able to be guaranteed?

Motivated by this question, in this chapter, we develop a new *Low-level, First-order Probabilistic Programming Language* (LF-PPL) [Zhou et al., 2019a], which is suited for the models containing a mix of continuous, discrete, and/or piecewise-continuous variables whose densities may not be fully differentiable. The key success of this language and its compilation scheme is in its ability to automatically distinguish parameters the density function is discontinuous with respect to, while further providing runtime checks for boundary crossings. This enables the introduction of new inference engines that are able to exploit gradient information, while remaining efficient for models which are not

everywhere differentiable. We demonstrate this ability by incorporating a discontinuous Hamiltonian Monte Carlo (DHMC) inference engine that is able to deliver automated and efficient inference for non-differentiable models. Our system is backed up by a mathematical formalism that ensures that any model expressed in this language has a density with measure zero discontinuities to maintain the validity of the inference engine.

5.1 Motivation

Non-differentiable densities arise in a huge variety of common probabilistic models [Mohasel Afshar et al., 2016; Gelman et al., 2013]. Often, but not exclusively, they occur due to the presence of discrete variables. In the context of probabilistic programming [Gordon et al., 2014; Goodman et al., 2008b; Wood et al., 2014; Gelman et al., 2015] such densities are often induced via branching, i.e. `if-else` statements, where the predicates depend upon the latent variables of the model. Unfortunately, performing efficient and scalable inference in models with non-differentiable densities is difficult and algorithms adapted for such problems typically require specific knowledge about the discontinuities [Afshar and Domke, 2015; Nishimura et al., 2017; Lee et al., 2018], such as which variables the target density is discontinuous with respect to and catching occurrences of the sampler crossing a discontinuity boundary. However, detecting when discontinuities occur is difficult and problem dependent. Consequently, automating specialized inference algorithms in probabilistic programming languages (PPLs) is challenging.

To address this problem, we introduce a new Low-level First-order Probabilistic Programming Language (LF-PPL), with a novel accompanying compilation scheme. Our language is based around carefully chosen mathematical constraints, such that the set of discontinuities in the density function of any model written in LF-PPL will have measure zero. This is an essential property for many inference algorithms designed for non-differentiable densities [Afshar and Domke, 2015; Nishimura et al., 2017; Lee et al., 2018; Dinh et al., 2017; Yi and Doshi-Velez, 2017]. Our accompanying compilation scheme automatically classifies discontinuous and continuous random variables for any model specified in our language. Moreover, this scheme can be used to detect transitions across discontinuity boundaries at runtime, providing important information for running such inference schemes.

Relative to previous languages, LF-PPL enables one to incorporate a broader class of specialized inference techniques as automated inference engines. In doing so, it removes the burden from the user of manually establishing which variables the target is not differentiable with respect to. Its low-level nature is driven by a desire to establish the minimum language requirements to support inference engines tailored to problems with measure-zero discontinuities, and to allow for a formal proof of correctness. Though still usable in its own right, our main intention is that it will be used as a compilation target for existing systems, or as an intermediate system for designing new languages.

There are a number of different derivative-based inference paradigms for which LF-PPL can help extend to non-differentiable setups [Afshar and Domke, 2015; Nishimura et al., 2017; Lee et al., 2018; Dinh et al., 2017; Yi and Doshi-Velez, 2017]. Of particular note, are stochastic variational inference (SVI) [Hoffman et al., 2013; Ranganath et al., 2014; Blei et al., 2017; Kucukelbir et al., 2015] and Hamiltonian Monte Carlo (HMC) [Duane et al., 1987; Neal, 2011], two of the most widely used approaches for probabilistic programming inference.

In the context of the former, [Lee et al., 2018] recently showed that the reparameterization trick can be generalized to piecewise differentiable models when the non-differentiable boundaries can be identified, leading to an approach which provides significant improvements over previous methods that do not explicitly account for the discontinuities. LF-PPL provides a framework that could be used to apply their approach in a probabilistic programming setting, thereby paving the way for significant performance improvements for such models. Similarly, many variants of HMC have been proposed in recent years to improve the sample efficiency and scalability when the target density is non-differentiable [Afshar and Domke, 2015; Nishimura et al., 2017; Zhang et al., 2012; Pakman and Paninski, 2013, 2014]. Despite this, no probabilitic programming systems (PPSs) support these tailored approaches at present, as the underlying languages are not able to extract the necessary information for their automation. The novel compilation approach of LF-PPL provides key information for running such approaches, enabling their implementation as automated inference engines. We realize this potential by implementing Discontinuous HMC (DHMC) [Nishimura et al.,

2017] as an inference engine in LF-PPL, allowing for efficient, automated, HMC-based inference in models with a mixture of continuous and discontinuous variables.

5.2 Background and Related Work

There exists a number of different approaches to probabilistic programming that are built around a variety of semantics and inference engines. Of particular relevance to our work are PPSs designed around derivative based inference methods that exploit automatic differentiation [Baydin et al.], such as Stan [Gelman et al., 2015], PyMC3 [Salvatier et al., 2016], Edward [Tran et al., 2017], Turing [Ge et al., 2018b] and Pyro [UberLabs, 2017]. Derivative based inference algorithms have been an essential component in enabling these systems to provide efficient and, in particular, scalable inference, permitting both large datasets and high dimensional models.

One important challenge for these systems occurs in dealing with probabilistic programs that contain discontinuous densities and/or variables. From the statistical perspective, dealing with discontinuities is often important for conducting effective inference. For example, in HMC, discontinuities can cause statistical inefficiency by inducing large errors in the leapfrog integrator, leading to potentially very low acceptance rates [Afshar and Domke, 2015; Nishimura et al., 2017]. In other words, though the leapfrog integrator remains a valid, reversible, MCMC proposal even when discontinuities break the reversibility of the Hamiltonian dynamics themselves, they can undermine the effectiveness of this proposal.

Different methods have been suggested to improve inference performance in models with discontinuous densities. For example, they use sophisticated integrators in the HMC setting to remain effective when there are discontinuities. Analogously, in the variational inference and deep learning literature, reparameterization methods have been proposed that allow training for discontinuous targets and discrete variables [Lee et al., 2018; Maddison et al., 2016].

However, these advanced methods are, in general, not incorporated in existing gradient-based PPSs, as existing systems do not have adequate support to deal with the discontinuities in the density functions of the model defined by probabilistic programs. This is usually necessary to guarantee the correct execution of those inference methods in an automated

fashion, as many require the set of discontinuities to be of measure zero. That is, the union of all points where the density is discontinuous have zero measure with respect to the Lebesgue measure. In addition to this, some further methods require knowledge of where the discontinuities are, or at least catching occurrences of discontinuity boundaries being crossed.

Of particular relevance to our language and compilation scheme are compilers which compile the program to an artifact representing a direct acyclic graphical model (DAG), such as those employed in BUGS [Spiegelhalter et al., 1996] and, in particular, the first order PPL (FOPPL) explored in [van de Meent et al., 2018]. Although the dependency structures of the programs in our language are established in a similar manner, unlike these setups, programs in our language will not always correspond to a DAG, due to different restrictions on our density factors, as will be explained in the next section. We also impose necessary constraints on the language by limiting the functions allowed to ensure that the advanced inference processes remain valid.

5.3 The Language

LF-PPL adopts a Lisp-like syntax, like that of Church [Goodman et al., 2008b] and Anglican [Wood et al., 2014]. The syntax contains two key special forms, **`sample`** and **`observe`**, between which the distribution of the query is defined and whose interpretation and syntax is analogous to that of Anglican.

More precisely, **`sample`** is used for drawing random variables, returning that variable, and **`observe`** factors the density of the program using existing variables and fixed observations, returning `zero`. Both special forms are designed to take a *distribution object* as input, with **`observe`** further taking an observed value. These distribution objects form the elementary random procedures of the language and are constructed using one of a number of internal constructors for common objects such as **`normal`** and **`bernoulli`**. Figure 5.1 shows an example of an LF-PPL program.

A distribution object constructor of particular note is **`factor`**, which can only be used with **`observe`**. Including the statement `(observe (factor log-p) _)` will factor the program density using the value of `(exp log-p)`, with no dependency on the observed value itself (here _). The significance of **`factor`** is that it allows the specification of

```
(let [x (sample (uniform 0 1))]
   (if (< (- q x) 0)
      (observe (normal 1 1) y)
      (observe (normal 0 1) y))
   (< (- q x) 0))
```

Figure 5.1: An example LF-PPL program sampling x from a uniform random variable and invoking a choice between two **observe** statements that factor the trace weight using different Gaussian likelihoods. The `(< (- q x) 0)` term, which is usually written as $((q - x) < 0)$, represents a Bernoulli variable parameterized by q and its boolean value also corresponds to which branch of the **if** statement is taken. The slightly unusual writing of the program is due to its deliberate low-level nature, with almost all syntactic sugar removed. One sugar that has been left in for exposition is an additional term in the let block, i.e. `(let [x e] e e)`, which can be trivially unraveled.

arbitrary unnormalized target distributions, quantified as `log-p` which can be generated internally in the program, and thus have the form of any deterministic function of the variables that can be written in the language.

Unlike many first-order PPLs, such as that of van de Meent et al. [2018], LF-PPL programs do not permit interpretation as DAGs because we allow the observation of internally sampled variables and the use of **factor**. This increases the range of models that can be encoded and is, for example, critical in allowing undirected models to be written. LF-PPL programs need not correspond to a correctly normalized joint formed by the combination of prior and likelihood terms. Instead we interpret the density of a program in the manner outlined by [Rainforth, 2017, §4.3.2 and §4.4.3], noting that for any LF-PPL program, the number of **sample** and **observe** statements (i.e. n_x and n_y in their notation) must be fixed, a restriction that is checked during the compilation.

To formalize the syntax of LF-PPL, let us use x for a real-valued variable, c for a real number, **op** for an analytic primitive operation on reals, such as **+**, **-**, *****, **/** and **exp**, and d for a distribution object whose density is defined with respect to a Lebesgue measure and is piecewise smooth under analytic partition (See Definition 1). Then the syntax of expressions e in our language are given as:

$$e ::= x \mid c \mid (\textbf{op}\ e\ \ldots\ e) \mid (\textbf{if}\ (<\ e\ 0)\ e\ e) \mid (\textbf{let}\ [x\ e]\ e)$$
$$\mid (\textbf{sample}\ (d\ e\ \ldots\ e)) \mid (\textbf{observe}\ (d\ e\ \ldots\ e)\ c)$$

Our syntax is deliberately low-level to permit theoretical analysis and aid the exposition of the compiler. However, common syntactic sugar such as `for`-loops and higher-

level branching statements can be trivially included using straightforward unravellings. Similarly, we can permit discrete variable distribution objects by noting that these can themselves be desugared to a combination of continuous random variables and branching statements. Thus, it is straightforward to extend this minimalistic framework to a more user-friendly language using standard compilation approaches, such that LF-PPL will form an intermediate representation.

5.4 Compilation Scheme

We now provide a high-level description of how the compilation process works. Specifically, we will show how it transforms an arbitrary LF-PPL program to a representation that can be exploited by an inference engine that makes of use of discontinuity information.

The compilation scheme performs three core tasks: a) finding the variables which the target is discontinuous with respect to, b) extracting the density of the program to a convenient form that can be used by an inference engine, and c) allowing boundary crossings to be detected at runtime. Key to providing these features is the construction of an internal representation of the program that specifies the dependency structure of the variables, the *Linearized Intermediate Representation* (LIR). The LIR contains vertices, arc pairs, and information of the `if` predicates. Each vertex of the LIR denotes a `sample` or `observe` statement, of which only a finite and fix number can occur in LF-PPL. The arcs of the LIR define both the probabilistic and `if` condition dependencies of the variables. The former of these are constructed in same was as is done in the FOPPL compiler detailed in van de Meent et al. [2018].

Using the dependency structure represented by the LIR, we can establish which variables are capable of changing the path taken by a program trace, that is the change the branch taken by one or more `if` statements. Because discontinuities only occur in LF-PPL through `if` statements, the target must be continuous with respect to any variables not capable of changing the traversed path. We can thus mark these variables as being “continuous”. Though it is possible for the target to still be continuous with respect to variables that appear in, or have dependent variables appearing in, the branching function of an `if`

statement, such cases cannot, in general, be statically established. We therefore mark all such variables as "discontinuous".

To extract the density to a convenient form for the inference engine, the compiler transforms the program into a collection four sets—Δ, Γ, D, and F—by recursively applying the translation rules given in Section 5.5.2. Here Δ specifies the set of all variables sampled in the program, while Γ specifies only the variables marked as discontinuous. D represents the density associated with all the `sample` statements in a program, while F represents the density factors originating for the `observe` statements, along with information on the program return value. These densities are themselves represented through a collection of smooth density terms and indicator functions truncating them into disjoint regions, each corresponding to a particular program path. This construction will be discussed in depth in Section 5.5.2.

To catch boundary crossings at run time, each `if` predicate is assigned a unique boolean variable within the LIR. We refer to these variables as *branching variables*. The boolean value of the branching variable denotes whether the current sample falls into the `true` or `false` branch of the corresponding `if` statement and is used to signal boundary crossings at runtime. Specifically, if one branching variable changes its boolean value, this indicates that at least one sampled variables effecting that `if` predicate has crossed the boundary. The inference engine can therefore track changes in the set of all Boolean values to catch the boundary crossings.

We finish the section by noting two limitations of the compiler and for discontinuity detection more generally. Firstly, we note that it is possible to construct programs which have piecewise smooth densities that contain regions of zero density. Though it is important to allow this ability, for example to construct truncated distributions, it may cause issues for certain inference algorithms if it causes the target to have disconnected regions of non-zero density. As analytic densities are either zero everywhere or "almost-nowhere" (see Section 7.5), we (informally) have that all realizations of a program that take a particular path will either have zero density or all have a non-zero density. Consequently, it is relatively straight forward to establish if a program has regions of zero density. However, whether these regions lead to "gaps" is far more challenging, and potentially impossible, to establish.

Moreover, constructing inference procedures for such problems is extremely challenging. We therefore do not attempt to tackle this issue in the current work.

A second limitation is that changes in the vector of branching variables is only a *sufficient* condition for the occurrence of a boundary crossing. This is because it is possible for *multiple* boundaries to be crossed in a single update that results in the new sample following the same path as the old one. For example, when moving from $x = -0.5$ to $x = 1.5$ then a branching variable corresponding to $x^3 - x > 0$ returns `true` in both cases even though we have crossed two boundaries. The problem of establishing with certainty that *no* boundaries have been crossed when moving between two points is mathematically intractable in the general case. As this problem is not specific to the probabilistic programming setting, we do not give it further consideration here, noting only that it is important from the perspective of designing inference algorithms that convergence is not undermined by such occurrences.

5.5 Mathematical Foundation and Compilation Details

Our story so far was developed by introducing a low-level first-order probabilistic programming language (LF-PPL) and its accompanying compilation scheme. We shall now expose the underlying mathematical details, which ensure that discontinuities contained within the densities of the programs one can compile in LF-PPL are of a suitable measure. This enables us to satisfy the requirements of several inference algorithms for non-differentiable densities. We also provide the formal translation rules of the LF-PPL, which are built around these mathematical underpinnings.

5.5.1 Piecewise Smooth Functions

A function $\mathcal{G} : \mathbb{R}^k \to \mathbb{R}$ is *analytic* if it is infinitely differentiable and its multivariate Taylor expansion at any point $x_0 \in \mathbb{R}^k$ absolutely converges to $\mathcal{G}$ point-wise in a neighborhood of x_0. Most primitive functions that we encounter in machine learning and statistics are analytic, and the composition of analytic functions is also analytic.

Definition 1. *A function $\mathcal{G} : \mathbb{R}^k \to \mathbb{R}$ is* piecewise smooth under analytic partition *if it has the following form:*

$$\mathcal{G}(x) = \sum_{i=1}^{N} \left(\prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0] \cdot h_i(x) \right)$$

where

1. *the* $p_{i,j}, q_{i,l} : \mathbb{R}^k \to \mathbb{R}$ *are analytic;*
2. *the* $h_i : \mathbb{R}^k \to \mathbb{R}$ *are smooth;*
3. N *is a positive integer or* ∞*;*
4. M_i, O_i *are non-negative integers; and*
5. *the indicator functions*

$$\prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0]$$

for the indices i *define a partition of* $\mathbb{R}^k$*, that is, the following family forms a partition of* $\mathbb{R}^k$*:*

$$\left\{ \left\{ x \in \mathbb{R}^k \;\middle|\; \forall j\, p_{i,j}(x) \geq 0,\ \forall l\, q_{i,l}(x) < 0 \right\} \;\middle|\; 1 \leq i \leq N \right\}.$$

Intuitively, $\mathcal{G}$ is a function defined by partitioning $\mathbb{R}^k$ into finitely or countably many regions and using a smooth function h_i within region i. The products of the indicator functions of these summands form a partition of $\mathbb{R}^k$, so that only one of these products gets evaluated to a non-zero value at x. To evaluate the sum, we just need to evaluate these products at x one-by-one until we find one that returns a non-zero value. Then, we have to compute the function h_i corresponding to this product at the input x. Even though the number of summands (regions) N in the definition is countably infinite, we can still compute the sum at a given x.

Theorem 2. *If an unnormalized density* $\mathcal{P} : \mathbb{R}^n \to \mathbb{R}_+$ *has the form of Definition 1 and so is piecewise smooth under analytic partition, then there exists a (Borel) measurable subset* $A \subseteq \mathbb{R}^n$ *such that* $\mathcal{P}$ *is differentiable outside of* A *and the Lebesgue measure of* A *is zero.*

Proof. Assume that $\mathcal{P}$ is piecewise smooth under analytic partition. Thus,

$$\mathcal{P}(x) = \sum_{i=1}^{N} \prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0] \cdot h_i(x) \tag{5.1}$$

for some N, M_i, O_i and $p_{i,j}, q_{i,l}, h_i$ that satisfy the properties in Definition 1.

We use one well-known fact: the zero set $\{x \in \mathbb{R}^n \mid p(x) = 0\}$ of an analytic function p is the entire $\mathbb{R}^n$ or has zero Lebesgue measure [Mityagin, 2015]. We apply the fact to each

$p_{i,j}$ and deduce that the zero set of $p_{i,j}$ is $\mathbb{R}^n$ or has measure zero. Note that if the zero set of $p_{i,j}$ is the entire $\mathbb{R}^n$, the indicator function $\mathbb{1}[p_{i,j} \geq 0]$ becomes the constant-1 function, so that it can be omitted from the RHS of equation (5.1). In the rest of the proof, we assume that this simplification is already done so that the zero set of $p_{i,j}$ has measure zero for every i, j.

For every $1 \leq i \leq N$, we decompose the i-th region

$$R_i = \{x \mid p_{i,j} \geq 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l\} \tag{5.2}$$

to

$$\begin{aligned} R'_i &= \{x \mid p_{i,j} > 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l\} \\ R''_i &= R_i \setminus R'_i. \end{aligned} \tag{5.3}$$

Note that R'_i is open because the $p_{i,j}$ and $q_{i,l}$ are analytic and so continuous, both $\{r \in \mathbb{R} \mid r > 0\}$ and $\{r \in \mathbb{R} \mid r < 0\}$ are open, and the inverse images of open sets by continuous functions are open. This means that for each $x \in R'_i$, we can find an open ball at x inside R'_i so that $\mathcal{P}(x') = h_i(x')$ for all x' in the ball. Since h_i is smooth, this implies that $\mathcal{P}$ is differentiable at every $x \in R'_i$.

For the other part R''_i, we notice that

$$R''_i \subseteq \bigcup_{j=1}^{M_i} \{x \mid p_{i,j}(x) = 0\}.$$

The RHS of this equation is a finite union of measure-zero sets, so it has measure zero. Thus, R''_i also has measure zero as well.

Since $\{R_i\}_{1 \leq i \leq N}$ is a partition of $\mathbb{R}^n$, we have that

$$\mathbb{R}^n = \bigcup_{i=1}^{N} R'_i \cup \bigcup_{i=1}^{N} R''_i.$$

The density $\mathcal{P}$ is differentiable on the union of R'_i's. Also, since the union of finitely or countably many measure-zero sets has measure zero, the union of R''_i's has measure zero. Thus, we can set the set A required in the theorem to be this second union. □

The target density being almost everywhere differentiable with discontinuities of measure zero is an important property required by many inference techniques for non-differentiable models [Nishimura et al., 2017]. As we shall prove in Section 5.5.2, any

program that can be compiled in LF-PPL constructs a density in the form of Definition 1, and thus satisfies this necessary condition.

5.5.2 Translation Rules

5.5.2.1 Overview

The compilation scheme $e \rightsquigarrow (\Delta, \Gamma, D, F)$ translates a program, which can be denoted as an expression e according to the syntax in Section 5.3, to a quadruple of sets (Δ, Γ, D, F). The first set Δ represents the set of all sampled random variables. All variables generated from `sample` statements in e will be recognized and stored in Δ. Variables that have not occurred in any `if` predicate are guaranteed to be continuous. Otherwise, they will be also put in $\Gamma \subseteq \Delta$, as the overall density is discontinuous with respect to them. D represents the densities from `sample` statements and has the form of a set of the pairs, i.e. $D = \{(\eta_1, k_1), \ldots, (\eta_{N_D}, k_{N_D})\}$, where N_D is the number of the pairs, η denotes a product of indicator functions indicating the partition of the space, and k represents the products of the densities defined by the `sample` statements. The last set F contains the densities from `observe` statements and the return expression of e. It is a set of tuples $F = \{(\zeta_1, l_1, v_1), \ldots, (\zeta_{N_F}, l_{N_F}, v_{N_F})\}$, where N_F is the number of the tuples, ζ functions similar to η, l is the product of the densities defined by `observe` statements and v denotes the returning expression. Note that it is a design choice to have v included in F.

Given $e \rightsquigarrow (\Delta, \Gamma, D, F)$, one can then construct the unnormalized density defined by the program e as

$$\mathcal{P} := \Big(\sum_{i=1}^{N_D} \eta_i \cdot k_i \Big) \cdot \Big(\sum_{j=1}^{N_F} \zeta_j \cdot l_j \Big) \tag{5.4}$$

which by Theorem 3 will be piecewise smooth under analytic partitions.

Recall that by assumption, the density of each distribution type d is piecewise smooth under analytic partition when viewed as a function of a sampled value and its parameters. Thus, we can assume that the probability density of a distribution has the form in Definition 1. For each distribution d, we define a set of pairs $\Phi^{(d)} = \{(\psi_1, \phi_1), \ldots, (\psi_{N_\Phi}, \phi_{N_\Phi})\}$ where N_Φ is the number of the partitions, ψ denotes the product of indicator functions indicating the partition of the space, taking the form of $\prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(\mathbf{x}) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(\mathbf{x}) < 0]$, and ϕ represents a smooth probability density function within that partition. One can then construct

the probability density function $\mathcal{P}_d$ for d from $\Phi^{(d)}$. For given parameters $x_1, \ldots, x_s$ of the distribution d and a given `sample` value x_0, we let $\mathbf{x} = (x_0, \ldots, x_s)$ and the probability density function defined by d is,

$$\mathcal{P}_d(x_0; x_1, \ldots, x_s) = \sum_{n=1}^{N_\Phi} \psi_n(\mathbf{x}) \cdot \phi_n(\mathbf{x})$$

For example, given a sample x_0 drawn from normal distribution $\mathcal{N}(\mu, \sigma)$, we have $\Phi^{(d)} = \{(1, \mathcal{N}(x_0\,; \mu, \sigma))\}$ and $\mathcal{P}_d(x_0; \mu, \sigma) = \mathcal{N}(x_0\,; \mu, \sigma)$. Similarly a uniform $\mathcal{U}(a, b)$ sampled variable x_0 has $\Phi^{(d)}$ as

$$\Big\{ (\mathbb{1}[x_0{-}a{<}0],\, 0)\,,\ (\mathbb{1}[b{-}x_0{<}0],\, 0)\,,\ (\mathbb{1}[x_0{-}a{\geq}0]{\cdot}\mathbb{1}[b{-}x_0{\geq}0],\, \mathcal{U}(x_0; a, b)) \Big\},$$

and $\mathcal{P}_d = \mathbb{1}[x_0{-}a{\geq}0]{\cdot}\mathbb{1}[b{-}x_0{\geq}0] \cdot \mathcal{U}(x_0; a, b)$. Note that in practice one can omit the pair (ψ_n, ϕ_n) in $\Phi^{(d)}$ when $\phi_n{=}\,0$ for simplicity and the probability density in the region denoting by the corresponding ψ_n is zero.

5.5.2.2 Formal Translation Rules

The translation process $e \rightsquigarrow (\Delta, \Gamma, D, F)$, is defined recursively on the structure of e. We present this recursive definition using the following notation

$$\frac{\text{premise}}{\text{conclusion}}$$

which says that if the premise holds, then the conclusion holds too. Also, for real-valued functions $f(x_1, \ldots, x_n)$ and $f'(x_1, \ldots, x_n)$ on real-valued inputs, we write $f[x_i := f']$ to denote the composition $f(x_1, \ldots, x_{i-1}, f'(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$. We now define the formal translation rules.

The first two rules define how we map the set of variables x and the set of constants c, to their unnormalized density and the values at which they are evaluated.

$$\frac{}{x \rightsquigarrow (\{x\}, \emptyset,\ \{(1,1)\},\ \{(1,1,x)\})}$$

$$\frac{}{c \rightsquigarrow (\emptyset, \emptyset,\ \{(1,1)\},\ \{(1,1,c)\})}$$

The third rule allows one to translate the primitive operations `op` defined in the LF-PPL, such as `+`, `-`, `*` and `/` with their argument expressions e_1 to e_n, where e_1 to e_n will be evaluated first. Note that $(\eta_i, k_i) \in D_i$ represents the enumeration of all (η_i, k_i) pairs in D_i and the result of this operation among all the D_i is the possible combination of all their elements. For example, given three sets D_1, D_2 and D_3 which have three, one

and two pairs respectively as their elements, the result set D' will have six pairs. This notation holds to the rest of the paper.

$$\frac{\begin{array}{l} e_i \rightsquigarrow (\Delta_i, \Gamma_i,\ D_i,\ F_i) \text{ for } 1 \leq i \leq n \\ D' = \{(\prod_{i=1}^n \eta_i,\ \prod_{i=1}^n k_i) \mid (\eta_i, k_i) \in D_i\} \\ F' = \{(\prod_{i=1}^n \zeta_i,\ \prod_{i=1}^n l_i,\ \texttt{op}\,(v_1, \ldots, v_n)) \mid (\zeta_i, l_i, v_i) \in F_i\} \end{array}}{(\texttt{op}\ e_1\ \ldots\ e_n) \rightsquigarrow (\bigcup_{i=1}^n \Delta_i, \bigcup_{i=1}^n \Gamma_i,\ D',\ F')}$$

The fourth rule for control flow operation **if** enables us to translate the predicate $(<\ e_1\ 0)$, its consequent e_2 and alternative e_3. This provides us with the semantics to correctly construct a piecewise smooth function, that can be evaluated at each of the partitions.

$$\frac{\begin{array}{l} e_i \rightsquigarrow (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, 2, 3 \\ D' = \{(\prod_{i=1}^3 \eta_i,\ \prod_{i=1}^3 k_i) \mid (\eta_i, k_i) \in D_i\} \\ F' = \{(\zeta_1 \cdot \zeta_2 \cdot \mathbb{1}[v_1 < 0],\ l_1 \cdot l_2,\ v_2), (\zeta_1 \cdot \zeta_3 \cdot \mathbb{1}[v_1 \geq 0],\ l_1 \cdot l_3,\ v_3) \mid (\zeta_i, l_i, v_i) \in F_i\} \end{array}}{(\texttt{if}\ (<\ e_1\ 0)\ e_2\ e_3) \rightsquigarrow (\bigcup_{i=1}^3 \Delta_i, \Delta_1 \cup \Gamma_2 \cup \Gamma_3,\ D',\ F')}$$

The translation rule for the **sample** statement generates a random variable from a specific distribution. During translation, we pick a fresh variable, i.e. a variable with a unique name to represent this random variable and add it to the Δ set. Then we compose the density of this variable according to the distribution d and corresponding parameters e_i.

$$\frac{\begin{array}{l} e_i \rightsquigarrow (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, \ldots, n \\ \text{pick a fresh variable } z \\ \Delta' = \{z\} \cup \bigcup_{i=1}^n \Delta_i, \quad \Gamma' = \bigcup_{i=1}^n \Gamma_i \\ D_0 = \{(\psi \cdot \prod_{i=1}^n \zeta_i,\ \phi[\mathbf{x} := (z, v_1, \ldots, v_n)]) \mid (\psi, \phi) \in \Phi^{(d)}, (\zeta_i, l_i, v_i) \in F_i\} \\ D' = \{(\prod_{i=0}^n \eta_i,\ \prod_{i=0}^n k_i) \mid (\eta_i, k_i) \in D_i\} \\ F' = \{(\prod_{i=1}^n \zeta_i,\ \prod_{i=1}^n l_i,\ z) \mid (\zeta_i, l_i, v_i) \in F_i\} \end{array}}{(\texttt{sample}\ (d\, e_1\ \ldots\ e_n)) \rightsquigarrow (\Delta', \Gamma', D', F')}$$

The translation rule for the **observe** statement, different from the **sample** expression, factors the density according to the distribution object, with all parameters e_i and the observed data c evaluated.

$$\frac{\begin{array}{l} e_i \rightsquigarrow (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, \ldots, n \\ \Delta' = \bigcup_{i=1}^n \Delta_i, \quad \Gamma' = \bigcup_{i=1}^n \Gamma_i \\ D' = \{(\prod_{i=1}^n \eta_i,\ \prod_{i=1}^n k_i) \mid (\eta_i, k_i) \in D_i\} \\ F' = \{(\psi \cdot \prod_{i=1}^n \zeta_i,\ \phi[\mathbf{x} := (c, v_1, \ldots, v_n)] \cdot \prod_{i=1}^n l_i,\ 0) \mid (\psi, \phi) \in \Phi^{(d)}, (\zeta_i, l_i, v_i) \in F_i\} \end{array}}{(\texttt{observe}\ (d\, e_1\ \ldots\ e_n)\ c) \rightsquigarrow (\Delta', \Gamma', D', F')}$$

The translation rule for **let** expressions first translates the definition e_1 of x and the body e_2 of **let**, and then joins the results of these translations. When joining the Δ and Γ sets,

the rule checks whether x appears in the sets from the translation of e_2, and if so, it replaces x by variable names appearing in e_1, an expression that defines x. Although **`let`** is defined as single binding, we can construct the rules to translate the **`let`** expression, defining and binding multiple variables by properly *desugaring*.

$$\frac{\begin{array}{l} e_i \rightsquigarrow (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, 2 \\ \Delta_0 = \{z \mid (\zeta_1, l_1, v_1) \in F_1 \text{ and } z \text{ occurs free in } v_1\} \\ \Delta' = \Delta_1 \cup (\Delta_2 \setminus \{x\}) \cup (\text{if } (x \in \Delta_2) \text{ then } \Delta_0 \text{ else } \emptyset) \\ \Gamma' = \Gamma_1 \cup (\Gamma_2 \setminus \{x\}) \cup (\text{if } (x \in \Gamma_2) \text{ then } \Delta_0 \text{ else } \emptyset) \\ D' = \{(\zeta_1 \cdot \eta_1 \cdot \eta_2[x := v_1],\ k_1 \cdot k_2[x := v_1]) \mid (\eta_i, k_i) \in D_i, (\zeta_1, l_1, v_1) \in F_1\} \\ F' = \{(\zeta_1 \cdot \zeta_2[x := v_1],\ l_1 \cdot l_2[x := v_1],\ v_2[x := v_1]) \mid (\zeta_i, l_i, v_i) \in F_i\} \end{array}}{(\textbf{let}\ [x\ e_1]\ e_2) \rightsquigarrow (\Delta',\ \Gamma',\ D',\ F')}$$

Theorem 3. *If e is an expression that does not contain any free variables and $e \rightsquigarrow (\Delta, \Gamma,\ D,\ F)$, then the unnormalized density defined by e is in the form of Equation 5.4. It is a real-valued function on the variables in Δ, which is non-negative and piecewise smooth under analytic partition as per Definition 1.*

Proof. As shown in Equation 5.4,

$$\mathcal{P} := \left(\sum_{i=1}^{N_D} \eta_i \cdot k_i\right) \cdot \left(\sum_{j=1}^{N_F} \zeta_j \cdot l_j\right)$$

it suffices to show that both factors are non-negative and piecewise smooth under analytic partition, because such functions are closed under multiplication.

We prove a more general result. For any expression e, let $\mathrm{Free}(e)$ be the set of its free variables. Also, if a function $\mathcal{G}$ in Definition 1 satisfies additionally that its h_i's are analytic, we say that this function $\mathcal{G}$ is *piecewise analytic* under analytic partition. We claim that for all expressions e (which may contain free variables), if $e \rightsquigarrow (\Delta, \Gamma, D, F)$, where $D = \{(\eta_i, k_i) \mid 1 \leq i \leq N_D\}$ and $F = \{(\zeta_j, l_j, v_j) \mid 1 \leq j \leq N_F\}$, then $\left(\sum_{i=1}^{N_D} \eta_i \cdot k_i\right)$ and $\left(\sum_{j=1}^{N_F} \zeta_j \cdot l_j\right)$ are non-negative functions on variables in $\mathrm{Free}(e) \cup \Delta$ and they are piecewise analytic under analytic partition, as k and l' in the sum are analytic. These two properties in turn imply that $\left(\sum_{i=1}^{N_D} \eta_i \cdot k_i\right) \cdot \left(\sum_{j=1}^{N_F} \zeta_j \cdot l_j\right)$ is a function on variables in $\mathrm{Free}(e) \cup \Delta$ and it is also piecewise analytic (and thus piecewise smooth) under analytic partition. Thus, the desired conclusion follows. Regarding our claim, we can prove it by induction on the structure of the expression e. □

By providing this set of mathematical translations we have been able to prove that any such program written in LF-PPL constructs a density in the form of Definition 1, which is piecewise smooth under analytic partitions. Together with Theorem 2, we further show that this density is almost everywhere differentiable and the discontinuities are of measure zero, a necessary condition for several inference schemes such as DHMC [Nishimura et al., 2017].

5.5.3 A Compilation Example

We now present a simple example of how the compiler transforms the program e_{pp} in Figure 5.1 to the quadruple $(\Delta_{pp}, \Gamma_{pp}, D_{pp}, F_{pp})$. The translation rules are applied recursively and within each rule, all individual components are compiled eagerly first. Namely, we step into each individual component and step out until it is fully compiled. A desugared version of e_{pp} is:

```
(let [x (sample (uniform 0 1))]
    (let [x_ (if (< (- q x) 0)
                 (observe (normal 1 1) y)
                 (observe (normal 0 1) y))]
       (< (- q x) 0)))
```

where q and y are constant and $x\_$ is not used. It follows the following steps.

i. *Rule* (**let** $[x\ e_{1,out}]\ e_{2,out}$). We start by looking at the outer let expressions, with $e_{1,out}$ being the **sample** statement and $e_{2,out}$ corresponding to the entire inner **let** block. Before we can generate the output of this rule, we step into $e_{1,out}$ and $e_{2,out}$ and compile them accordingly.

ii. *Rule* (**sample** $(d\ e_1\ e_2)$). We then apply the **sample** rule on $e_{1,out}$ from i with each of its components evaluated first, for which we have $e_{1,out}$:= `(sample (uniform 0 1))`. For `(uniform 0 1)`, 0 and 1 are constant and we have $0 \rightsquigarrow (\emptyset, \emptyset, \{(1,1)\}, \{(1,1,0)\})$ and $1 \rightsquigarrow (\emptyset, \emptyset, \{(1,1)\}, \{(1,1,1)\})$. d represents **uniform** distribution and has the form $\Phi^{(d)} = \{(\mathbb{1}[x \geq 0] \cdot \mathbb{1}[1-x \geq 0], \mathcal{U}(\cdot\,; 0, 1))\}$. After combining each set following the rule, with a fresh variable z, we have $e_{1,out} \rightsquigarrow \Big(\{z\}, \emptyset, \{(\mathbb{1}[z \geq 0] \cdot \mathbb{1}[1-z \geq 0], \mathcal{U}(z; 0, 1))\}, \{(1,1,z)\}\Big)$.

iii. *Rule* (**let** $[x\ e_{1,in}]\ e_{2,in}$). We now step into $e_{2,out}$ from i with itself being a **let** expression. $e_{1,in}$ is the entire **if** statement and $e_{2,in}$ is the returning value `(< (- q x) 0)`. Similarly, we need to compile $e_{1,in}$ and $e_{2,in}$ first before having the result for $e_{2,out}$.

iv. *Rule* $(\texttt{if}\ (<\ e_1\ 0)\ e_2\ e_3)$. To apply the `if` rule on $e_{1,in}$, we again need to compile its each individual component first. We start with its predicate e_1:=`(- q x)`, which follows the rule $(\texttt{op}\ e_1\ e_2)$. Then $e_1 \rightsquigarrow \big(\{x\}, \emptyset, \{(1,1)\}, \{(1,1,(q-x))\}\big)$ with $(q-x)$ as a operation `-` applied to q and x. e_2 and e_3 both follow $(\texttt{observe}\ (d\ e_1\ e_2)\ c)$. Take $e_2 := (\texttt{observe}\ (\texttt{normal}\ 1\ 1)\ y)$ as an example, 1 is constant and d is the `normal` distribution and has $\Phi^{(d)} = \{(1, \mathcal{N}(\cdot\,; 1, 1))\}$. Combining each set gives us $e_2 \rightsquigarrow \big(\emptyset, \emptyset, \{(1,1)\}, \{(1, \mathcal{N}(y;\, 1, 1), 0)\}\big)$. Similarly, we can obtain $e_3 \rightsquigarrow (\emptyset, \emptyset, \{(1,1)\}, \{(1, \mathcal{N}(y;\, 0, 1), 0)\})$. With e_1, e_2 and e_3 all evaluated, we can now continue the `if` rule. The key features are to extract variables in e_1 and put into Γ and to construct the indicator functions from e_1 and take the densities on each branch respectively. As a result, $e_{1,in}$ compiles to $\Delta = \{x\},\ \Gamma = \{x\},\ D = \{(1,1)\}$ and $F = \Big\{\big(\mathbb{1}[q{-}x{<}0], \mathcal{N}(y;1,1), 0\big), \big(\mathbb{1}[q{-}x{\geq}0], \mathcal{N}(y;0,1), 0\big)\Big\}$.

v. *Rule* $(\texttt{op}\ e_1\ \ldots\ e_n)$. For $e_{2,in}$ in iii, `(< (- q x) 0)` compiles to the following quadruples, $(\{x\}, \emptyset, \{(1,1)\}, \{(1,1,(q{-}x<0))\})$.

vi. *Result of the inner* `let`. Together with the outcome from iv and v, we can continue compiling the inner `let` block as in iii, and it is translated to

$$\begin{aligned}
&\Delta = \{x\},\ \Gamma = \{x\},\\
&D = \Big\{(\mathbb{1}[q{-}x < 0],\ 1), (\mathbb{1}[q{-}x \geq 0],\ 1)\Big\}\\
&F = \Big\{(\mathbb{1}[q{-}x < 0],\ \mathcal{N}(y;\ 1,\ 1), (q{-}x < 0)),\\
&\qquad (\mathbb{1}[q{-}x \geq 0],\ \mathcal{N}(y;\ 0,\ 1), (q{-}x < 0))\Big\}
\end{aligned}$$

vii. *Result of the outer* `let`. Finally, with $e_{1,out}$ compiled in ii and $e_{2,out}$ in vi, we step out to i. It is worth to emphasize that the variables Δ are the sampled ones rather than what are named in the `let` expression, i.e. x and $x\_$. Here x is replaced by z as declared in $e_{1,out}$ by following the `let` rule, and we have the final quadruple output:

$$\begin{aligned}
&\Delta_{pp} = \{z\},\ \Gamma_{pp} = \{z\},\\
&D_{pp} = \Big\{\big(\mathbb{1}[z{\geq}0]{\cdot}\mathbb{1}[1{-}z{\geq}0]{\cdot}\mathbb{1}[q{-}z{<}0], \mathcal{U}(z;0,1)\big),\\
&\qquad \big(\mathbb{1}[z{\geq}0]{\cdot}\mathbb{1}[1{-}z{\geq}0]{\cdot}\mathbb{1}[q{-}z{\geq}0], \mathcal{U}(z;0,1)\big)\Big\},\\
&F_{pp} = \Big\{\big(\mathbb{1}[q{-}z{<}0], \mathcal{N}(y;\ 1,\ 1), (q{-}z{<}0)\big),\\
&\qquad \big(\mathbb{1}[q{-}z{\geq}0], \mathcal{N}(y;\ 0,\ 1), (q{-}z{<}0)\big)\Big\}
\end{aligned}$$

From the quadruple, we have the overall density as

$$\mathcal{P} = \mathbb{1}[z \geq 0] \cdot \mathbb{1}[1 - z \geq 0] \cdot \mathbb{1}[q - z < 0] \cdot \mathcal{U}(z; 0, 1) \cdot \mathcal{N}(y; 1, 1) +$$
$$\mathbb{1}[z \geq 0] \cdot \mathbb{1}[1 - z \geq 0] \cdot \mathbb{1}[q - z \geq 0] \cdot \mathcal{U}(z; 0, 1) \cdot \mathcal{N}(y; 0, 1).$$

We can also detect when any random variable in Γ, in this case z, has crossed the discontinuity, by checking the boolean value of the predicate of the **if** statement `(< (- q x) 0)`, as discussed in Section 5.4.

5.6 Example Inference Engine: Discontinuous HMC

We shall now demonstrate an example inference algorithm that is compatible with LF-PPL. Specifically, we provide an implementation of discontinuous HMC (DHMC) [Nishimura et al., 2017], a variant of HMC for performing statistically efficient inference on probabilistic models with non-differentiable densities, using LF-PPL as a compilation target. This satisfies the necessary requirement of DHMC that the target density being piecewise smooth with discontinuities of measure zero. Given the quadruple output from LF-PPL, DHMC updates variables in Γ by the coordinate-wise integrator and the rest of the variables in $\Delta \backslash \Gamma$ by the standard leapfrog integrator. In an existing PPS without a special support, the user would be required to manually specify all the discontinuous and continuous variables, in addition to implementing DHMC accordingly. See Appendix B.1 for further details.

5.6.1 Gaussian Mixture Model

In our first example, we demonstrate how a classic model, namely a Gaussian mixture model (GMM), can be encoded in LF-PPL. The density of the GMM contains a mixture of continuous and discrete variables, where the discrete variables lead to discontinuities in the density. We construct the GMM as follows:

$$\mu_k \sim \mathcal{N}(\mu_0, \sigma_0), \ k = 1, \ldots, K$$
$$z_n \sim \text{Categorical}(p_0), \ n = 1, \ldots, N$$
$$y_n \,|z_n, \mu_{z_n} \sim \mathcal{N}(\mu_{z_n}, \sigma_{z_n}), \ n = 1, \ldots, N$$

where $\mu_{1:K}, z_{1:N}$ are latent variables, $y_{1:N}$ are observed data with K as the number of clusters and N the total number of data. The Categorical distribution is constructed by a combination of uniform draws and nested **if** expressions, as shown in Appendix B.2. For our experiments, we considered a simple case with $\mu_0 = 0$, $\sigma_0 = 2$, $\sigma_{z_{1:N}} = 1$ and $p_0 = [0.5, 0.5]$,

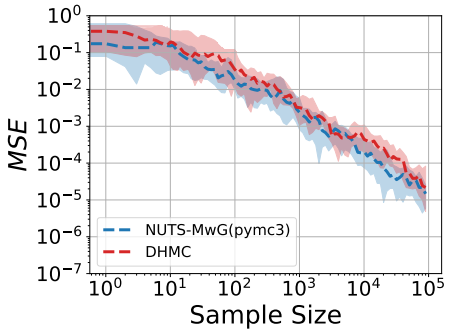


Figure 5.2: Mean Squared Error for the posterior estimates of the true posterior of the cluster means $\mu_{1:2}$. We compare the results from our unoptimized DHMC and the optimized PyMC3 NUTS with Metropolis-within-Gibbs, and show that the performance between the two is comparable for the same computation budget. The median of MSE (dashed lines) with $20\%/80\%$ confidence intervals (shaded regions) over 20 independent runs are plotted.

along with the synthetic dataset: $y_{1:N} = [-2.0, -2.5, -1.7, -1.9, -2.2, 1.5, 2.2, 3, 1.2, 2.8]$. We compared the Mean Squared Error (MSE) of the posterior estimates for the cluster means of both an unoptimized version of DHMC and an optimized implementation of NUTS with Metropolis-within-Gibbs (MwG) in PyMC3 [Salvatier et al., 2016], with the same computation budget. We take 10^5 samples and discard 10^4 for burn in. We find that our DHMC implementation, performs comparable to the NUTS with MwG approach. The results are shown in Figure 5.2 as a function of the number of samples.

5.6.2 Heavy Tail Piecewise Model

In our next example we show how the efficiency of DHMC improves, relative to vanilla HMC, on discontinuous target distributions as the dimensionality of the problem increases. We consider the following density[Afshar and Domke, 2015] which represents a hyperbolic-like potential function,

$$\pi(\boldsymbol{x}) = \begin{cases} \exp(-\sqrt{\boldsymbol{x}^T A \boldsymbol{x}}) & \text{if } ||\boldsymbol{x}||_\infty \leq 3 \\ \exp(-\sqrt{\boldsymbol{x}^T A \boldsymbol{x}} - 1) & \text{if } 3 < ||\boldsymbol{x}||_\infty \leq 6 \\ 0 & \text{otherwise} \end{cases}$$

It generates planes of discontinuities along the boundaries defined by the `if` expressions. To write this as a density in our language we make use of the `factor` distribution object as shown in Appendix B.2.

The results in Figure 5.3 provide a comparison between the DHMC and the standard HMC on the worst mean absolute error [Afshar and Domke, 2015] as a function of the

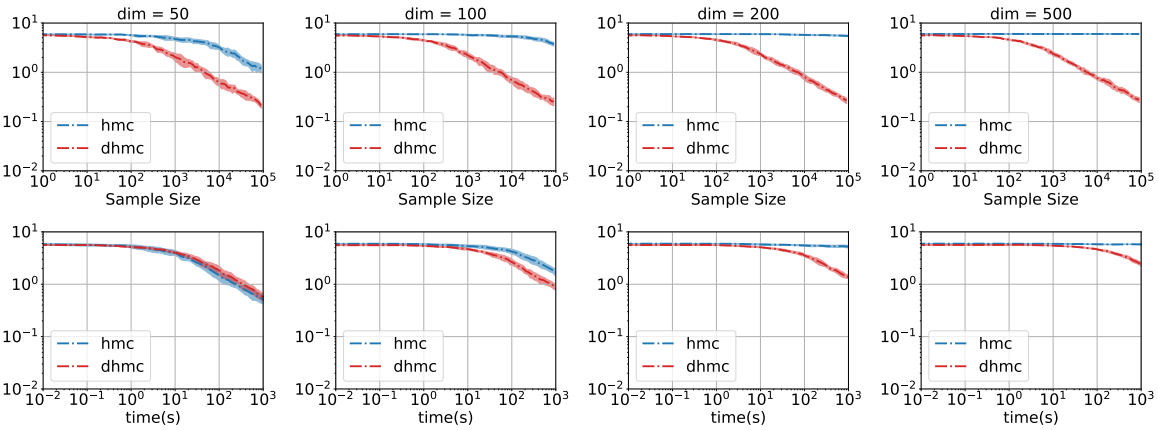


Figure 5.3: We compare DHMC against HMC on the worst mean absolute error (dashed lines) with the $20\%/80\%$ confidence intervals (shaded regions) over 20 independent runs for dimensions $D = 50, 100, 200, 500$ (left to right). We demonstrate how the sample efficiency decreases with respect to sample size (*top* row) and with respect to runtime (*bottom* row) respectively as dimensionality increases. We see that the performance of HMC deteriorates significantly more than DHMC as the dimensionality increases.

number of iterations and time, $\mathrm{WMAE}(N) = \frac{1}{N} \max_{d=1,\ldots,D} \left| \sum_{n=1}^{N} \boldsymbol{x}_d^{(n)} \right|$. We see that as the dimensionality of the model increases, the per-sample performance of HMC deteriorates rapidly as seen in the top row of Figure 5.3. Even though DHMC is more expensive per iteration than HMC due to its sequential nature, in higher dimensions, the additional time costs occurred by DHMC is much less than the rate at which HMC performance deteriorates. The reason for this is that the acceptance rate of the HMC sampler degrades with increasing dimension, while the coordinate-wise integrator of the DHMC sampler circumvents this.

5.7 Conclusion

In this paper we have introduced a Low-level First-order Probabilistic Programming Language (LF-PPL) and an accompanying compilation scheme for programs that have non-differentiable densities. We have theoretically verified the language semantics via a series of translations rules. This ensures programs that compile in our language contain only discontinuities that are of measure zero. Therefore, our language together with the compilation scheme can be used in conjunction with other scalable inference algorithms such as adapted versions of HMC and SVI for non-differentiable densities, as we have demonstrated with one such variant of HMC called discontinuous HMC. It provides a road

map for incorporating other inference algorithms into PPSs and shows the performance improvement of these inference algorithms over existing ones.

6

Universal Probabilistic Programming

Chapter 4 introduced a class of restricted PPSs which mainly focus on inference efficiency, but strictly constrain the type of supported models in order to achieve this. Many useful models, such as models with recursions, stochastic loops or higher order functions are excluded from such PPSs. Can we completely relieve the restrictions and create a system which is able to support any possible model that people might want to use? Motivated by this question, PPSs designed for the universal and general purpose have gradually drawn the attention. Unlike the restricted counterpart, universal PPSs aim to support the widest possible class of models which may define any computable distribution. The rich modeling language frees the concerns of domain experts and they can design any complicated models as needed. However, this expressivity also results in a problem that automating inference for any possible model in such these systems is extremely challenging: many conventional, efficient inference methods are no longer applicable whereas the basic, general options struggle when the model density becomes complicated or high-dimensional. Therefore, a central question of universal PPSs for the practitioners is how to come up with an inference algorithm that is generically applicable as well as reasonably efficient, or whether such inference algorithm exists in the first place.

In this chapter, we will introduce the basic aspects of universal PPSs, highlight their differences compared to the restricted counterparts, demonstrate the uses of the expres-

sivity of universal PPLs via a few examples, and discuss the challenges in performing inference in such systems.

6.1 Language Expressivity

Instead of hand-crafting the modeling language centered around the inference engine, PPSs in this category try to encode the widest possible range of models. This would require an expressive underlying PPL to encode complicated models. One way to design a general-purpose PPL is to start with an expressive, Turing-complete non-probabilistic language, and extend that language with the features for probabilistic procedures. For example, the PPL Anglican [Tolpin et al., 2016] is implemented by extending the programming language, Clojure [Hickey, 2008], and making use of the macro facilities to construct the `sample` and `observe` special forms for generating random values from distributions and conditioning random variables on certain values respectively. Unlike restricted PPLs in which one wants to regulate the language carefully to satisfy specific requirements, a universal PPL wants to exploit the expressivity and richness of the host language to denote models with complex features. Recursions, branching statements, stochastic loops and higher-order functions are all free to use when defining a model.

However, the richness of models also causes an issue for later on inference procedures. It is now much more difficult to extract useful features from a model and abstract this information in a convenient representation, since, for example, a universal probabilistic program is no longer guaranteed to correspond to a graphical model. One cannot compile a universal program to an internal graph-based data structure as in Section 4.3.1 or even a static computation graph that one can manipulate easily. In fact, the number of total variables itself may be a random variable and it may be undeterminable at compile time. It is not trivial to imply what probabilistic distribution a universal probabilistic program is encoding before inference is carried out, but meanwhile it is also difficult to decide which inference algorithm to run without knowing this information.

Before we introduce a mechanism of how to analyze universal PPLs more formally, you might wonder what we can actually gain for establishing such expressive language, or whether it is worth the effort to allow these complicated features of the language if inference

in such systems becomes extremely difficult. It turns out to be the case that such expressivity not only enables more complex models beyond DAGs, but also enables models that are otherwise difficult to encode, such as the models with nested conditioning. We will have a look at a few examples to reveal the advantages of the universal PPLs. Throughout this chapter, we will use the PPL Anglican to demonstrate some key aspects of universal PPLs while the fundamental ideas apply to others languages as well.

6.1.1 Open-universe Model

An important feature of a universal PPL is that, it no longer requires the number of the random variables or their dependency structure to be fixed statically. It is therefore possible to define the so called *open-universe* models where the number of random variables is stochastic. This type of model is particularly useful if one is not sure about the total number of objects or the existence of objects. Such feature can be encoded, for instance, in a way that the existence of a random variable depends on the bound of a recursive function, or the total number is a random variable. When the bound of the recursion is finite, it is less problematic as one can always specify an upper bound of the number of variables and turn the model into a fixed-dimensional case. However, this is infeasible when the bound becomes infinite, in which case one needs to instantiate variables and their dependency relationships dynamically.

One typical example of this type is an open-universe Gaussian Mixture Model [van de Meent et al., 2018; Le et al., 2017]. Unlike the GMM we have seen in Section 5.6.1 where we have a fixed number, K, of the mixtures in total, this number is now a random variable itself. Such model can be defined as

$$
\begin{aligned}
K &\sim \text{Poisson}(\lambda) + 1, \\
\mu_k &\sim \mathcal{D}_\mu(\theta_\mu), \ \text{for}\, k = 1 : K \\
y_n &\sim \mathcal{D}_y(y_n | K, \mu_{1:K}, \theta_y), \ \text{for}\, n = 1 : N_y
\end{aligned}
$$

where the prior for each μ_k is some distribution $\mathcal{D}_\mu$, the likelihood function for observed data y is $\mathcal{D}_y$, and λ, θ_μ and θ_y are all hyper parameters. The code snippet in Anglican of such model is shown in Figure 6.1. The for loop at line 7 has a stochastic bound depending on the variable K, which has a shifted `poisson` prior. The total number of instantiations of

```
(defquery gmm-open [data]
  (let [;; sample the total number of total clusters
        poi-rate 9
        K (+ 1 (sample (poisson poi-rate)))

        ;; sample the mean for each k-th cluster
        mus (loop [k 0
                   mus []]
              (if (= k K)
                mus ;; return mus
                (let [mu-k (sample (mu-dist))]
                  (recur (inc k) (conj mus mu-k)))))
        obs-std 0.1]

    ;; evaluate the log likelihood
    (map (fn [y] (observe (lik-dist mus obs-std) y)) data)

    ;; output
    (cons K mus)))
```

Figure 6.1: Program code for open-universe Gaussian Mixture Model in Anglican.

the subsequent variables for the mean of each cluster, μ_k, will change according to K, and we might have infinitely many possible clusters if K is infinitely large. We have essentially constructed a varying-depth model with a stochastic structure. This causes a problem that some previously introduced efficient inference algorithms, such as HMC, can no longer be applicable to this kind of models. We will explore some other options later in this chapter as well as propose a novel inference scheme in Chapter 7.

6.1.2 Inverting Simulators

Another important view of probabilistic programming is about inverting simulators. In this scenario, we assume a stochastic simulator of an existing system can be understood as a probabilistic generative model. When fixing the input or the latent parameters of a simulator and run it forward, it will produce specific output or observation data following a certain distribution. One might ask what latent parameters are likely to be, after observing a particular output. This involves an backward reasoning process to invert the simulator, which is equivalent to the inference procedure where one tries to characterize the probability distribution of the unknowns of a model conditioned on some observations. Universal

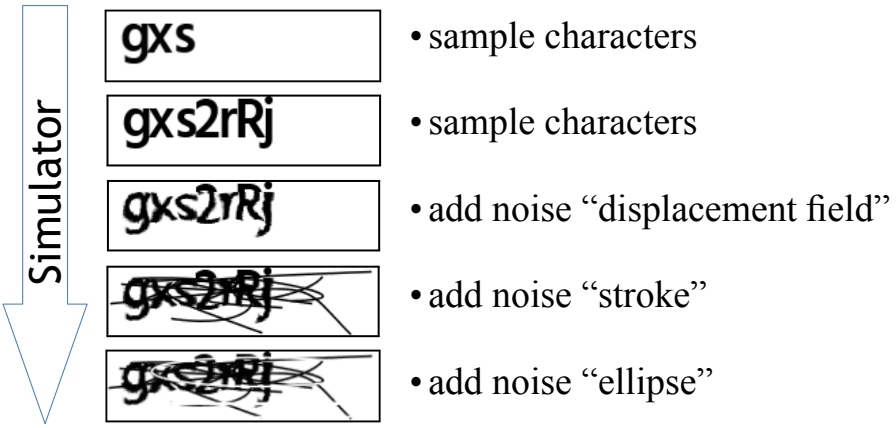


Figure 6.2: The diagram of an artificial simulator of Captchas adapted from [Le et al., 2017]. To generate an Captcha image, we first draw a few characters and numbers, then randomly sample various noises, before rendering into an image in the bottom.

```
(defquery captcha [observed-image]
  (let [num-letters (sample (uniform-discrete 6 8))
        font-size (sample (uniform-discrete 38 44))
        kerning (sample (uniform-discrete -2 2))
        letter-ids (repeatedly
          num-letters
          #(sample (uniform-discrete 0 (count letter-dict))))
        letters (apply str (map (partial nth letter-dict)
                                letter-ids ))
        rendered-image (render letters font-size kerning)]
    (observe (abc-dist rendered-image abc-sigma)
             observed-image)
    letters))
```

Figure 6.3: Program code for the Captcha example in Anglican taken from [Le, 2019].

PPSs provide a convenient framework where a simulator can be specified as a forward probabilistic program, and the backward reasoning can be achieved in a convenient way by applying automated inference engines.

We will take a detailed look at a simulator of Captchas to understand this scenario. Captchas are images of twisted, noisy numbers and/or characters that are used to test whether the subject in front of a computer is a human or a robotic when we try to connect to a specific website. See the small image at the bottom of Figure 6.2 for an example. One can design a simulator following the steps shown in Figure 6.2 to generate Captchas, and the corresponding probabilistic program written in Anglican is shown in Figure 6.3.

The simulator works as follows. It firstly draws the parameters indicating the total number of dsiplaying letters, font-size and kerning of the letters (line 2–4), and then randomly samples each letter from a collection of characters and numbers (line 5–9) to form the string to display. After that, it takes the parameters and renders the string to an Captcha image, where the rendering function is defined in a Captcha library. The approximate Bayesian computation (ABC) style "likelihood" function is defined at line 11 which measures a scaled L_2 distance between the rendered image and the observed one. The stochastic simulator described above essentially defines a generative model of the latent variables and the rendered image. The goal is to infer possible values of the latent variables once an observed Captcha image is provided. This task might be difficult to solve by hand, but is conveniently solved using universal PPSs since one can leverage the existing code of the simulator and automated inference engines are provided.

6.1.3 Mutually Recursive Problem

Apart from the flexibility in model structures, the richness of universal PPLs also enables expressing models that are otherwise difficult to encode. A good example is modeling the theory of mind with nested conditioning [Stuhlmüller and Goodman, 2014]. Suppose Amy and Bob are two friends who want to catch up at either a local pub or a Starbucks but they forget to chat about which one to go before leaving home. The question we are interested in is whether they can meet up successfully at the pub. Since they do not have the exact information where the other person will go, they will need to reason within their own mind to estimate whether pub or Starbucks is a better choice.

Assume the following conditions hold between Amy and Bob. Both Amy and Bob prefer the pub slightly if they choose independently (say 60%/40% for pub and Starbucks respectively), they both want to go to where the other person goes, and they both know the other person knows this and also recognize the fact they know the other knows. Starting with Bob, because he knows Amy wants to go to the pub, if he goes to the pub, he is more likely to see Amy. Amy knows that Bob knows she prefers the pub and Bob wants to go where she might go. If she goes to the pub, she will increase the likelihood to see Bob. We can go back to Bob and think what he might think and then Amy again, and keep the meta-reasoning iterating forever.

```
(defdist location [pub-preference] []
  ;; customize a distribution object
  (sample* [this]
    (if (sample* (flip pub-preference)) :pub :starbucks))
  (observe* [this value]
    (observe* (flip pub-preference) (= value :pub))))

(declare amy bob)
(with-primitive-procedures [location]
  (defm amy [depth]
    (let [amy-location (location 0.6)
          bob-location (bob (dec depth))]
      (observe amy-location bob-location)
      bob-location))

  (defm bob [depth]
    (let [bob-location (location 0.6)]
      (if (> depth 0)
        (let [amy-location (amy depth)]
          (observe bob-location amy-location)
          amy-location)
        (sample bob-location)))))
```

Figure 6.4: Program code for the coordination game in Anglican.

This model would be inconvenient to formalize and reason about in a standard statistical notation due to the nested and recursive nature. Universal PPLs provide a beautiful and straightforward way to specify the model using a probabilistic program similar to our intuitive thought process above. As we can see in Figure 6.4 (which is is adapted from [Stuhlmüller and Goodman, 2014] and is part of the tutorial of Anglican), this model can be defined in just a few lines in PPS Anglican. We encode the decision making process for Amy and Bob in two recursive functions which call each other. To estimate how likely both of them go to the pub eventually, we can perform inference in the model using automated inference engines. We will see that depending on the depth of the meta-reasoning, the probably that they meet in the pub changes; the deeper they reason before taking actions, they more likely they will choose the pub, and the probability goes towards one as the depth goes to infinity.

6.2 Execution-based Formalization

As we have seen in the previous section, a universal program, in general, does not correspond to a graphical model anymore. Especially, it is possible to construct programs with infinite many variables, in which case graph-based reasoning is not appropriate immediately. In order to reason about models in the generic setting, we will introduce *evaluation-based* inference [van de Meent et al., 2018], which exploits the *execution* of a program. The intuition behind such scheme is that even if a program can encode an infinite model, each execution would still contain only a finite subset of all random variables, and therefore we will not have the problem of trying to characterize an infinite graph.

Probably the first question to ask is what it means to execute a a probabilistic program. Conceptually, in PPS Anglican, one can think of this procedure akin to running a purely deterministic program but with a few exceptions: we execute the statements one by one in a standard way at most places, but invoke specific operations at the probabilistic constructs such as `sample` and `observe` special forms, and we return the value of the final statement which we shall call the *output* of the probabilistic program. Note that we have assumed that our programs will always terminate.

6.2.1 Addressing Scheme

In FOPPL, we always have a finite set V containing all random variables, and we can identify each variable by simply numbering them according to the dependency relationships in the graphical model representation G. In a universal scenario, although we cannot enumerate all the random variables, we would still need some way to distinguish the sampled variables as such information is necessary for many inference algorithms. For instance, to implement the MH sampler, one needs to design a transition kernel to propose a new sample sequence based on the current one and calculate the acceptance probability. This would require a unique identifier for the `sample` statement of each latent variable, and this information will be used in both constructing the transition kernel as well as evaluating related probability density functions.

One can design the addressing or naming scheme in different ways, dynamically as in Wingate et al. [2011] or statically as in Wood et al. [2014] and Le et al. [2017]. We will take

the latter option in this thesis, where the address of the i-th **sample** statement is defined by a unique name, a_i, according to its lexical position at compile time.

6.2.2 Execution Traces

For simplicity, we assume that during the execution, we only draw samples when encountering **sample** statements for now and skip all the **observe** ones. Akin to FOPPL [van de Meent et al., 2018], **sample** statement in Anglican also takes a distribution object, `dist`, as an argument, and returns a value drawn from that distribution. We use i to index the i-th triggered **sample** statement with an address a_i, and x_i for the generated sample following the distribution $\texttt{dist}_{a_i}$. Supposing we have invoked **sample** statements n_x times[1] in total in a single execution, we can then obtain the *execution trace* for this run recognized as $[a_i, x_i]_{i=1}^{n_x}$ where $a_{1:n_x}$ indicates the *path* we have been through and $x_{1:n_x}$ are the *draws* that have been generated at each **sample** statements respectively. Note that $a_{1:n_x}$ and n_x can vary from execution to execution. We may land on different paths or encounter different set of latent variables during execution because of the possible recursions, branching statements, stochastic loops, and higher-order functions. We say two executions have the *same path* only when they have exactly the same $a_{1:n_x}$, i.e. each a_i, the order of a_is and the length n_x being the same. It is also important to highlight that, the sequence of draws $x_{1:n_x}$ is sufficient to recognize a trace; this is because the program between the **sample** statement is purely deterministic, and therefore, it is deterministically determinable which **sample** statement will be invoked next given the results from the **sample** statements encountered so far.

Take the open-universe GMM from Section 6.1.1 as an example. If we execute the program shown in Figure 6.1, the first **sample** statement we invoke is always the one at line 4 and we use $\#l_4$ to denote the lexical address. We draw a sample of $\hat{K}$ from **poisson**$(9) + 1$ and suppose its value is 6. This is followed by the for loop at $\#l_7$ where we will call the **sample** statement at $\#l_{11}$ six times before exiting the loop. Suppose the drawn values for μs are $[0.1, 2.3, -1.5, 0.6, 1.8, -2.9]$. We have then obtained the trace for this execution with the draws $\hat{x}_{1:7}^{(t)} = [6, 0.1, 2.3, -1.5, 0.6, 1.8, -2.9]$ and path $\hat{a}_{1:7}^{(t)} = [\#l_4, \#l_{11}, \#l_{11}, \#l_{11}, \#l_{11}, \#l_{11}, \#l_{11}]$, and the trace length n_x is 7. As we have seen here, the trace length is decided by the value of variable K. When we execute the

[1] One **sample** statement may be triggered more than once such as the one in the for loop.

program again, we might generate a different value of K and therefore we would follow a different path of the new execution trace.

6.2.3 Probability Distribution of Execution Traces

We can now formalize the probability density that a universal probabilistic program defines via its execution traces. For the **`sample`** statement, apart from drawing samples as we have seen above, we can also evaluate the density f_{a_i} according to $\texttt{dist}_{a_i}$ with the newly generated sample x_i. For **`observe`**, it takes two arguments – a distribution object $\texttt{dist}_{b_j}$ and an observed data y_j, where we can evaluate the density g_{b_j} defined by $\texttt{dist}_{b_j}$. The probability density of an execution trace is

$$\gamma(x) = \prod_{i=1}^{n_x} f_{a_i}\left(x_i \mid \eta_i\right) \prod_{j=1}^{n_y} g_{b_j}\left(y_j \mid \phi_j\right) \tag{6.1}$$

where η_i and ϕ_j contain all the information needed to evaluating f_{a_i} and g_{b_j} including the distribution types and the evaluated parameters. Unlike N_x in (4.1) which represents the total number of latent variables in the model, n_x here may be a random variable and represents the number of latent variables on a particular trace. Also observed data y now may occur in η, in which case the product of all f terms no longer corresponds to a conventional prior $p(x)$. Therefore $\gamma(x)$ is not necessary a joint distribution $p(x, y)$ as in the standard Bayesian model, but we can still interpret it as an unnormalized density where the target distribution $\pi(x)$ can be obtained in a similar manner. A nice upshot of the execution-based formalization is that though it might be difficult to characterize the target distribution the program defines upfront, or we do not even know how many variables it might include overall, each single forward execution will contain only a finite number of evaluations of fs and gs, and therefore, we can define the probability w.r.t. to each finite trace to form the overall density.

One important point when manipulating the execution trace is to ensure its *validity*. This will not be problematic if we always use evaluation-based inference methods in the PPS as all the outcomes are directly drawn from the model; it will, however, when we try to evaluate the unnormalized density using samples generated or updated externally. A *valid trace* requires that it corresponds to a complete and valid path, i.e. it cannot be only part of a path, or a path that does not exist when running program, or a path with ill-defined or un-defined densities. Take the program defined in Figure 6.5 as an example. Both paths have a length

```
1 (let [x (sample (normal 0 2))]
2    (if (> x 0)
3       (sample (exponential x))
4       (sample (beta 2 2))))
```

Figure 6.5: Example program to demonstrate invalid trace.

two so any trace with only one entry like $[x_1 = 0.3]$ is incomplete and thus invalid. Moreover, suppose we are on a path corresponding to the `true` branch, $\hat{x}_{1:2} = [0.3, -1.9]$ is not a valid trace because **`exponential`** distribution only has support on $[0, +\infty)$, -1.9 would not be a valid outcome generated from **`exponential`**. Another example could be suppose we have the outcomes $\hat{x}_{1:2} = [0.3, 1.9]$ now and we propose an update for x_1 to -1.1 and re-evaluating the density. However, if we have not realized the resulting path changing and still calculate the density of x_2 using the **`exponential`** distribution, we will end up with an ill-defined distribution. It is important to ensure that none of the invalid traces are returned by the inference engine; they should always have weight zero or never been accepted.

6.3 Universal Inference Engines

Recall that our goal is to obtain the conditional distribution $\pi(x)$ from the unnormalized version $\gamma(x)$. Since we do not impose any constraint of the modeling language as we did in restricted PPSs, a universal program no longer corresponds to a graphical model and therefore it is no longer viable to attempt to compile the program into a graphical model representation G as we have seen in Section 4.3. Some conventional inference options that work well for the graphical model scenario may no longer be applicable in the universal setting (due to the possible complicated model structures). Even worse, the so called graph-based implementations inference algorithms which are designed to work with a graphical model data structure are not suitable for universal PPSs where programs may have complicated features such as stochastic loops and unbounded recursions.

Aware of these problems, we will have a look at a different strategy–an *evaluation-based* implementation–for the inference techniques that are still applicable in universal PPSs. At a high-level, such methods work by executing the universal probabilistic programs and manipulating their execution traces to obtain desired outputs.

6.3.1 Separation between Program Code and Inference Engine

To perform inference in an automated and generic fashion, we would still require some form of abstraction between the model program and the inference algorithm. Because the computation graph, the dependency structure, and the existence of all the variables are not deterministically computable at compile time, it is infeasible anymore to use the restricted, graphical model representation as the compilation target. However, we still want to extract or encapsulate the information about the model program, such as types of the latent variables and their dependency structures, in a certain way that can be easily accessed and used by the inference engine.

One popular choice is to make use of the partial execution of the program. Rather than interacting with a static graphical model representation, the inference procedure now functions more like a controller: it is able to call the program, interrupt the execution at a certain point and resume the execution from a particular program point. This is because, for instance, in the Metropolis-Hastings, we might want to propose an update for a given variable which can be at any place along the trace. Or, in SMC, we might want to carry out more than one execution traces pause all of them at each **`sample`** statement to perform re-sampling steps. We will describe the design framework shared in both Tolpin et al. [2016] and van de Meent et al. [2018] as an example in the following.

To allow the interaction between the program code and the inference engine, the program is stored as a continuation object $\mathcal{P}$, and the inference algorithm is implemented as a backend $\mathcal{B}$. Most components of the program are deterministic and can be operated in a standard manner apart from the probabilistic special forms, i.e. the **`sample`** and **`observe`** statements. At such special forms, the inference backend steps in and interrupts the execution; these points are called *checkpoints*. What happens at checkpoints depends on the specific inference choice. At a high level, the program $\mathcal{P}$ is paused at the checkpoint and passes necessary information to the inference backend $\mathcal{B}$. The backend $\mathcal{B}$ then undertakes certain operations and returns a value to $\mathcal{P}$ if required, and $\mathcal{P}$ continues (with the returned value) from previous pausing point. For the **`sample`** checkpoint, $\mathcal{B}$ receives a distribution object `dist` with the distribution type and all the parameters evaluated, and returns some value x to $\mathcal{P}$. For the **`observe`** checkpoint, aside from the distribution object `dist`, $\mathcal{B}$

also receives a constant value y. Unlike **`sample`** statement, it usually carries out some calculations and returns `nil` or nothing. When $\mathcal{P}$ terminates, it returns the output to $\mathcal{B}$ which can be the draws generated on the trace or the deterministic functions of the trace.

6.3.2 Importance Sampling as a Universal Engine

Importance Sampling (IS) is probably the most basic but also widely applicable inference engine that is supplied in most PPSs. As we have introduced in Section 2.3.1, it works by generating samples from a proposal distribution, q, and calculating their weights, $\hat{w}$ accordingly to the target distribution. To implement IS as an automated engine, we first need a convenient way to construct a valid proposal distribution that is easy to sample from. This is not trivial as it may sound since it is almost impossible to determine the support of latent variables under the target distribution, and therefore, it is possible to obtain samples with invalid weights if the proposal distribution is not designed carefully (eg. the weight can be infinite large if the tail of q is much lighter than the target distribution). Also, for the universal program, it is unlikely to know where the mode of the target might be so it would be difficult to construct a good proposal at the first place. One way to implement the IS backend is to use the prior distribution, $f_{a_i}(\cdot|\theta_i)$ from the **`sample`** statements as the proposal. This is guaranteed to be valid as the samples are drawn from the generative model itself. Since the "prior" is now the proposal, this term cancels out and the weight only contains the "likelihood" terms specified in the **`observe`** statement; such implementation of IS is also known as *likelihood weighting* in the probabilistic programming context.

Given what we have introduced previously, designing the IS backend is straightforward: at each **`sample`** checkpoint, $\mathcal{P}$ is interrupted by $\mathcal{B}$ which then generates sample(s) from the corresponding distribution object and returns the drawn samples to $\mathcal{P}$; for the **`observe`** special forms, $\mathcal{B}$ computes the log probability density of the specified distribution, $g_{b_j}\left(y_j \mid \phi_j\right)$, with all the parameters evaluated; the IS engine accumulates the log density from all the **`observe`** statements along the execution trace, and returns the drawn samples $\hat{x}_{1:n_x}$ and the corresponding weight w as the output.

6.3.3 A Single-site Metropolis Hastings

As we have introduced in Section 2.3.3, MCMC methods are one important class of inference methods widely applied in PPSs. To implement such engine, we need to achieve two tasks: to propose a new sample x' based on the current trace, and to evaluate the acceptance ratio $\mathcal{A}$ as per (2.22). For Gibbs sampler and HMC, we implicitly presume that the model can be characterized by a graphical model representation, with the former further requiring the extraction of the Markov blanket of the latent variables, and the latter the gradient of the log density. Such properties do not hold for universal probabilistic programs. In this section, we describe an evaluation-based Metropolis Hastings (MH) algorithm known as the *single-site MH*, the *lightweight MH* (LMH) or the random database algorithm (RDB) [Wingate et al., 2011], as well as its variant, the random-walk MH (RMH) [Le, 2015], which are easy to implement and generally applicable.

LMH constructs a Markov chain over program execution traces. The single site proposal updates the trace in the Metropolis-within-Gibbs manner, that is, it updates one variable at a time following by an accept/reject step. We start by running the program forward to obtain an initial execution trace. At each iteration, based on the current trace $x^{(t)} = x_{1:n_x^{(t)}}$, the proposal distribution $k(x'|x^{(t)})$ for generating a new trace x' works as following. It first uniformly chooses a variable in the trace, $i \in 1 : n_x^{(t)}$, and the inference backend $\mathcal{B}$ interrupts the program $\mathcal{P}$ at the corresponding checkpoint with an address a_i which is referred as the *entry point*. It then proposes a new value of that variable, $x_i^{(t)} \rightarrow x_i'$, using a reversible transition kernel $k_i(x_i'|x_i^{(t)})$. We then continue the program execution $\mathcal{P}$ with the updated, partial trace $[x_{1:i-1}^{(t)}, x_i']$. For every downstream **`sample`** statement at a_l, if it has been encountered before and the ERP type $\texttt{dist}_{a_l}$ remains the same, we can reuse the old value x_l and re-calculating or rescoring its probability given all previous samples if necessary. Otherwise, if $\texttt{dist}_{a_l}$ has changed, or a new variable is encountered, we draw new value from the prior and incorporate its probability accordingly. Intuitively, we want to reuse as much information as we can from the current trace such that the proposed new trace is "close" to the current one. Following the notation from Wood et al. [2014], we use $x' \cap x^{(t)}$ to denote all re-used samples, and $x' \backslash x^{(t)}$ for all newly generated ones.

We therefore have the proposal density as

$$k(x'|x^{(t)}) = \frac{1}{n_x^{(t)}} k_i(x'_i|x_i^{(t)}) \frac{p(x' \backslash x^{(t)}|x' \cap x^{(t)})}{p(x'_i|x' \cap x^{(t)})}. \tag{6.2}$$

Informally, the right most fraction in (6.2) describes the probability of newly drawn samples after the entry point i. We use $\mathbb{D}(l) = 1$ to indicates that the l-th sample is newly generated, and therefore we can rewrite the proposal as

$$k(x'|x^{(t)}) = \frac{1}{n_x^{(t)}} k_i(x'_i|x_i^{(t)}) \prod_{l=i+1}^{n'_x} \left(f_{a_j}(x'_l|\theta'_l)\right)^{\mathbb{D}(l)=1}. \tag{6.3}$$

We then accept the new trace x' as the proposed sample $x^{(t+1)}$ with the probability

$$\mathcal{A}(x', x) = \min\left(1, \frac{\gamma(x')k(x^{(t)}|x')}{\gamma(x^{(t)})k(x'|x^{(t)})}\right). \tag{6.4}$$

For the standard LMH set up, $k_i(x'_i|x_i^{(t)})$ is simply the prior distribution $f_{a_i}(x'_i|\theta_i)$, where x'_i is independent from $x_i^{(t)}$ to a certain extent. In doing so, we have lost the important locality information at x'_i, which might result in the fact that the downstream part of the new trace, $x'_{i+1:n'_x}$, shares little information with the corresponding part of the existing trace, $x^{(t)}_{i+1:n_x}$ (e.g. the new trace might switch paths after x'_i). Especially in high-dimensional problems, it is unlikely to land in "good" regions with possible high posterior mass by just proposing from the prior. Consequently, one might experience a low acceptance rate for a LMH engine.

Motivated by such a problem, Le [2015] proposes an alternate of the original LMH, known as the Random-walk Metropolis Hastings (RMH). For the variable at the entry point i, it employs a proposal distribution being a mixture of a local kernel and the prior with a probability of α and $1 - \alpha$, respectively. For a local kernel, $k_i(x'_i|x_i^{(t)})$ is a distribution parameterized by the existing value $x_i^{(t)}$, and one proposes the new value in a random walk manner. For instance, to propose a small movement for $x_i^{(t)}$, k_i can be a normal distribution $\mathcal{N}(x_i^{(t)}, 1)$ which we use to draw x'_i. RMH behaves akin to LMH when the prior is used, whereas it is able to establish local movements when using the local kernel. The rest of how to continue the program as well as how to calculate the acceptance ratio is similar to the setting of LMH.

7 Divide, Conquer, and Combine

Universal probabilistic programming systems (PPSs) provide a powerful framework for specifying rich probabilistic models. They further attempt to automate the process of drawing inferences from these models, but doing so successfully is severely hampered by the wide range of non–standard models they can support. As a result, one can easily fall into the awkward situation that although one can specify complex models in a universal PPS, the provided inference engines often fall far short of what is required. In particular, we show that they produce surprisingly unsatisfactory performance for models where the support varies between executions, often doing no better than importance sampling from the prior. To address this, we introduce a new inference framework: Divide, Conquer, and Combine [Zhou et al., 2019b], which remains efficient for such models, and shows how it can be implemented as an automated and generic PPS inference engine. We empirically demonstrate substantial performance improvements over existing approaches on three examples.

7.1 Motivation

Probabilistic programming systems (PPSs) provide a flexible platform where probabilistic models are specified as programs and inference procedures are performed in an automated manner. Some systems, such as BUGS [Tesauro et al., 2012] and Stan [Carpenter et al., 2017], are primarily designed around the efficient automation of a small number of

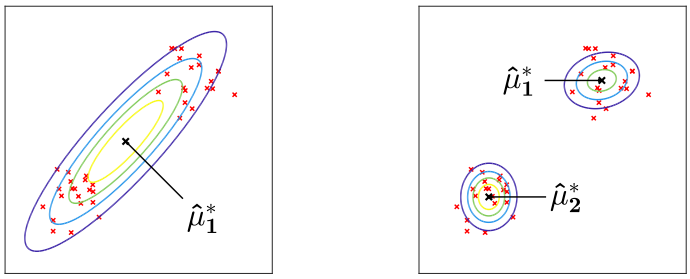


Figure 7.1: MAP estimate of the means and covariances of a Gaussian mixture model in the cases of $K = 1$ and $K = 2$ clusters. If K is itself random, the model has stochastic support as the parameters of the second cluster, e.g. its mean, only exist when $K = 2$.

inference strategies and the convenient expression of models for which these inference strategies are suitable.

Universal PPSs, such as Church [Goodman et al., 2008b], Venture [Mansinghka et al., 2014], Anglican [Wood et al., 2014], Birch [Murray and Schön, 2018], Turing.Ji [Ge et al., 2018a], and Pyro [Bingham et al., 2018], on the other hand, are set up to try and support the widest possible range of models a user might wish to write. Though this means that such systems can be used to write models which would be otherwise difficult to encode, this expressiveness comes at the cost of significantly complicating the automation of inference. In particular, models may contain random variables with mixed types or have varying, or even unbounded, dimensionalities; characteristics which cause significant challenges at the inference stage.

In this section, we aim to address one of the most challenging model characteristics: variables whose very existence is stochastic, often, though not always, leading to the overall dimensionality of the model varying between realizations. Many practical models posses this characteristic. For example, many models contain a variable controlling an allowed number of states, such as the number of clusters in a mixture model (see Figure 7.1, Richardson and Green 1997; Nobile and Fearnside 2007), or the number of states in a HMM or change point model [Fox et al., 2008]. More generally, many inference problems involve some sort of Bayesian model averaging where the constituent models do not share the exact same set of parameters. Other models are inherently defined on spaces with non-static support, such as probabilistic context free grammars (PCFGs) [Manning and Schutze, 1999], program

induction models [Perov and Wood, 2014], kernel or function induction models [Schaechtle et al., 2016; Janz et al., 2016], many Bayesian non-parametric models [Roy et al., 2008; Teh, 2010], and a wide range of simulator–based models [Le et al., 2017; Baydin et al., 2019].

These models can be easily expressed in universal PPSs via branching statements, stochastic loops, or higher-order functions (see e.g. Figure 7.2). However, performing inference in them is extremely challenging, with the desire of PPSs to automate this inference complicating the problem further.

A number of automated inference engines have been proposed to provide consistent estimates in such settings [Wingate et al., 2011; Wood et al., 2014; Tolpin and Wood, 2015; Ge et al., 2018a; Bingham et al., 2018]. However, they usually only remain effective for particular sub-classes of problems. In particular, we will show that they can severely struggle on even ostensibly simple problems, often doing no better, and potentially even worse, than importance sampling from the prior.

To address these shortfalls, we introduce a completely new framework—***Divide, Conquer, and Combine*** (DCC)—for performing inference in such models. DCC works by ***dividing*** the program into separate straight-line sub-programs with fixed support, ***conquering*** these separate sub-programs using an inference strategy that exploits the fixed support to remain efficient, and then ***combining*** the resulting sub-estimators to an overall approximation of the posterior. The main motivation behind this approach is that the difficulty in applying Markov chain Monte Carlo (MCMC) strategies to such programs lies in the transitioning between variable configurations; within a given configuration efficient inference might still be challenging, but will be far easier than tackling the whole problem directly. Furthermore, this approach also allows us to introduce meta-strategies for allocating resources between sub-programs, thereby explicitly controlling the exploration-exploitation trade-off for the inference process in a manner akin to Rainforth et al. [2018] and Lu et al. [2018]. To demonstrate its potential utility, we implement a specific realization of our DCC framework as an automated and general-purpose inference engine in the PPS Anglican [Wood et al., 2014], finding that it is able to achieve substantial performance improvements and tackle more challenging models than existing approaches.

```
(let [z0 (sample (normal 0 2))
      y1 9]
  (if (< z0 0)
    (let [z1 (sample (normal -5 2))]
      (observe (normal z1 2) y1)
      [z0 z1])
    (let [z2 (sample (normal 5 2))
          z3 (sample (normal z2 2))]
      (observe (normal z3 2) y1)
      [z0 z2 z3])))
```

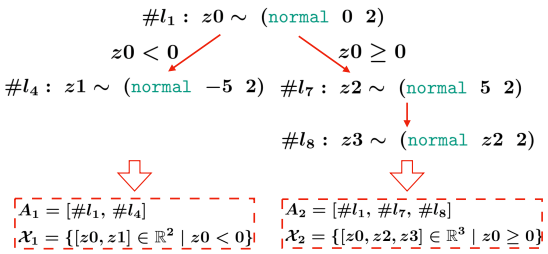


Figure 7.2: Example program with stochastic support (left) and its possible execution traces (right). The two branches of the **if** each produce a different sample path, denoted as A_1 and A_2, and each have a different supports denoted as $\mathcal{X}_1$ and $\mathcal{X}_2$ respectively.

7.2 Background

In universal PPSs [Goodman et al., 2008b; Mansinghka et al., 2014; Wood et al., 2014; Bingham et al., 2018; Ge et al., 2018a], a program may denote a model with varying support: different realizations of a program may lead to different sets of variables being sampled. For example, in Figure 7.2, there are two possible variable configurations, $[\texttt{z0}, \texttt{z1}]$ and $[\texttt{z0}, \texttt{z2}, \texttt{z3}]$, due to the stochastic control-flow.

To aid exposition and formalize these programs, we will reiterate the formalization of a particular universal PPS Anglican [Wood et al., 2014; Tolpin et al., 2016], but note that our ideas are applicable to other universal PPSs. Anglican extends the syntax of Clojure with two special forms: **sample** and **observe**, between which the distribution of the program is defined. **sample** statements are used to draw random variables, while **observe** statements are used to condition on data. Informally, they can be thought of as prior and likelihood terms, respectively.

The density of an Anglican program is derived by executing it in a forward manner, drawing from **sample** statements when encountered, and keeping track of density components that originate from the **sample** and **observe** terms. Specifically, let $\{x_i\}_{i=1}^{n_x} = (x_1, \ldots, x_{n_x})$ represent the random variables generated from the encountered **sample** statements, where the i^{th} statement among them has lexical program address a_i, input η_i, and density $f_{a_i}(x_i|\eta_i)$. Analogously, let $\{y_j\}_{j=1}^{n_y} = (y_1, \ldots, y_{n_y})$ represent the observed values of the n_y **observe** statements encountered during execution, which have lexical

addresses b_j and corresponding densities $g_{b_j}(y_j|\phi_j)$, where ϕ_j is analogous to η_i. The density is now given by $\pi(x) = \gamma(x)/Z$ where

$$\gamma(x) := \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j), \tag{7.1}$$

$$Z := \int \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j) dx_{1:n_x}, \tag{7.2}$$

and the associated reference measure is implicitly defined through the executed `sample` statements. Note that everything here (i.e. n_x, n_y, $x_{1:n_x}$, $y_{1:n_y}$, $a_{1:n_x}$, $b_{1:n_y}$, $\eta_{1:n_x}$, and $\phi_{1:n_y}$) is a random variable, but each is deterministically calculable given $x_{1:n_x}$ (see §4.3.2 of Rainforth [2017]). See Section 6.2 for a more detailed explanation about execution-based formalization.

From this, we see that it is sufficient to denote an ***execution trace*** (i.e. realization) of an Anglican program by the sequence of the addresses of the encountered `sample` statements and the corresponding sampled values, namely $[a_i, x_i]_{i=1}^{n_x}$.[1] For clarity, we refer to the sequence $a_{1:n_x}$ as the ***path*** of an execution trace and $x_{1:n_x}$ as the ***draws***. A program with ***stochastic support*** can now be more formally defined as one for which the path $a_{1:n_x}$ varies between realizations: different values of the path correspond to different ***configurations*** of variables being sampled.

7.3 Shortcomings of Existing Inference Engines

In general, existing inference engines that can be used for (at least some) problems with stochastic support can be grouped into five categories: importance/rejection sampling, particle based inference algorithms (e.g. SMC, PG, PIMH, PGAS, IPMCMC, RM-SMC, PMMH, SMC^2), MCMC approaches with automated proposals (e.g. LMH, RMH), MCMC approaches with user–customized proposals (e.g. RJMCMC), and variational approaches (VI, BBVI). More details on each are provided in Appendix C.1.

Importance/rejection sampling approaches can straightforwardly be applied in stochastic support settings by using the prior as the proposal, but their performance deteriorates rapidly as the dimensionality increases. Particle–based approaches offer improvements for

[1] Strictly speaking, the addresses a_i can be deterministically derived from the sampled values for a given program. However, for our purposes it will be convenient to think about first sampling the path $a_{1:n_x}$ and then sampling $x_{1:n_x}$ conditioned on this path.

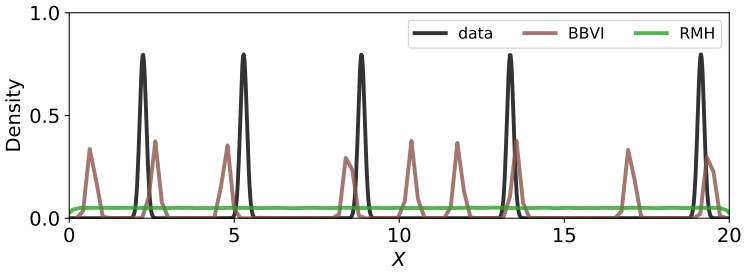


Figure 7.3: Kernel density estimation of the synthetic data (black) of the univariate mixture model and the posterior predictive distribution from BBVI (brown) and RMH (green) in Anglican.

models with sequential structure [Wood et al., 2014], but become equivalent to importance sampling when this is not the case.

Some variational inference (VI) approaches can, at least in theory, be used in the presence of stochastic support [Bingham et al., 2018; Cusumano-Towner et al., 2019], but this can require substantial problem–specific knowledge to construct a good guide function, requires a host of practical issues to be overcome [Paige, 2016, Chapter 6], and is at odds with PPSs desire for automation. Furthermore, these approaches produce inconsistent estimates and current engines often give highly unsatisfactory practical performance as we will show later. When using such methods, it is common practice to side–step the complications originating from stochastic support by approximating the model with a simpler one with fixed support. For example, though Pyro can ostensibly support running VI in models with stochastic support, their example implementation of a Dirichlet process mixture model (`https://pyro.ai/examples/dirichlet_process_mixture.html`) uses fixed support approximations by assuming a bounded number of mixture components before stripping clusters away.

Because of these issues, arguably the current go–to approaches for programs with stochastic support are specialized MCMC approaches [Wingate et al., 2011; Roberts et al., 2019]. However, these are themselves far from perfect and, as we show next, they often given performance far worse than one might expect.

A demonstrative example Consider the following simple mixture model with an unknown number of clusters K

$$K \sim \text{Poisson}(9)+1,$$

$$\mu_k \sim \text{Uniform}\left(\frac{20(k-1)}{K}, \frac{20k}{K}\right),$$
$$z_n \sim \text{Cat}(\{1/K, ..., 1/K\}),$$
$$y_n \sim \mathcal{N}(\mu_{z_n}, 0.1^2).$$

Here $\mu_{1:K}$ are the cluster centers, $z_{1:N}$ are the cluster assignments, and $y_{1:N}$ is the observed data. When conducting inference, we can analytically marginalize out the cluster assignments $z_{1:N}$ and perform inference on K and $\mu_{1:K}$ only. However, as the prior on K is a Poisson distribution, the number of parameters in the model is unbounded. Using the model itself, we generated a synthetic dataset of $y_{1:150}$ for an one-dimensional mixture of five clusters (i.e. $K = 5$).

We now wish to perform inference over both the number of clusters K and the cluster means $\mu_{1:K}$, so that we can make predictions from the posterior predictive distribution. Two approaches we might try are VI, for which we use Anglican's black-box variational inference (BBVI) implementation [Ranganath et al., 2014; Paige, 2016], and MCMC, for which we take its RMH algorithm [Le, 2015], a derivative of the single-site MH algorithm of [Wingate et al., 2011] (see below). Unfortunately, as we see in Figure 7.3, both approaches fail spectacularly and produce posterior predictive distributions that bare little resemblance to the data. In particular, they fail to properly encapsulate the number of clusters. As we will show in §7.7.1, importance sampling and particle–based inference engines fare no better for this problem. In fact, we are not aware of any fully *automated* inference engine that is able to give satisfactory performance, despite the apparent simplicity of the problem.

7.3.1 Why is MCMC so Hard with Stochastic Support?

To run MCMC on a program with stochastic support, one needs to be able to construct a transition kernel that is able to switch between the configurations; many popular MCMC methods, like Hamiltonian Monte Carlo (HMC), cannot be applied. One can either look to construct this transition kernel manually through a carefully chosen user–specified trans–dimensional proposal and then using a reversible jump MCMC (RJMCMC) scheme [Green, 1995, 2003; Roberts et al., 2019; Cusumano-Towner et al., 2019], or use a general–purpose kernel that works on all models [Wingate et al., 2011; Goodman and Stuhlmüller, 2014].

The predominant approaches for the latter are the single-site MH (a.k.a. LMH) algorithm [Wingate et al., 2011] and its extensions [Yang et al., 2014; Le, 2015; Tolpin et al., 2015; Ritchie et al., 2016b]. LMH is based around a Metropolis-within-Gibbs (MwG) approach on the program traces [Brooks et al., 2011], whereby one first samples a variable in the execution trace, $i \in 1 : n_x$, uniformly at random and then proposes a MwG transition to this variable, $x_i \to x_i'$, followed by an accept/reject step. Anglican's RMH is a particular case of this LMH approach where the proposal is a mixture of resampling x_i from the prior $f_{a_i}(x_i|\eta_i)$ and a local random walk proposal $p(x_i'|x_i)$.

The problem with LMH approaches is that if the transition of x_i influences the downstream control flow of the program, the downstream draws no longer produce a valid execution trace and so must be redrawn, typically using the prior. This can cause the mixing of the sampler over configurations to become extremely slow; the need to transit between configurations bottlenecks the system.

This problem is also far from specific to the exact transition kernel used by LMH samplers: it is also extremely challenging to hand–craft RJMCMC proposals to be effective. Namely, proposing changes in the configuration introduces new variables that might not be present in the current configuration, such that our proposal for them effectively becomes an importance sampling proposal. Furthermore, the posterior on the other variables may shift substantially when the configurations changes.

In short, one loses a notion of locality: having a sample in a high density region of one configuration typically provides little information about which regions have a high density for another configuration. For example, in a mixture model shown in Figure 7.1, having a good characterization of $\mu_1|K = 1$ provides little information about the distribution of $\mu_1|K = 2$, as shown by the substantial change in their mode, μ_1^*. It is thus extremely difficult to design proposals which maintain a high acceptance rate when proposing a new configuration: once in a high density region of one configuration, it becomes difficult to switch to another configuration. This problem is further compounded by the fact that RJMCMC only estimates the relative mass of each configuration through the relative frequency of transitions, giving a very slow convergence for the overall sampler.

7.4 Divide, Conquer, and Combine

The challenges for running MCMC methods on programs with stochastic support stem from the difficultly in transitioning *between* configurations properly. To address this, we now introduce a completely new inference framework for these programs: ***Divide, Conquer, and Combine*** (DCC). Unlike most existing inference approaches which directly target the full program density (i.e. (7.1)), DCC breaks the problem into individual sub-problems with ***fixed*** support and tackles them separately. Specifically, it ***divides*** the overall program into separate straight-line sub-programs according to their execution paths, ***conquers*** each sub-program by running inference locally, and ***combines*** the results together to form an overall estimate in a principled manner.

In doing this, DCC transfers the problem of designing an MCMC proposal which both efficiently transitions between paths (i.e. varying configurations) and mixes effectively over the draws on that path, to that of a) performing inference locally over the draws of each given path, and b) learning the relative marginal posterior mass of these paths. This separation brings the benefit that the inference for a given path can typically be performed much more efficiently than when using a global sampler, as it can exploit the fixed support and does not need to deal with changes in the variable configuration. Furthermore, it allows the relative posterior mass to be estimated more reliably than with global MCMC schemes, for which this is estimated implicitly through the relative frequency of the, typically infrequent, transitions.[2]

We now explain the general setup for each component of DCC. Specific strategies for each will be introduced in §7.6, while an overview of the approach is given in Algorithm 7.1.

7.4.1 Divide

The aim of DCC's divide step is to split the given probabilistic program into its constituent straight-line programs (SLPs), where each SLP is a partition of the overall program corresponding to a particular sequence of sample addresses encountered during execution,

[2]Note that though sharing a similar name, our DCC scheme is fundamentally different from the Divide-and-Conquer SMC (D&C-SMC) algorithm from Lindsten et al. [2017]. The latter works with factor graphs where it decomposes a graphical model into sub-graphs and constructs corresponding auxiliary intermediate target distributions as in SMC. The DCC framework that we propose here targets at models with stochastic support. It divides according to program execution traces, rather than subsets of variables, and how inference is carried out locally in DCC is completely different than in D&C-SMC.

Algorithm 7.1 Divide, Conquer, and Combine (DCC)

Input: Program ***prog***, number of iterations T
Output: Posterior approx $\hat{\pi}$, ML estimate $\hat{Z}$

1: Obtain initial set of discovered SLPs $\hat{A}$ ▷ §7.4.1, §7.6.2
2: Compute initial estimates $\forall A_k \in \hat{A}$ ▷ §7.4.2, §7.6.1
3: **for** $t = 1, \ldots, T$ **do**
4: Choose an SLP $A_k \in \hat{A}$ to update ▷ §7.6.3
5: Update local estimates $\hat{\pi}_k$ and $\hat{Z}_k$ ▷ §7.4.2, §7.6.1
6: [Optional] Look for undiscovered SLPs (e.g. using a global proposal), add any found to $\hat{A}$ ▷ §7.6.2
7: **end for**
8: Combine local approximations as per (7.5) ▷ §7.4.3

i.e. a particular path $a_{1:n_x}$. Each SLP has a fixed support as the set of variables it draws are fixed by the path, i.e. the program draws from the same fixed set of **`sample`** statements in the same order.

Introducing some arbitrary indexing for the set of SLPs, we use A_k to denote the path for the k^{th} SLP (i.e. $a_{1:n_{x,k}} = A_k$ for every possible realization of this SLP). The set of of all possible execution paths is now given by $A = \{A_k\}_{k=1}^{K}$, where K must be countable (but need not be finite). For the example in Figure 7.2, this set consists of two paths $A_1 = [\#l_1, \#l_4]$ and $A_2 = [\#l_1, \#l_7, \#l_8]$, where we use $\#l_j$ to denote the lexical address of the **`sample`** statement is on the j^{th} line. Note that, for a given program, each SLP is uniquely defined by its corresponding path A_k; we will sometimes use A_k to denote an SLP.

Dividing a program into its constituent SLPs implicitly partitions the overall target density into disjoint regions, with each part defining a sub-model on the corresponding subspace. The unnormalized density $\gamma_k(x)$ of the SLP A_k is defined with respect to the variables $\{x_i\}_{i=1}^{n_{x,k}}$ that are paired with the addresses $\{a_i\}_{i=1}^{n_{x,k}}$ of A_k (where we have used the notation $n_{x,k}$ to emphasize that this is now fixed). We use $\mathcal{X}_k$ to denote its corresponding support. Note that the union of all the $\mathcal{X}_k$ is the support of the original program, $\mathcal{X} = \bigcup_{k=1}^{K} \mathcal{X}_k$.

Analogously to (7.1), we now have that the density of SLP k is $\pi_k(x) = \gamma_k(x)/Z_k$ where

$$\begin{aligned}\gamma_k(x) &:= \gamma(x)\mathbb{I}[x \in \mathcal{X}_k] \\ &= \mathbb{I}[x \in \mathcal{X}_k] \prod_{i=1}^{n_{x,k}} f_{A_k[i]}(x_i|\eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j), \qquad (7.3)\end{aligned}$$

$$Z_k := \int_{x \in \mathcal{X}_k} \gamma_k(x)dx. \qquad (7.4)$$

Unlike for (7.1), $n_{x,k}$ and A_k are now, critically, deterministic variables so that the support of the problem is fixed. Though b_j and n_y may still be stochastic, they do not effect the reference measure of the program and so this does not cause a problem when trying to perform MCMC sampling.

Following our example in Figure 7.2, for A_1 we have $x_{1:2} = [z0, z1]$, $\mathcal{X}_1 = \{[x_1, x_2] \in \mathbb{R}^2 \mid x_1 < 0\}$, and $\gamma_1(x) = \mathcal{N}(x_1; 0, 2)\mathcal{N}(x_2; -5, 2)\mathcal{N}(y_1; x_2, 2)\mathbb{I}[x_1 < 0]$. For A_2, we instead have $x_{1:3} = [z0, z2, z3]$, $\mathcal{X}_2 = \{[x_1, x_2, x_3] \in \mathbb{R}^3 \mid x_1 \geq 0\}$ and $\gamma_2(x) = \mathcal{N}(x_1; 0, 2)\mathcal{N}(x_2; 5, 2)\mathcal{N}(x_3; x_2, 2)\mathcal{N}(y_1; x_3, 2)\mathbb{I}[x_1 \geq 0]$.

To implement this divide step, we now need a mechanism for establishing the SLPs. This can either be done by trying to extract them all upfront, or by dynamically discovering them as the inference runs, see §7.6.2.

7.4.2 Conquer

Given access to the different SLPs produced by the divide step, DCC's conquer step looks to carry out the local inference for each. Namely, it aims to produce a set of estimates for the individual SLP densities $\pi_k(x)$ and the corresponding marginal likelihoods Z_k. As each SLP has a fixed support, this can be achieved with conventional inference approaches, with a large variety of methods potentially suitable. Note that $\pi_k(x)$ and Z_k need not be estimated using the same approach, e.g. we may use an MCMC scheme to estimate $\pi_k(x)$ and then introduce a separate estimator for Z_k. One possible estimation strategy is given in §7.6.1.

An important component in carrying out this conquer step effectively is to note that it is not usually necessary to obtain estimates of equally-high fidelity for all SLPs. Specifically, SLPs with small marginal likelihoods Z_k only make a small contribution to the overall density and thus do not require as accurate estimation as SLPs with large Z_k. As such, it will typically be beneficial to carry out ***resource allocation*** as part of the conquer step,

that is, to generate our estimates in an online manner where at each iteration we use information from previous samples to decide the best SLP(s) to update our estimates for. See §7.6.3 for one possible such strategy.

7.4.3 Combine

The role of DCC's combine step is to amalgamate the local estimates from the individual SLPs to an overall estimate of the distribution for the original program. For this, we can simply note that, because the supports of the individual SLPs are disjoint and their union is the complete program, we have $\gamma(x) = \sum_{k=1}^{K} \gamma_k(x)$ and $Z = \sum_{k=1}^{K} Z_k$, such that the unnormalized density and marginal likelihoods are both additive. Consequently, we have

$$\begin{aligned} \pi(x) &= \frac{\sum_{k=1}^{K} \gamma_k(x)}{\sum_{k=1}^{K} Z_k} = \frac{\sum_{k=1}^{K} Z_k \pi_k(x)}{\sum_{k=1}^{K} Z_k} \\ &\approx \frac{\sum_{k=1}^{K} \hat{Z}_k \hat{\pi}_k(x)}{\sum_{k=1}^{K} \hat{Z}_k} =: \hat{\pi}(x) \end{aligned} \tag{7.5}$$

where $\hat{\pi}_k(x)$ and $\hat{Z}_k$ are the SLP estimates generated during the conquer step. Note that, by proxy, this also produces the overall marginal likelihood estimate $\hat{Z} := \sum_{k=1}^{K} \hat{Z}_k$.

When using an MCMC sampler for $\pi_k(x)$, $\hat{\pi}_k(x)$ will take the form of an empirical measure comprising of a set of samples, i.e. $\hat{\pi}_k(x) = \frac{1}{N_k} \sum_{m=1}^{N_k} \delta_{\hat{x}_{k,m}}(x)$. If we use an importance sampling or particle filtering based approach instead, our empirical measure will compose of weighted samples. We note that in this case, the $\hat{Z}_k$ term in the numerator of (7.5) will cancel with any potential self-normalization term used in $\hat{\pi}_k(x)$, such that we can think of using the estimate $\pi(x) \approx (\sum_{k=1}^{K} \hat{\gamma}_k(x))/(\sum_{k=1}^{K} \hat{Z}_k)$.

7.5 Theoretical Correctness

We now demonstrate that the outlined general DCC approach is consistent (as $T \to \infty$ where T is the number of iterations) given some simple assumptions about the individual component strategies. At a high level, these assumptions are that the estimators used for each SLP, $\hat{\pi}_k$ and $\hat{Z}_k$, are themselves consistent, we use an SLP extraction strategy that will eventually uncover all of the SLPs with finite probability mass, and our resource allocation strategy selects each SLP infinitely often given an infinite number of iterations. More formally we have the following result

Theorem 4. *If Assumptions 4-8 hold, then the empirical measure, $\hat{\pi}(\cdot)$, produced by DCC converges weakly to the conditional distribution of the program in the limit of large number of iterations T :*

$$\hat{\pi}(\cdot) \xrightarrow{d} \pi(\cdot) \quad as \quad T \to \infty.$$

The proof of this result is as follows. We start by explaining the required assumptions, before going on to the main theoretical result. Our first assumption, which may initially seem highly restrictive but turns out to be innocuous, is that we only split our program into a finite number of sub-programs:

Assumption 4. *The total number of sub-programs K is finite.*

We note that this assumption is not itself used as part of the consistency proof, but is a precursor to Assumptions 7 and 8 being satisfiable. We can always ensure the assumption is satisfied even if the number of SLPs is infinite; we just need to be careful about exactly how we specify a sub-program. Namely, we can introduce a single sub-program that combines all the SLPs whose path is longer than n_{thresh}, i.e. those which invoke $n_x > n_{\text{thresh}}$ sample statements, and then ensure that the local inference run for this specific sub-program is suitable for problems with stochastic support (e.g. we could ensure we always use importance sampling from the prior for this particular sub-program). If n_{thresh} is then set to an extremely large value such that we can safely assume that the combined marginal probability of all these SLPs is negligible, this can now be done without making any notable adjustments to the practical behavior of the algorithm. In fact, we do not even envisage this being necessary for actual implementations of DCC: it is simply a practically inconsequential but theoretically useful adjustment of the DCC algorithm to simplify its proof of correctness. Moreover, the fact that we will only ever have finite memory to store the SLPs means that practical implementations will generally have this property implicitly anyway.

For better notational consistency with the rest of the paper, we will use the slightly inexact convention of referring to each of these sub-programs as an SLP from now on, such that the K^{th} "SLP" may actually correspond to a collection of SLPs whose path length is above the threshold if the true number of SLPs is infinite.

Our second and third assumptions simply state that our local estimators are consistent given sufficient computational budget:

Assumption 5. *For every SLP $k \in \{1, \ldots, K\}$, we have a local density estimate $\hat{\pi}_k$ (taking the form on an empirical measure) which converges weakly to the corresponding conditional distribution of that SLP π_k in limit of large allocated budget S_k, where $\pi_k(x) \propto \gamma(x)\mathbb{1}[x \in \mathcal{X}_k]$, $\gamma(x)$ is the unnormalized distribution of the program, and $\mathcal{X}_k$ is the support corresponding to the SLP k.*

Assumption 6. *For every SLP, we have a local marginal probability estimate $\hat{Z}_k$ which converges in probability to the corresponding to true marginal probability $Z_k = \int \gamma(x)\mathbb{1}[x \in \mathcal{X}_k]dx$ in limit of large allocated budget S_k. We further assume that if $Z_k = 0$, then $\hat{Z}_k$ also equals 0 with probability 1 (i.e. we never predict non-zero marginal probability for SLPs that contain no mass).*

The final part of this latter assumption, though slightly unusual, will be satisfied by all conventional estimators: it effectively states that we do not assign finite mass in our estimate to any SLP for which we are unable to find any valid traces with non-zero probability.

Our next assumption is that the SLP extraction strategy will uncover all K SLPs in finite time.

Assumption 7. *Let T_{found} denote the number of iterations the DCC approach takes to uncover all SLPs with $Z_k > 0$. We assume that T_{found} is almost surely finite, i.e. $P(T_{found} < \infty) = 1$.*

A sufficient, but not necessary, condition for this assumption to hold is to use a method for proposing SLPs that has a non-zero probability of proposing a new trace from the prior. This condition is satisfied by LMH style proposals like the RMH proposal we adopt in practice. We note that as with Assumption 4, this assumption is not itself used as part of the consistency proof, but is a precursor to Assumption 8 (below) being satisfiable.

Our final assumption is that our resource allocation strategy asymptotically allocates a finite proportion of each of its resources to each SLP with non-zero marginal probability:

Assumption 8. *Let T denote the number of DCC iterations and $S_k(T)$ the number of times that we have chosen to allocate resources to an SLP k after T iterations. We assume that there exists some $\epsilon > 0$ such that*

$$\forall k \in \{1, \ldots, K\}.\ Z_k > 0 \implies \frac{S_k(T)}{T} > \epsilon. \tag{7.6}$$

Given these assumptions, we are now ready to demonstrate the consistency of the DCC algorithm as follows:

Proof. By Assumption 8, we have that for all k with $Z_k > 0$, $S_k \to \infty$ as $T \to \infty$. Using this along with Assumptions 5 and 6 gives us that, in the limit $T \to \infty$,

$$\hat{Z}_k \xrightarrow{p} Z_k \quad \forall k \in \{1, \ldots, K\} \tag{7.7}$$

$$\hat{\pi}_k(\cdot) \xrightarrow{d} \pi_k(\cdot) \quad \forall k \in \{1, \ldots, K\} \text{ with } Z_k > 0 \tag{7.8}$$

so that all our local estimates converge.

The result now follows through a combination of Equation 7.5, linearity, and Slutsky's theorem. Namely, let us consider an arbitrary bounded continuous function $f : \mathcal{X} \to \mathbb{R}$, for which we have

$$\int f(x)\hat{\pi}(dx) = \frac{\int f(x) \sum_{k=1}^{K} \hat{Z}_k \hat{\pi}_k(dx)}{\sum_{k=1}^{K} \hat{Z}_k} = \frac{\sum_{k=1}^{K} \int f(x) \hat{Z}_k \hat{\pi}_k(dx)}{\sum_{k=1}^{K} \hat{Z}_k}.$$

Given Equations (7.7) and (7.8), using Slutsky's theorem, we can conclude that as $T \to \infty$, the above integral converges to

$$\frac{\sum_{k=1}^{K} \int f(x) Z_k \pi_k(dx)}{\sum_{k=1}^{K} Z_k} = \frac{\int f(x) \sum_{k=1}^{K} Z_k \pi_k(dx)}{Z} = \frac{\int f(x)\gamma(dx)}{Z} = \mathbb{E}_{\pi(x)}[f(x)].$$

We thus see that the estimate for the expectation of $f(x)$, which is calculated using our empirical measure $\hat{\pi}(\cdot)$, converges to its true expectation under $\pi(\cdot)$. As this holds for an arbitrary integrable $f(x)$, this ensures, by definition, that $\hat{\pi}(\cdot)$ converges in distribution to $\pi(\cdot)$, thereby giving the desired result. □

7.6 DCC in Anglican

We now outline a particular realization of our DCC framework. It is implemented in Anglican and can be used to run inference automatically for any valid Anglican program. As part of this, we suggest particular strategies for the individual components left unspecified in

Algorithm 7.2 An Implementation of DCC in Anglican

Input: Program ***prog***, number of iterations T, inference hyper-parameters Φ (e.g. number of initial iterations T_0, times proposed threshold C_0),

Output: Posterior approximation $\hat{\pi}$ and marginal likelihood estimate $\hat{Z}$

Execute ***prog*** forward multiple times (i.e. ignore observes) to obtain an initial set of discovered SLPs A^{total}
for $t = 1, \ldots, T$ **do**
    Select the model(s) k' in A^{total} whose proposed times $C_{k'} \geq C_0$ and add them into A^{active}
    if exist any new models **then**
        Initialize the inference with N parallel MCMC chains
        Perform T_{init} optimization step for each chain by running a "greedy" RMH locally and only accepting samples with higher $\hat{\gamma}_k$ to boost burning-in; store only the last MCMC samples as the initialization for each chain
        Draw M importance samples using each previous MCMC samples as proposal to generate initial estimate for $\hat{Z}_k$
    end if
    Step 1: select a model $\mathbf{A}_{\mathbf{k}^*}$
    Choose a sub-model with index k^* with the maximum utility value as per Equation 3.2
    Step 2: perform local inference on $\mathbf{A}_{\mathbf{k}^*}$
    Perform one or more RMH step locally for all N chains of model K^* to update $\hat{\pi}_{k^*}$
    Draw M importance samples using each previous MCMC samples as proposal to update $\hat{Z}_{k^*}$
    Step 3: explore new models (optional)
    Perform one RMH step using a global proposal for all N chains to discover more SLPs $A_{k''}$
    Add $A_{k''}$ to A^{total} if $A_{k''} \notin A^{total}$; increment the $C_{k''}$
end for
Combine local approximations into overall posterior estimate $\hat{\pi}$ and overall marginal likelihood estimate $\hat{Z}$ as per (7.5)

the last section, but emphasize that these are far from the only possible choices; DCC should be viewed more as a general framework. An algorithm block is provided in Algorithm 7.2.

7.6.1 Local Estimators

Recall that the goal for the local inference is to estimate the local target density $\pi_k(x)$ and the local marginal likelihood Z_k. Straightforward choices include (self-normalized) importance sampling and SMC as both return a marginal likelihood estimate $\hat{Z}_k$. However, knowing good proposals for these a priori is challenging and, as we discussed in §7.3, naïve choices are unlikely to perform well.

Thankfully, each SLP has a fixed support, which means many of the complications that make inference challenging for universal PPSs no longer apply. In particular, we can use conventional MCMC samplers—such as MH, HMC, or MwG—to approximate $\pi_k(x)$. Due to the fact that individual variable types may be unknown or not even fixed, we have elected to use MwG in our implementation, but note that more powerful inference approaches like HMC may be preferable when they can be safely applied.

To estimate the local target density $\pi_k(x)$ for a given SLP A_k, DCC establishes a multiple-chain MCMC sampler in order to ensure a good performance in the setting with high dimensional and potentially multi-modal local densities. Explicitly, we perform one RMH step in each chain for N independent chains in total at each iteration. Suppose the total iteration to run local inference in A_k is T_k. With all the MCMC samples $(\hat{x}^{(k)}_{1:N,1:T_k})$ within A_k, we then have the estimator

$$\hat{\pi}_k(x) := \frac{1}{NT_k} \sum_{n=1}^{N} \sum_{t=1}^{T_k} \delta_{\hat{x}^{(k)}_{n,t}}(\cdot). \tag{7.9}$$

As MCMC samplers do not directly provide an estimate for Z_k, we must introduce a further estimator that uses these samples to estimate it. For this, we use PI-MAIS [Martino et al., 2017]. Though ostensibly an adaptive importance sampling algorithm, PI-MAIS [Martino et al., 2017] is based around using the set of N proposals each centered on the outputs of an MCMC chain. It can be used to generates marginal likelihood estimates from a set of MCMC chains, as we require.

More precisely, given the series of previous generated MCMC samples, $\hat{x}^{(k)}_{1:N,1:T_k}$, PI-MAIS introduces a mixture proposal distribution for each iteration of the chains by using the combination of separate proposals (e.g. a Gaussian) centered on each of those chains:

$$q^{(k)}_t(\cdot|\hat{x}^{(k)}_{1:N,t}) := \frac{1}{N} \sum_{n=1}^{N} q^{(k)}_{n,t}(\cdot|\hat{x}^{(k)}_{n,t}) \quad \text{for } t \in \{1, 2, \ldots, T_k\}. \tag{7.10}$$

This can then be used to produce an importance sampling estimate for the target, with Rao-Blackwellization typically applied across the mixture components, such that M samples, $(\tilde{x}^{(k)}_{n,t,m})_{m=1}^{M}$, are drawn separately from each $q^{(k)}_{n,t}$ with weights

$$\tilde{w}^{(k)}_{n,t,m} := \frac{\gamma_k(\tilde{x}^{(k)}_{n,t,m})}{q^{(k)}_t(\tilde{x}^{(k)}_{n,t,m}|\hat{x}^{(k)}_{n,t})} \text{ with } \tilde{x}^{(k)}_{n,t,m} \sim q^{(k)}_{n,t}(\cdot|\hat{x}^{(k)}_{n,t}), \tag{7.11}$$

$$\text{for } m \in \{1, \ldots, M\} \text{ and } n \in \{1, \ldots, N\}. \tag{7.12}$$

We then have the marginal likelihood estimate $\hat{Z}_k$ as

$$\hat{Z}_k := \frac{1}{NT_kM} \sum_{n=1}^{N} \sum_{t=1}^{T_k} \sum_{m=1}^{M} \tilde{w}_{n,t,m}^{(k)}. \tag{7.13}$$

An important difference for obtaining $\hat{Z}_k$ using an adaptive IS scheme with multiple MCMC chain as the proposal, compared to vanilla importance sampling (from the prior), is that MCMC chains form a much more efficient proposal than the prior as they gradually converge to the local target distribution $\pi_k(x)$. These chains are running locally, i.e. restricted to the SLP A_k, which means that the issues of the LMH transitioning between SLPs as discussed in §7.3 no longer apply and local LMH could maintain a much higher mixing rate where it becomes the standard MwG sampler on a fixed set of variables. Furthermore, the benefit of having multiple chains is that they will approximate a multi-modal local density better. With N chains, we no longer require one chain to discover all the modes but instead only need each mode being discovered by at least one chain. As a result, Equation 7.13 provides a much more accurate estimator than basic methods.

An interesting point of note is that one can also use the importance samples generated by the PI-MAIS for the estimate $\hat{\pi}_k(x)$, where $\hat{\pi}_k(x)$ from Equation 7.9 will be

$$\hat{\pi}_k(x) := \sum_{n=1}^{N} \sum_{t=1}^{T_k} \sum_{m=1}^{M} \bar{w}_{n,t,m}^{(k)} \delta_{\tilde{x}_{n,t,m}^{(k)}}(\cdot), \quad \text{where } \bar{w}_{n,t,m}^{(k)} := \tilde{w}_{n,t,m}^{(k)} \Big/ \sum_{n=1}^{N} \sum_{t=1}^{T_k} \sum_{m=1}^{M} \tilde{w}_{n,t,m}^{(k)}. \tag{7.14}$$

The relative merits of these approaches depend on the exact problem. For problems where the PI-MAIS forms an efficient adaptive importance sampler, the estimate it produces will be preferable. However, in some cases, particularly high-dimensional problems, this sampler may struggle, so that it is more effective to take the original MCMC samples. Though it might seem that we are doomed to fail anyway in such situations, as the struggling of the PI-MAIS estimator is likely to indicate our Z_k estimates are poor, this is certainly not always the case. In particular, for many problems, one SLP will dominate, i.e. $Z_{k^*} \gg Z_{k \neq k^*}$ for some k^*. In that case, we do not necessarily need accurate estimates of the Z_k's to achieve an overall good approximation of the posterior. We just need to identify the dominant Z_k.

7.6.2 Discovering SLPs

To divide a given model into its constituent sub-models expressed by SLPs, we need a mechanism for discovering them automatically. One possible approach would be to analyze

the source code of the program using static analysis techniques [Chaganty et al., 2013; Nori et al., 2014], thereby extracting the set of possible execution paths of the program at compilation time. Though potentially a viable choice in some scenarios, this can be difficult to achieve for all possible programs in a universal PPS. For example, the number of possible paths may be unbounded. We therefore take an alternative approach that discovers SLPs dynamically at run–time as part of the inference. In general, it maintains a set of SLPs encountered so far, and *remembers* any new SLP discovered by the MCMC proposals at each iteration.

Our approach starts by executing the program forward for T_0 iterations to generate some sample execution traces from the prior. The paths traversed by these sampled traces are recorded, and our set of SLPs is initialized as that of these recorded paths. At subsequent iterations, after each local inference iteration, we then perform one *global* MCMC step based on our current sub-model and trace, producing a new trace with path $A_{k'}$ that may or may not have changed. If $A_{k'}$ corresponds to an existing SLP, this sample is discarded (other than keeping count of the number of proposed transitions into each SLP). However, if it corresponds to an unseen path, it is added to our set of SLPs as a new sub-model, followed by T_w MCMC steps restricted to that path to burn-in. The sample of the final step will be stored as initiation for the future local inference on that path.

The key difference between our strategy and running a single global MCMC sampler (e.g. RMH), is that we do not need this new sample to be *accepted* for the new SLP to be "discovered". Because, as we explained in § 7.3, making effective proposals into a high–density region of a new configuration is very challenging, it is unlikely that the new trace sample we propose has high density: even if it corresponds to a path with large Z_k, we are unlikely to immediately sample a good set of draws to accompany it. Therefore, the global MCMC sampler is very likely to miss or ***forget*** new SLPs since it accepts/rejects movements based on one sample.

DCC, on the other hand, overcomes this problem by first ***remembering*** the path information of any newly proposed SLP, and then carrying out a few local warm-up iterations, before deciding whether an SLP is a promising sub-model or not in later exploration. As a result, DCC does not suffer from the reliance on forward sampling as per RMH to discover

new SLPs. Moreover, our scheme inherits the hill-climbing behavior of MCMC to discover SLP in a more efficient way: it can find the sub-models of high posterior mass even under an extremely small prior probability, as will be shown in §7.7.2.

To better understand our SLP extraction procedure in Anglican, one can imagine that we maintain two stacks of information of SLPs: one `total` stack and one `active` stack (A^{total} and A^{active} respectively in Algorithm 7.2). The `total` stack records all the information of all discovered SLPs and the `active` stack keeps the SLPs that are believed to be promising so far.

Let's now have a detailed look at Algorithm 7.2 together. To prevent the rate of models being generated from outstripping our ability to perform inference on current models, we probably only want to perform inference on a subset of all possible sub-models given finite computational budget, which is A^{active}. However, to determine which SLP might be good, i.e. have high posterior mass, is somewhat part of the job of the inference. Therefore, in DCC, we propose that if a model in $\{A^{total} \setminus A^{active}\}$ is "close" enough to the promising models discovered so far as in A^{active}, it would be regarded as being *potentially good* and added to A^{active}. To quantify the closeness, we count how many times a discovered SLP not in A^{active} gets proposed by a model in A^{active} during the global exploration step (at line $\#l_{14}$, Algorithm 7.2). We will add a newly discovered SLP into A^{active} for the resource allocation only when its count reaches some threshold C_0 ($\#l_4$).

One might worry the number of models in A^{active} might still go beyond the capacity of the inference engine. We have considered the following design choices to avoid this situation in the DCC in Anglican. The first one is to increase C_0 accordingly as the iteration grows. Intuitively, an SLP needs to be proposed more as more computational resources is available to demonstrate that it might be a good one. Another design choice provided is to control the total number of sub-models in A^{active}. For instance, before one can add a new model in A^{active}, one needs to take an SLP out of A^{active}, e.g. the one with the least $\hat{\gamma}_k$, if the upper bound of the total "active" number has reached. Our DCC also randomly chooses one "non-active" SLPs to perform local inference to ensure the overall correctness.

These design choices, though, are not always necessary if the number of possible sub-models does not explode naturally. For example, in the GMM example, this number is

controlled by the value of a Poisson random variable which diminishes quickly, in which case we do not need to further bound the number of active sub-models. But when it comes to the GMM with misspecified prior where the possible number of active models can easily grow quickly, these design choices become essential. They prevent too much computation resources from waste on keeping discovering new models rather than used to perform inference in the discovered ones.

From the practical perspective, one might also want to "split" on discrete variables as their values are likely to affect the downstream program path. This means that for a specific discrete variable(s), not only their addresses but also their values are included in defining a program path. It equivalently transforms sampling a discrete variable to observing the variable being a fixed value. The benefit of doing so is when performing inference locally, we no longer need to propose changes for that discrete variable but instead evaluate its conditional probability. Therefore, we can "avoid" proposing samples out of current SLP too often to waste the computation. The posterior of that discrete variable can be obtained from the local marginal likelihood estimates. In our implementation of DCC in Anglican, we enable this feature but would require the user to specify which discrete variables that they want to "split" on. Automatically distinguishing which discrete variable will or will not affect program paths is beyond the scope of this paper, and we shall leave it for future work.

7.6.3 Allocating Resources Between SLPs

At each iteration we must choose an SLP from the discovered set to perform local inference on. Though valid, it is not wise to split our computational resources evenly among all SLPs; it is more important to ensure we have accurate estimates for SLPs with large Z_k. Essentially, we have a multi-armed bandit problem where we wish to develop a strategy of choosing SLPs that will lead to the lowest error in our *overall* final estimate $\hat{\pi}$. Though it might seem that this is a problem that DCC has introduced, it is actually an inherent underlying problem that must always be solved for models with stochastic support; DCC is simply making the problem explicit. Namely, we do not know upfront which SLPs have significant mass and so any inference method must deal with the computational trade–off involved in figuring this out. Conventional approaches do this in an implicit, and typically highly inefficient, manner.

For example, MCMC relies on the relatively frequency of individual transitions between SLPs to allocate resources, which will generally be extremely inefficient for finite budgets.

To address this, we introduce a resource allocation scheme based on an upper confidence bounding (UCB) approach developed in Rainforth et al. [2018]. Specifically, we use the existing SLPs estimates to construct a utility function that conveys the relative merit of refining the estimates for each SLP, balancing the need for ***exploitation***, that is improving the estimates for SLPs currently believed to have large relative Z_k, and ***exploration***, that is improving our estimates for SLPs where our uncertainty in Z_k is large.

At each iteration, we then update the estimate for the SLP which has the largest utility, defined as

$$U_k := \frac{1}{S_k}\left(\frac{(1-\delta)\hat{\tau}_k}{\max_k\{\hat{\tau}_k\}} + \frac{\delta\hat{p}_k}{\max_k\{\hat{p}_k\}} + \frac{\beta\log\sum_k S_k}{\sqrt{S_k}}\right)$$

where S_k is the number of times we have previously performed local inference on A_k; $\hat{\tau}_k$ is the current estimate of the "reward" of A_k, incorporating both how much mass the SLP contains and how efficient our estimates are for it; $\hat{p}_k$ is a targeted exploration term that helps identify promising SLPs that we are yet to establish good estimates for; $0 \leq \delta \leq 1$ is a hyperparameter controlling the trade–off between these terms; and $\beta > 0$ is the standard optimism boost hyper-parameter.

As proved by Rainforth et al. [2018, §5.1], the optimal asymptotic allocation strategy is to choose each A_k in proportion to $\hat{\tau}_k = \sqrt{Z_k^2 + (1+\kappa)\sigma_k^2}$ where κ is a smoothness hyper-parameter, Z_k is the local marginal likelihood, and σ_k^2 is the variance of the weights of the individual samples used to generate Z_k. Intuitively, this allocates resources not only to the SLPs with high marginal probability mass, but also to the ones having high variance on our estimate of it. We normalize each $\hat{\tau}_k$ by the maximum of $\hat{\tau}_{1:K}$ as the reward function in UCB is usually in $[0, 1]$.

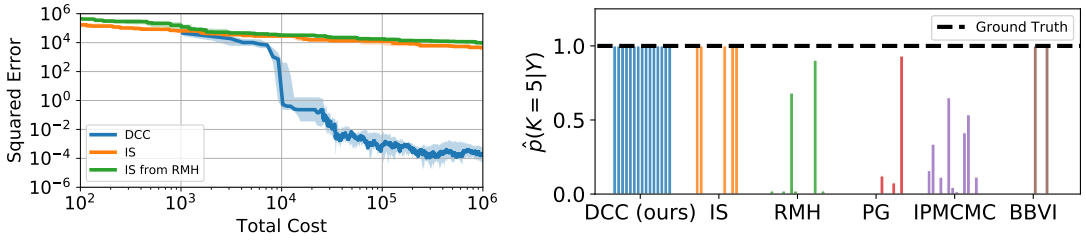
The target exploration term $\hat{p}_k$ is a subjective tail-probability estimate on how much the local inference *could improve* in estimating the local marginal likelihood if given more computations. This is motivated by the fact that estimating Z_k accurately is difficult, especially at the early stage of inference. One might miss substantial modes if only relying on optimism boost to undertake exploration. As per Rainforth et al. [2018], we realize

this insight by extracting additional information from the log weights. Namely, we define $\hat{p}_k := P(\hat{w}_k(T_a) > w_{th}) \approx 1 - \Psi_k(\log w_{th})^{T_a}$, which means the probability of obtaining at least one sample with weight w that exceeds some threshold weight w_{th} if provided with T_a "look-ahead" samples. Here $\Psi_k(\cdot)$ is a cumulative density estimator of the log local weights (eg. the cumulative density function for the normal distribution), T_a is a hyperparameter, and w_{th} can be set to the maximum weight so far among all SLPs. If $\hat{p}_k$ is high, it implies that there is a high chance that one can produce higher estimates of Z_k given more budget.

Note that for our implementation of DCC, there is an optimization step when a new model is discovered at the first time. (See line 6 of Algorithm 7.2.) Specifically, once a new candidate of SLP has been selected to be added into A_{active}, DCC firstly performs T_{init} optimization steps to boost initialization of inference. Informally, we want to burn in the MCMC chains quickly such that the initialization of each chain would be close to the local mode of the target distribution. By doing so, DCC applies a "greedy" RMH where it only accepts the MCMC samples with the larger $\hat{\gamma}_k$, which enforces the hill-climbing behavior in MCMC to discover modes. Note that only the MCMC samples of the last step in the optimization will be stored as the initialization of each chain and therefore this optimization strategy will not affect the correctness of the local inference.

We finish this section by noting that our choices for the particular DCC implementation in Anglican straightforwardly ensure that these assumptions are satisfied (provided we take the aforementioned care around Assumption 4). Namely:

- Using RMH for the local inferences will provide $\hat{\pi}_k$ that satisfies Assumption 5.
- Using PI-MAIS will provide $\hat{Z}_k$ that satisfies Assumption 6 provided we construct this with a valid proposal.
- The method for SLP extraction has a non-zero probability of discovering any of the SLPs with $Z_k > 0$ at each iteration because it has a non-zero probability of proposing a new trace from prior, which then itself has a non-zero probability of proposing each possible SLP.



(a) Squared error $\| \log \hat{Z} - \log Z_{true} \|^2$. (b) Estimates $\hat{p}(K = 5 \mid y_{1:N_y})$ for 15 different runs.

Figure 7.4: Results for DCC and baselines for the GMM example outlined in §7.3. [Left] Convergence in squared error in log marginal likelihood estimate $\| \log \hat{Z} - \log Z_{true} \|^2$. The solid line corresponds to the median across 15 runs and the shading region $25\% - 75\%$ quantiles. Note that none of IS, RMH, PG, IPMCMC, and BBVI provide such at estimate, hence their omission; an additional baseline of drawing importance samples from a proposal centered on and RMH chain was considered instead. [Right] Final estimates for $p(K = 5 \mid y_{1:N_y})$ for each of the 15 runs, for the ground truth is roughly 0.9998. In both cases, the ground truth was estimated using a very large number of importance samples with a manually adapted proposal. We see that DCC substantially outperforms the baselines.

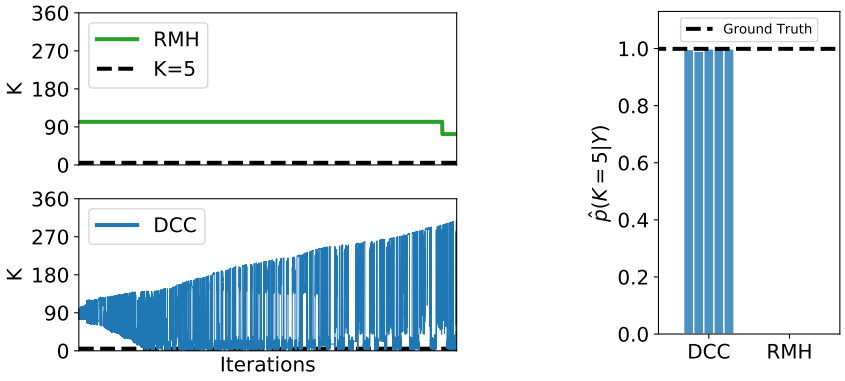
- The resource allocation strategy will eventually choose each of its possible actions with non-zero rewards (which in our case are all SLPs with $Z_k > 0$) infinitely often, as was proven in Rainforth et al. [2018].

7.7 Experiments

7.7.1 Gaussian Mixture Model (GMM)

We now further investigate the GMM example with an unknown number of clusters introduced in §7.3. Its program code written in Anglican is provided in Appendix C.2.1. We compare the performance of DCC against five baselines: importance sampling (from prior) (IS), RMH [Le, 2015], Particle Gibbs (PG) [Andrieu et al., 2010], interacting Particle MCMC (IPMCMC) [Rainforth et al., 2016b], and Black-box Variational Inference (BBVI) [Paige, 2016], taking the same computational budget of 10^6 total samples for each method.

We first examine the convergence of the overall marginal likelihood estimate $\hat{Z}$. Here IS is the only baseline which can be used directly, but we also consider drawing importance samples centered around the RMH chain in a manner akin to PI-MAIS. Figure 7.4 [Left] shows that DCC outperforms both by many orders of magnitude. The sudden drop in the error for DCC is because the dominant sub-model with $K = 5$ is typically discovered



(a) SLP visit order

(b) $\hat{p}(K = 5 \mid y_{1:N_y})$

Figure 7.5: Comparison of DCC to RMH on GMM with misspecified prior. [Left] visit order of SLPs (i.e. sampled value of K at each iteration) for single run. [Right] final posterior estimates for 5 different runs.

after taking around 10^4 samples. Further investigation of that SLP allows DCC to improve the accuracy of the estimate. DCC has visited 23 to 27 sub-models (out of *infinitely* many) among all 15 runs.

We next examine the posterior distribution of K and report the estimates of $p(K = 5 \mid y_{1:N_y})$ in Figure 7.4 [Right]. We see that all methods other than DCC struggle. Here , the accuracy of the posterior of K reflects the accuracy in estimating the relative masses of the different SLPs, i.e. Z_k, explicitly or implicitly. The dimension of this model varies between one and infinity and the posterior mass is concentrated in a small sub-region ($K = 5$) with small prior mass. It is therefore challenging for the baselines to either to learn each marginal likelihood simultaneously (eg. in IS) or to estimate the relative masses implicitly through transitions between configurations using an MCMC sampler. By breaking down the model into sub-problems, DCC is able to overcome these challenges and provide superior posterior estimator for the overall model.

7.7.2 GMM with Misspecified Prior

To further test the capability of each method to discover SLPs—and to examine the MCMC-esque behavior for DCC in SLP space in particular—we adjust the GMM example above slightly so that K now has a, high misspecified, prior of $\text{Poisson}(90)+1$, keeping everything else the same. The dominant SLP is still $K = 5$ (with around 0.9976 posterior mass), but his now has an extremely low prior probability (around 10^{-14}). Consequently, finding this

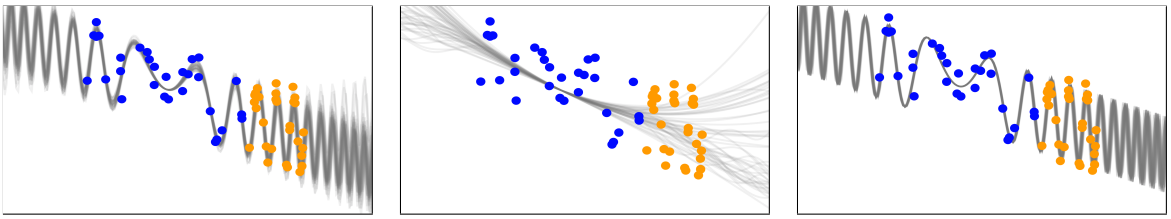
dominant SLP is only practically possible if the algorithm exhibits an effective hill climbing behavior in SLP space. We only now compare DCC to RMH on the basis that: a) none of the baselines could deal with simpler case before, such that they will inevitably not be able to deal with this harder problem; and b) RMH is the only baseline where one might expect to see some hill climbing behavior in SLP space.

Figure 7.5(a) shows the trace plot for the SLP visit history (i.e. sampled K at each iteration) of each method. As we can see in the bottom plot, DCC starts from the SLPs of K around 90, influenced by the prior, and gradually discovers smaller Ks with higher posterior mass, also exploring large values of K as well. This implies a MCMC-esque hill-climbing behavior guided by our SLP discovery scheme. Moreover, the trace plot also demonstrates the resource allocation within DCC where it gradually spends more computation for SLPs with lower Ks while still maintaining a degree of exploration. Both factors are essential for the resulting accurate posterior approximation shown in Figure 7.5(b).

By comparison, RMH gets stuck in its initialized SLP (Figure 7.5(a) top): for the one run shown it only makes one successful transition to another SLP and never gets anywhere close to region of SLPs with significant mass. Equivalent behavior was experience in all the other runs (not shown). As a result, it does not produce a reasonable posterior estimate as shown Figure 7.5(b); in fact, it always returns an estimate of exactly 0 as it never discovers this SLP. It is worth noting that the local mixing of RMH between SLPs here is even worse than in the previous example. This is because the larger K at which the sampler is initialized induces a higher dimensional space on program draws, i.e. $\mu_{1:K}$. This is catastrophic for RMH because it is effectively importance sampling when transitioning between SLPs and thus suffers acutely from the curse of dimensionality. DCC, meanwhile, gracefully deals with this because of its ability to remember SLPs that are proposed but not accepted and then subsequently perform effective localized inference that exploits hill–climbing effects in the space of the draws of that SLP.

7.7.3 Function Induction

Function induction is an important task for automated machine learning [Duvenaud et al., 2013; Kusner et al., 2017]. In PPSs, it is typically tackled using a probabilistic context free grammar (PCFG) [Manning and Schutze, 1999]. Here we consider such a model where



(a) DCC (ours) (b) IS (c) RMH

Figure 7.6: Posterior distribution $p(\Theta|D)$ estimated by DCC, IS, and RMH under the same computation. Blue points represent the observed data D and orange ones the test data D'. Grey lines are the posterior samples of the functions from the run with the highest LPPD among 15 independent runs of the three algorithms.

we specify the structure of a candidate function using a PCFG and a distribution over the function parameters, and estimate the posterior of both for given data. Our PCFG consists of four production rules with fixed probabilities: $e \to \{x \mid x^2 \mid \sin(a * e) \mid a * e + b * e\}$, where x and x^2 are terminal symbols, a and b are unknown coefficient parameters, and e is a non-terminal symbol. The model also has prior distributions over each coefficient parameters. See Appendix C.2.2 for details.

To generate a function from this model, we must sample both a PCFG rollout and the corresponding parameters. Let Θ be the collection of all the latent variables used in this generative process. That is, Θ consists of the sequence of the discrete variables recording the choices of the grammar rules and all coefficients in the sampled structure. Conditioned on txhe training data D, we want to infer the posterior distribution $p(\Theta|D)$, and calculate the posterior predictive distribution $p(D'|D)$ for test data $D' = \{x_n, y_n\}_{n=1}^N$.

In our experiment, we control the number of sub-models by requiring that the model use the PCFG in a restricted way: a sampled function structure should have depth at most 3 and cannot use the plus rule consecutively. We generate a synthetic dataset of 30 training data points from the function $f(x) = -x + 2\sin(5x^2)$ and compare the performance of DCC to our baselines on estimating the posterior distribution and the posterior predictive under the same computational budget of 10^6 samples and 15 independent runs. BBVI is omitted from this experiment due to it failing to run at all.

Figure 7.6 shows the posterior samples generated by DCC, IS, and RMH for one run, with the training data D marked blue and the test data D' in orange. The DCC samples

Table 7.1: Mean and one standard derivation of the LPPD over 15 independent runs.

	DCC (ours)	IS	RMH	PG	IPMCMC
LPPD	**-28.56 ± 0.41**	-73.18 ± 1.08	-32.69 ± 8.51	-200.82 ± 126.63	-70.58 ± 4.71

capture the periodicity of the data and provides accurate extrapolation, while retaining an appropriate degree of uncertainty. This indicates good inference results on both the structure of a function and the coefficients. Though RMH does find some good functions, it becomes stuck in a particular mode and does not fully capture the uncertainty in the model, leading to poor predictive performance.

Table 7.1 shows the test log posterior predictive density (LPPD), defined as

$$\sum_{n=1}^{N} \log \int_{\Theta} p(y_n|x_n, \Theta)p(\Theta|D)d\Theta,$$

of all approaches. DCC substantially outperforms all the baselines both in terms of predictive accuracy and stability. IS, PG, and IPMCMC all produced very poor posterior approximations leading to very low LPPDs. RMH had an LPPD that is closer to DCC, but which is still substantially inferior.

A further issue with RMH was its high variance of the LPPD. This is caused by this model being multi-modal and RMH struggling to move: it gets stuck in a single SLP and fails to capture the uncertainty. Explicitly, 4 sub-models (out of 26) contain most of the probability mass. Two of them are functions of the form used to generate the data, $f(x) = a_1 x + a_2 \sin(a_3 x^2)$, modulo symmetry of the $+$ operator. The other two have the form $f(x) = a_1 \sin(a_2 x) + a_3 \sin(a_4 x^2)$, which can also match the training data well in the region $(-1.5, 1.5)$ as $a_1 \sin(a_2 x) \approx a_1 a_2 x$ for small values of $a_2 x$. Note that the local distributions are also multi-modal due to various symmetries, for example $a_1 \sin(a_2 x^2)$ and $-a_1 \sin(-a_2 x^2)$, meaning the local inference task is non-trivial even in low dimensions.

To test the effectiveness of the resource allocation strategy, we further investigate the computational resources spent for each SLP by looking at the convergence of the local marginal likelihood estimates $\hat{Z}_k$. In Figure 7.7, the sub-models 15 and 18 correspond to the form $f(x) = a_1 x + a_2 \sin(a_3 x^2)$ and its mirror, which contain the most posterior mass. The sub-models 23 and 24 correspond to $f(x) = a_1 \sin(a_2 x) + a_3 \sin(a_4 x^2)$ and are the second largest modes. Figure 7.7 implies that DCC indeed spends more computational

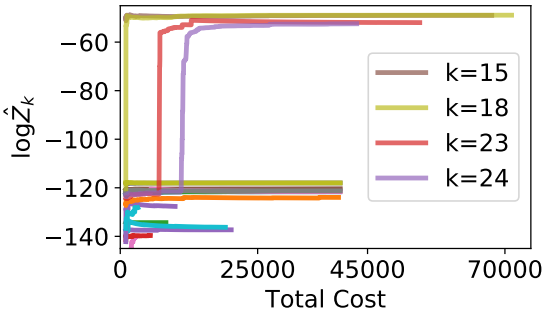


Figure 7.7: Convergence of DCC's $\log Z_k$ estimate for each SLP and corresponding total amount of resources spent.

resource on sub-models with high probability mass (as signified by the higher final total cost), while also exploring the other sub-models occasionally.

7.8 Conclusion

In this paper, we have proposed *Divide, Conquer, and Combine (DCC)*, a new inference strategy for probabilistic programs with stochastic support. We have shown that, by breaking down the overall inference problem into a number of separate inferences of sub-programs with fixed support, the DCC framework can provide substantial performance improvements over existing approaches which directly target the full program. To realize this potential, we have shown how to implement a particular instance of DCC as an automated engine in the PPS Anglican and shown that this outperforms existing baselines on three example problems.

8

Conclusion and Future Directions

In this thesis, we have mainly focused on non–standard probabilistic models and automating inference in such models. We have made contributions in designing more tractable inference algorithms both in a probabilistic programming context and more generally.

In Chapter 3, we proposed a new class of adaptive inference algorithms called inference tress (ITs) building upon ideas from Monte Carlo tree search. Compared to conventional adaptive methods which mainly emphasize exploitation, we argue that exploration is also important, especially in the areas where we are uncertain about. By carrying out explicit control over the exploitation–exploration, ITs are able to alleviated common pathologies in existing baselines, and have outperformed previous state-of-the-art methods on tested models with multiple and far separated modes. The self-adaptive nature also makes ITs possible to be implemented in an automated manner.

In Chapter 5, we investigated the design of a probabilistic programming system (PPS) and alleviated the common drawback among existing restricted PPSs where the class of models is extremely constrained. We proposed a novel low-level, first-order probabilistic programming language (LF-PPL), which provides a principled way to extend these systems such that more complicated models can be allowed and more advanced inference methods can be incorporated. The underlying mathematical formalism of LF-PPL enables it not only to be used in its own right, but also to serve as a promising compilation target for existing

systems. As a result, one is able to handle a wider range of models in restricted PPSs as well as maintain the efficiency of the inference procedure following the idea of LF-PPL.

In Chapter 7, we took a look on universal PPSs, which provide a powerful framework for specifying rich and complicated models. However, performing inference in such model is extremely hard, let alone automate this process. In particular, we have investigated a class of important but challenging models where their support may vary between executions. For these models, we have shown that existing inference engines produce surprisingly unsatisfactory performance, often doing no better than importance sampling from the prior. To this end, we introduced a novel inference scheme, Divide, Conquer and Combine (DCC), which is able to remain efficient for such models, and can be implemented as an automated and generic PPS inference engine. At a high-level, DCC alleviates the difficulties of handling varying support by dividing an overall program into sub-programs, conquering about each sub-program before combining local results together.

There are still many open research questions to be answered about automating inference for non–standard models. The first question is how to extract useful information from a complex model. The reason for doing so is that, to implement an inference algorithm in a PPS as an automated engine, it is essential that any necessary information can be extracted and accessed in a convenient way. For models satisfying the restricted PPS setup, one can compile the model into a convenient abstraction such as a graphical model. But this becomes infeasible when the model goes beyond a certain scope, and many universal PPSs employ an execution based approach. Can we extract more useful information of a universal probabilistic program beyond its execution trace? Or, can we transform the program in a way that it is more convenient to manipulate by the inference engine? Existing tools such as static analysis from programming language community might have the potential to offer improvements [Luo et al., 2020].

Moreover, as we have seen earlier, models with a varying number of dimensions are extremely challenging for a PPS, and we have proposed DCC, which is a step further for the sampling-based inference paradigm. But such models still cause troubles for other inference approaches such as variational inference methods. For instance, although the PPS pyro [Bingham et al., 2018] is designed for a generic purpose that is able to support

such model, they actually turn such model into a fixed dimensional case by upper bounding the number of variables when applying the variational engine. Can we still interpret the variational approximation in the standard way? Are the standard estimators for the ELBO and its gradient still valid when the model has a varying dimension? As a first step forward, it might be worth to re-consider ideas from DCC and formulate the problem according to variational inference. This is because DCC fundamentally breaks the stochastic support problem into a few fixed support sub-problem. Within a sub-problem, conventional methods can be used as normal because the support is fixed, and so is variational inference. Therefore, one might sidestep this problem by taking the DCC framework.

Appendices

A

Appendix for Inference Trees

A.1 Discrete Variables

As explained in §3.4, our method for generating partitions means that they are not always disjoint as required by Assumption 1, most notably when x is discrete. Fortunately, we can still deal with this case by noting that the required properties of Assumption 1 *do* hold in the space of $v_{1:D}$. This will require no algorithmic changes, but will require additional consideration in the proof. In this case we replace Assumption 1 with the following

Assumption 9. *Let $v_{1:D} \sim u(v_{1:D})$ be uniformly distributed on the unit hypercube $\mathfrak{Z}_D = [0,1]^T$ and let $x = g(v_{1:D})$ have density $q(x)$ and support $x \in \mathbb{X}$, where $q(x)$ is a valid importance sampling proposal for $\gamma(x)$ (see e.g. [Owen, 2013]). For every independent estimator set $\ell \in \{1, \ldots, L\}$, we are given a) a partitioning $\{B_{\ell,i}\}_{i \in \mathcal{I}_\ell}$ of $\mathfrak{Z}_D$ such that $B_{\ell,i} \cap B_{\ell,j} = \emptyset$ for $i \neq j$ and $\bigcup_{i \in \mathcal{I}_\ell} B_{\ell,i} = \mathfrak{Z}_D$, and b) a family $\{\hat{\varphi}_{\ell,i}^{N_{\ell,i}}\}_{i \in \mathcal{I}_\ell}$ of estimated measures on $\mathfrak{Z}_D$ for all $N \geq 1$:*

$$\hat{\varphi}_{\ell,i}^{N_{\ell,i}}(\cdot) := \frac{1}{N_{\ell,i}} \sum_{n=1}^{N_{\ell,i}} w_{\ell,i}^n \, \delta_{\hat{v}_{1:D,\ell,i}^n}(\cdot)$$

for some random variables $w_{\ell,i}^n$ and $\hat{v}_{1:D,\ell,i}^n$ such that each $\hat{\varphi}_{\ell,i}^{N_{\ell,i}}$ converges weakly to the following measure on $\mathfrak{Z}_D$ as $N_{\ell,i} \to \infty$

$$\frac{\gamma(g(v_{1:D}))\mathbb{1}(v_{1:D} \in B_{\ell,i})u(v_{1:D})}{q(g(v_{1:D}))}.$$

Further each marginal probability estimate converges in probability as follows

$$\hat{Z}_{\ell,i}^{N_{\ell,i}} := \frac{1}{N_{\ell,i}} \sum_{n=1}^{N_{\ell,i}} w_{\ell,i}^n \xrightarrow{p} \int_{\mathbb{3}_D} \frac{\gamma(g(v_{1:D}))\mathbb{1}(v_{1:D} \in B_{\ell,i})}{q(g(v_{1:D}))} u(dv_{1:D}).$$

Corollary 2. *Let $\hat{\gamma}_{\ell,i}^{N_{\ell,i}}$ denote the pushforward measure of $\hat{\varphi}_{\ell,i}^{N_{\ell,i}}$ (as per $\hat{\gamma}_{\ell,i}^{N_{\ell,i}}$ in Assumption 1), then if Assumptions 9, 2 and 3 hold,*

$$\hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}} := \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \hat{\gamma}_{\ell,i}^{N_{\ell,i}} \tag{A.1}$$

converges weakly to the measure $\gamma(x)$ on $\mathbb{X}$ as $R \to \infty$.

Proof. As per Lemma 1, the estimates for $\ell \notin \mathcal{L}_0$ need not converge but do not affect the final estimate. We again demonstrate the result by considering an arbitrary continuous function $f : \mathbb{X} \to \mathbb{R}$ for which we have

$$\begin{aligned}
\int f(x)\, \hat{\gamma}^{\{N_{\ell,i}\}_{\ell,i}}(dx) &= \int f(x) \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \hat{\gamma}_{\ell,i}^{N_{\ell,i}}(dx) \\
&= \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \int f(x) \hat{\gamma}_{\ell,i}^{N_{\ell,i}}(dx) \\
&= \frac{1}{\sum_{\ell=1}^{L} k_\ell\left(N_\ell\right)} \sum_{\ell=1}^{L} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \int f(g(v_{1:D})) \hat{\varphi}_{\ell,i}^{N_{\ell,i}}(dv_{1:D})
\end{aligned}$$

which using Assumptions 9 and 2 converges as $R \to \infty$ to

$$\begin{aligned}
&\frac{1}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)} \sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \sum_{i \in \mathcal{I}_\ell} \int \frac{f(v_{1:D})\gamma(g(v_{1:D}))\mathbb{1}(v_{1:D} \in B_{\ell,i})}{q(g(v_{1:D}))} u(dv_{1:D}) \\
&\quad = \frac{1}{\sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right)} \sum_{\ell \in \mathcal{L}_0} k_\ell\left(N_\ell\right) \int \frac{f(v_{1:D})\gamma(g(v_{1:D}))}{q(g(v_{1:D}))} u(dv_{1:D}) \\
&\quad = \int \frac{f(v_{1:D})\gamma(g(v_{1:D}))}{q(g(v_{1:D}))} u(dv_{1:D}) = \int \frac{f(x)\gamma(x)}{q(x)} q(dx) = \int f(x)\, \gamma(dx)
\end{aligned}$$

as required. □

Given this corollary, we can now trivially extend Theorem 1 to the setting where Assumption 9 holds instead of Assumption 1 using the same arguments.

A.2 Additional Details on Theory

To further see why Lemma 1 holds, we provide more details of the proof.

Let $X \sim \gamma(x)$, $X_1 = X\mathbb{1}(x \in A_1)$ and $X_2 = X\mathbb{1}(x \in A_2)$ where A_1 and A_2 are disjoint partitions and their union is the complete space, such that X and $X_1 + X_2$ are equal

in distribution as we always have exactly $X = X_1 + X_2$. We now consider $X > 0$ only for simplicity where $X_2 = 0$ implies $X \in A_1$ and thus $X \notin A_2$ due to the disjointness of the partitions, and we will generalize the result shortly. Now define the random variable $\Theta = \Theta_1 + \Theta_2$ where Θ_1 and Θ_2 are random variables that we assume converge respectively to X_1 and X_2 (which is equivalent to Assumption 1). The central question to answer here is whether Θ converges in distribution to X. To show this, we firstly have

$$\int f(\theta_1 + \theta_2) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2) = \int f(\theta_1 + \theta_2) \mathbb{1}(\theta_1 \neq 0, \theta_2 = 0) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2) +$$
$$\int f(\theta_1 + \theta_2) \mathbb{1}(\theta_1 = 0, \theta_2 \neq 0) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2) + \int f(\theta_1 + \theta_2) \mathbb{1}(\theta_1 \neq 0, \theta_2 \neq 0) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2)$$

as the sum of these three identity functions is always 1. The third term must further be zero in the limit because, by construction, we have in the limit that either $\theta_1 = 0$ or $\theta_2 = 0$. This property results from how we form our estimators, and it is essential for the proof since one would not necessarily be able to eliminate the last term in a general setting.

Further, the first term equals to

$$\begin{aligned}
&\int f(\theta_1 + \theta_2) \left(1 - \mathbb{1}(\theta_1 = 0)\right) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2) \\
&= \int f(\theta_1) \left(1 - \mathbb{1}(\theta_1 = 0)\right) p_{\theta_1\theta_2}(d\theta_1 + d\theta_2) = \int f(\theta_1) \left(1 - \mathbb{1}(\theta_1 = 0)\right) p_{\theta_1}(d\theta_1) \\
&= \int f(\theta_1) p_{\theta_1}(d\theta_1)
\end{aligned}$$

which converges weakly to $\int f(x_1) p_{x_1}(dx_1)$ because Θ_1 converges to X_1 in distribution. Applying similar logic, we have the second term converging to $\int f(x_2) p_{x_2}(dx_2)$. Taking the sum and applying similar logic for θ in a reverse order, we have

$$\begin{aligned}
&\int f(x_1) p_{x_1}(dx_1) + \int f(x_2) p_{x_2}(dx_2) \\
&= \int f(x_1) \mathbb{1}(x_1 \neq 0) p_{x_1}(dx_1) + \int f(x_2) \mathbb{1}(x_1 \neq 0) p_{x_2}(dx_2) \\
&= \int \left(f(x_1) \left(1 - \mathbb{1}(x_1 = 0)\right) + f(x_2) \left(1 - \mathbb{1}(x_2 = 0)\right)\right) p_{x_1 x_2}(dx_1 + dx_2) \\
&= \int f(x_1 + x_2) p_{x_1 x_2}(dx_1 + dx_2) = \int f(x) p_x(dx)
\end{aligned}$$

where $\mathbb{1}(x_2 = 0)$ implies $\mathbb{1}(x_1 \neq 0, x_2 = 0)$, and the last line is because we have $X_1 + X_2$ being equivalent in distribution to X. We have then proven that $\Theta_1 + \Theta_2$ converges in distribution to X.

We can now generalize this result to the case that goes beyond $X > 0$. In doing so, we adjust our definition to $X_1 = h(X)\mathbb{1}(X \in A_1)$, and $X_2 = h(X)\mathbb{1}(X \in A_2)$ where h is a bijective mapping to strictly positive numbers (e.g., for one dimensional X, we can take $h(X) = \exp(X)$) such that $X_2 = 0$ still implies that $X_1 \in A_1$ and thus $X_1 \notin A_2$. Following the same logic, we have $\Theta_1 + \Theta_2$ converges to $h(X)$, and thus $h^{-1}(\Theta_1 + \Theta_2)$ converges to X. We can now generalize this result to the case with more than two partitions, which is comparatively straightforward as we can apply the same logic recursively. Consequently, we can show the validity of combining separate estimators over disjoint space partitions (i.e. summing over i in (3.17)).

Moreover, as defined in Assumption 1, each estimator set ℓ is independent of each other. Therefore, we can show that the combination of separate estimators for the same target (i.e. summing over ℓ in (3.17)) which each individually converges to that target also converges to the target.

Putting all together, we have enhanced the proof of Lemma 1. We reiterate that a central point of the proof results from the disjoint nature of separate estimators such that the only term that depends on the joint distribution can be eliminated.

A.3 Estimates for Empirical Variance and Effective Sample Size

When calculating terms such as the effective sample size (ESS) [Owen, 2013], we need to take care about the fact that our traversal strategy implies additional implicit weights through the N_j and c_j. In short, our "expected squared weight" should not be simply calculated using $\frac{1}{N_j}\sum_{n=1}^{N}(w_i^n)^2$ but instead using the scheme we now introduce. Given this expected squared weight estimator, a number of useful estimators will follow naturally.

We start by introducing an alternative formulation of the combined marginal likelihood estimate of a node as follows

$$\hat{Z}_j = (1-c_j)\frac{1}{N_j}\sum_{n=1}^{N_j} w_j^n + c_j\left(\hat{Z}_{\ell_j} + \hat{Z}_{r_j}\right) = \frac{1}{\mathfrak{N}_j}\sum_{\mathfrak{n}=1}^{\mathfrak{N}_j} w_{\mathfrak{n}} k_{\mathfrak{n}} \frac{\mathfrak{N}_j}{N_{j(\mathfrak{n})}} \tag{A.2}$$

where $\mathfrak{N}_j$ is the total number of samples from the current node j and its decedents that have been used for estimation, $\{w_{\mathfrak{n}}\}_{\mathfrak{n}=1}^{\mathfrak{N}_j}$ is the union of all the weights, $k_{\mathfrak{n}}$ is a child preference weight associated with sample $\mathfrak{n}$ (e.g. $(1-c_j)$ for a sample form the current node local

estimate, $c_j(1-c_j)$ for a sample from the local estimate of a child if that node is an internal node, etc.), and $N_{j(\mathfrak{n})}$ is the number of local samples generated the node $j(\mathfrak{n})$ which the sample $\mathfrak{n}$ belongs to (the sum of all numbers of the local samples from the current node and its decedents equals to $\mathfrak{N}_j$). We thus see that the *true* sample weights in our combined estimator are $w_{\mathfrak{n}}k_{\mathfrak{n}}\mathfrak{N}_j/N_{j(\mathfrak{n})}$ and so our estimator for the squared weight is

$$\begin{aligned}\hat{\varsigma}_j^2 :=& \frac{1}{\mathfrak{N}_j}\sum_{\mathfrak{n}=1}^{\mathfrak{N}_j}\left(w_{\mathfrak{n}}k_{\mathfrak{n}}\frac{\mathfrak{N}_j}{N_{j(\mathfrak{n})}}\right)^2 = \mathfrak{N}_j\sum_{\mathfrak{n}=1}^{\mathfrak{N}_j}\left(\frac{w_{\mathfrak{n}}k_{\mathfrak{n}}}{N_{j(\mathfrak{n})}}\right)^2 \\ =& \mathfrak{N}_j\left((1-c_j)^2\frac{1}{N_j^2}\sum_{n=1}^{N_j}\left(w_j^n\right)^2 + c_j^2\left(\frac{\hat{\varsigma}_{\ell_j}^2}{\mathfrak{N}_{\ell_j}} + \frac{\hat{\varsigma}_{r_j}^2}{\mathfrak{N}_{r_j}}\right)\right). \end{aligned} \tag{A.3}$$

$\hat{\varsigma}_j^2/\mathfrak{N}_j$ can be propagated in a similar fashion to other estimates, allowing $\hat{\varsigma}_j^2$ to be estimated at any node.

Given $\hat{\varsigma}_j^2$, we can straightforwardly construct various useful estimators. For example, the Monte Carlo estimator for the variance of the weight produced by a given traversal, used in (3.13), is given by

$$\sigma_j^2 := \frac{\mathfrak{N}_j}{\mathfrak{N}_j - 1}\left(\hat{\varsigma}_j^2 - \hat{Z}_j^2\right) \tag{A.4}$$

where the first term is Bessel's correction. The ESS, on the other hand, is

$$\text{ESS}_j := \frac{\mathfrak{N}_j\hat{Z}_j^2}{\hat{\varsigma}_j^2}. \tag{A.5}$$

A.4 Derivation of the Pure-Exploitation Target

For this derivation, it will be convenient to first consider the case where the children we are deciding between are both leaf nodes and that there is some arbitrary (unknown) target function f, such that combined child estimate (not including the parent) is given by

$$\hat{\mu}_{\text{ch}} := \frac{1}{N_\ell}\sum_{n=1}^{N_\ell} w_\ell^n \hat{f}_\ell^n + \frac{1}{N_r}\sum_{n=1}^{N_r} w_r^n \hat{f}_r^n \tag{A.6}$$

where $\hat{f}_i^n := f(\hat{x}_i^n)$. Now the mean squared error (MSE) of our estimator decomposes in the standard manner

$$\mathbb{E}[(\hat{\mu}_{\text{ch}} - \mu_j)^2] = \text{Var}[\hat{\mu}_{\text{ch}}] + (\mathbb{E}[(\hat{\mu}_{\text{ch}} - \mu_j)])^2$$

where the second term is the biased squared and all terms are implicitly conditioned on N_ℓ and N_r. Though the finite sample bias of our estimator is difficult to assert, we know that it

vanishes as $N_\ell, N_r \to \infty$ and, due to the central limit theorem, we can safely assume this happens faster than the standard deviation vanishes. Thus asymptotically, we only need to consider the variance to minimize the MSE. Now, invoking the conditional independence given N_ℓ and N_r of each child estimator and each sample within those estimators, we have

$$\begin{aligned}\mathrm{Var}[\hat{\mu}_{\mathrm{ch}}] &= \mathrm{Var}\left[\frac{1}{N_\ell}\sum_{n=1}^{N_\ell} w_\ell^n \hat{f}_\ell^n\right] + \mathrm{Var}\left[\frac{1}{N_r}\sum_{n=1}^{N_r} w_r^n \hat{f}_r^n\right] \\ &= \frac{1}{N_\ell}\mathrm{Var}\left[w_\ell^1 \hat{f}_\ell^1\right] + \frac{1}{N_r}\mathrm{Var}\left[w_r^1 \hat{f}_r^1\right].\end{aligned}$$

Using the stratified sampling results of, for example, [Carpentier et al., 2015], it is straightforward to show that the subsequent optimal strategy is to set

$$N_\ell \propto \sqrt{\mathrm{Var}\left[w_\ell^1 \hat{f}_\ell^1\right]} \quad \text{and} \quad N_r \propto \sqrt{\mathrm{Var}\left[w_r^1 \hat{f}_r^1\right]}.$$

Now assuming that the weights and evaluations are independent (remembering that we are considering an arbitrary f) we have

$$\begin{aligned}\mathrm{Var}\left[w_\ell^1 \hat{f}_\ell^1\right] &= \mathbb{E}\left[\left(\hat{f}_\ell^1\right)^2\right]\mathbb{E}\left[\left(w_\ell^1\right)^2\right] - \left(\mathbb{E}\left[\hat{f}_\ell^1\right]\right)^2\left(\mathbb{E}\left[w_\ell^1\right]\right)^2 \\ &= \mathbb{E}\left[\left(\hat{f}_\ell^1\right)^2\right]\mathrm{Var}\left[w_\ell^1\right] + \mathrm{Var}\left[\hat{f}_\ell^1\right]\left(\mathbb{E}\left[w_\ell^1\right]\right)^2 \\ &= \mathrm{Var}\left[\hat{f}_\ell^1\right]\left(\mathrm{Var}\left[w_\ell^1\right]\left(1 + \frac{\left(\mathbb{E}\left[\hat{f}_\ell^1\right]\right)^2}{\mathrm{Var}\left[\hat{f}_\ell^1\right]}\right) + \left(\mathbb{E}\left[w_\ell^1\right]\right)^2\right)\end{aligned}$$

and similarly for $\mathrm{Var}\left[w_r^1 \hat{f}_r^1\right]$. We thus have that the optimal strategy is to set (using σ to denote standard deviation)

$$N_\ell \propto \frac{\sigma\left[\hat{f}_\ell^1\right]}{\sigma\left[\hat{f}_\ell^1\right] + \sigma\left[\hat{f}_r^1\right]}\sqrt{\left(\mathrm{Var}\left[w_\ell^1\right]\left(1 + \frac{\left(\mathbb{E}\left[\hat{f}_\ell^1\right]\right)^2}{\mathrm{Var}\left[\hat{f}_\ell^1\right]}\right) + \left(\mathbb{E}\left[w_\ell^1\right]\right)^2\right)}. \tag{A.7}$$

Here the first term depends only on the unknown target function. Though one might want to potentially postulate a particular dependence of $\sigma\left[\hat{f}_\ell^1\right]$ on the relative volume of the nodes, we will just presume the ratio is unknown and conservatively set it to 1, falling in line with standard approaches for Bayesian inference where we aim to sample in proportion to the posterior, rather than artificially producing more samples in larger areas of the space to account for the potential of higher variation in the target function.

The second term depends only on the statistics of the sample weights and the ratio $\left(\mathbb{E}\left[\hat{f}_\ell^1\right]\right)^2 / \mathrm{Var}\left[\hat{f}_\ell^1\right]$. As f is unknown, we also do not know this ratio. However, we

do know it must vary between 0 (when $\mathbb{E}\left[\hat{f}_\ell^1\right] = 0$ or $\mathrm{Var}\left[\hat{f}_\ell^1\right] \to \infty$) and ∞ (when $\mathrm{Var}\left[\hat{f}_\ell^1\right] = 0$, i.e. the function is flat). These two respective extremes give

$$N_\ell \propto \sqrt{\left(\mathrm{Var}\left[w_\ell^1\right] + \left(\mathbb{E}\left[w_\ell^1\right]\right)^2\right)} = \sqrt{\mathbb{E}[(w_\ell^1)^2]} \quad \text{and} \quad N_\ell \propto \sigma[w_\ell^1].$$

The latter of these corresponds to the optimal strategy for estimating the marginal likelihood, as would be expected from considering the stratified sampling results of Carpentier et al. [2015] applied to estimating $\mathbb{E}[w]$. However, this strategy gives no consideration of the need to produce samples from areas of high posterior density to capture possible variations in the target function and so is highly inappropriate. Assuming the former extreme is more conservative and spends time sampling in regions of high probability mass and also those of high weight uncertainty.

Rather than taking a particular extreme, we treat

$$\kappa := \frac{\left(\mathbb{E}\left[\hat{f}_\ell^1\right]\right)^2}{\mathrm{Var}\left[\hat{f}_\ell^1\right]}, \quad \kappa \in [0, \infty]$$

explicitly as a parameter of the traversal algorithm, where higher values of κ give more emphasis to estimating the marginal likelihood and to accurate prediction of expectations of smoothly varying functions, while lower values of κ give more emphasis to sampling regions in proportion to their marginal probabilities. We note the interesting, and perhaps counter-intuitive, result that even when κ is its minimum possible value, the optimal traversal strategy is still not to sample in proportion to marginal probability, except in the special case where the variance of the weights is zero.

Thus far we have omitted the fact that we eventually want a normalized estimator. We deal with the former by noting that we intend to *separately* propagate the unnormalized estimate and the marginal likelihood estimate. Thus, except at the root node, our aim is to propagate low variance estimates of both, rather than simply low variance estimates of the ratio. Though we do not do further analysis to assess this, we choose by default to set $\kappa = 1$, to reflect the fact that we thus always explicitly care about the marginal likelihood estimate.

We have also thus far omitted the fact that we need to calculate traversal strategies when the children are not leaves. Here we can use the same analysis but need to replace $\mathbb{E}\left[w_\ell^1\right]$ and $\mathrm{Var}\left[w_\ell^1\right]$ with appropriate combined estimators. For the former, we can simply

use $\hat{Z}_\ell$. For the latter, we need a notion of a "single-traversal" variance in the marginal likelihood estimate. Such a metric was derived as $\hat{\sigma}_\ell^2$ in Appendix A.3. We thus arrive at our derivation of the unnormalized exploitation reward of node ℓ as

$$\hat{\tau}_\ell := \sqrt{\hat{Z}_\ell^2 + (1+\kappa)\hat{\sigma}_\ell^2}. \tag{A.8}$$

A.5 Additional Density Estimation Details

A.5.1 Additional Heuristics

Even though we cannot calculate it, we know that there is maximum possible log weight for each node, namely

$$\log w_j^* = \max_{v_{1:D} \in B_j} \log \gamma(g(v_{1:D})) + \log \|B_j\| - \log q(g(v_{1:D})).$$

Consequently, our density estimator (which is defined on the full real line) will typically slightly overestimate the probability of a sampling falling above the threshold. In particular, if there is a large number of samples at the node and we are only using a simple density estimator for ψ, we may continue to except to exceed the threshold even when previous samples suggest a saturation below the threshold.

Let $\mathrm{e}(T)$ to denote the event $\left\{\max(w_j^{1:D}) > w_{\text{th}}\right\}$, i.e. the event that one of T independent samples exceeds the threshold if we draw T samples. We now have $P(\mathrm{e}(T)) = 1 - \Psi(\log w_{\text{th}})^T$. We can further condition this on the event that we have not already seen the threshold exceeded using the likelihood $P(\neg\mathrm{e}(N_j)|\mathrm{e}(T))$. To define this, we introduce an additional parameter $\log w_{\text{gap}}$ and define our likelihood to condition on the fact that none of our N_j samples fall above $\log w_{\text{th}}$ with Ψ truncated at $\log w_{\text{tr}} := \log w_{\text{th}} + \log w_{\text{gap}}$ to reflect the fact that the true log weights are bounded, giving

$$P(\neg\mathrm{e}(N_j)|\mathrm{e}(T)) = \left(\frac{\Psi(\log w_{tr}) - \Psi(\log w_{\text{th}})}{\Psi(\log w_{tr})}\right)^{N_j},$$

with Bayes' rule in turn yielding

$$P(\mathrm{e}(T)|\neg\mathrm{e}(N_j)) = \frac{\left(1 - (1 - \Psi(\log w_{\text{th}}))^T\right) P(\neg\mathrm{e}(N_j)|\mathrm{e}(T))}{(1 - (1 - \Psi(\log w_{\text{th}}))^T)\, P(\neg\mathrm{e}(N_j)|\mathrm{e}(T)) + (1 - \Psi(\log w_{\text{th}}))^T}. \tag{A.9}$$

The full definition of $\hat{p}_j^s$ actually used is then given by

$$\hat{p}_j^s := (1 - c_j)\, \frac{P(\mathrm{e}(T)|\neg\mathrm{e}(N_j))}{\mathrm{ESS}_j} + c_j \left(\hat{p}_{\ell_j}^s + \hat{p}_{r_j}^s - \hat{p}_{\ell_j}^s \hat{p}_{r_j}^s\right). \tag{A.10}$$

A.5.2 Additional Intuition and Parameters

At first it might seem counter intuitive to include an ESS scaling term in $\hat{p}_j^s$ as a classic failure case for the ESS as a performance metric is if there multiple modes. However, the scenario where the *local* estimate has a high ESS and multiple modes is expected to be rare. Instead, one will typically have a low local ESS for any node with multiple modes but it may have children with a high ESS giving it a high combined ESS estimate. In these cases, the combined significant probability estimate $\hat{p}_j^s$ should still be high if there is any descendant i with a high $\hat{p}_i^s$ and a low ESS_i. Thus in practice, scaling by the ESS does not cause the high nodes in the tree to miss multiple mode cases, while providing a more reliable metric for nodes low down in the tree.

In our approach, T and $\log w_{\text{gap}}$ constitute fixed parameters which we set to 1000 and 10 respectively as default. On the other hand, w_{th} naturally needs to change as the training progresses. We make the simple choice of setting w_{th} to being the highest weight generated at any node, scaled to adjust for differences $\|B_j\|$ makes to the weight. An unfortunate feature of this choice is that whenever the MAP estimate changes, the $\hat{p}_j^s$ for all nodes must be updated. However, the regularity that this occurs diminishes with the number of iterations, such that it, in practice, does not lead to an increasing per-iteration computational cost as the tree is run longer.

A.6 Additional Details on Refinement Strategy

To define our entropy metric more precisely, recall that the entropy of a continuous uniform distribution $U(s_1, s_2)$ is

$$\textsc{Entropy}(U(s_1, s_2)) = -\int_{s_1}^{s_2} q(v_d) \log q(v_d) \, \mathrm{d}v_d = \ln(s_2 - s_1). \tag{A.11}$$

Assume that we propose a split at a point $s \in (s_1, s_2)$, and that we will later go to the left of this split with a probability P_ℓ and to the right with a probability $P_r = 1 - P_\ell$. This splitting and the traversal strategy give rise to a proposal of a mixture of two uniform distributions that has the following density

$$q_s(v_d) = \begin{cases} d_\ell & \text{if } v_d < s, \text{ where } d_\ell = \frac{P_\ell}{s - s_1} \\ d_r & \text{otherwise, where } d_r = \frac{1 - P_\ell}{s_2 - s} \end{cases} \tag{A.12}$$

The entropy of this proposal is:

$$\text{ENTROPY}(q_s) = -\int_{s_1}^{s} d_\ell \log d_\ell \, \mathrm{d}v_d - \int_{s}^{s_2} d_r \log d_r \, \mathrm{d}v_d = P_\ell \log \frac{1}{d_\ell} + P_r \log \frac{1}{d_r}. \quad \text{(A.13)}$$

We can now use our empirical estimates $\hat{Z}_\ell = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(v_{d,i} \in B_\ell) w_i$ and similarly $\hat{Z}_r = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(v_{d,i} \in B_r) w_i$ to form the empirical estimates of $\hat{P}_\ell$ and $\hat{P}_r$, which defines our entropy metric as

$$\text{ENTROPY}(q_s) = \hat{P}_\ell \log \frac{s - s_1}{\hat{P}_\ell} + \hat{P}_r \log \frac{s_2 - s}{\hat{P}_r} \text{ where } \hat{P}_\ell = \frac{\hat{Z}_\ell}{\hat{Z}_\ell + \hat{Z}_r} \text{ and } \hat{P}_r = \frac{\hat{Z}_r}{\hat{Z}_\ell + \hat{Z}_r},$$

which is trivially equivalent to the loss given in (3.15) up to a normalization constant. We then choose the split $s^* = \arg\min_s \text{ENTROPY}(q_s)$ where the minimization is over our randomly sampled candidate splits.

As a minor additional heuristic aimed at avoiding splits where a small but significant proportion of the tail is contained within one child, we do not in practice use s^* directly, instead reducing the size of the child with lower probability mass by 25% to form a more conservative split.

After choosing the best split among all candidates and separating the space in to B_ℓ and B_r, we run inference restricted to B_ℓ and B_r separately. Then we compare the empirical estimates of the marginal likelihood for each child using a t-test, which shows how likely the results are samples from two different distributions. If the p-value is small, it suggests the split is meaningful. In that case, we accept the split, creating two new child nodes and converting the current leaf node to the parent one. Otherwise, we discard the split and combine the samples, adding them to the estimate of the current node. When the node is revisited, new splits are suggested and the process continues in the same way.

B
Appendix for LF-PPL

B.1 Discontinuous Hamiltonian Monte Carlo

The discontinuous HMC (DHMC) algorithm was proposed by Nishimura et al. [2017]. It uses a coordinate-wise integrator, Algorithm B.1, coupled with a Laplacian momentum to perform inference in models with non-differentiable densities. The algorithm works because the Laplacian momentum ensures that all discontinuous parameters move in steps of $\pm m_b \epsilon$ for fixed constants m_b and step size ϵ, where the index b is associated to each discontinuous coordinate. These properties are advantages because they remove the need to know where the discontinuity boundaries between each region are; the change of the potential energy in the state before and after the $\pm m_b \epsilon$ move provides us with information of whether we have enough kinetic energy to move into this new region. If we do not have enough energy we reflect backwards $\mathbf{p}_b = -\mathbf{p}_b$. Otherwise, we move to this new region with a proposed coordinate update $\mathbf{x}_b^*$ and momentum $\mathbf{p}_b = m_b \cdot sign(\mathbf{p}_b) \cdot \Delta U$. This is in contrast to algorithms such as Reflect, Refract HMC [Afshar and Domke, 2015], that explictly need to know where the discontinuities boundaries are. Hence, it is important to have a compilation scheme that enables one to do that.

The addition of the random permutation ϕ of indices b is to ensure that the coordinate-wise integrator satisfies the criterion of reversibility in the Hamiltonian. Although the integrator does not reproduce the exact solution, it nonetheless preserves the Hamiltonian

Algorithm B.1 Coordinate-wise Integrator. A random permutation ϕ on $\{1, \ldots, B\}$ is appropriate if the induced random sequences $(\phi(1), \ldots, \phi(|B|))$ and $(\phi(|B|), \ldots, \phi(1))$ have the same distribution

function COORDINATEWISE($\mathbf{x}, \mathbf{p}, \epsilon, U$)
  pick an appropriate random permutation ϕ on B
  for $i = 1, \ldots, B$ **do**
    $b \leftarrow \phi(i)$
    $\mathbf{x}^* \leftarrow \mathbf{x}$
    $\mathbf{x}^*_b \leftarrow \mathbf{x}^*_b + \epsilon m_b \cdot sign(\mathbf{p}_b)$
    $\Delta U \leftarrow U(\mathbf{x}^*) - U(\mathbf{x})$
    if $K(\mathbf{p}_b) = m_b|\mathbf{p}_b| > \Delta U$ **then**
      $\mathbf{x}_b \leftarrow \mathbf{x}^*_b$
      $\mathbf{p}_b \leftarrow \mathbf{p}_b - m_b \cdot sign(\mathbf{p}_b) \cdot \Delta U$
    else
      $\mathbf{p}_b \leftarrow -\mathbf{p}_b$
    end if
  end for
  return $\mathbf{x}_b, \mathbf{p}_b$
end function

exactly, even if the density is discontinuous. See Lemma 1 and Theorems 2-3 in Nishimura et al. [2017]. This yields a rejection-free proposal.

Then the DHMC algorithm adpated for LF-PPL and our compilation scheme is shown in Algorithm B.2.

Algorithm B.2 Discontinuous HMC Integrator for the LF-PPL.
χ is a map from random-variable names n in Δ to their values $\mathbf{x}_n$, H is the total Hamiltonian, $\epsilon > 0$ is the step size, and L is the trajectory length.

function DHMC-LFPPL($\Delta, \Gamma, D, F, \mathbf{x}, \mathbf{p}, H, \epsilon, L$)
  $B = \Gamma$; $A = \Delta \setminus \Gamma$
  for $a \in A$ **do** ▷ A represents the set of continuous variables
    $\mathbf{x}_a^0 \leftarrow \mathbf{x}_a$; $\mathbf{p}_a \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$
  end for
  for $b \in B$ **do** ▷ B represents the set of discontinuous variables
    $\mathbf{x}_b^0 \leftarrow \mathbf{x}_b$; $\mathbf{p}_b \sim Laplace(\mathbf{0}, \mathbf{1})$
  end for
  $\forall a \in A,\ \mathbf{x}_a^0 \leftarrow \mathbf{x}_a$; $\mathbf{p}_a \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$
  $\forall b \in B,\ \mathbf{x}_b^0 \leftarrow \mathbf{x}_b$; $\mathbf{p}_b \sim Laplace(\mathbf{0}, \mathbf{1})$
  $U \leftarrow -\textsc{LogJointDensity}(D, F)$
  for $i = 1$ to L **do**
    $U_A \leftarrow U$ with names in B replaced by their values in $\mathbf{x}_B^i$
    $(\mathbf{x}_A^i, \mathbf{p}_A^i) \leftarrow \textsc{HalfStep1}(\mathbf{x}_A^{i-1}, \mathbf{p}_A^{i-1}, \epsilon, U_A)$
    $U_B \leftarrow U$ with names in A replaced by their values in $\mathbf{x}_A^i$
    $(\mathbf{x}_B^i, \mathbf{p}_B^i) \leftarrow \textsc{Coordinate-wise}(\mathbf{x}_B^{i-1}, \mathbf{p}_B^{i-1}, \epsilon, U_B)$
    $U_A \leftarrow U$ with names in B replaced by their values in $\mathbf{x}_B^i$
    $(\mathbf{x}_A^i, \mathbf{p}_A^i) \leftarrow \textsc{HalfStep2}(\mathbf{x}_A^i, \mathbf{p}_A^i, \epsilon, U_A)$
  end for
  $\mathbf{x}^L \leftarrow \mathbf{x}_A^L \cup \mathbf{x}_B^L,\ \mathbf{p}^L \leftarrow \mathbf{p}_A^L \cup \mathbf{p}_B^L$;
  $\mathbf{x}^*, \mathbf{p}^* \leftarrow \textsc{Evaluate}(F,\ \mathbf{x}^L, \mathbf{p}^L)$
  $\alpha \sim Uniform(0, 1)$
  if $\alpha > \min\{1, \exp(H(\mathbf{x}, \mathbf{p}) - H(\mathbf{x}^*, \mathbf{p}^*))\}$ **then**
    return $\mathbf{x}^*, \mathbf{p}^*$
  else
    return $\mathbf{x}, \mathbf{p}$
  end if
end function
function HALFSTEP1($\mathbf{x}, \mathbf{p}, \epsilon, U$)
  $\mathbf{p}' \leftarrow \mathbf{p} - \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x})$
  $\mathbf{x}' \leftarrow \mathbf{x} + \frac{\epsilon}{2}\nabla_{\mathbf{p}'} K(\mathbf{p}')$
  return $(\mathbf{x}', \mathbf{p}')$
end function
function HALFSTEP2($\mathbf{x}, \mathbf{p}, \epsilon, U$)
  $\mathbf{x}' \leftarrow \mathbf{x} + \frac{\epsilon}{2}\nabla_{\mathbf{p}} K(\mathbf{p})$
  $\mathbf{p}' \leftarrow \mathbf{p} - \frac{\epsilon}{2}\nabla_{\mathbf{x}'} U(\mathbf{x}')$
  return $(\mathbf{x}', \mathbf{p}')$
end function

B.2 Program code

```
(let [y (vector -2.0 -2.5 ... 2.8)
      pi [0.5 0.5]
      z1 (sample (categorical pi))
      ...
      z10(sample (categorical pi))
      mu1 (sample (normal 0 2))
      mu2 (sample (normal 0 2))
      mus (vector mu1 mu2)]
  (if (< (- z1) 0)
      (observe (normal mu1 1) (nth y 0))
      (observe (normal mu2 1) (nth y 0)))
  ...
  (if (< (- z10) 0)
     (observe (normal mu1 1) (nth y 9))
     (observe (normal mu2 1) (nth y 9)))
  (mu1 mu2 z1 ... z10))
```

Figure B.1: The LF-PPL version of the Gaussian mixture model detailed in Section 5.6.

```
(let [x (sample (uniform -6 6))
      abs-x (max x (- x))
      z (- (sqrt (* x (* A x))))]
   (if (< (- abs-x 3) 0)
      (observe (factor z) 0)
      (observe (factor (- z 1)) 0))
  x)
```

Figure B.2: The LF-PPL version of the heavy-tailed model detailed in Section 5.6.

C
Appendix for DCC

C.1 Existing PPSs and Inference Engines for Probabilistic Programs with Stochastic Support

In Table C.1 above, we have listed which inference engines from the five categories in §7.3 are supported by each of the most popular universal PPSs. The fundamental difficulty in performing inference on probabilistic models with stochastic support is that the posterior mass of such models is usually concentrated in one or many separated sub-regions which are non-trivial to be fully discovered, especially in a high dimensional space. Moreover, even if a variable exists in different variable configurations, the posterior might shift substantially like the mean μ_1 for the GMM shown in the Figure 7.1, which complicates the design for a "proper" proposal distribution, let alone automate this procedure in PPS. We now have a more detailed look at each category with the related PPSs and uncover the reasons why these engines are not suitable for probabilistic programs with stochastic support.

Importance/rejection sampling Basic inference schemes, such as importance sampling (IS) and rejection sampling (RS), are commonly supported by many PPSs due to their generality and simplicity. See Table C.1. They can be directly applied to a model with stochastic support, but their performance deteriorates rapidly (typically exponentially) as the dimension of the model increases; they suffer acutely from the curse of dimensionality. Furthermore, their validity relies on access to a valid proposal. This can often be easily ensured, e.g. by

Table C.1: List of popular universal PPSs and their supported inference algorithms for models with stochastic support.

	Rejection	IS	SMC	PMCMC	VI	Customized proposal MCMC	Automated proposal MCMC
Venture	✓						
WebPPL	✓		✓	✓			✓
MonadBayes			✓	✓			
Anglican		✓	✓	✓	✓		✓
Turing.jl		✓	✓	✓		✓	✓
Pyro		✓	✓		✓	✓	
Gen		✓	✓		✓	✓	✓
Hakaru		✓				✓	
Stochaskell						✓	

sampling from the prior, but doing this while also ensuring the proposal is efficient can be very challenging; the prior is only a practically viable proposal for very simple problems. Though schemes for developing more powerful proposals in amortized inference settings have been developed [Le et al., 2017; Ritchie et al., 2016a], these are not appropriate for conventional inference problems.

Particle based methods Particle based inference methods, such as Sequential Monte Carlo (SMC) [Doucet et al., 2001], can offer improvements for models with natural sequential structure [Wood et al., 2014; Rainforth et al., 2016b]. More explicitly, SMC improves IS when there exist interleaved observe statements in the model program. However, it similarly rapidly succumbs to the curse of dimensionality in the more general case where this does not occur. Such methods also cannot be used at all if the number of observe statements is not fixed, which can be common for stochastic support problems.

One might then tend to more sophisticated methods such as particle MCMC (PMCMC) methods [Andrieu et al., 2010] but unfortunately these suffer from the same underlying issues. The basic setups of PMCMC include the Particle Independent Metropolis Hasting (PIMH) (in Anglican and Turing.jl) and Particle Gibbs (PG) (in Anglican and Turing.jl [Ge et al., 2018a]), where one uses, respectively, *independent* and *conditional* SMC sweeps as the proposal within the MCMC sampler. These building-block sweeps

suffer from exactly the same issue as conventional SMC, and thus offer no advantage over simple importance sampling without interleaved observe statements.

These advanced variants do though allow one to treat global parameters separately: they update the global parameters using a Metropolis–within–Gibbs (MwG) step and then update the rest latent variables using a (conditional) SMC sweep with those global parameter values. When combined with PIMH, this leads to Particle Marginal Metropolis Hasting (PMMH) algorithm (available in WebPPL [Goodman and Stuhlmüller, 2014], Turing.jl and MonadBayes [Ścibior et al., 2018]). It already constitutes a valid step for PG. Unfortunately, in both cases this MwG suffers from exactly the same issues as those already discussed for LMH. As such, these approaches again offer little advantage over importance sampling without sequential problem structure.

Variational inference Following the discussion in § 7.3, three universal PPSs under our survey that allow Variational Inference (VI) in stochastic support settings are Pyro [Bingham et al., 2018], Gen [Cusumano-Towner et al., 2019] and Anglican [Wood et al., 2014] (note many others allow VI for statistic support, but cannot be used when the support varies). Pyro supports Stochastic Variational Inference (SVI) [Hoffman et al., 2013; Wingate and Weber, 2013; Kucukelbir et al., 2017] and allows an auto-generated guide function (AutoGuide), i.e. the variational proposal, or a user-specified guide. However, AutoGuide only works for some basic probabilistic programs, which we cannot directly apply for the GMM example in § 7.3. With the customized guide, one cannot deal with the exact same model because of the need to upper-bound the number of the variational parameters according to the tutorial[1].

Both Gen and Anglican support the Black box Variational Inference (BBVI) [Ranganath et al., 2014]. We have tested the Anglican's BBVI with the GMM example in § 7.3, but it does not provide very accurate estimates as can be seen in the Figure 7.3.

One key challenge for implementing VI dealing with models with stochastic support is that one might still have never-before-seen variables after finite number of training steps. Moreover, it is usually very challenging in stochastic support settings to ensure that the variational family is defined in a manner that ensures the KL is well defined. For example,

[1] `https://pyro.ai/examples/dirichlet_process_mixture.html`

if using $\mathrm{KL}(q||p)$ (for proposal q and target p), then the KL will be infinite if q places support anywhere p does not. Controlling this with static support is usually not especially problematic, but in stochastic support settings it can become far more challenging.

Overcoming these complications is beyond the scope of our paper but could be a potential interesting direction for future work.

MCMC with customized proposal To perform Markov chain Monte Carlo (MCMC) methods [Metropolis and Ulam, 1949] on the models with stochastic support, one needs to construct the transitional kernel such that the sampler can switch between configurations. A number of PPSs such as Hakaru, Turing.jl, Pyro and Gen allow the user to customize the kernel for Metropolis Hastings (a.k.a programmable kernel) whereas Stohaskell explicitly supports reversible jump Markov chain Monte Carlo (RJMCMC) [Green, 1995, 2003] methods. However, their application is fundamentally challenging as we have discussed in § 7.3 due to the difficulty in designing proposals which can transition efficiently as well as the posterior shift on the variable under different configuration.

MCMC with automated proposal One MCMC method that can be fully automated for PPSs is the Single-site Metropolis Hastings or the Lightweight Metropolis Hastings algorithm (LMH) of [Wingate et al., 2011] and its extensions [Yang et al., 2014; Tolpin et al., 2015; Le, 2015; Ritchie et al., 2016b], for which implementations are provided in a number of systems such as Venture [Mansinghka et al., 2014], WebPPL and Anglican. In particular, Anglican supports LMH and its variants, Random-walk lightweight Metropolis Hastings (RMH) [Le, 2015], which uses a mixture of prior and local proposal to update the selected entry variable x_i along the trace $x_{1:n_x}$.

Many shortcomings of LMH and RMH have been discussed in §7.3 and we reiterate a few points here. Though widely applicable, LMH relies on proposing from the prior whenever the configuration changes for the downstream variables. This inevitably forms a highly inefficient proposal (akin to importance sampling from the prior), such that although samples from different configurations might be proposed frequently, these samples might not be "good" enough to be accepted. This usually causes LMH get stuck in one sub-mode and struggles to switch to other configurations. For example, we have observed this behavior

in the GMM example in Figure 7.4. As a result, LMH typically performs very poorly for programs with stochastic support, particularly in high dimensions.

Note that LMH/RMH could have good mixing rates for many models with fixed support in general akin to standard Metropolis-within-Gibbs methods. This is because when x_i is updated, the rest of the random variables can still be re-used, which ensures a high acceptance rate. That is also why we can still use LMH/RMH as the local inference algorithm within the DCC framework for a fixed SLP to establish substantial empirical improvements.

C.2 Details on Experiments

C.2.1 Gaussian Mixture Model

The Gaussian Mixture Model defined in §7.3 can be written in Anglican as in Figure C.1. The kernel density estimation of the data is shown Figure 7.3 (black line) with the raw data file provided in the code folder.

C.2.2 Function Induction

The Anglican program for the function induction model in in §7.7.3 is shown in Figure C.2. The prior distribution for applying each rule $R = \{e \to x \mid x^2 \mid \sin(a * e) \mid a * e + b * e\}$ is $P_R = [0.3, 0.3, 0.2, 0.2]$. To control the exploding of the number of sub-models, we set the maximum depth of the function structure being three and prohibit consecutive plus. Both our training data (blue points) and test data (orange points) displayed in Figure 7.6 are generated from $f(x) = -x + 2\sin(5x^2)$ with observation noise being 0.5 and the raw data files are provided in the code folder.

```
(defdist lik-dist
   [mus std-scalar]
   []   ; auxiliary bindings
   (sample* [this] nil)  ;; not used
   (observe* [this y]  ;; customize likelihood
      (reduce log-sum-exp
         (map #(- (observe* (normal %1 std-scalar) y)
                  (log (count mus)))
                mus))))

(with-primitive-procedures [lik-dist]
  (defquery gmm-open [data]
    (let [poi-rate 9
          ;; sample the number of total clusters
          K (+ 1 (sample (poisson poi-rate)))
          lo 0.
          up 20.
          ;; sample the mean for each k-th cluster
          mus (loop [k 0
                     mus []]
                (if (= k K)
                   mus  ;; return mus
                  (let [mu-k
                     (sample (uniform-continuous
                     (+ lo (* (/ k K) (- up lo)))
                     (+ lo (* (/ (+ k 1) K) (- up lo))))))]
                   (recur (inc k) (conj mus mu-k)))))
           obs-std 0.1]
        ;; evaluate the log likelihood
        (map (fn [y]
            (observe (lik-dist mus obs-std) y)) data)
        ;; output
        (cons K mus))))
```

Figure C.1: Code example of GMM in Anglican

```
(defm gen-prog [curr-depth max-depth prev-type]
    (let [expr-type
          (if (< curr-depth max-depth)
              (if (= prev-type 3)
                  (sample (discrete [0.35  0.35  0.3]))
                  (sample (discrete [0.3  0.3  0.2 0.2])))
              (sample (discrete [0.5 0.5])))]
       (cond
         (= expr-type 0)
         (if (nil? prev-type)
             (let [_ (sample (normal 0 1))]  'x)
             'x)
         (= expr-type 1)
         (let [_ (sample (normal 0 1))]
            (list '* 'x 'x))
         (= expr-type 2)
         (let [a (sample (normal 0 1))
               curr-depth (+ curr-depth 1)
               expr-sin
               (list 'Math/sin (list '* a
                 (gen-prog curr-depth max-depth expr-type)))]
           expr-sin)
         (= expr-type 3)
         (let [a (sample (normal 0 1))
               b (sample (normal 0 1))
               curr-depth (+ curr-depth 1)
               expr-plus (list '+
               (list '* a
                 (gen-prog curr-depth max-depth expr-type))
               (list '* b
                 (gen-prog curr-depth max-depth expr-type)))]
           expr-plus))))

(defm gen-prog-fn [max-depth]
    (list 'fn ['x] (gen-prog 1 max-depth nil)))

(defquery pcfg-fn-new [ins outs]
    (let [obs-std 0.5
          max-depth 3
          f (gen-prog-fn max-depth)
          f-x (mapv (eval f) ins)]
        (map #(observe (normal %1 obs-std) %2) f-x outs)
        f))
```

Figure C.2: Code example of the Function Induction model in Anglican

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Hadi Mohasel Afshar and Justin Domke. Reflection, refraction, and hamiltonian monte carlo. In *Advances in Neural Information Processing Systems*, pages 3007–3015, 2015.

Shipra Agrawal and Navin Goyal. Analysis of Thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory*, pages 39–1, 2012.

Christophe Andrieu and Gareth O Roberts. The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics*, pages 697–725, 2009.

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

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2010.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine learning*, 47(2-3):235–256, 2002.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18 (153):1–153.

Atilim Gunes Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, et al. Efficient probabilistic inference in the quest for physics beyond the standard model. In *Advances in Neural Information Processing Systems*, pages 5460–5473, 2019.

Jean Bérard, Pierre Del Moral, and Arnaud Doucet. A lognormal central limit theorem for particle approximations of normalizing constants. *Electronic Journal of Probability*, 19(94):1–28, 2014.

Donald A Berry and Bert Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5:71–87, 1985.

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

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

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.

Luke Bornn, Pierre E Jacob, Pierre Del Moral, and Arnaud Doucet. An adaptive interacting wang–landau algorithm for automatic density exploration. *Journal of Computational and Graphical Statistics*, 22(3):749–773, 2013.

Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

Monica F Bugallo, Victor Elvira, Luca Martino, David Luengo, Joaquin Miguez, and Petar M Djuric. Adaptive importance sampling: the past, the present, and the future. *IEEE Signal Processing Magazine*, 34(4):60–79, 2017.

Olivier Cappé, Arnaud Guillin, Jean-Michel Marin, and Christian P Robert. Population Monte Carlo. *Journal of Computational and Graphical Statistics*, 13(4):907–929, 2004.

Olivier Cappé, Randal Douc, Arnaud Guillin, Jean-Michel Marin, and Christian P Robert. Adaptive importance sampling in general mixture classes. *Statistics and Computing*, 18(4):447–459, 2008.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

Alexandra Carpentier and Rémi Munos. Finite time analysis of stratified sampling for monte carlo. In *Advances in Neural Information Processing Systems*, pages 1278–1286, 2011.

Alexandra Carpentier, Remi Munos, and András Antos. Adaptive strategy for stratified monte carlo sampling. *Journal of Machine Learning Research*, 16:2231–2271, 2015.

Arun Chaganty, Aditya Nori, and Sriram Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*, pages 153–160, 2013.

Julien Cornebise, Éric Moulines, and Jimmy Olsson. Adaptive Methods for Sequential Importance Sampling with Application to State Space Models. *Statistics and Computing*, 18(4):461–480, 2008.

Jean Cornuet, JEAN-MICHEL MARIN, Antonietta Mira, and Christian P Robert. Adaptive Multiple Importance Sampling. *Scandinavian Journal of Statistics*, 39(4):798–812, 2012.

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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019.

Bruno De Finetti. La prévision: ses lois logiques, ses sources subjectives. In *Annales de l'institut Henri Poincaré*, volume 7, pages 1–68, 1937.

Vu Dinh, Arman Bilge, Cheng Zhang, IV Matsen, and A Frederick. Probabilistic path hamiltonian monte carlo. *arXiv preprint arXiv:1702.07814*, 2017.

Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

Arnaud Doucet, Mark Briers, and Stéphane Sénécal. Efficient block sampling strategies for sequential Monte Carlo methods. *Journal of Computational and Graphical Statistics*, 15(3):693–711, 2006.

Arnaud Doucet, Michael Pitt, George Deligiannidis, and Robert Kohn. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. *Biometrika*, page asu075, 2015.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics letters B*, 1987.

David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. *arXiv preprint arXiv:1302.4922*, 2013.

Víctor Elvira, Luca Martino, David Luengo, and Jukka Corander. A gradient adaptive population importance sampler. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4075–4079. IEEE, 2015.

Víctor Elvira, Luca Martino, David Luengo, and Mónica F Bugallo. Improving population monte carlo: Alternative weighting and resampling schemes. *Signal Processing*, 131:77–91, 2017.

Pierre Etoré and Benjamin Jourdain. Adaptive optimal allocation in stratified sampling methods. *Methodology and Computing in Applied Probability*, 12(3):335–360, 2010.

Pierre Etore, Gersende Fort, Benjamin Jourdain, and Eric Moulines. On adaptive stratification. *Annals of operations research*, 189(1):127–154, 2011.

Emily B Fox, Erik B Sudderth, Michael I Jordan, and Alan S Willsky. An hdp-hmm for systems with state persistence. In *Proceedings of the 25th international conference on Machine learning*, pages 312–319, 2008.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, 2018a.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 1682–1690, 2018b. URL `http://proceedings.mlr.press/v84/ge18b.html`.

Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*. Chapman and Hall/CRC, 2013.

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.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6): 721–741, 1984.

Charles J Geyer. Markov chain monte carlo maximum likelihood. 1991.

Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 2015.

Mark Girolami and Ben Calderhead. Riemann manifold langevin and hamiltonian monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2011.

Yuyun Gong and Qi Zhang. Hashtag recommendation using attention-based convolutional neural network. In *IJCAI*, pages 2782–2788, 2016.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2020-2-6.

Noah D Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.

Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. 2008a.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A Language for Generative Models. In *In UAI*, pages 220–229, 2008b.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 2014.

Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)*, 140(2):107–113, 1993.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

Peter J Green. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

Peter J Green. Trans-dimensional markov chain monte carlo. *Oxford Statistical Science Series*, pages 179–198, 2003.

Shixiang Gu, Zoubin Ghahramani, and Richard E Turner. Neural adaptive sequential Monte Carlo. In *NIPS*, pages 2629–2637, 2015.

W Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

David Janz, Brooks Paige, Tom Rainforth, Jan-Willem van de Meent, and Frank Wood. Probabilistic structure discovery in time series data. *arXiv preprint arXiv:1611.06863*, 2016.

Reiichiro Kawai. Asymptotically optimal allocation of stratified sampling with adaptive variance reduction by strata. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(2): 9, 2010.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR*, 2014.

Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *ECML*, volume 6, pages 282–293. Springer, 2006.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. Automatic variational inference in Stan. In *NIPS*, pages 568–576, 2015.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic differentiation variational inference. *The Journal of Machine Learning Research*, 18(1):430–474, 2017.

Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1945–1954. JMLR. org, 2017.

Tuan Anh Le. Inference for higher order probabilistic programs. *Masters thesis, University of Oxford*, 2015.

Tuan Anh Le. *Amortized inference and model learning for probabilistic programming*. PhD thesis, University of Oxford, 2019.

Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. In *Artificial Intelligence and Statistics*, pages 1338–1348. PMLR, 2017.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. Reparameterization gradient for non-differentiable models. In *NIPS*, 2018.

Stéphanie Lefèvre, Dizan Vasquez, and Christian Laugier. A survey on motion prediction and risk assessment for intelligent vehicles. *ROBOMECH journal*, 1(1):1–14, 2014.

Faming Liang, Chuanhai Liu, and Raymond Carroll. *Advanced Markov Chain Monte Carlo Methods: Learning from Past Samples*, volume 714. John Wiley & Sons, 2011.

Ming Lin, Rong Chen, Jun S Liu, et al. Lookahead Strategies for Sequential Monte Carlo. *Statistical Science*, 28(1):69–94, 2013.

Fredrik Lindsten, Adam M Johansen, Christian A Naesseth, Bonnie Kirkpatrick, Thomas B Schön, JAD Aston, and Alexandre Bouchard-Côté. Divide-and-conquer with sequential monte carlo. *Journal of Computational and Graphical Statistics*, 26(2):445–458, 2017.

Xiaoyu Lu, Tom Rainforth, Yuan Zhou, Jan-Willem van de Meent, and Yee Whye Teh. On exploration, exploitation and learning in adaptive importance sampling. *arXiv preprint arXiv:1810.13296*, 2018.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. Winbugs-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000.

Yicheng Luo, Antonio Filieri, and Yuan Zhou. Sympais: Symbolic parallel adaptive importance sampling for probabilistic program analysis. *arXiv preprint arXiv:2010.05050*, 2020.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Luca Martino, Victor Elvira, David Luengo, and Jukka Corander. An adaptive population importance sampler: Learning from uncertainty. *IEEE Transactions on Signal Processing*, 63(16):4422–4437, 2015.

Luca Martino, Victor Elvira, David Luengo, and Jukka Corander. Layered adaptive importance sampling. *Statistics and Computing*, 27(3):599–623, 2017.

Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

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.

Błażej Miasojedow, Eric Moulines, and Matti Vihola. An adaptive parallel tempering algorithm. *Journal of Computational and Graphical Statistics*, 22(3):649–664, 2013.

T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

Boris Mityagin. The zero set of a real analytic function. *arXiv preprint arXiv:1512.07276*, 2015.

Hadi Mohasel Afshar et al. Probabilistic inference in piecewise graphical models. 2016.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

Lawrence M Murray. Bayesian state-space modelling on high-performance hardware using libbi. *arXiv preprint arXiv:1306.3277*, 2013.

Lawrence M Murray and Thomas B Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 2018.

Radford M Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2, 2011.

James Neufeld. *Adaptive Monte Carlo Integration*. PhD thesis, University of Alberta, 2016.

James Neufeld, András György, Dale Schuurmans, and Csaba Szepesvári. Adaptive Monte Carlo via bandit allocation. *arXiv preprint arXiv:1405.3318*, 2014.

Akihiko Nishimura, David Dunson, and Jianfeng Lu. Discontinuous hamiltonian monte carlo for sampling discrete parameters. *arXiv preprint arXiv:1705.08510*, 2017.

Agostino Nobile and Alastair T Fearnside. Bayesian finite mixtures with an unknown number of components: The allocation sampler. *Statistics and Computing*, 17(2):147–162, 2007.

Aditya Nori, Chung-Kil Hur, Sriram Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

Art B. Owen. *Monte Carlo theory, methods and examples*. 2013.

Timothy Brooks Paige. *Automatic inference for higher-order probabilistic programs*. PhD thesis, University of Oxford, 2016.

Ari Pakman and Liam Paninski. Auxiliary-variable exact hamiltonian monte carlo samplers for binary distributions. In *Advances in neural information processing systems*, pages 2490–2498, 2013.

Ari Pakman and Liam Paninski. Exact hamiltonian monte carlo for truncated multivariate gaussians. *Journal of Computational and Graphical Statistics*, 23(2):518–542, 2014.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Yura N Perov and Frank D Wood. Learning probabilistic programs. *arXiv preprint arXiv:1407.2646*, 2014.

Michael K Pitt, Ralph dos Santos Silva, Paolo Giordani, and Robert Kohn. On some properties of Markov chain Monte Carlo simulation methods based on the particle filter. *Journal of Econometrics*, 171(2):134–151, 2012.

Martyn Plummer et al. Jags: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, 2003.

Tom Rainforth. *Automating Inference, Learning, and Design using Probabilistic Programming*. PhD thesis, University of Oxford, 2017.

Tom Rainforth, Tuan Anh Le, Jan-Willem van de Meent, Michael A Osborne, and Frank Wood. Bayesian Optimization for Probabilistic Programs. In *Advances in Neural Information Processing Systems*, pages 280–288, 2016a.

Tom Rainforth, Christian A Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem van de Meent, Arnaud Doucet, and Frank Wood. Interacting particle Markov chain Monte Carlo. *ICML*, 48, 2016b.

Tom Rainforth, Yuan Zhou, Xiaoyu Lu, Yee Whye Teh, Frank Wood, Hongseok Yang, and Jan-Willem van de Meent. Inference trees: Adaptive inference with exploration. *arXiv preprint arXiv:1806.09550*, 2018.

Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial Intelligence and Statistics*, pages 814–822, 2014.

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.

Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016a.

Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. C3: lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, pages 28–37, 2016b. URL `http://proceedings.mlr.press/v51/ritchie16.html`.

David Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic programming. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 634–643, 2019.

Daniel M Roy, Yee Whye Teh, et al. The mondrian process. In *NeurIPS*, pages 1377–1384, 2008.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic Programming in Python Using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

Ulrich Schaechtle, Feras Saad, Alexey Radul, and Vikash Mansinghka. Time series structure discovery via probabilistic program synthesis. *arXiv preprint arXiv:1611.07051*, 2016.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. Bugs 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Cambridge*, 1996.

Stan Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4, 2014.

Andreas Stuhlmüller and Noah D Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

Robert H Swendsen and Jian-Sheng Wang. Replica monte carlo simulation of spin-glasses. *Physical review letters*, 57(21):2607, 1986.

Yee Whye Teh. Dirichlet process., 2010.

Gerald Tesauro, VT Rajan, and Richard Segal. Bayesian inference in Monte-Carlo tree search. *arXiv preprint arXiv:1203.3519*, 2012.

David Tolpin and Frank Wood. Maximum a Posteriori Estimation by Search in Probabilistic Programs. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank D. Wood. Output-sensitive adaptive metropolis-hastings for probabilistic programs. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II*, pages 311–326, 2015. doi: 10.1007/978-3-319-23525-7_19. URL `https://doi.org/10.1007/978-3-319-23525-7_19`.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, page 6. ACM, 2016.

Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Uber. Pyro, a Deep Probabilistic Programming Language, 2017.

UberLabs. Pyro, a universal probabilistic programming language. `https://github.com/uber/pyro`, 2017.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

Guy Van den Broeck and Kurt Driessens. Automatic discretization of actions and states in monte-carlo tree search. In *Proceedings of the ECML/PKDD 2011 Workshop on Machine Learning and Data Mining in and around Games*, pages 1–12, 2011.

Fugao Wang and DP Landau. Determining the density of states for classical statistical models: A random walk algorithm to produce a flat histogram. *Physical Review E*, 64(5):056101, 2001.

David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, 2011.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *AISTATS*, pages 2–46, 2014.

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient MCMC kernels from probabilistic programs. In *AISTATS*, pages 1068–1076, 2014.

Kexin Yi and Finale Doshi-Velez. Roll-back hamiltonian monte carlo. *arXiv preprint arXiv:1709.02855*, 2017.

Yichuan Zhang, Zoubin Ghahramani, Amos J Storkey, and Charles A Sutton. Continuous relaxations for discrete hamiltonian monte carlo. In *Advances in Neural Information Processing Systems*, pages 3194–3202, 2012.

Yuan Zhou, Bradley 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 *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 148–157, 2019a.

Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. *arXiv preprint arXiv:1910.13324*, 2019b. ICML 2020 (to appear).