Extending Probabilistic Programming Systems and Applying Them to Real-World Simulators

Bradley Gram-Hansen

Harris Manchester College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2021

Abstract

Probabilistic programming is a paradigm that enables us to efficiently write probabilistic models as program code that we can sample, infer underlying parameters and predict outcomes based on complete or incomplete observations. Naturally, stochastic simulators, a special sub-class of simulators containing random variables, internal inference procedures, and the simulation of observations, are structurally rich probabilistic models. However, most simulators are not written in the probabilistic programming paradigm, as they are written in arbitrary programming code. This means that it is challenging to automatically update the variables in these simulators to account for observations from conducted experiments, which limits the simulators' use.

Furthermore, there are two components to a probabilistic programming system i) the language and compilation procedure, ii) the inference procedures. These components can limit our ability to compile particular classes of probabilistic models, such as models that contain mixtures of parameter types, due to restrictions in the expressiveness of the language. Restrictions in the expressivity of the language can also inhibit our ability to generate efficient inferences, as this naturally influences the design of the probabilistic programming system and the set of available inference backends. Creating probabilistic programming systems that are expressive enough for different probabilistic models leads to the creation of many different probabilistic programming systems, which is inefficient - it would be more efficient if we could repurpose existing probabilistic programming systems.

In this thesis, we develop three pieces of original work through four papers. The first piece of work describes how to extend differentiable first-order probabilistic programming systems to perform statistically correct and computationally efficient inference on models with mixtures of continuous and non-continuous parameters, without having to modify the underlying language, or develop an entirely new probabilistic programming system. The second describes how to translate real-world stochastic simulators written in arbitrary program languages to probabilistic programming systems. And finally, in the third piece of work, we develop two new Bayesian inference schemes to make inference more computationally and statistically efficient in nested models, models that contain probabilistic programs, within probabilistic programs, which arise in many real-world stochastic simulators.

Acknowledgements

I would like to thank first and foremost my supervisors Tom Rainforth, Atılım Güneş Baydin, Philip Hilaire Sean Torr, Yee Whye Teh and Frank Wood for their support and the knowledge that they helped me acquire throughout my P.h.D. Special thanks goes to Tom Rainforth for his endearing mentorship, support and friendship that without, would have made it challenging to complete this P.h.D, and to Atılım Güneş Baydin and Tobias Kohn whose friendship and mentorship have been critical. To Phil, for being full of energy and emotionally supportive, and to Yee Whye for his mentorship and kindness. To Wendy Poole, whose support and conversations have helped carry me through this P.h.D, helping me with admin, to life quandaries, and for planning so many wonderful AIMS events, I will be forever thankful for her support and kind-being. To my friends and lab colleagues, who have been incredibly supportive and some of the best friends that I could have ever asked for, in no specific order; Adam Golínski, Triantafyllos Afouras, Andrea Patané, Fabian Fuchs, Shuyu Lin, Xu Ji, Edward Wagstaff, Sasha Salter, Oliver Bent, Yuan Zhou, Rob Zinkov, Christian Schroeder de Witt, David Martínez-Rubio, Mario Lezcano-Casado, Mike Teng, Andrew Warrington, Tuan Anh Le, Prateek Gupta and members of the OXCSML and Torr Vision groups. To my examiners, Luke Ong and John Winn for their feedback and time. Finally, I am thankful to all the teachers that helped to shape me into the way I am today, for my Auntie Val who provided a roof for me when I was homeless and for her love and support growing up, my sister Charlie for all her inspiration, to my dad - even though we did not see eye-to-eye I thank him for trying his best, to my partner Tse for all her love, inspiration and care, EPSRC for funding this work and for Steve Roberts, because if he had not responded to an e-mail in 2015 I would have never applied to the 2016 AIMS program.

Publications

Portions of this thesis are based on previously published work by the author, performed in collaboration with others. On a chapter-by-chapter basis:

- Chapter 4 Yuan Zhou*, **Bradley J Gram-Hansen***, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. * Equal contribution, LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models, *In Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS), 2019*
- Chapter 5 comprises of two closely linked papers:
  - Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, **Bradley Gram-Hansen**, Lei Shao, Saeid Naderiparizi,Andreas Munk, Jialin Liu, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, Frank Wood, Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model, *In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS), 2019.* In this paper we develop the probabilistic programming execution protocols, which were necessary to connect arbitrary real-world simulators with probabilistic programming systems.
  - Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, **Bradley Gram-Hansen**, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Frank Wood, Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale, *In Proceedings of the International Conference for High Performance Computing (SC), 2019.* In this paper we show that these protocols scale in a high-performance computing environment.
- Chapter 6 is an extended manuscript of **Bradley Gram-Hansen**, Christian Schroeder de Witt, Robert Zinkov, Saeid Naderiparizi, Adam Scibior, Andreas Munk, Frank Wood, Mehrdad Ghadiri, Philip Torr, Yee Whye Teh, Atilim Gunes Baydin, Tom Rainforth, Efficient Bayesian Inference for Nested Simulators, *In Proceedings of the Advances in Approximate Bayesian Inference, (AABI), 2019* [Gram-Hansen et al., 2019b], that is currently under review at the Conference on Uncertainty in Artificial Intelligence (UAI) 2021.

The following published works are excluded from this thesis, as they do not align with the core thesis objective:

- **Bradley J Gram-Hansen***, Patrick Helber*, Indhu Varatharajan, Faiza Azam, Alejandro Coca-Castro, Veronika Kopackova, Piotr Bilinski, *Equal contribution, Mapping informal settlements using machine learning and low resolution multi-spectral data, *In Proceedings of the 2019 Association for the Advancement of Artificial (AAAI) and ACM Journal of AI, Ethics, and Society, 2019*

- Alan Blackwell, Tobias Kohn, Martin Erwig, Atilim Gunes Baydin, Luke Church, James Geddes, Andy Gordon, Maria Gorinova, **Bradley Gram-Hansen**, Neil Lawrence, Vikash Mansinghka, Brooks Paige, Tomas Petricek, Diana Robinson, Advait Sarkar, Oliver Strickson, Usability of Probabilistic Programming Languages, *Psychology of Programming Interest Group 30th Annual Workshop, PPIG, 2019*

- Christian Schroeder de Witt, **Bradley Gram-Hansen**, Nantas Nardelli, Andrew Gambardella, Rob Zinkov, Puneet Dokania, N Siddharth, Ana Belen Espinosa-Gonzalez, Ara Darzi, Philip Torr, Atılım Güneş Baydin, Simulation-Based Inference for Global Health Decisions, *International Conference of Machine Learning workshop on ML for Global Health, 2020*

- **Bradley Gram-Hansen***, Christian Schröder de Witt*, Tom Rainforth, Philip HS Torr, Yee Whye Teh, Atılım Güneş Baydin, Hijacking Malaria Simulators with Probabilistic Programming, *International Conference of Machine Learning workshop for AI for Social Good, 2019*

This thesis is presented as an integrated thesis, in which my publications are included in their camera-ready form, that is, as they appear in the proceedings from their publication venues, with the exception of Chapter 6, which is an extension of a published manuscript that is currently under review at Conference on Uncertainty in Artificial Intelligence (UAI) 2021. Following each publication chapter is a signed statement of authorship detailing my contributions.

Contents

1 Introduction

As we are born into the world, as we open our eyes and take in the surrounding sounds and sights, we begin to form a simulation of the world around us, encoding a prior belief of what we expect to observe. For example, we model the expected output from a set of electromagnetic waves that interact with different surfaces, and as sounds propagate through our eardrums, our brain creates a model for language and a model for how we should construct sounds. As we progress through life and interact with the world around us, models appear everywhere, from the dynamics of particle collisions to the beating of a heart. Some models are deterministic, but many are stochastic and have known-unknown parameters that we need to infer, given our observations. But, how do we model? How do we infer things about models? How can we use our models to predict things? Bayesian reasoning, which provides a powerful mechanism for modelling, such as learning from data, combined with a concept called probabilistic programming, which provides a mechanism for automating Bayesian inference in such models, provides an answer. At the intersection of computer science, engineering, and statistics, probabilistic programming enables users to solve complex models in simple, autonomous ways, by combining data-driven techniques with statistical modelling via sophisticated compilers and programming languages. However, constructing models is still problematic, even if we have an abundance of data, and reasoning about those models, given our world view, is even more complicated; it's an NP-hard problem [Roth, 1996].

1.1 Why model?

Models are humanity's way of describing *things* and to construct models we take a series of statements to form a description of that thing. The language we typically use to describe those statements is mathematics, whether the thing is abstract, such as using algebraic geometry to model the concept of space, or something more concrete such as describing the dynamics of particle interactions or forecasting the weather. Mathematics is a flexible tool for constructing models, and if we want to automate the evaluation of mathematical statements, computers provide a flexible framework to do so. By writing our models as program code we can automate the simultion of the model under different parameterizations.

1.2 Why use simulators?

Simulators are models, and a special type of simulator that we shall explore throughout this thesis is called the stochastic simulator, whose parameters are a mixture of deterministic and stochastic variables. Stochastic simulators are forward-run probabilistic generative models and can be designed to model everything from biological effects to physical events. In particular, we will explore strategies and inference procedures that are designed to be applied to stochastic simulators that model a specific event, such as the decaying of a particle or multiple particles [Gleisberg et al., 2008], or contain nested structures [Jäckel, 2002; Smith et al., 2008; Stuhlmüller and Goodman, 2014].

Simulators are important due to the comprehensive knowledge that they contain, generated through many millennia, centuries, decades, and years of extensive study and observation, which means simulators provide a level of interpretability. This is particularly important when we need to understand a given biological or physical process that we cannot treat as an unknown-oracle that only provides answers, as is the case in much of deep-learning [Goodfellow et al., 2016].

In addition to this, they enable us to model complex phenomena that are not easily observed or expensive to generate in a lab environment. This enables researchers to simulate their models without the need for large, expensive particle-colliders (although we are happy that they exist), and generate events for processes too rare to observe when running a single physical experiment. Because simulators can be deployed on a computer, we can leverage thousands of computers in parallel to simultaneously run multiple experiments, increasing our odds of observing the initial conditions that led to the rare event.

Finally, as they are written as program code, we can run simulators anywhere where we have access to some form of computation, from low-powered embedded computers and smartphones, to energy-hungry super-computers, as we shall demonstrate in Chapter 5. As stated previously, the simulators explored in this thesis will all be stochastic simulators, which are by design probabilistic models.

1.3 Why write programs?

We write programs to automate procedures, such as switching off a light-emitting diode, or executing trades as part of a complex financial trading system. Similar to modelling, writing programs is centered on defining a model, whose dynamics and output are repeatable and understandable. The program's objective is to significantly reduce the manual time cost for the given task and perform operations that may not be feasible for a given human. Furthermore, a program is compact, re-usable, and can be deployed in different environments.

However, one problem with writing programs is choosing the framework and language specification. Is speed an issue? If so, then the user may write their program in C++ [ISO, 1998], but, if the user cares about readability and maintainability, then they may write their program in Python [Van Rossum and Drake Jr, 1995]. These decisions have lasting consequences on how the program can evolve, what frameworks the program can be run in, and the transferability of the program[1]. Even though programs are "re-usable" in the sense that a user can run a program repeatedly, it may not be possible to embed that program in an existing framework, or adaptively modify that program when new information appears. Simulators are often coded in legacy frameworks, making it difficult to embed them in modern systems and adaptively modify them for new observations. This can be problematic as simulators can be tens, to millions of lines of code, but as they grow in complexity, typically maintained by a small number of people, the ability to proactively change them becomes more convoluted and challenging. This hinders the user's ability to include new physics into existing simulators, and/or, to infer properties about the simulators, in terms of the initial states given some observations. Due to the complexity of some simulators, it is difficult to understand the program's structure, rendering aspects of the simulator un-interpretable. We will present steps towards a solution to these challenges in Chapter 5.

1.4 What is Bayesian inference?

Bayesian modelling starts with a simple, yet powerful formula called Bayes' theorem [Bayes, 1763]:

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})} = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x}}{\int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x}} \tag{1.1}$$

where $\mathbf{x}$ represents our latent variables, our known-unknowns, and $\mathbf{y}$ represents our observations, our known-knowns. When using Bayes' theorem, there are two ways we can think about solving

[1]Containerised environments such as Docker and Singularity have made this less of an issue.

our problem; one is *forward probabilities*, the other is *inverse probabilities* [MacKay, 1998, Chapter 2].

Forward probability problems involve a *generative model* that describes a process that is assumed to give rise to some observations; the task is to compute the probability distribution or expectation of some quantity that depends on the data, that is $p(\mathbf{y}|\mathbf{x})$. Like forward probability problems, inverse probability problems involve a generative model of a process. Instead of computing the probability distribution of some quantity produced by the process, we compute the conditional probability of one or more of the latent variables in the process, given the observed variables, that is $p(\mathbf{x}|\mathbf{y})$.

Figure 1.1 provides the directed acyclic graphical models for each, in the simplest case, where we have two random variables $\mathbf{x}$ and $\mathbf{y}$ that have a directional relationship between one another. The stochastic simulators that we will explore in this thesis are generative models, probabilistic model that contain random variables to simulate a process and define a joint density $p(\mathbf{x}, \mathbf{y})$. When the simulator is run forward we generate observations which should be consistent with our measured observations of a potentially complex procedure, but this is often not the case, as models by construction are imperfect. To optimally determine the parameters that would lead to the ground-truth observations, instead of relying on heuristic values, we need to "invert" the simulator to determine the form of the latent variables that would generate such observations. One way of doing this is via statistical inference procedures, but doing this automatically for different types of stochastic simulators is challenging, and led to the developments presented in Chapters 4, 5 and 6.

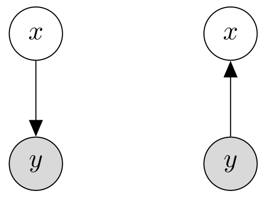


Figure 1.1: *Left*: The forward probability.
Right: The inverse probability. The grey-circles represent observations and the non-grey circles represent known-unknown quantities.

As a conceptual example of Bayes theorem, imagine we have a worn coin and want to write a simulator to model this worn coin - so that we don't have keep performing the flipping manually. To do this we would need to build an accurate simulator, which means that we have to invert the flipping of the coin to determine how biased it is. Provided we have some experimental observations, Bayesian statistics provides a solution. We begin to generate our experimental observations and flip the coin to observe each outcome, for the first we observe

Heads (H), $\mathbf{y}_1 = H$ and for the next we observe Tails (T), $\mathbf{y}_2 = T$. With these observations we begin to hypothesize that our coin is not biased. In fact, we make several more observations, $\mathbf{y}_{1:10} = HTHHHTTTTH$, it still looks unbiased, but after a hundred observations we see we observe only thirteen-tails and eighten-seven heads, and so our beliefs begin to change.

Bayes' theorem provides a useful framework to capture that change in belief as we update our view of the world, based on the data we gather, which could be through simulation, or experiment. However, we need a starting point for our beliefs, the prior. In the case where all outcomes are equally probable, that is, we have no additional information that advantages one outcome over another, we state that we have a Uniform, $U(a, b)$ prior over the range of possible values, with a, representing the lowest outcome and, b, the highest. We call this a non-informative prior. In many cases, we have prior information about our model and the problem it specifies, such as the coin is worn. We will see the effects of informative priors in Section 2.4. Next, we need to construct a likelihood, that will tell us how likely a given set of variables are for some data, $p(\mathbf{x}|\mathbf{y})$, or, how likely the data is, given some choice of variables $p(\mathbf{y}|\mathbf{x})$, dependent upon whether we are calculating forward, or inverse probabilities. In either case, how we construct both the likelihood and the prior is subjective. Hence the phrase, all models are *wrong*, but some are just less wrong [Box, 1976].

1.5 Can we draw inferences from simulators using programming constructs?

Simulators are usually run forward and generate data relating to that forward-generating process. To draw inferences from simulators we need to extract information from the simulator source code, so that the model structure and stochastic variables in the simulator can be found.

One such paradigm that enables us to draw inferences and extract this information directly from source code is probabilistic programming. Probabilistic programming lies at the intersection of machine learning, programming languages and statistics, and provides an automated way to model and extract inferences from probabilistic models utilising Bayesian statistics. It provides us with a mechanisms of drawing inferences from simulators rather than just running them forwards. It allows us to ask questions like "what is the implied distribution of x given that $y = c$?", that is *conditioning* to only produce outputs consistent with certain restrictions. This means that the system can automatically calibrate the underlying simulator parameters via conditioning constructs, enabling us to tune the simulator output so that it matches our observations in the physical world. Furthermore, probabilistic programming systems dramatically reduce both the amount of code that has to be written and the expertise required to perform complex statistical inference procedures in simulators [Gordon et al., 2014; Minka et al., 2014; Pfeffer, 2001; Rainforth, 2018a; Wood et al., 2014], as the probabilistic programming backend deals with this.

Naturally, as probabilistic programs are probabilistic models, they have a density $p(\mathbf{x}, \mathbf{y})$, in terms of the latent variables, the samples, $\mathbf{x}$ and observations, the data, $\mathbf{y}$, which in the context of probabilistic programs we refer to as the program density. Thus, simulators are a natural fit for probabilistic programming systems, as stochastic simulators are probabilistic programs. In order for the system to extract a program density, which is required for inference, probabilistic programming system use a combination of sophisticated compilers and coroutines to transform the program to different representations; the probabilistic and computational graphical model. Once constructed, the program density can be extracted, which can then be used with the inference back-end.

Probabilistic programming systems come in many different designs and flavours, but fundamentally have two parts: a language semantics and a set of generalised inference procedures that can run automatically on models specified in the language of the probabilistic programming system [Ackerman et al., 2019; Bingham et al., 2019; Goodman and Stuhlmüller, 2014; Goodman et al., 2008; Le et al., 2017; Li and Russell, 2013; Pfeffer, 2001; Rainforth, 2018a].

The probabilistic programming systems that we will explore in this thesis leverage Bayesian statistics and use modified inference algorithms that can be applied in generalised ways, enabling inference in different classes of probabilistic models in an autonomous fashion. This relationship between the inference procedures and probabilistic programming systems often means that the language of a system is entirely built around the specification of the inference procedures implemented in the system [Carpenter et al., 2017; Lunn et al., 2000; Spiegelhalter et al., 1996], which means when a user writes a simulator in the syntax of a probabilistic programming system, the system can place guarantees on the generated inferences. In other systems, the model is the centerpiece [Bingham et al., 2019; Cusumano-Towner et al., 2019; Goodman et al., 2008; Le et al., 2017; Wood et al., 2014], and the user is provided arbitrary freedom in the models that they can write, but do not get strong guarantees on the inference results.

1.6 Why not write simulators as programs?

Given probabilistic programming and Bayesian inference, one may ask, why in machine learning do we not solve all simulators in the probabilistic programming paradigm? There are two main reasons for this. The first is that inference - computing the posterior distribution in Equation 1.1 — can only be performed exactly and efficiently in a very small set of simulators: simulators where the posterior $p(\mathbf{x}, \mathbf{y})$ form certain types of graphical models [Murphy, 2012, Chapter 20] and simulators where a conjugacy relationship occurs between the prior and likelihood function, which leads to a posterior that has the same functional form as the prior. Outside of this small set of models, exact inference in complex models often requires evaluating intractable integrals or NP-hard computation requiring enumeration of exponentially large combinatorial spaces. This requires one to resort to approximate Bayesian inference schemes, some of which lead to the true answer in the limit as the number of samples tends to infinity; others only lead to a bounded approximation. Which methods perform well is often model-dependent, leading to many difficulties when trying to design probabilistic programming systems. It can cause some languages to be restrictive, but models compiled in those languages will have certain statistical guarantees, compared to languages that allow for more model expressivity that make trade-offs in inference efficiency. We develop solutions to this in Chapters 4 and 6, to extend the model space of existing probabilistic programming systems so that they can perform efficient, automated Bayesian inference without having to be rebuilt or made obsolete.

Second, the specification of simulators in arbitrary languages that are not written in probabilistic programming systems is a significant practical hurdle. To circumvent this issue, we develop a compilation procedure and a set of coroutines that connects simulators written in non-probabilistic programming languages to probabilistic programming systems, Chapter 5. This enables the simulator to be evaluated and run as if it had been written as a probabilistic program in a probabilistic programming system, allowing us to perform inference in the given simulator and removing the need to re-write the simulator in the probabilistic programming system, and the need to define input parameters heuristically.

1.7 Overview of thesis

Throughout this thesis we shall provide innovative solutions to the challenges outlined above by developing tools to aid engineers and scientists via new compilation and inference schemes.

In Chapter 2 we explore the different types of probabilistic programming systems and the role compilation plays in converting probabilistic programs into graphical models that enable automatable inference. Next, in Chapter 3 we explore several Monte Carlo based-inference schemes that are utilised throughout this thesis, we also define the program density of a probabilistic program and introduce purpose built inference engines for probabilistic programming systems that leverage the forward execution of programs. Then we shall move on to the core contributions Chapters 4, 5 and 6. In Chapter 4, we develop an innovative compiler that enables constrained-by-inference probabilistic programming systems to be more liberal in the class of simulators they can operate over, without having to be redesigned, or sacrifice inference efficiency. In Chapter 5, we develop a new compilation process and a series of coroutines for transforming real-world simulators that were not directly amenable to inference into computational graphical models that are amenable to inference in probabilistic programming systems, without having to rewrite the existing, often complex, simulator inside of the probability programming system. In Chapter 6, we introduce two new Bayesian inference procedures to perform efficient inference on a special class of stochastic simulators that have nested internal inference procedures. And finally, we finish with a set of concluding remarks, Chapter 7.

2 Probabilistic Programming and Simulation

Simulators arise in all aspects of life, from complicated simulators that model epidemics and fundamental physics, to seemingly simple simulators that model the flip of a coin. However, designing and building simulators poses many challenges. First, the simulator needs defining, which requires us to understand how a set of latent variables $\mathbf{x}$, map to a set of observations $\mathbf{y}$. Some of these relationships may be known, though we must make an educated guess about this mapping in many instances. Second, we have to determine how we will design our simulator; what language will our simulator be written in? Will there be stochastic choices, or will everything be deterministic? Design decisions like this will have direct implications for the types of function mappings allowed within the simulator, directly affecting the set of inference strategies that can be deployed. Finally, how do we infer the latent variables of the simulator, given new observations?

Our framework for constructing and inferring properties of simulators will utilise probabilistic programming frameworks [Bingham et al., 2019; Carpenter et al., 2017; Minka et al., 2014; Pfeffer, 2005; Rainforth, 2018a; Tran et al., 2017; Van de Meent et al., 2018] that use Bayesian inference [Bayes, 1763; Gelman et al., 2013]. While these are not the only frameworks that exist [Milch et al., 2005], they are flexible and convenient tools for solving statistical problems. In particular, probabilistic programming provides a way of quickly prototyping a model in program code and extracting answers at the click of a button using Bayesian methodology.

2.1 Probabilistic Programming

When a program such as a simulator is executed in the standard programming paradigm, it runs forward using its input parameters and maps those inputs to some outputs. In contrast, statistical inference runs backwards, it takes the output and infers what input led to that output. In an automated way, probabilistic programming links the forward execution to the inference procedure that is run backwards. This enables the users to accelerate iteration over probabilistic models, as code is easier to read and write than math, and removes the requirement to write the complicated inference back-ends as the inference methods are intertwined with the probabilistic programming system, reducing the technical barrier to probabilistic modelling. Probabilistic programming is able to automate this cycle by turning inference procedures into compiler optimisations. Thus, the semantics of the language determines the expressivity of the system i.e. what simulators can be written, and the inference engines that can be deployed in the system. As such, the work of this thesis focuses on the development of new compilers, coroutines and inference procedures that enable further automation, while providing greater efficiency in terms of the statistical inference performed and the ability to re-use existing simulators, and probabilistic programming systems. Throughout this thesis, when we refer to a program in the context of a probabilistic programming system, we implicitly imply a stochastic simulator that has some joint density, unless otherwise stated.

As stated previously, a key component of a probabilistic programming system is the language. Probabilistic programming languages (PPLs) extend general-purpose programming languages with constructs to perform sampling and conditioning of random variables [Gordon et al., 2014; Van de Meent et al., 2018]. The purpose of the PPLs is to decouple model specification from inference: a probabilistic program is implemented by the user as a regular program in the host programming language, specifying a simulator that produces samples from a generative process at each execution. In other words, the program produces samples from a joint distribution $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$, which the program implicitly defines. The program is then executed using a general-purpose inference engine available within the probabilistic programming system. The compiler and coroutines of a PPL check that the model adheres to the semantics and expressivity of the PPL, while also being a valid program in the host language. Then, inference is performed to obtain $p(\mathbf{x}|\mathbf{y})$, the posterior distribution of latent variables $\mathbf{x}$ conditioned on the observed variables $\mathbf{y}$.

In certain types of PPLs we must restrict certain functions to inhibit recursion, leading to two distinct groups of PPLs. In one group, we have the first-order probabilistic programming languages (FOPPLs) [Staton et al., 2016; Van de Meent et al., 2018], such as BUGS [Spiegelhalter et al., 1996] and Stan [Carpenter et al., 2017] that have purpose-built compilers that ensure the models written in the system only compile if they are valid directed acyclic graphical models [Koller et al., 2009] and as such provide computational and statistical guarantees on the generated inferences. In the second group, we have the universal probabilistic programming languages (UPPLs) [Staton, 2017] such as Church [Goodman et al., 2008], Anglican [Wood et al., 2014] and PyProb [Baydin et al., 2019b; Le et al., 2017] which introduce no explicit constraints on the models that can be expressed, and are universal in the sense that they can be used to specify any computable distribution.

2.1.1 First-order probabilistic programming languages

First-order probabilistic programming languages (FOPPLs) are probabilistic programming languages that are restricted to only allowing first-order functions [Van de Meent et al., 2018], which means that functions cannot accept other functions as arguments, nor allow non-terminable recursion. To impose such restrictions, the PPL is usually a reduced subset of the full-language with added constructs, such as **`sample`** and **`observe`**, which represent sampling from a distribtuion and conditioning on observations. By restricting functions to be first-order certain types of programs that depend on potentially infinite recursion, and, or higher-order functions, which is to say that user-defined functions cannot accept other functions as arguments, cannot be compiled in a FOPPL. Consider the problem of drawing a sample from a geometric distribution, representing the number of failures before the first success in a sequence of Bernoulli trials. Most probabilistic programming languages that support discrete latent variables would already include a geometric random primitive, or if not, one could be created via tools for implementing new random primitives. However, what if we want to implement this not as a random primitive, but as a program in our probabilistic language? The direct way of sampling this is to repeatedly sample from a Bernoulli distribution until success, Program 2.6, and return the total number of trials required. This could be expressed as:

```
def sample_geometric(p):
  if coin_flip(p):
    return 0
  else:
    return sample_geometric(p) + 1
```

Where Program 2.6, the coin-flip function, is a constructor for a Bernoulli distribution over the values `true` and `false`. This function recurses to an arbitrary depth: although it halts with probability 1 for any $p \in (0, 1]$, one could not construct a finite graph outlining the computation for all possibilities. Even familiar simple distributions are sufficiently complex that we are unable to implement them in first-order probabilistic programming languages using only smaller building-blocks; in contrast, universal languages in theory only require a single primitive (for example, uniform on the [0, 1] interval) to be able to draw samples from any such distribution.

Furthermore, the position of the observation variables need to be provided at compile time, because when the compiler constructs the computational graph it must be able to determine the structure of the program in order to leverage efficient inference schemes, as if this information was not known then the directional relationships between the x's and y's could not be established and as such a directed acyclic graph could not be constructed. Although the values of the observations can be added dynamically during runtime. Each of these limitations restricts the model expressivity of FOPPL-based probabilistic programming systems. These restrictions do have advantages, as they enable inference schemes to be implemented in an efficient manner via compiler optimisations as programs that compile in a FOPPL have joint densities $p(\mathbf{x}, \mathbf{y})$ that correspond directly to directed acyclic graph models, for which there exists efficient statistical inference algorithms that converge quickly to the correct posterior distribution [Geyer, 1992; Koller et al., 2009].

2.1.2 Universal probabilistic programming languages

In contrast to FOPPLs, universal probabilistic programming languages (UPPLs) allow for the expression of unrestricted probabilistic models [Borgström et al., 2016; Goodman et al., 2008; Le et al., 2017; Wood et al., 2014]. They can express models with an unbounded number of random variables, which means that random variables are not fixed statically and can be created dynamically during execution. This flexibility means that UPPLs can represent any computable distribution [Borgström et al., 2016], as there are no restrictions on recursion, stochastic control flow and the type of allowable functions. As functions are first-class values in UPPLs, they can define higher-order functions, functions that accept other functions as arguments. Thus any computable program can be run in a UPPL. This means any simulator of arbitrary complexity that can be defined in a Turing-complete programming language can be evaluated in a UPPL, making them powerful tools for modelling. We will see an example of this in Chapter 5. Because they allow for infinite recursion, these systems facilitate simulators that contain different layers of nesting, which arises in a large number of real-world simulators, where agents reason about other agents decisions; based on the perceived decision an agent will take, given an initial action [Rainforth et al., 2018; Smith et al., 2008; Stuhlmüller and Goodman, 2014]. However, performing inference in nested models is challenging, as even though the distribution implied by any nested simulator always has a direct form that could be written down as an unnested simulator, it is doubly intractable [Murray et al., 2006] and the program that expresses it might

be infinitely complicated, that is it is non-terminable in a finite amount of time.
Thus, even though nested simulators can be written in UPPLs, developing efficient inference schemes for such simulators is non-trivial [Rainforth, 2018b; Rainforth et al., 2018] and in Chapter 6 we develop two efficient Bayesian inference schemes for high-dimensional nested simulators.

In all languages presented in this thesis, the constructs `sample` and `observe` will possess a special semantic meaning that will play a vital role in the program execution, as they will act as labels for the random variables in the given simulator. We call these labels "addresses" and will discuss them in more detail in Section 2.3.1.

2.2 Compiling a probabilistic program

For the purpose of demonstration we will introduce a simple FOPPL utilising both the Clojure [Hickey, 2008] and Python [Van Rossum and Drake Jr, 1995] programming languages and includes standard language features such as conditional statements (e.g. `if`), assignment (e.g. `(let [a 2])` in Clojure is equivalent to $a = 2$ in Python etc.), primitive operations (e.g. `+`, `-`, `*`, `/` where $a + b$ is the Pythonic expression and the equivalent Clojure expression is `(+ a b)` etc.), and user-defined functions, in Python `def` and in Clojure `fn` which are restricted to be first order. Here, `(fn [args] body)` takes a set of arguments and a set of procedures in the body to evaluate during the forward execution, see Programs 2.1 and 2.8 for examples of simple FOPPL programs.
Non-probabilistic languages can be either compiled, whereby high-level source code is converted to a lower level language (e.g. byte-code) before being evaluated, or interpreted, whereby an interpreter reads the program and directly evaluates it based on the language semantics. The same is true for PPLs, for which compilation usually comprises of converting the program, $\mathcal{P}$, to a model artifact in a host language in which the inference algorithms are written.

viding a common representation of simulators. We start by taking a program $\mathcal{P}$ as input, this could be either a Clojure, or Python program for our system, Chapter 4, and through a Linearised Intermediate Representation (LIR) we internally construct the probabilistic graph of the program, $G(V, E)$, consisting of vertices, V and arcs, E, from which we can extract the computational graph and program density for the back-end inference engines. Where the LIR is a reduced set of statements which suffice to express any graphcal model. Each vertex of the LIR denotes a `sample` or `observe` statement, of which only a finite and fixed number can occur in a FOPPL, in more expressive languages there are no restrictions. The arcs of the LIR define both the probabilistic and conditional dependencies of the program variables. We can express a graphical model as a linear program, comprising three types of statements, `sample` and `observe` , and if we have conditioning, then the third type of statements are conditional statements, i.e. `if` statements. Figure 2.1 shows an example hierarchical structure of a probabilistic program compiler.

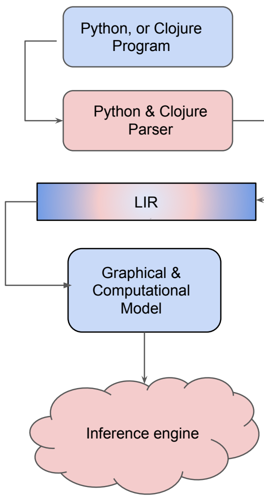


Figure 2.1: The compiler uses several passes to transform a probabilistic FOPPL Clojure / Python program to a graphical and computational model. The computational model is then used as an interface to connect with an inference engine.

A sampling statement has the form `(let [xi sample(d, e1, ... , en)])`, where a sample from the distribution `d` is taken and stored under the name $\mathbf{x}_i$. The sample construct represents a latent variable. It accepts a distribution object d, which must evaluate to a distribution object and a set of expressions e representing the distribution inputs, i.e. $\mathbf{x} \sim \text{Uniform}(0, 1)$ is represented as `sample(uniform 0 1)`, and returns a value that is a sample from this distribution object. Distributions are constructed using primitives provided by the FOPPL. The distribution can depend deterministically on any previously sampled variable $\mathbf{x}_k$ $(k < i)$ via the expression e_i. The second type of statements are conditioning statements, which have the form `observe( (d e1, ... , en)c)`. In contrast to the `sample` construct, `observe` factors the density according to the distribution `d`, with all parameters `e1,...,` `en` and the observed data `c` and represents an observed random variable. It accepts an argument d, which must evaluate to a distribution, and conditions on the next argument c, which is the value of the random variable, our observation y. For example, `(observe (normal x 1.0)2)` is equivalent to $p(y|0, 1^2) = \text{Normal}(y = 2|\mu = x, \sigma^2 = 1^2)$. The third type of statement are conditions of the form `(if (< e1 0)e2 e3)`, where `(< e1 0)` is the predicate and `e2 e3` are the consequent and alternative. For example, `(let [x sample(uniform 0 1)] (if (< x-0.5 0)(observe (normal x 1.0)2)(observe (Beta x 5)2))))`, which in the conditional is equivalent to $Normal(x, 1|2)^{\mathbb{I}[x<0.5]} Beta(x, 5|2)^{\mathbb{I}[x-0.5>0]}$, where $\mathbb{I}[\cdot]$ represents the indicator function.

2.2.1 The compilation output

Combining these statements together we can now compile any program that can be written in a FOPPL. As an example, we provide an output from the compilation scheme after compiling a program that contains all three types of statements, Program 2.1, representing a probabilistic model for determining the mean of a population, given some observations $y1, y2$, under the assumption that the population is normally distributed. The `mean`, represents the latent variable in our model.

Program 2.1: A Cloure-based FOPPL Gassuian-Unknown mean program containing samples, observes and conditionals. The file name of this program is *gaussian_unknown_mean_branching_model.clj*, which we will need to import into our Python script.

```
(let [mean (sample (normal 0 1))
        y1 1
        y2 -1 ]
(if (> mean 0)
        (observe (normal mean 1) y1)
        (observe (normal mean -1) y2))
[mean, y1, y2])
```

The directed acyclic graphical model structure outputted is that of Program 2.2 and the generated computational graph that is used to interact with the inference engine and extract the program density can be seen in Program 2.3. In our system Python and Clojure syntax is interchangeable and so we can write our probabilistic model in Clojure, and run it natively in Python code.

```
from pyppl import compile_model
import gaussian_unknown_mean_branching_model as branching_model
compiled_python = compile_model(branching_model, language="clojure")
\# Vertices : 3, \#Arcs: 2
 Vertices V:
Vertex x30001 [Sample]
   Name:           x30001
   Ancestors:
   Cond−Ancs.:
   Dist−Args:      \{'loc': '0', 'scale': '1'\}
   Dist−Code:       dist .Normal(0, 1)
   Dist−Name:      Normal
   Dist−Type:       DistributionType .CONTINUOUS
   Sample−Size:    1
   Orig. Name:     mean
Vertex y30003 [Observe]
   Name:           y30003
   Ancestors:      x30001
   Conditions:     cond\_30002=True
   Cond−Ancs.:     x30001
   Cond−Nodes:     cond\_30002
   Dist−Args:      \{'loc': "state['x30001']", 'scale': '1'\}
   Dist−Code:       dist .Normal(state['x30001'], 1)
```

```
  Dist−Name:        Normal
  Dist−Type:        DistributionType .CONTINUOUS
  Sample−Size:      1
  Observation:      1
Vertex  y30004 [Observe]
  Name:             y30004
  Ancestors:        x30001
  Conditions:       cond\_30002=True
  Cond−Ancs.:       x30001
  Cond−Nodes:       cond\_30002
  Dist−Args:        \{ 'loc': " state ['x30001']",  'scale':  '−1'\}
  Dist−Code:        dist .Normal(state ['x30001'],  −1)
  Dist−Name:        Normal
  Dist−Type:        DistributionType .CONTINUOUS
  Sample−Size:      1
  Observation:      −1
Arcs  A:
  (x30001, y30003),  (x30001, y30004)

Conditions  C:
Condition
  Name:           cond\_30002
  Ancestors:      x30001
  Condition:      ( state ['x30001'] > 0)
  Function:       state ['x30001']
  Op:             >
```

Program 2.2: The directed acyclic graph $G(V, E)$ of Program 2.1. We can see how the vertices are formed, what types of variables they are, `sample` , `observe` , or conditional (conditions), and how the different variables relate to each other, and their types, are they continuous, or dis-continuous variables, which is critical to know for certain types of inference engines, Subsection 3.1.6.

```python
import torch
import torch. distributions  as  dist


class Model():

    def __init__( self ,  vertices : set, arcs: set, data: set, conditionals : set):
        super().__init__()
        self . vertices  =  vertices
        self . arcs  =  arcs
        self . data  =  data
        self . conditionals  =  conditionals

    def __repr__(self):
        V = '\n'. join(sorted([repr(v) for v in self.vertices]))
        A = ',␣'. join(['({},␣{})'.format(u.name, v.name) for (u, v) in self. arcs ])  if len(self.arcs) > 0 else '␣␣−'
        C = '\n'. join(sorted([repr(v) for v in self.conditionals])) if len(self. conditionals ) > 0 else '␣␣−'
        D = '\n'. join([repr(u) for u in self.data ])  if len(self.data) > 0 else '␣␣−'
        graph = ' Vertices␣V:\n{V}\nArcs␣A:\n␣␣{A}\n\nConditions␣C:\n{C}\n\nData␣D:\n{D}\n'.format(V=V, A=A, C=C, D=D)
        graph = '# Vertices:␣{},␣#Arcs:␣{}\n'.format(len(self.vertices), len(self. arcs)) + graph
        return graph

    def gen_cond_bit_vector( self ,  state ):
        result  = 0
        for cond in self. conditionals :
            result  = cond. update_bit_vector ( state ,  result )
        return result

    def gen_cond_vars(self):
        return [c.name for c in self. conditionals ]

    def gen_cont_vars( self ):
        return [v.name for v in self. vertices  if v. is_continuous  and not v. is_conditional  and v.is_sampled]

    def gen_disc_vars( self ):
        return [v.name for v in self. vertices  if v. is_discrete  and v.is_sampled]

    def gen_if_vars( self ):
        return [v.name for v in self. vertices  if v. is_conditional  and v.is_sampled and v. is_continuous ]

    def gen_log_prob(self,  state ):
        try:
            log_prob  = 0
            dst_  =  dist .Normal(loc=0,  scale=1)
            log_prob  = log_prob  + dst_ .log_prob( state ['x30001'])
            state ['cond_30002'] = ( state ['x30001'] > 0)
            dst_  =  dist .Normal(loc=state ['x30001'],  scale=1)
            if state ['cond_30002']:
                log_prob  = log_prob  + dst_ .log_prob( state ['y30003'])
            dst_  =  dist .Normal(loc=state ['x30001'],  scale=−1)
            if not state['cond_30002']:
                log_prob  = log_prob  + dst_ .log_prob( state ['y30004'])
            return log_prob
        except(ValueError, RuntimeError) as  e:
            print('****Warning:␣Target␣density␣is␣ill−defined****')

    def gen_prior_samples( self ):
        state  = {}
        dst_  = dist .Normal(loc=0,  scale=1)
        state ['x30001'] = dst_.sample()
        state ['cond_30002'] = ( state ['x30001'] > 0)
        dst_ = dist .Normal(loc=state ['x30001'],  scale=1)
        state ['y30003'] = 1
        dst_ = dist .Normal(loc=state ['x30001'],  scale=−1)
        state ['y30004'] = −1
        return state

    def get_arcs( self ):
        return self.arcs

    def get_arcs_names(self):
        return [(u.name, v.name) for (u, v)  in self . arcs ]

    def get_conditions ( self ):
        return self. conditionals

    def get_vars( self ):
        return [v.name for v in self. vertices  if v.is_sampled]

    def get_vertices ( self ):
        return self. vertices

    def get_vertices_names( self ):
        return [v.name for v in self. vertices ]
```

Program 2.3: The computational graph derived from Program 2.1. The compiler automatically generates the log probaiblity density of the program, which is typically used for inference, Chapter 3, for numerical stability. Furthermore, the output also enables sampling from the program, depedent on which condition is triggered.

The compilation target generated from our system generates a target that can be used to extend existing languages, such as Stan [Carpenter et al., 2017], to a larger class of models and enables us to integrate more computationally efficient inference engines, without having to redesign the underlying language, we present the full details in Chapter 4.

2.3 An interface for probabilistic programming

Previously, we saw that when we `sample` random variables we utilise the `sample` construct and when we need to condition on random-variables we utilise the `observe` construct. Formally, we state that `sample` will be used to make raw random draws $x_j \sim f_{a_j}(x_j|\phi_j)$ at a location in the program a_j, where a_j is an address in the execution trace and j is the index of the random draw in the execution trace, we will formally introduce the concept of a trace and address in Subsection 2.3.1. The raw random draws imply that they directly come from a distriubution type, rather than generated via a process, or separate sampling procedure. Here, $f_{a_j}(x_j, \phi_j)$ denotes a density, or mass function, depending on the type of the random variable, and ϕ_j is a subset of the variables in scope at the point of sampling, which may include distribution parameters, internal observations and other deterministic program variables. In a FOPPL, $f_{a_j}(x_j, \phi_j)$ terms directly correspond to the prior in the conventional sense, $p(x_j|\phi_j) = f_{a_j}(x_j, \phi_j)$ as there are no observation terms in ϕ_j, however, in UPPLs, as observations can appear in the ϕ_j terms this can lead to $p(x_j|\phi_j) \neq f_{a_j}(x_j, \phi_j)$ and so $f_{a_j}(\cdot)$ acts like a prior, but may not be a one-to-one mapping with a conventional prior, $p(\mathbf{x})$. Formally, the `observe` statements, denoted by

$g_{b_k}(y_k|\psi_k)$, represent the terms in the simulator that condition on some variables, with b_k serving a similar purpose to a_j and ψ_k shares the same definition as ϕ_j. When the language is a FOPPL, $g_{b_k}(.)$ will directly correspond to a likelihood term $p(\mathbf{x}|\mathbf{y})$, in the standard Bayesian sense. In contrast, as UPPLs allow for conditioning on stochastic observations on execution paths of the program, $g_{b_k}(\cdot)$ will not always directly correspond to a likelihood in the standard Bayesian sense and so the term likelihood, in this sense, is ill-defined. More generally, each observation y_k factors the program density by $g_{b_k}(y_k|\psi_k)$, where b_k is the location of this observation statement, and ψ_k are parameters of the factorisation.

2.3.1 Tracking random variables and program traces

Addresses a_j can be thought of as an index that enables the inference procedures implemented in the back-end of the system to understand the ordering of the random calls, or where the latent variables occur, so that the program density can be constructed correctly for inference. Associated with an address a_j are various quantities: such as the distribution type, is it continuous or dis-continuous, and the sampled value after execution, at that location. Formally, the set of address $\{a_1, \ldots, a_{n_x}\}$, from one forward run, forms a trace in the program comprised of a series of samples $\mathbf{x} = \{x_j\}_{j=1:n_x}$, where n_x is the number of latent variables in the given forward evaluation and can vary on each execution. The trace enables us to uniquely identify the elements of a sequence of samples, which is required in many sampling-based inference schemes. For example, Markov chain Monte Carlo based algorithms such as Lightweight Metropolis-Hastings [Wingate et al., 2011], Subsection 3.3.2, require a transition kernel which is a conditional distribution of one sample sequence given another sample sequence, to calculate the acceptance probability of accepting a new sample sequence given by the transition kernel. As ordering is important, by uniquely identifying each sample statement we can correctly align elements from the old and the new sample sequences. As such, each sampled value is then uniquely identified using its static address, and, when the language is dynamically evaluated using a trace-based inference scheme, an instance number, i_j. The instance number refers to the number of times the same sample statement, including the current one, has been encountered. In contrast, probabilistic programming systems with inference backends based on gradient-based Markov chain Monte Carlo methods, such as Hamiltonian Monte Carlo, do not require instance numbers, even if dynamically evaluated, as it does not depend on the trace construction, but requires knowledge about the types of the latent variables associated to each address.

As a concrete example lets consider the following Gaussian unknown-mean example, Program 2.4, where the goal is to find the means, μ_1 and μ_2, that best describe the distribution of the observations $\mathbf{y}$.

Program 2.4: Gaussian Unknown Means with Gamma prior.

```
y = 10
mu_1 = sample(normal(3,2))
mu_2 = sample(gamma(mu_1,1))
observe(normal(mu_2,5), y)
```

When the modelled is compiled the directed acyclic graph, Program 2.5 is created.

```
# Vertices : 3, #Arcs: 2
Vertices V:
Vertex x30001 [Sample]
  Name:          x30001
  Ancestors:
  Cond−Ancs.:
  Dist−Args:     {'loc': '3', 'scale': '2'}
  Dist−Code:     dist.Normal(3, 2)
  Dist−Name:     Normal
  Dist−Type:     DistributionType.CONTINUOUS
  Sample−Size:   1
  Orig. Name:    mu_1
Vertex x30002 [Sample]
  Name:          x30002
  Ancestors:     x30001
  Cond−Ancs.:
  Dist−Args:     {'alpha': "state['x30001']", 'beta': '1'}
  Dist−Code:     dist.Gamma(state['x30001'], 1)
  Dist−Name:     Gamma
  Dist−Type:     DistributionType.CONTINUOUS
  Sample−Size:   1
  Orig. Name:    mu_2
Vertex y30003 [Observe]
  Name:          y30003
  Ancestors:     x30002
  Conditions:
  Cond−Ancs.:
  Cond−Nodes:
  Dist−Args:     {'loc': "state['x30002']", 'scale': '5'}
  Dist−Code:     dist.Normal(state['x30002'], 5)
  Dist−Name:     Normal
  Dist−Type:     DistributionType.CONTINUOUS
  Sample−Size:   1
  Observation:   10
Arcs A:
  (x30001, x30002), (x30002, y30003)
```

Program 2.5: The graph $G(V, E)$ of Program 2.4

We see in the compiled output, Program 2.5, that all random variables have a unique identifier, i.e. the sampled variable μ_1 is represented by `Vertex x30001`. This is the address index, i.e. $a_1 = x30001$, and the information associated with the address tells us the distribution type, continuous for μ_1, the arguments ϕ in the distribution object, 0 and 1. Furthermore, if random variables are dependent on one-another, the compiled graph also details this; such as the execution of μ_2, which is an ancestor of μ_1.

Extracting the Addresses from Real-world Simulators

Here we provide an example of what the addresses and traces look like for a real-world simulator, a Malaria virus vector simulator, OpenMalaria [Smith, 2008], generated using the Probabilistic Programming eXecution (PPX) protocols and corresponding coroutines that we develop in

Chapter 5. In Figure 2.2, we show the trace graphs after a forward execution of each simulator, connected via the PPX protocols to PyProb [Le et al., 2017], a Python-based UPPL.

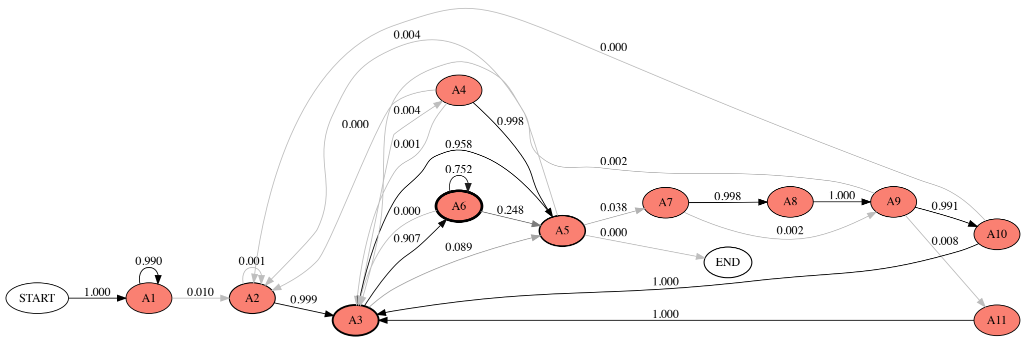


Figure 2.2: Here we run the OpenMalaria simulator and plot the outputted trace paths and path probabilities, generated by thousands of calls to each address $A_1, \ldots, A_{n_x}$.

For each trace we have the addresses $A_1, \ldots, A_{n_x}$, where the addresses represent the sampled variables in the trace $\mathbf{x} = \{x_j\}_{j=1:n_x}$ and along each edge in the graph denotes the trace probability. For the OpenMalaria simulator, in Table 2.1, we can see the distribution object for each address, the parameters to that distribution object and the location in the physical code base.

Table 2.1: An example of an address generated for the model run forward in the OpenMalaria simulator. We can see that for each address we extract its distribution types, parameters to the distribution, location information of the code and the related function calls to that address.

Address ID	Full address

A1	[forward()+0x204; OM::Simulator::
	start(scnXml::Monitoring const)+0x28a;
	OM::Population::createInitialHumans()+0x94;
	OM::Population::newHuman(OM::SimTime)+0x5c;
	OM::Host::Human::Human(OM::SimTime)+0x12b;
	OM::WithinHost::WHInterface::createWithinHostModel(double)+0x99;
	OM::WithinHost::DescriptiveWithinHostModel::DescriptiveWithinHostModel(double)+0x3a;
	OM::WithinHost::WHFalciparum::WHFalciparum(double)+0xe6;
	OM::util::random::gauss(double, double)+0xb4]__Normal

⋮	⋮
⋮	⋮

A5	[forward()+0x204; OM::Simulator::start(scnXml::Monitoring const&)+0x468;
	OM::Population::update1(OM::SimTime)+0xff;
	OM::Host::Human::update(bool)+0x2bc;
	OM::Clinical::ClinicalModel::update(OM::Host::Human&, double, bool)+0x96;
	OM::Host::NeonatalMortality::eventNeonatalMortality()+0x9;
	OM::util::random::uniform_01()+0xc0]__Uniform

⋮	⋮

2.4 Simulators in a probabilistic programming environment

"one stocastic variable 'x'". The variable for the bias is called 'bias' in Listing 2.6 but 'x' here. And then on the next page, 'x' is also used to indicate the variable for the outcome of the coin toss and also the value of this variable.¬ This needs to be corrected throughout to distinguish between these three quantities e.g. use 'bias' for the bias, 'x' for the coin toss and 'x=H' when referring to values of 'x'

Returning to the coin-flip example in Section 1.4, where we have a worn coin and want to learn how bias the coin is, so that we can construct a stochastic simulator to emulate the coin, Program 2.6. We could develop a deterministic model that relies on physical dynamics to determine the outcome of a flipped coin for a given set of physical variables, and experimentally determine the coin's bias depending on how it follows the true physics. However, this is incredibly complex, as there are many different variables that we have to account for, some of which may be unknown-unknowns; that is we are blind to their existence and how they affect a process. Alternatively, we can remove aspects of this complexity by ignoring the physical dynamics, and modelling the coin as a stochastic decision-maker that has some level of bias to be learnt, Program 2.6.

```
def coin_flip(bias):
  return sample(bernoulli(bias))
```

Program 2.6: The coin-flip flipper simulator.

where the simulator coin−flip takes as input a bias, and returns 1, for heads, or 0, for tails depending on the outcome of the draw from the `bernoulli` distribution. If we learn the correct bias by conditioning on our experimental observations, then collecting outputs from this program will lead to the same posterior generated from conditioning on the observations of the flipped coins. Thus, the program defines a forward generative model that is now capable of simulating the outcome of a coin-flip, and enables us to gather observations from a coin with a given bias, without ever having to flip a real coin.

In this model we assume that we need one stochastic variable $\mathbf{x}$, the probability that the outcome is heads, $p(\mathbf{x} = H)$. We know that we can observe two possible outcomes, $\mathbf{y} = H$, heads, or $\mathbf{y} = T$, tails and we assume it is impossible for the coin to land on its side, $\mathbf{y} = S$, $p(\mathbf{x} = S) = 0$.

If the coin were fair, no bias, we would see that the probability of $p(\mathbf{x}) = p(\neg\mathbf{x}) = 0.5$, but if the coin is biased, then this will not be the case. So how do we determine the bias, which is represented by our latent vairable x? Do we experimentally flip the coin many times and assume that the end result, after some frequency of heads, n_h, and number of flips, n_f, to be the correct probability i.e., $p(\mathbf{x}) = \frac{n_h}{n_f}$? We could, but looking solely at the frequency would only give us a point estimate, so we would not know how "good" our estimate is. Or, do we encode our *prior* beliefs into the model, that the coin is worn, thus potentially biased and let this subjective view determine the posterior prediction?

In order to leverage the Bayesian framework, Section 1.4, we require a prior $p(\mathbf{x})$ and a likelihood term $p(\mathbf{y}|\mathbf{x})$. Once these are defined our objective is to calculate the posterior, $p(\mathbf{x}|\mathbf{y})$ to determine if the coin is biased, i.e is $p(\mathbf{x} = H) > 0.5$, or $p(\mathbf{x} = H) < 0.5$ a valid assumption, given our observations.
We initially assume that we have a non-informative prior, that is the probability that the outcome is heads, $p(\mathbf{x} = H)$, follows a `uniform` distribution on the interval [0,1], so we are favouring no particular initial belief about the biasedness of the coin, thus the form of the prior is $p(\mathbf{x}) = U(0, 1)$ and we sample from this prior during each forward execution of the model, $\mathbf{x} \sim U(0, 1)$. If we believe that the coin prefers one outcome more than another, we can incorporate that belief with a prior that follows a `beta` distribution and recalculate the posterior. With probabilistic programming this is made incredibly simple, see Programs 2.7 and 2.8.

Once we have chosen our prior we construct the likelihood of our model. We have a natural distribution that describes our problem set-up for the biased coin, the binomial distribution $Bin(n, \mathbf{x})$. The Binomial distribution is a discrete distribution that enables us to derive the probability of determining n successes, in this case, the number of heads, $n = n_h$, from $N = n_h + n_t$ independent runs of our model, flipping the coin, where the result of each forward run is either H, with probability $p(\mathbf{x} = H) = \mathbf{x}$, or T with probability $p(\neg\mathbf{x}) = 1 - p(\mathbf{x} = H)$.
We can now calculate the posterior. If we did not have access to a probabilistic programming system we would have to analytically derive the posterior for each new prior, or likelihood, we choose. We don't have to do that here and leave that labourious exercise to Appendix A.1. With a probabilistic programming system all we have to do is run our program code, Program 2.7, and the posterior required to determine the biasedness of the coin will be returned.

```
def model(N, data):
  x = sample(uniform(0, 1))
  observe(binomial(N, x), data)
  return x
```

Program 2.7: The coin-flip model with Uniform prior.

In Program 2.7, N is the total number of flips and $data$ is generated from experimental observations, but this could equally be generated from the simulated coin-flip, Program 2.6, for a given $bias$. The $data = [5, 87, 670]$, which represents the number of heads observed, n_h, from $N = [10, 100, 1000]$ coin flips. When we execute the program in Program 2.7 the code is compiled, which enables us to extract the program density, as we saw previously, Section 2.3. Using this compilation output, we can utilise PyMC3's inference back-end to calculate the posterior, see Figure A.1. In this instance, the program density will be the numerator of Equation A.3, in Appendix A.1. However, this posterior will change as we change the structure of the code with different model features, priors and likelihoods. Nonetheless, the probabilistic programming system will take care of this.

Modelling with Different Priors

As noted previously, if we had seen that we have more heads than tails, as we saw in the second and third set of observations, then we could use a `beta` distribution as the prior, skewed towards the left tail, instead of the original `uniform` prior, this amounts to changing one line of code in the probabilistic program in Program 2.7.

```
def model(n, data):
  x = sample(beta(2, 5))
  observe(binomial(n, x), data)
  return x
```

Program 2.8: The coin-flip FOPPL program with Beta prior

We compile this program in our FOPPL system and we get our posterior samples for the $bias$. See Figure 2.3 for the posterior outputs and trace plots of the sampled value for each iteration, for each of these models, under the same set of observations as the original model. We see that the posteriors are equivalent to our hand-calculated case, Appendix A.1, as expected and the expected value for the bias parameter of Programs 2.7 is $\mathbb{E}[bias] = 0.67$. We note that by changing our subjective prior beliefs, the posterior of the model evolves differently as it consumes new data. Eventually, both prior choices lead to the same final posterior after a large number of observations, but for lower numbers of observations each model predicts a different expected level of bias for the coin. This is the subjective component of Bayesian modelling.

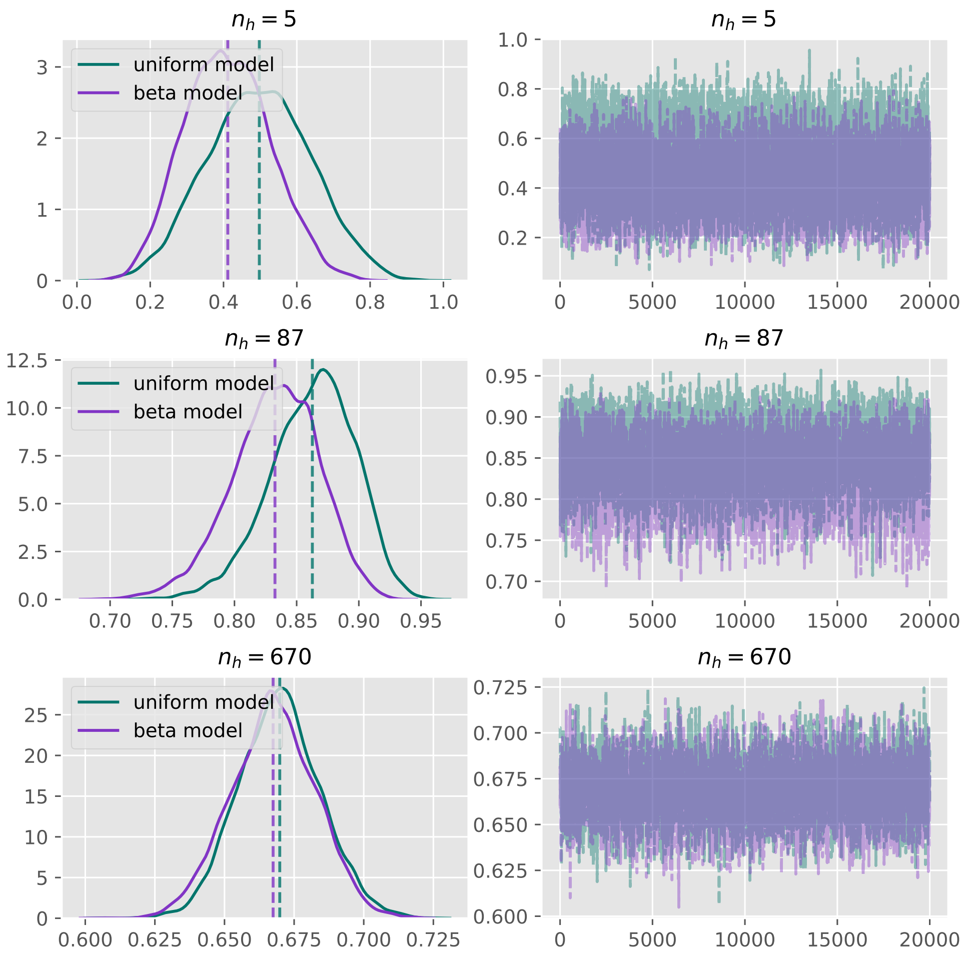


Figure 2.3: Posterior plots, left, and inference trace plots for $20,000$ samples from Bayesian inference algorithm, right, for the bias parameter of the coin-flip program, for both a Uniform, green, and $Beta$, purple, prior.

2.4.1 Inference challenges

In Appendix A.1 we describe how to calculate the posterior for the coin-flip model with the uniform prior and show how doing Bayesian inference, even in simple models, without probabilistic programming is laborious. Unfortunately, there are many probabilistic models of practical interest for which exact inference is impossible, at least with known methods, and so we must revert to numerical inference algorithms, Chapter 3. These algorithms can be broken down into two distinct classes: exact and approximate inference schemes. Exact inference schemes generate the exact marginals when the underlying factors of the graphical model form a tree, although methods such as cut-set conditioning, and junction tree do not require this. However, such methods like cut-set conditioning and junction trees only work well for models with a small number of variables. These schemes typically utilise message-passing schemes to exchange information between nodes in the graphical model [Minka et al., 2014] and are effective and computationally efficient, for certain types of graphical models that can be factored. In particular, models where the latent variables are from the exponential families [Murphy, 2012]. However, the complexity of exact inference on arbitrary graphical models is NP-hard and so we can only perform exact inference on a small subset of probabilistic models.

Thus, many problems can only be solved with approximate inference schemes. These schemes come in two forms: sampling-based, which include Markov chain Monte Carlo and particle based methods, and optimisation-based, which include variational methods. We focus on the former in this thesis. Optimisation-based methods exploit local and global structures to determine the optimal parameters for approximating an analytical form to the posterior of the latent variables, while simultaneously computing the lower bound of the marginal likelihood - where the higher the marginal likelihood the better. While these methods can be effective, efficient and scalable to large dimensions, it is difficult to know how far away you are from the true posterior of your given model, or program, as you only know that you are in some vicinity of the posterior, which may mean samples generated from the learned generative model will not be entirely representative of the observations that the program is trying to describe. In contrast, sampling-based methods will asymptotically converge to the correct posterior, in the limit of a large number of samples, or particles. This too can be framed in a optimisation perspective, as you are learning the best set of latent variables from some set of feasible alternatives that describe your data in the best context under a pre-defined objective function. Typically, sampling based methods struggle to scale to big-data, high-dimensional latent spaces, and to probabilistic models that contain mixtures of latent variable types i.e. continuous and discrete. But, as we shall see in Chapters 4, 5 and 6, we develop compilation techniques and inference procedures to mitigate these problems for a large class of probabilistic models.

3 Inference Algorithms

In this chapter we will explore a number of important sampling algorithms based on the principles of Monte Carlo (MC) [Metropolis and Ulam, 1949], that form the foundation of many of the sampling schemes utilised throughout the presented works in this thesis, and in general probabilistic programming systems, Section 3.3. Monte Carlo methods are ubiquitous across statistics, computational physics, numerical integration and many other domains, due to their simplicity and ability to generate samples from high-dimensional functions. The core idea of Monte Carlo methods is to use some form of proposal distribution that we can easily sample from and then make appropriate adjustments to achieve, in most cases approximate, samples from the posterior. Sampling schemes constructed from the principles of Monte Carlo will help us to evaluate integrals that we cannot necessarily analytically integrate, but can evaluate in a point-wise fashion, enabling us to numerically estimate their value. In the context of probabilistic inference, these integrals typically involve expectations, which are the quantities of interest when we want to learn the expected posteriors of the latent variables in our simulator, when utilising Bayes' theorem. Unfortunately, integrating out particular variables to extract the marginals from the joint density, the denominator in Bayes' theorem, Equation 1.1, is intractable for most real-world problems and so we must find away to estimate the integral instead. Utilising Monte Carlo methods will enable us to calculate those marginals, under light assumptions that we can perform point-wise evaluations of the function of interest such that they are independent from one another. Although, as we shall see in Subsections 3.1.3 and 3.1.4, we do not actually need to calculate the marginal as we can generate posterior samples through unnormalised versions of the posterior.

3.1 Monte Carlo

We now introduce the basic principal of Monte Carlo methods, that is Monte Carlo integration, which states given some function of interest $\eta(\mathbf{x})$, and a set of N independent and identically distributed (IID) samples $\hat{\mathbf{x}}_n \sim \pi(\mathbf{x})$ where $n = 1, \ldots, N$, from a target distribution $\pi(\mathbf{x})$, the expected value of any function of a random variable can be approximated as

$$I = \mathbb{E}[\eta(\mathbf{x})] = \int \eta(\mathbf{x})\pi(\mathbf{x})dx \approx I_N = \frac{1}{N}\sum_{n=1}^{N} \eta(\hat{\mathbf{x}}_n) \tag{3.1}$$

In our setting, the target distribution will always be a posterior distribution, typically $p(\mathbf{x}|\mathbf{y})$. By varying $\eta(\mathbf{x})$ we can calculate many quantities of interest such as the expected value $\mathbb{E}[\mathbf{x}] = \frac{1}{N}\sum_{n=1}^{N} \hat{\mathbf{x}}_n$ and variance $Var[\mathbf{x}] = \frac{1}{N}\sum_{n=1}^{N}(\hat{\mathbf{x}}_n - \mathbb{E}[\mathbf{x}])^2$, which is required when we need to construct the credible interval, if we are in the Bayesian regime, or confidence interval if we are in the frequentist regime.

A nice property of Monte Carlo integration is that it leads to an unbiased estimator because as $N \to \infty$, the expected value of the integral goes to the true value of the integral, $\mathbb{E}[I_N] = I$, provided the samples generated are IID. This means that Monte Carlo does not introduce any bias into to the approximation, that is, it will not be an over, or under estimate. If the samples are not IID then the estimator will be biased and so the converse is true. The fact that Monte Carlo generates an unbiased approximation, when samples are IID, is because of the law of large numbers, which states that if the samples are IID, then the empirical mean $\frac{1}{N}\sum_{n=1}^{N} \eta(\hat{\mathbf{x}}_n)$, equals the true expected value $\mathbb{E}[\eta(\mathbf{x})]$ for the function $\eta(\mathbf{x})$.

3.1.1 Rejection sampling

Rejection sampling is one of the most trivial Monte Carlo methods and provides a way to sample from distributions that we are unable to directly evaluate the cumulative density function of. It is the de facto way to sample in traditional stochastic simulators due to its ability to generate exact samples from the target density $\pi(\mathbf{x}) = \frac{\gamma(\mathbf{x})}{Z}$, even when we can only evaluate the unnormalised target density $\gamma(\mathbf{x})$, where Z is the normalisation constant, which is possibly unknown. To generate exact samples from $\pi(\mathbf{x})$ we introduce a proposal $q(\mathbf{x})$ that can be simulated by some known method and satisfies $\gamma(\mathbf{x}) \leq cq(\mathbf{x})$ for all $\mathbf{x}$, and some constant c. Thus, to perform rejection sampling we introduce a simple function $q(\mathbf{x})$ such that it encompasses all of $\gamma(\mathbf{x})$. We can then generate a sample from $\pi(\mathbf{x})$ in the following way:

1. Generate $\hat{\mathbf{x}}_n \sim q(\mathbf{x})$ and $u \sim Uniform(0, 1)$ (picking a random $\hat{\mathbf{x}}_n$ location and height $\mathbf{y} = u$ under the envelope $cq(\mathbf{x})$)

2. `if` $u \leq \frac{\gamma(\hat{\mathbf{x}}_n)}{cq(\mathbf{x})}$; `return` $\hat{\mathbf{x}}_n = \mathbf{y}$

```
3. else; Go to step 1
```

As we generate with probability $q(\mathbf{x})$ and accept with probability $\frac{\gamma(\mathbf{x})}{cq(\mathbf{x})}$, the probability of acceptance is

$$\mathbf{E}[A(\mathbf{x})] = \int \frac{\gamma(\mathbf{x})}{cq(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} = \frac{1}{c} \int \gamma(\mathbf{x}) d\mathbf{x} \tag{3.2}$$

and so we want to choose c to be as small as possible, such that $cq(\mathbf{x})$ is larger than $\gamma(\mathbf{x})$ and ensuring the initial constraint is satisfied, to ensure $A(\mathbf{x}) \approx 1$. However, even if we choose a good c this may still result in a large number of rejections, especially if $\mathbf{x}$ is high-dimensional, because the probability of landing in the acceptance region decreases exponentially as the number of dimensions increases. This is in contrast to Markov chain Monte Carlo methods, that we will explore in Subsection 3.1.4, which work well in high-dimensional spaces.

As an example of rejection sampling, consider we are in the Bayesian setting and our target is the posterior $\pi(\mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})/p(\mathbf{y})$, thus $\gamma(\mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$, and we choose a proposal to have the same form as the prior $q(\mathbf{x}) = p(\mathbf{x})$ and set $c = p(\mathbf{y}|\tilde{\mathbf{x}})$, where $\tilde{\mathbf{x}} = \arg\max p(\mathbf{y}|\mathbf{x})$ is the maximum likelihood estimate, suggested by Smith and Gelfand [1992]. This leads to an acceptance probability of $A(\mathbf{x}) = \frac{p(\mathbf{y}|\mathbf{x})}{p(\mathbf{y}|\tilde{\mathbf{x}})}$ as the $p(\mathbf{x})$ terms cancel. Thus, samples from the prior that have a higher likelihood are more likely to be retained in the posterior. However, if there is a large mis-match between the prior and posterior, then $A(\mathbf{x}) < 1$ and so sampling will be very inefficient.

The next Monte Carlo method that we will explore is importance sampling, which like rejection sampling uses a proposal distribution q, but instead of an accept, or reject step, we generate importance weights for each sample, which we can then use to generate an estimate of the posterior distribution, or the value of an integral.

3.1.2 Importance sampling

In importance sampling we are working under the assumption that we cannot sample from the target density, $\pi(\mathbf{x})$, but we can do point evaluations of it and can also sample from the proposal distribution $q(\mathbf{x})$. This allows us to construct a different Monte Carlo estimator and evaluate the expectation as

$$\mathbb{E}[\eta(\mathbf{x})] = \int \eta(\mathbf{x}) \frac{\pi(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \approx \frac{1}{N} \sum_{n=1}^{N} \omega_n \eta(\hat{\mathbf{x}}_n) = \hat{I} \tag{3.3}$$

where $\omega_n = \frac{\pi(\hat{\mathbf{x}})}{q(\hat{\mathbf{x}})}$ are the importance weights and $\hat{\mathbf{x}}_n \sim q(\mathbf{x})$, and is valid, provided $\frac{q(\mathbf{x})}{q(\mathbf{x})} = 1$ for all points where $q(\mathbf{x}) \neq 0$.

The key idea behind importance sampling is to draw samples $\mathbf{x}$ in regions that have high probability $\pi(\mathbf{x})$ and $|\eta(\mathbf{x})|$ is large, which leads to a low-variance estimator provided that a proposal that generates importance weights that have a value close to one. Moreover, the importance sampling estimate is unbiased, which can be seen as follows

$$\mathbb{E}[\frac{1}{N}\sum_{n=1}^{N}\frac{\pi(\hat{\mathbf{x}}_n)}{q(\hat{\mathbf{x}}_\mathbf{n})}\eta(\hat{\mathbf{x}}_n)] = \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_q[\frac{\pi(\hat{\mathbf{x}}_n)}{q(\hat{\mathbf{x}}_n)}\eta(\hat{\mathbf{x}}_n)] = \mathbb{E}_\pi[\eta(\mathbf{x})] \tag{3.4}$$

where we have used the law of large numbers in the last equality.

A common challenge that occurs in importance sampling is how to choose the right proposal, this is a complex question to answer in practice, but it can be shown that the optimal proposal $q^*(\mathbf{x})$ is [Owen, 2013, Chapter 9]

$$q^*(\mathbf{x}) = \frac{|\eta(\mathbf{x})|\pi(\mathbf{x})}{\mathbb{E}_{\pi(\mathbf{x})}[\eta(\mathbf{x})]} \tag{3.5}$$

Further challenges in importance sampling occur when we cannot evaluate $\pi(\mathbf{x})$ and only know the unnormalised target density $\gamma(\mathbf{x}) = \frac{\pi(\mathbf{x})}{Z}$. Fortunately, we can still utilise importance sampling even if we cannot directly evaluate the target through a process called self-normalised importance sampling, Subsection 3.1.3, however, utilising this method gives rise to a biased estimator.

3.1.3 Self-normalised importance sampling

When we are in the Bayesian inference setting, the marginal likelihood, normalisation constant, is typically intractable, which means importance sampling, Subsection 3.1.2, is not applicable to settings where the marginal likelihood is not known, as we need to be able to evaluate $\pi(\mathbf{x})$ in a point-wise fashion. However, we can typically evaluate the joint density $p(\mathbf{x}, \mathbf{y})$, which is proportional to the posterior density $p(\mathbf{x}, \mathbf{y}) \propto p(\mathbf{x}|\mathbf{y})$. Thankfully, we can adapt importance sampling so that we can evaluate expectations without having to sample directly from the target distribution $\pi(\mathbf{x}) = \frac{\gamma(\mathbf{x})}{Z}$, by self-normalising the importance weights. To do this, we start by creating Monte Carlo estimators for both Z and $Z\mathbb{E}_{\pi(\mathbf{x}}[\eta(\mathbf{x})]$. For Z, the Monte Carlo estimator is $Z_N = \frac{1}{N}\sum_{n=1}^{N}\omega_n$ and so $\mathbb{E}[Z_N] = Z$ as the Monte Carlo estimator converges to the true expectation as the number of samples increases. Thus, for $Z\mathbb{E}_{\pi(\mathbf{x}}[\eta(\mathbf{x})]$, we note $\mathbb{E}_q[\frac{\gamma(\mathbf{x})}{q(\mathbf{x})}\eta(\mathbf{x})] = Z\mathbb{E}_\pi[\eta(\mathbf{x})]$, and so if we take the ratio of these two estimators then we can construct an expectation for $\eta(\mathbf{x})$ under the distribution $\pi(\mathbf{x})$ that does not require the normalisation constants as follows

$$I = \mathbb{E}_\pi[\eta(\mathbf{x})] \approx \hat{I} = \frac{\frac{1}{N}\sum_{n=1}^{N}\omega_n\eta(\hat{\mathbf{x}}_n)}{\frac{1}{N}\sum_{n=1}^{N}\omega_n} = \sum_{n=1}^{N}\bar{\omega}_n\eta(\hat{\mathbf{x}}_n) \tag{3.6}$$

where $\hat{\mathbf{x}}_n \sim q$ and $\omega_n = \frac{\gamma(\hat{x}_n)}{q(\hat{x}_n)}$ are the unnormalised importance weights. The normalised importance weights are defined as $\bar{\omega}_n = \frac{\omega_n}{\sum_n \omega_n}$ as $\sum_{n=1}^{N}\bar{\omega}_n = 1$. However, as the integral is now a ratio of two estimates and the numerator and denominator are correlated - since they are using the same set of samples, this estimator is biased for a finite N, and so the expectation $\mathbb{E}[\hat{I}] \neq I$.

As we did in importance sampling we can also construct an optimal proposal q^* for a self-normalised importance sampler, which is given by [Owen, 2013][Chapter 9]

$$q^*(\mathbf{x}) = \frac{\pi(\mathbf{x})|\eta(\mathbf{x}) - I|}{\int \pi(\mathbf{x})|\eta(\mathbf{x}) - I|d\mathbf{x}} \tag{3.7}$$

However, in many cases $\eta(\mathbf{x})$ is not known ahead of time and we might want to evaluate I for multiple $\eta(\mathbf{x})$, so we still need to generate samples for future use. Fortunately, we can carry out importance sampling in the same vein by sampling from q and generating a set of weighted samples $\{\hat{\mathbf{x}}_n, \omega_n\}_{n=1:N}$ that can be used to approximate the posterior $\pi(\mathbf{x})$ as a weighted sum of delta distributions centered at $\hat{\mathbf{x}}_n$

$$\pi(\mathbf{x}) \approx \hat{\pi}(\mathbf{x}) = \sum_{n=1}^{N}\bar{\omega}_n\delta_{\hat{x}_n}(\hat{\mathbf{x}}_n - \mathbf{x}). \tag{3.8}$$

When we do not have access to η there is no single optimal proposal distribution. Instead, we often consider an optimal proposal to be

$$q^*(\mathbf{x}) = \pi(\mathbf{x}). \tag{3.9}$$

If we manage to achieve this, then the self-normalised importance estimator has the same form as the standard Monte Carlo estimator, Equation 3.1, that is

$$\hat{I} = \frac{1}{N}\sum_{n=1}^{N}\eta(\hat{\mathbf{x}}_n) \text{ where } \hat{\mathbf{x}}_n \sim q^*. \tag{3.10}$$

Unfortunately, similar to rejection sampling, as the dimensionality of the variables increases, the sampling efficiency decreases and renders importance sampling ineffective. For importance sampling to be sample efficient, that is ensuring a significant proportion of the weights are non-neglible, requires choosing a good proposal distribution, which is challenging to do in practice.

If the proposal is chosen poorly, then the likelihood of a given weight being greater than zero diminishes as the dimensionality grows, thus sample efficiency decreases. In Subsection 3.3.3, we introduce amortized importance sampling, which removes the difficulty of defining a proposal, by learning offline a family of proposal distributions in a data-driven way, such that the proposal learnt offline can then be used online to generate fewer non-neglible weights, making importance sampling more sample efficient. In Subsection 3.1.4 we introduce a set of sampling techniques called Markov chain Monte Carlo that utilise the Markov property and Monte Carlo to construct efficient samplers that can explore the target distribution more effectively as the dimensionality of the variables increase.

3.1.4 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods describe a family of popular numerical inference techniques, that provide methods for evaluating functions correctly in some asymptotic limit [Gilks et al., 1995; Hastings, 1970; Metropolis et al., 1953]. They are used widely in Bayesian inference due to their ability to scale, without significantly reducing sample efficiency when our probabilistic model has high-dimensional observations and latent variables, see Chapters 4,5,6 and Subsection 3.1.6. MCMC methods work by leveraging the Markov property and by constructing a valid Markov chain that has the target distribution as its equilibrium distribution. The Markov property, Equation 3.11, states that each new state $\mathbf{x}_n$ is only conditioned on the last state $\mathbf{x}_{n-1}$, and is independent of all states prior to the $\mathbf{x}_{n-1}$ state, that is

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}) \tag{3.11}$$

and for the Markov chain, $\mathbf{x}_1, \ldots, \mathbf{x}_n$, each state is determined solely from the probability of the initial state $p(\mathbf{x}_1)$, and the probabilities to transition between each state are given by $p(\mathbf{x}_{n+1}|\mathbf{x}_n)$. When each transition probability is the same $p(\mathbf{x}_{n+1} = j|\mathbf{x}_n = i) = p(\mathbf{x}_n = i|\mathbf{x}_{n-1} = j)$, then the Markov chain is called homogeneous, as it only depends on i and j, not n. (time) homogeneous The term $p(\mathbf{x}_{n+1}|\mathbf{x}_n)$ is call the transition kernel (or proposal distribution), $\kappa(\mathbf{x}_n \to \mathbf{x}_{n+1})$, and describes the probability of moving between states. For the Markov chain to converge to the target distribution $\pi(\mathbf{x})$, we need the chain to converge to some equilibrium value, that is $\lim_{n\to\infty} p(\mathbf{x}_n = \mathbf{x}) = \pi(\mathbf{x})$ for any possible initial state $\mathbf{x}_1$, and we require that $\pi(\mathbf{x})$ be a stationary distribution of the Markov chain. For this to be satisfied the following condition must met

$$\pi(\mathbf{x}) = \int \kappa(\mathbf{x} \to \mathbf{x}')\pi(\mathbf{x}')d\mathbf{x} \tag{3.12}$$

If this condition is met, then the target distribution is invariant with respect to the transition kernel. Thus, after some time $t = n$ our chain will produce samples from the target distribution, such that if $p(\mathbf{x}_t) = \pi(\mathbf{x})$, then all subsequent $\mathbf{x}_{t+1}$ will have come from the target distribution. This property, that all starting points converge to the target distribution, is known as ergodicity and requires the Markov chain to be irreducible, that is, all states with non-neglible probability in the chain can be reached in a finite number of steps, and aperiodic, that no states can only be reached at certain periods of time. This means that the transition probabilities are non-neglible for all sufficiently large n. Much of the difficulty in MCMC lies in constructing valid transition kernels and a common, but not necessary, condition for constructing valid Markov chains is to ensure the chain satisfies the condition of detailed balance [Hastings, 1970; Metropolis et al., 1953], which is satisfied if given a target distribution $\pi(\mathbf{x})$ satisfies

$$\pi(\mathbf{x}')\kappa(\mathbf{x}' \to \mathbf{x}) = \pi(\mathbf{x})\kappa(\mathbf{x} \to \mathbf{x}'). \tag{3.13}$$

Any chain that satisfies this property is called reversible, although there are classes of non-reversible chains such as the bouncy particle sampler [Bouchard-Côté et al., 2018], which can converge faster to the equilibrium distribution [Hwang et al., 2005], but we shall not discuss them in this thesis. The MCMC methods that we introduce in this thesis, Subsections 3.1.6,3.1.5 and 3.3.2 all produce reversible chains. By imposing these conditions MCMC methods are able to overcome the curse of dimensionality. As the transition between states is independent of all states in the chain, except the previous state, it means that moves in the chain occur locally with respect to the current position, and so rather than trying to independently sample from the target distribution, as in the case of rejection and importance sampling, we instead use the local knowledge of the space to guide us. That is, most of the probability mass, or density, in a high-dimensional space will be concentrated at the same location, so it makes sense to use the local information to guide your search of the state space. MCMC still struggles when the target distribution has multiple modalities, as if certain modes have a lower mass, or density, it can be difficult to make the local move, unless the proposal q accounts for it.

3.1.5 Metropolis Hastings

The Metropolis-Hastings (MH) algorithm is the backbone of many prominent MCMC methods, as it is simple to implement and satisfies the detailed balance criteria, provided a suitable kernel, proposal distribution, is used. The algorithm works by proposing new locations to explore in the state space $\mathbf{x}'$, and then determines whether or not moving to that new location is sensible given the current location $\mathbf{x}$, according to a proposal $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x})$, which is conditioned on the current location via an accept-reject criteria, with the probability of accepting determined by

$$A(\mathbf{x} \to \mathbf{x}') = \min(1, \frac{\gamma(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}) \tag{3.14}$$

and if the sample is accepted, then at iteration n we set $\mathbf{x}_{n+1} = \mathbf{x}'$, else we remain at $\mathbf{x}$. The full MH algorithm is shown in Algorithm 1. Notice that the normalisation constant Z cancels out, as $Z\pi(\mathbf{x}) = \gamma(\mathbf{x})$, thus we can evaluate $\gamma(\mathbf{x})$ to generate samples from the target distribution $\pi(\mathbf{x})$, without requiring the normalisation constant. The acceptance ratio defined in Equation 3.14 is commonly used in MCMC algorithms to ensure that detailed balance is satisfied in the given inference scheme. However, it should be noted that the sample efficiency of MH is directly dependent on the choice of the proposal, for example, if q was independent of $\mathbf{x}$ then it will essentially ignore local information and produce a sampler that is less sample efficient than importance sampling, as the samples are generated in the same way as importance sampling, but information is lost in the accept-reject step [Rainforth, 2018a]. Furthermore, if the target contains more than one mode, it can take the sampler a long time to explore all the modes and generate a sufficient amount of samples, so a lot of work on MCMC goes into finding efficient ways to explore the state space.

Algorithm 1 Metropolis-Hasting algorithm

```
1: Initialise $\mathbf{x}_1$                                  ▷ Typically done by sampling from the prior
2: for $n = 1, 2, 3, \dots, N$ do
3:     $\mathbf{x} \to \mathbf{x}_n$
4:     $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x})$
5:     $\alpha = min(1, \frac{\gamma(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x})})$        ▷ Acceptance condition
6:     $u \sim U(0, 1)$
7:     $\mathbf{x}_{n+1} = \begin{cases} \mathbf{x}' \text{ if } u < \alpha \\ \mathbf{x} \text{ if } u \geq \alpha \end{cases}$
8: end for
9: return $\mathbf{x}_{n+1}$
```

Why Metropolis-Hastings works

To prove that MH generates samples from the target density $\gamma(\mathbf{x})$, we start by nothing that the MH algorithm defines a Markov chain with the following transition kernel

$$\kappa(\mathbf{x}'|\mathbf{x}) = \kappa(\mathbf{x} \to \mathbf{x}') = \begin{cases} q(\mathbf{x}'|\mathbf{x})A(\mathbf{x} \to \mathbf{x}') \text{ if } \mathbf{x}' \neq \mathbf{x} \\ q(\mathbf{x}|\mathbf{x}) + \int_{\mathbb{R}} q(\mathbf{x}''|\mathbf{x})(1 - A(\mathbf{x} \to \mathbf{x}''))\mathbb{I}[\mathbf{x}'' \neq \mathbf{x}] \text{ otherwise} \end{cases} \tag{3.15}$$

This means, if we propose a state $\mathbf{x}'$ with probability $q(\mathbf{x}'|\mathbf{x})$ then it must have been accepted with probability $A(\mathbf{x} \to \mathbf{x}')$, otherwise you stay in state $\mathbf{x}$ because you either proposed this state with $q(\mathbf{x}'|\mathbf{x})$, or the proposed state was rejected with probability $1 - A(\mathbf{x} \to \mathbf{x}')$.

We know that if detailed balance is satisfied, Equation 3.13, then MH defines a valid transition function and $\gamma(\mathbf{x})$ is stationary distribution. To show this we need to show that the transition kernel is ergodic and reversible. Consider two states $\mathbf{x}$ and $\mathbf{x}'$, either $\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) < \gamma(\mathbf{x}')q(\mathbf{x}'|\mathbf{x})$, or $\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) > \gamma(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')$. Without loss of generality we assume the latter, hence

$$A(\mathbf{x} \to \mathbf{x}') = \frac{\gamma(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} < 1 \text{ or } A(\mathbf{x}' \to \mathbf{x}) = 1 \tag{3.16}$$

In order to move from $\mathbf{x}$ to $\mathbf{x}'$ we first propose $\mathbf{x}'$ and then accept it. That is

$$\kappa(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}'|\mathbf{x})A(\mathbf{x} \to \mathbf{x}') = \frac{q(\mathbf{x}'|\mathbf{x})\gamma(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\gamma(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{\gamma(\mathbf{x}')}{\gamma(\mathbf{x})}q(\mathbf{x}|\mathbf{x}') \tag{3.17}$$

which means

$$\gamma(\mathbf{x})\kappa(\mathbf{x}'|\mathbf{x}) = \gamma(\mathbf{x})q(\mathbf{x}|\mathbf{x}') \tag{3.18}$$

The backwards probability is $\kappa(\mathbf{x}|\mathbf{x}') = q(\mathbf{x}|\mathbf{x}')A(\mathbf{x}' \to \mathbf{x}) = q(\mathbf{x}|\mathbf{x}')$ as $A(\mathbf{x}' \to \mathbf{x}) = 1$ and so putting this into Equation 3.18 we get

$$\gamma(\mathbf{x})\kappa(\mathbf{x}'|\mathbf{x}) = \gamma(\mathbf{x}')\kappa(\mathbf{x}|\mathbf{x}') \tag{3.19}$$

Thus, detailed balance holds with respect to $\gamma(\mathbf{x})$, so it is a stationary distribution and because detail balance holds, the chain is also ergodic and irreducible.

We shall leverage the MH algorithm when utilising a trace-based approach to performing inference over program traces in probabilistic programming systems, Subsection 3.3.2 and in Chapter 6 for performing inference in nested models, Subsection 3.2.2, once we have learnt proposals for the nested program.

3.1.6 Hamiltonian Monte Carlo

Of particular relevance to our work are probabilistic programming systems designed around derivative based inference methods that exploit automatic differentiation [Baydin et al., 2018], such as PyMC3 [Salvatier et al., 2016], Stan [Carpenter et al., 2017], Edward [Tran et al., 2017], Turing [Ge et al., 2018] and Pyro [Bingham et al., 2019]. Automatic differentiation, see Baydin et al. [2018], is a tool that enables us to differentiate programmatic code to any order for which a valid result is returned and enables the implementation of generalisable inference algorithms that require gradient information [Blei et al., 2017; Carpenter et al., 2017; Hoffman and Gelman, 2014; Le et al., 2017]. Derivative based inference algorithms, such as Hamiltonian Monte Carlo (HMC), have been an essential component in enabling these probabilistic programming systems to provide efficient and, in particular, scalable inference, permitting both high-dimensional observations and latent variables.

Hamiltonian Monte Carlo is a physics inspired inference algorithm that utilises Hamiltonian mechanics, where the Hamiltonian of a physical system is defined completely in terms of the set of points $(\mathbf{x}, \mathbf{p})$, where $\mathbf{x}$ is the position, and $\mathbf{p}$ is the momentum variable. These points span what is called the phase space, a manifold that enables us to see how the dynamical system evolves with respect to $\mathbf{x}$ and $\mathbf{p}$, and how it is constrained by the total energy within the system - this constraint on total energy is key to the scalability of HMC. We state that the total energy of the system is given by the Hamiltonian $H(\mathbf{x}, \mathbf{p}) = K(\mathbf{p}) + U(\mathbf{x})$, where $K(\mathbf{p})$ represents the kinetic energy and $U(\mathbf{x})$ is the potential energy. To solve problems involving Hamiltonian dynamics we must solve the following set of differential equations

$$\nabla_{\mathbf{p}} H = \dot{\mathbf{x}} \tag{3.20}$$

$$\nabla_{\mathbf{x}} H = -\dot{\mathbf{p}} \tag{3.21}$$

However, solving Hamiltons equations in practice is challenging and so we must resort to numerical integration schemes that satisfy three physical constraints of the Hamiltonian: time reversibility, invariance of the Hamiltonian and volume preservation. In order to use these schemes we must discretise Equations 3.20-3.21, which naturally induces errors. One such low-order error integrator that satisfies all of these properties is the Leapfrog integrator [Duane et al., 1987; Neal, 2011], which works by updating the momentum and position variables via gradient information from the Hamiltonian along a trajectory of length, L, that takes t steps to complete, with each step being ϵ in length

$$\mathbf{p}(t+\frac{\epsilon}{2}) = \mathbf{p}(t) - \left(\frac{\epsilon}{2}\right)\nabla_{\mathbf{x}}U(\mathbf{x}(t)) \tag{3.22}$$

$$\mathbf{x}(t+\epsilon) = \mathbf{x}(t) + \epsilon\nabla_{\mathbf{p}}K(\mathbf{p}(t+\frac{\epsilon}{2})) \tag{3.23}$$

$$\mathbf{p}(t+\epsilon) = \mathbf{p}(t+\frac{\epsilon}{2}) - \left(\frac{\epsilon}{2}\right)\nabla_{\mathbf{x}}U(\mathbf{x}(t+\epsilon)) \tag{3.24}$$

This generates a new proposed state and in order to decide whether we should accept or reject this proposal, we use the following acceptance criteria[Duane et al., 1987]

$$\min[1, \exp(-H(\mathbf{x}^{'}, \mathbf{p}^{'}) + H(\mathbf{x}, \mathbf{p})] = \min[1, \exp(-U(\mathbf{x}') + U(\mathbf{x}) - K(\mathbf{p}^{'}) + K(\mathbf{p}))] \tag{3.25}$$

where $(\mathbf{x}^{'}, \mathbf{p}^{'})$ is the proposed state and $(\mathbf{x}, \mathbf{p})$ is the current state. In order to simulate Hamiltonian dynamics correctly, for each $\mathbf{x}$ we must introduce an auxiliary momentum variable $\mathbf{p}$. The momentum is usually sampled from a normal distribution $\mathbf{p} \sim \mathcal{N}(0, \mathbb{I})$, which corresponds to a kinetic energy resembling the mean-field approximation $K(\mathbf{p}) = \frac{\mathbf{p}^T M^{-1}\mathbf{p}}{2}$, where M, the mass matrix, is a symmetric, positive definite and typically diagonal matrix, and the potential energy $U(\mathbf{x}) = -\log\gamma(\mathbf{x}) = -\log(p(\mathbf{x})p(\mathbf{y}|\mathbf{x}))$ represents the target density that we want to generate samples $\mathbf{x}$ from.

We present the HMC sampling algorithm in Algorithm 2. The parameters ϵ and L are parameters that need to be tuned. Theoretically speaking, in order to ensure that the proposal is symmetric, we should negate the momentum variables at the end of the trajectory, to ensure that the Metropolis proposal is symmetrical, which is needed for the acceptance probability to be valid. However, in practice we do not need to perform this negation since $K(\mathbf{p}) = K(-\mathbf{p})$ for the Gaussian momentum and after each iteration the momentum will be replaced before it is used again. Hence, we leave it out of Algorithm 2.

Hamiltonian Monte Carlo at Discontinuous Points

One important challenge for these gradient-based probabilistic programming systems occurs in probabilistic programs that contain discontinuous densities and/or variables. 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., 2020]. Though the leapfrog integrator remains a valid, reversible MCMC proposal, even

Algorithm 2 Hamiltonian Monte Carlo

```
1: procedure HMC($\mathbf{x}_0$, $\epsilon$, $L$, $U$, $N$)
2:     for $t = 1$ to $N$ do
3:         $\mathbf{p}_t \sim \mathcal{N}(0, 1)$
4:         $\mathbf{x}_t \to \mathbf{x}_{t-1}$
5:         for $i = 1$ to $L$ do
6:             $(\mathbf{x}_{i+1}, \mathbf{p}_{i+1}) \to$ LEAPFROG($\mathbf{x}_i, \mathbf{p}_i, \epsilon$)
7:         end for
8:         $\alpha = \min\{1, \exp(H(\mathbf{x}_L, \mathbf{p}_L)) - H(\mathbf{x}_t, \mathbf{p}_t)\}$
9:         $u \sim \text{Uniform}(0, 1)$
10:        if $u < \alpha$ then
11:            $\mathbf{x}_t \to \mathbf{x}_L$                                ▷ Accept
12:        else
13:            $\mathbf{x}_t \to \mathbf{x}_t$                                ▷ Reject
14:        end if
15:    end for
16:    return $(\mathbf{x}_1, \dots, \mathbf{x}_N)$
17: end procedure
18:
19: procedure LEAPFROG($\mathbf{x}$, $\mathbf{p}$, $\epsilon$)
20:    $\mathbf{p}' \to \mathbf{p} - \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x})$
21:    $\mathbf{x}' \to \mathbf{x} + \epsilon \nabla_{\mathbf{p}} K(\mathbf{p}')$
22:    $\mathbf{p}'' \to \mathbf{p}' - \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x}')$
23:    return ($\mathbf{x}'$, $\mathbf{p}''$)
24: end procedure
```

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 [Afshar and Domke, 2015; Nishimura et al., 2020].

However, these advanced methods are, in general, not incorporated in existing gradient-based probabilistic programming systems, as existing systems do not have adequate support to deal with the discontinuities in the density functions of the probabilistic model defined by the probabilistic program. This support 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, see Chapter 4 for more details. In addition to this, some further methods require knowledge of where the discontinuities are, or at least catching occurrences of discontinuity boundaries being crossed. The work presented in Chapter 4 addresses some of these issues.

3.2 Nested Monte Carlo

In many simulators nested structures arise, where an outer program, is dependent on one, or many nested inner programs, and the outer and inner programs are probabilistic in nature. Problems that have this nested structure are often called "doubly intractable", which is equivalent to one layer of nesting, see Subsection 3.2.1. Finding ways to perform exact, or approximate inference in doubly intractable integrals [Moller and Waagepetersen, 2003; Murray et al., 2006] and probabilistic models with nested structures [Stuhlmüller and Goodman, 2014] has seen increased interest in recent years [Lyne et al., 2015], as probabilistic models with such structures are used across the sciences. This includes theory of the mind experiments [Stuhlmüller and Goodman, 2014] where we reason about human behaviour, in financial simulators [Raberto et al., 2001] where traders reason about other traders market strategies, poker games [Rainforth et al., 2018], in epidemiology simulators [Smith et al., 2008] where we have rejection sampler programs that are dependent on the results of other probabilistic programs, and several areas of quantum physics and epidemiology [Bhanot and Kennedy, 1985; Green and Richardson, 2001; Lyne et al., 2015]. In these problems we can no longer rely on MCMC, or at least MCMC strategies that use a Metropolis-Hastings transition kernel. Adaptations do exist, but these impose a large number of constraints on the underlying model [Moller and Waagepetersen, 2003] and so cannot be used for applications of interest, as the normalisation constant of the likelihood is no longer constant, so it does not cancel in the MH proposal. To see this, we note that the form of the posterior in doubly intractable problems is given by

$$\pi(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})} = \frac{P^{in}(\mathbf{y}|\mathbf{x})}{Z(\mathbf{x})} \tag{3.26}$$

where $Z(\mathbf{x}) = \int P^{in}(\mathbf{y}|\mathbf{z})d\mathbf{z}$ is typically intractable and $P^{in}(\mathbf{y}|\mathbf{x})$ may or may not be dependent on x, $p(\mathbf{y})$ is also intractable, hence the name doubly intractable. If we try and construct the standard MH acceptance proposal, Equation 3.14, we have the following

$$A(\mathbf{x} \to \mathbf{x}') = \frac{\pi(\mathbf{x}'|\mathbf{y})q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x}|\mathbf{y})q(\mathbf{x}'|\mathbf{x})} = \frac{p(\mathbf{x})P^{in}(\mathbf{y}|\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p(\mathbf{x})P^{in}(\mathbf{y}|\mathbf{x})q(\mathbf{x}'|\mathbf{x})} \times \frac{Z(\mathbf{x})}{Z(\mathbf{x}')} \tag{3.27}$$

and because the normalisation terms $Z(\mathbf{x})$ no longer cancel, in addition to its intractability, means that we can not construct a valid transition kernel and cannot leverage any of the existing MCMC frameworks. To address these issues, in Chapter 6 we develop two new Bayesian inference schemes that not only scale in dimensionality, but also enable one to leverage MCMC inference in doubly intractable and multi-layered nesting probabilistic models.

One way to ensure convergence and generate expectations of nested models is through Nested Monte Carlo, which ensures that at each layer of nesting a sufficient number of samples are drawn to ensure overall convergence. This is important as each layer is using samples of an approximation, of an approximation, see [Rainforth et al., 2018] for more details.

3.2.1 Constructing a Nested Monte Carlo estimator

A Nested Monte Carlo (NMC) estimate, for D layers of nesting and real-valued functions $\eta_0 \ldots \eta_D$ can be recursively formalised as [Rainforth et al., 2018]:

$$I_D(\mathbf{x}^{0:D-1}) = \frac{1}{N_D} \sum_{n_D=1}^{N_D} \eta_D(\mathbf{x}^{0:D-1}, \mathbf{x}^D_{n_d}) \tag{3.28}$$

$$I_k(\mathbf{x}^{0:k}) = \frac{1}{N_k} \sum_{n_k=1}^{N_k} \eta_k(\mathbf{x}^{0:1}, \mathbf{x}^k_{n_k}, I_{k+1}(\mathbf{x}^{0:k}, \mathbf{x}^k_{n_k})) \tag{3.29}$$

for $0 \leq k < D$, where each $\mathbf{x}^k_n \sim p(\mathbf{x}^k | \mathbf{x}^{0:k})$ is drawn independently and integral I_0 is a NMC estimate using a total sample budget of $T = N_0 \ldots N_D$. When $D = 0$ we recover the standard Monte Carlo estimate, Equation 3.1, and for clarity, when we have a single level of nesting, the doubly intractable case, $\eta(\mathbf{x}^0) = \eta_0(\mathbf{x}^0, \mathbb{E}[\eta_1(\mathbf{x}^0, \mathbf{x}^1)|\mathbf{x}^0])$, thus

$$I_0 = \mathbb{E}[\eta_0(\mathbf{x}^0, \mathbb{E}[\eta_1(\mathbf{x}^0, \mathbf{x}^1)|\mathbf{x}^0)])] \approx \frac{1}{N_0} \sum_{n=1}^{N_0} \eta_0(\mathbf{x}^0_n, \frac{1}{N_1} \sum_{m=1}^{N_1} \eta_1(x^0_n, \mathbf{x}^1_{n,m})) \tag{3.30}$$

where each $\mathbf{x}^1_{n,m} \sim p(\mathbf{x}^1 | \mathbf{x}^0_n)$ is drawn independently and I_0 is an NMC estimate using a budget of $T = N_0 N_1$ samples.

Now that we have introduced NMC, we will explore how we can use this to perform inference in nested models.

3.2.2 Nested inference

Nested inference problems, which we will explore in more detail in Chapter 6, are unique as nested inference problems include terms with unknown, parameter dependent, marginal likelihoods.

To formally define nested inference problems we follow the logic of [Rainforth et al., 2018] and let $\mathbf{y}$ and $\mathbf{x}$ denote all the random variables of an outer program that are respectively passed or not passed to the inner program, let $\mathbf{z}$ denote all the random variables in the inner program, and assume without loss of generality that these are all returned to the outer program. Under these assumptions, the unnormalised density for the outer program can be written as

$$\gamma^{out}(\mathbf{x},\mathbf{y},\mathbf{z}) = \psi(\mathbf{x},\mathbf{y},\mathbf{z})P^{in}(\mathbf{z}|\mathbf{y}) \tag{3.31}$$

where $P^{in}(\mathbf{z}|\mathbf{y})$ is the normalised density of the outputs of the inner program and $\psi(\mathbf{x},\mathbf{y},\mathbf{z})$ encapsulates all other terms influencing the trace probability of the outer program and the inner program defines an unnormalised density $\gamma^{in}(\mathbf{y},\mathbf{z})$ that can be evaluated pointwise and so

$$P^{in}(\mathbf{z}|\mathbf{y}) = \frac{\gamma^{in}(\mathbf{y},\mathbf{z})}{\int \gamma^{in}(\mathbf{y},\mathbf{z}')d\mathbf{z}'} \quad \text{leads to} \tag{3.32}$$

$$\pi^{out}(\mathbf{x},\mathbf{y},\mathbf{z}) \propto \gamma^{out}(\mathbf{x},\mathbf{y},\mathbf{z}) = \frac{\psi(\mathbf{x},\mathbf{y},\mathbf{z})\gamma^{in}(\mathbf{y},\mathbf{z})}{\int \gamma^{in}(\mathbf{y},\mathbf{z}')d\mathbf{z}'} \tag{3.33}$$

where $\pi^{out}(\mathbf{x},\mathbf{y},\mathbf{z})$ is the posterior distribution. The denominator here is intractable due to the dependence on $\mathbf{y}$ and $\mathbf{z}$, and we must evaluate it separately for each value of $\mathbf{y}$.

To perform inference in nested settings we can utilise self-normalised importance sampling. Again, following closely the work of Rainforth et al. [2018], given a proposal $q(\mathbf{x},\mathbf{y},\mathbf{z}) = q(\mathbf{x},\mathbf{y})q(\mathbf{z}|\mathbf{y})$ we can calculate the expectation of some function $\eta(\mathbf{x},\mathbf{y},\mathbf{z})$ under $\pi^{out}(\mathbf{x},\mathbf{y},\mathbf{z})$ as

$$\begin{aligned}\mathbb{E}_{\pi^{out}(\mathbf{x},\mathbf{y},\mathbf{z})}\left[\eta(\mathbf{x},\mathbf{y},\mathbf{z})\right] &= \frac{\mathbb{E}_{q(\mathbf{x},\mathbf{y},\mathbf{z})}\left[\frac{\eta(\mathbf{x},\mathbf{y},\mathbf{z})\gamma^{out}(\mathbf{x},\mathbf{y},\mathbf{z})}{q(\mathbf{x},\mathbf{y},\mathbf{z})}\right]}{\mathbb{E}_{q(\mathbf{x},\mathbf{y},\mathbf{z})}\left[\frac{\gamma^{out}(\mathbf{x},\mathbf{y},\mathbf{z})}{q(\mathbf{x},\mathbf{y},\mathbf{z})}\right]} \\ &= \frac{\mathbb{E}_{q(\mathbf{x},\mathbf{y},\mathbf{z})}\left[\frac{\eta(\mathbf{x},\mathbf{y},\mathbf{z})\psi(\mathbf{x},\mathbf{y},\mathbf{z})\gamma^{in}(\mathbf{y},\mathbf{z})}{q(\mathbf{x},\mathbf{y},\mathbf{z})\mathbb{E}_{\mathbf{z}'\sim q(\mathbf{z}|\mathbf{y})}\left[\gamma^{in}(\mathbf{y},\mathbf{z}')/q(\mathbf{z}'|\mathbf{y})\right]}\right]}{\mathbb{E}_{q(\mathbf{x},\mathbf{y},\mathbf{z})}\left[\frac{\psi(\mathbf{x},\mathbf{y},\mathbf{z})\gamma^{in}(\mathbf{y},\mathbf{z})}{q(\mathbf{x},\mathbf{y},\mathbf{z})\mathbb{E}_{\mathbf{z}'\sim q(\mathbf{z}|\mathbf{y})}\left[\gamma^{in}(\mathbf{y},\mathbf{z}')/q(\mathbf{z}'|\mathbf{y})\right]}\right]}\end{aligned} \tag{3.34}$$

As stated in Subsection 3.1.3, $\eta(\mathbf{x},\mathbf{y},\mathbf{z})$ is not known upfront so we instead return weighted samples that can be used to estimate the integral of interest, where the the unnormalised importance weights are defined as

$$\omega_{j,k} = \frac{\psi(\mathbf{x}_j,\mathbf{y}_j,\mathbf{z}_{j,k})\gamma^{in}(\mathbf{y}_j,\mathbf{z}_{j,k})}{q(\mathbf{x}_j,\mathbf{y}_j,\mathbf{z}_{j,k})\frac{1}{N_1}\sum_{\ell=1}^{N_1}\frac{\gamma^{in}(\mathbf{y}_j,\mathbf{z}_{j,\ell})}{q(\mathbf{z}_{j,\ell}|\mathbf{y}_j)}}. \tag{3.35}$$

where we sample $(\mathbf{x}_j,\mathbf{y}_j) \sim q(\mathbf{x},\mathbf{y})$ and $\mathbf{z}_{j,k} \sim q(\mathbf{z}|\mathbf{y}_j)$ and return all samples $(\mathbf{x}_j,\mathbf{y}_j,\mathbf{z}_{j,k})$, then each $(\mathbf{x}_j,\mathbf{y}_j)$ is duplicated N_1 times in the set of samples. We can then use these importance weights to approximate the posterior as weighted sum of delta distributions, as we did in Equation 3.8,

$$\pi_{out}(\mathbf{x},\mathbf{y},\mathbf{z}) \approx \frac{\sum_{j=1}^{N_0}\sum_{k=1}^{N_1}\omega_{j,k}\delta_{(\mathbf{x}_j,\mathbf{y}_j,\mathbf{z}_{j,k})}(\cdot)}{\sum_{j=1}^{N_0}\sum_{k=1}^{N_1}\omega_{j,k}} \tag{3.36}$$

Although nested self-normalised IS is efficient in low-dimensional latent spaces, as the dimension of the latent space increases the efficiency decreases substantionally [Gram-Hansen et al., 2019b]. In Chapter 6, we develop two Bayesian inference algorithms to make inference in nested probabilistic models more computationally efficient in nested models with high-dimensional latent spaces by enabling the use of efficient MCMC based samplers in nested models.

3.3 Inference in probabilistic programming systems

We have now introduced some fundamental inference algorithms based on the principles of Monte Carlo, but how do we connect the inference back-end to the probabilistic program? In Chapters 1 and 2, we explored the different types of PPLs and saw that we can construct a graphical model of the probabilistic program by using special constructs, Section 2.2, and we can then use that to perform inference in simulators that are probabilistic programs, Section 2.4. To perform inference we take those special constructs and use them to generate a joint density that represents a probabilistic program, $\mathcal{P}$, which is then evaluated by the inference back-end in the probabilistic programming system.

The choice of inference back-ends will determine the type of information that needs to be extracted from the probabilistic program in order to run inference. For some inference back-ends, such as HMC, we do not require a trace, only addresses that are unique to `sample` statements defined in the program, and the ability to differentiate the specified target distribution - which is possible in languages that have automatic differentiation libraries [Bingham et al., 2019; Carpenter et al., 2017]. Other inference back-ends that are traced-based, will require access to the program traces, addresses, and instance counters, to keep track of how many times we have visited a given construct.

In the proceeding subsections we will describe how we define the density of a probabilistic program, and introduce two general-purpose inference backends that perform inference, not always efficiently, for any simulator that can be compiled in a UPPL, one being Lightweight Metropolis Hastings [Wingate et al., 2011] that is based on the MH algorithm, Subsection 3.1.5, and the other, amortized importance sampling (AIS) [Le et al., 2017] based on importance sampling 3.1.2. Both methods work directly with the trace and need the address stack to track changes in the underlying random variables, with AIS, sometimes referred to as inference compilation, requiring the underlying system to generate gradients, but not directly from the program code.

3.3.1 Probability over program traces

For many simulators it is not possible to explicitly determine the joint density, so how do we perform Bayesian inference for a general probabilistic program? Rather than constructing the explicit joint density of the probabilistic model, we can instead perform Bayesian inference over the set of possible executions of the probabilistic program.

For a general probabilistic program, $\mathcal{P}$, the program trace, Subsection 2.3.1, also called the execution trace, is a `sample` sequence of the forward evaluation of a probabilistic program. Using the trace, we can express the probabilistic program as a sequence of samples and addresses $(x_j, a_j)_{j=1:n_x}$, for the jth entry in a given trace of length n_x. Sometimes we will also require an instance counter $(i_j)_{j=1:n_x}$, which counts the number of `sample` statements with the same address as the current address, a_j, encountered during the forward evaluation, that is $i_j = \sum_{l=1}^{j} \mathbb{I}[a_j = a_l]$. Thus, when we execute $\mathcal{P}$, it directly describes a joint probability distribution between the latent variables x_j and the observations y_k, for which we encounter n_x `sample` statements and n_y `observe` statements. This leads to the formation of sets $(f_{a_i}, \phi_j)_{j=1:n_x}$ for `sample` statements and $(g_{b_k}, \psi_k)_{k=1:n_y}$ for `observe` statements, and so the probability of a program execution, provided the program trace is valid, which means all `sample` statements in that trace have been evaluated and that all generated distribution objects are valid, is defined as

$$\pi(\mathbf{x}) \propto \gamma(\mathbf{x}) = \prod_{j=1}^{n_x} f_{a_j}(x_j|\phi_j) \prod_{k=1}^{n_y} g_{b_k}(y_k|\psi_k) \tag{3.37}$$

Here, all terms—i.e., x_j, n_x, a_j, ϕ_j, n_y, b_k, y_k, and ψ_k—may be random variables, but each is deterministically calculable from the trace $x_{1:n_x}$ (see, e.g., Rainforth [2018a, Section 4.3.2]), and γ represents the unnormalised program density. Only in FOPPLs is the program density defined in Equation 3.37 equivalent to the joint-density $\gamma(\mathbf{x}) = p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$, as FOPPL programs are by construction directed acyclic graphical models [Van de Meent et al., 2018] and so may be composed in terms of prior terms, $p(x_{1:n_x}) = \prod_{j=1}^{n_x} f_{a_j}(x_j|\phi_j)$ and likelihood terms $p(x_{1:n_x}|y_{1:n_y}) = \prod_{k=1}^{n_y} g_{b_k}(y_k|\psi_k)$. However, in an UPPL, $g_{b_k}(\cdot)$ is not necessarily a likelihood, see Section 2.3 and so does not define the standard joint-density. In the same way we calculated the normalised target distribution previously, we can also define the posterior program as

$$\pi(\mathbf{x}) = \frac{\gamma(\mathbf{x})}{Z} \tag{3.38}$$

where Z is the normalisation constant and can be calculated by integrating over all possible program execution traces

$$Z = \int \gamma(\mathbf{x}) d\mathbf{x} \tag{3.39}$$

As we discussed previously, the normalisation factor, the marginal likelihood, over program execution traces is typically intractable, and so we must revert to approximate inference schemes, such as MCMC and importance sampling to calculate it. In the same vien as we described previously, we can also calculate expectations for a probabilistic program, by taking the program output, which is a deterministic function, $\eta(\mathbf{x})$, given the sampled values $\mathbf{x}$ in the execution trace. Thus, we can use the posterior distribution over the execution traces $\pi(\mathbf{x})$ to characterise the distribution over program outputs given the observations $\mathbf{y}$ to calculate the posterior expectation

$$\mathbb{E}[\eta(\mathbf{x})] = \int \eta(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \frac{1}{Z}\int \eta(\mathbf{x})\gamma(\mathbf{x})dx \tag{3.40}$$

As this is characterised over the outputs of the probabilistic program, it means that when we run the probabilistic program forward we are directly defining a joint-density. Many sampling-based inference algorithms used for probabilistic programming require uniquely identifying the elements of a sequence of samples, and so by constructing the program density of a probabilistic program via the execution traces, we can now explore two purpose-built probabilistic programming inference schemes, Light-weight Metropolis-Hastings [Wingate et al., 2011] and amortized importance sampling [Le et al., 2017].

3.3.2 A trace-based inference scheme

Lightweight Metropolis-Hastings (LMH) [Wingate et al., 2011], is a general-purpose trace-based MCMC inference algorithm that has been designed around execution traces of probabilistic programs, where a single random variable drawn in the course of a particular execution of a probabilistic program, which is modified via a standard MH proposal, is accepted depending on the probability generated when comparing the values of the joint program density evaluated at the old and proposed program trace. LMH differs from component-wise MH algorithms in that other random variables may also have to be modified, depending on the structural dependencies in the probabilistic program.

We initialise LMH by running a program, $\mathcal{P}$, forward once to generate an initial trace $\mathbf{x}^1$ of length n_{x^1}. We then run the program forward a large number of times to generate multiple traces. Given a trace $\mathbf{x}^l$ of length n_{x^l}, we define a reversible proposal $q(\mathbf{x}'|\mathbf{x}^l)$ for sampling a new candidate trace and begin by picking a single random choice x^l_m from the trace $\mathbf{x}^l$ that is of length n_{x^l}, by drawing m uniformly from the set of integers $\{1, \ldots, n_{x^l}\}$. Then, we apply a reversible proposal $q_m(x'_m|x^l_m)$ to propose a new value at that specific random choice. Keeping all **sample** variables $m+1$ onwards fixed, we re-run the remainder of the program to generate a new proposal trace $\mathbf{x}'$ of length $n_{x'}$. This leads to the probability of acceptance for a proposed trace $\mathbf{x}'$ as

$$A(\mathbf{x} \rightarrow \mathbf{x}') = min(1, \frac{\gamma(x'_{1:n_{x'}}) n_{x^l} q_m(x^l_m|x'_m) \prod_{j=m+1}^{n_{x^l}} f^l_{a_j}(x^l_j|\phi^l_j)}{\gamma(x^l_{1:n_{x^l}}) n_{x'} q_m(x'_m|x^l_m) \prod_{j=m+1}^{n_{x'}} f'_{a_j}(x'_j|\phi'_j)}) \tag{3.41}$$

In our implementation of LMH used in Chapter 5 the proposal is generated by forward simulation of the prior $f_{a_m}(x_m|\phi_m)$, which leads to the acceptance probability

$$A(\mathbf{x} \rightarrow \mathbf{x}') = min(1, \frac{\gamma(x'_{1:n_{x'}}) n_{x^l} \prod_{j=m}^{n_{x^l}} f^l_{a_j}(x_j|\phi_j)}{\gamma(x^l_{1:n^l_x}) n_{x'} \prod_{j=m}^{n_{x'}} f'_{a_j}(x'_j|\phi'_j)}) \tag{3.42}$$

For models with significant prior-posterior mismatch, such an approach could be slow to mix as it does not allow for any locality in the moves. To improve the efficiency of LMH Le [2015] introduced Random-walk MH (RMH), which uses the distribution type of the object associated with x_m to automatically construct a valid random walk proposal that allows for improved exploration on the individual updates.

3.3.3 Amortized importance sampling

Once a simulator is expressed as a probabilistic program $\mathcal{P}$, that is a joint program density $\gamma(\mathbf{x}) = p(\mathbf{x}, \mathbf{y})$ is defined, we are interested in performing inference in order to get posterior distributions $\pi(\mathbf{x})$ of the latent variables $\mathbf{x}$ conditioned on observed variables $\mathbf{y}$. However, performing inference in complex models is typically slow and must be re-run for every new set of observations, which is natural bottleneck when performing inference in probabilistic programs. Moreover, many inference problems have shared structure, and we should be able to leverage that shared structure when calculating the posterior under a different set of observations. To exploit this structure we can amortize the cost of inference by learning a family of proposals, offline, for a range of observations, which is expensive, but in return makes online inference computationally cheap and efficient, as the proposal should better match the posterior [Cappé et al., 2008]. In the probabilistic programming paradigm, because we have direct access to the underlying simulator, it is computationally cheap to run $\mathcal{P}$ forward to generate a database of observations, which can then be used to learn a general family of proposals $q_{a_j}(\cdot)$ that can be utilised during online-inference for any given set of observations $\mathbf{y}$.

Amortized importance sampling (AIS) is an efficient importance sampling-based method [Le et al., 2017] that leverages amortization of the inference problem when we have direct access to $\mathcal{P}$. Rather than manually selecting a proposal distribution, we can instead leverage the learned family of parameterised proposal distributions characterised by both simple and sophisticated Neural Network (NN) architectures that match the underlying posterior density. Thus, in theory, AIS should reduce the proportion of importance weights equal to zero as we generate proposals that are similar to the posterior, which makes AIS more sample efficient, when compared to standard and self-normalised importance sampling.

Formally speaking, in Subsection 3.1.3 we saw that we can take a set of weighted samples $\{(\hat{\mathbf{x}}_n, \omega_n)_{n=1:N}\}$ and construct an empirical approximation of the posterior distribution $\hat{\pi}(\mathbf{x}) = \sum_{n=1}^{N} \hat{\omega}_n \delta(\mathbf{x}_n - \mathbf{x})$, where δ is the Dirac delta function and $\hat{\omega}_n$ are the normalised weights. The importance weight for each execution trace of $\mathcal{P}$ is

$$w_n = \prod_{k=1}^{n_y} g_{b_k}(y_k | x_{1:n_{x'}}) \prod_{j=1}^{n_x} \frac{f_{a_j}(x_j | x_{1:j-1})}{q_{a_j, i_j}(x_j | x_{1:j-1})} \ , \tag{3.43}$$

where $q_{a_j,i_j}(\cdot | x_{1:j-1})$ is known as the proposal distribution and may be identical to the prior f_{a_j}, as in regular importance sampling. The proposal distributions in AIS are learnt using a recurrent NN, $\zeta(\cdot)$, that receives the observed values $\mathbf{y}$ and returns a set of adapted proposals $q_{a_j,i_j}(x_j | x_{1:j-1}, \zeta(\kappa, \mathbf{y}))$ such that the joint proposal distribution $q(\mathbf{x} | \zeta(\kappa, \mathbf{y}))$, which is factored into the product of proposals $q_{j,i_j}(\cdot)$ whose type depends on the type of the prior f_{a_j}, is close to the true posterior $\pi(\mathbf{x})$. As we need to choose parameters κ that perform well across all observations $\mathbf{y}$, we construct an objective function $\mathcal{L}(\kappa)$ by minimising the expected value of the Kullback–Leibler divergence from *p-to-q* under $p(\mathbf{y})$, that is

$$\mathcal{L}(\zeta) := \mathbb{E}_{p(\mathbf{y})} \left[D_{\mathrm{KL}} \left(\pi(\mathbf{x}) \ || \ q(\mathbf{x} | \zeta(\kappa, \mathbf{y})) \right) \right] \tag{3.44}$$

$$= \int p(\mathbf{y}) \int \log \frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\zeta(\kappa, \mathbf{y}))} d\mathbf{x} d\mathbf{y} = \mathbb{E}_{\gamma(\mathbf{x})} \left[-\log q(\mathbf{x} | \zeta(\kappa, \mathbf{y})) \right] + \text{const.} \tag{3.45}$$

which has gradient

$$\nabla_{\zeta} \mathcal{L}(\zeta) = \mathbb{E}_{\gamma(\mathbf{x})} \left[-\log q(\mathbf{x} | \zeta(\kappa, \mathbf{y})) \right] \tag{3.46}$$

where $\zeta(\kappa, \mathbf{y})$ is the NN and the parameters κ are the weights of the NN, which are optimised to minimise this objective by continually drawing training pairs $(\mathbf{x}, \mathbf{y}) \sim \mathcal{P}$ from the probabilistic program. As with all inference schemes there are limitations to the AIS approach. In AIS training, we may designate a subset of all addresses (a_j, i_j) to be "controlled" (learned) by the NN, leaving all remaining addresses to use the prior f_{a_j} as proposal during inference. Expressed in simple terms, taking an observation $\mathbf{y}$ as input, the NN learns to control the random number draws of latents $\mathbf{x}$ during the programs execution in such a way that makes the observed outcome likely. First, if your simulator is not well-definied then the output of the program will lead a family of proposal distrubutions that are not representiative of the true posterior. By utilising real-world simulators we can be confident in the model structure, due to the intensive research phase that goes to constructing such models, limiting the impact from ill-defined programs. Second, AIS does not handle nested programs, which is problematic for a large class of real-world simulators.

In Chapters 5 we use AIS 3.3.3, to learn a family of proposal distribution to make computing the importance weights more sample and computationally efficient, and in Chapter 6 we use amortized inference to learn surrogate densities so that can we can replace nested inference procedures and utilise efficient MCMC algorithm to perform inference in nested simulators.

3.4 Challenges in inference

As stated previously in Subsection 2.4.1, constructing inference algorithms for arbitrary probabilistic models is NP-hard. Furthermore, as the dimensionality of the observations increases, and/or, the dimensionality of the latent space, additional complexities arise, due to sample efficiency. Probabilistic programs add an additional component to this complexity, as no longer are we dealing with functions explicitly, but instead we are working with stochastic traces, which can change in structure during each execution. This is further complicated by having to perform inference in models of arbitrary complexity, as in the case of UPPLs. Thus, it is more sensible to focus on making inference procedures in probabilistic programming system more sample efficient and computationally faster, in generalisable ways. The work in Chapters 4, 5 and 6 focuses on finding solutions to such challenges by exploring how we can extend language semantics to exploit program structure in real-world simulators for efficient inferences, and how we can transform programs into representations that make them amenable to efficient inference schemes, even if parts of the underlying simulator have to be approximated.

4
Extending Constrained-by-Inference Probabilistic Programming Systems

LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models

Yuan Zhou[*,1] **Bradley J. Gram-Hansen**[*,1] **Tobias Kohn**[2,†] **Tom Rainforth**[1] **Hongseok Yang**[3] **Frank Wood**[4]

[1]University of Oxford [2]University of Cambridge [3]KAIST [4]University of British Columbia

Abstract

We develop a new Low-level, First-order Probabilistic Programming Language (LF-PPL) suited for models containing a mix of continuous, discrete, and/or piecewise-continuous variables. 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.

1 Introduction

Non-differentiable densities arise in a huge variety of common probabilistic models [1, 2]. Often, but not exclusively, they occur due to the presence of discrete variables. In the context of probabilistic programming [3, 4, 5, 6] 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 [7, 8, 9], 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 [7, 8, 9, 10, 11]. 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 [7, 8, 9, 10, 11]. Of particular note, are stochastic variational inference

[*]Equal contribution, [†]Work completed while at Oxford.

Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89. Copyright 2019 by the author(s).

(SVI) [12, 13, 14, 15] and Hamiltonian Monte Carlo (HMC) [16, 17], two of the most widely used approaches for probabilistic programming inference.

In the context of the former, [9] 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 [7, 8, 18, 19, 20]. 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) [8] as an inference engine in LF-PPL, allowing for efficient, automated, HMC-based inference in models with a mixture of continuous and discontinuous variables.

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 [21], such as Stan [6], PyMC3 [22], Edward [23], Turing [24] and Pyro [25]. 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 [7, 8]. 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 [9, 26].

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 [27] and, in particular, the first order PPL (FOPPL) explored in [28]. 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.

3 The Language

LF-PPL adopts a Lisp-like syntax, like that of Church [4] and Anglican [5]. 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 1 shows an example of an LF-PPL program.

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

Figure 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.

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 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 [28], 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 [29, §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:

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

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. For implementation and code, see `https://github.com/bradleygramhansen/PyLFPPL`.

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 [28].

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 5.2. 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 5.2.

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 5.1), 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 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.

5.1 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 1. *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.*

The proof is given in Appendix A. 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 [8]. As we shall prove in Section 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.

5.2 Translation Rules

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 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. $D = \{(\eta_1, k_1), \dots, (\eta_{N_D}, k_{N_D})\}$, 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 $F = \{(\zeta_1, l_1, v_1), \dots, (\zeta_{N_F}, l_{N_F}, v_{N_F})\}$, 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.

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

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

which by Theorem 2 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 $\Phi^{(d)} = \{(\psi_1, \phi_1), \dots, (\psi_{N_\Phi}, \phi_{N_\Phi})\}$ where N_Φ is the number of the partitions, ψ denotes the product of indicator functions indicating the partition of the space 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 $x_1, \dots, x_s$ of the distribution d and a given `sample` value x_0, we let $\mathbf{x} = (x_0, \dots, x_s)$ and have $\Phi^{(d)}$ to be the following set:

$$\Phi^{(d)} := \Big\{ \big(\psi_n(\mathbf{x}),\ \phi_n(\mathbf{x})\big) \Big|\ 1 \leq n \leq N_\Phi, \\ \psi_n(\mathbf{x}) := \prod_{j=1}^{M_i} \mathbb{1}[p_{n,j}(\mathbf{x}) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{n,l}(\mathbf{x}) < 0] \Big\}.$$

The probability density function defined by d is,

$$\mathcal{P}_d(x_0; x_1, \dots, x_s) = \sum\nolimits_{n=1}^{N_\Phi} \psi_n(\mathbf{x}) \cdot \phi_n(\mathbf{x}) \tag{2}$$

For example, given 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

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

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.

5.2.2 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 $f(x_1, \dots, x_n)$ and $f'(x_1, \dots, x_n)$ on real-valued inputs, we write $f[x_i := f']$ to denote the composition $f(x_1, \dots, x_{i-1}, f'(x_1, \dots, x_n), x_{i+1}, \dots, x_n)$. 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 D_is 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.

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

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.

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

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.

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

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.

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

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.*

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

Theorem 2. *If e is an expression that does not contain any free variables and $e \leadsto (\Delta, \Gamma,\, D,\, F)$, then the unnormalized density defined by e is in the form of Equation 1. 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.*

The proof is provided in Appendix B. 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 1, 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 [8].

5.3 A Compilation Example

We now present a simple example of how the compiler transforms the program e_{pp} in Figure 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 $e_{1,out}$:= `(sample (uniform 0 1))` from i, with each of its components evaluated first. 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* (`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 (`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 (`observe` $(d\ e_1\ e_2)\ c$). Take e_2 := `(observe (normal 1 1) y)` as an example, 1 is constant and d is the `normal` distribution and has $\Phi^{(d)} = \{(1, \mathcal{N}(\cdot\,; 1, 1))\}$. We combine each set and have $e_2 \rightsquigarrow \big(\emptyset, \emptyset, \{(1,1)\}, \{(1, \mathcal{N}(y; 1, 1), 0)\}\big)$. Similarly, $e_3 \rightsquigarrow (\emptyset, \emptyset, \{(1,1)\}, \{(1, \mathcal{N}(y; 0, 1), 0)\})$.

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

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 4

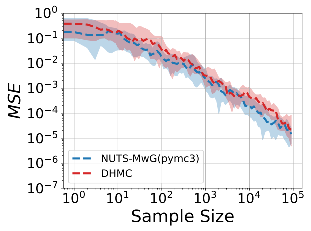


Figure 2: Mean Squared Error for the posterior estimates of the true posterior of the cluster means $\mu_{1:2}$. We compare the results from our unoptimized DHMC and the optimized PyMC3 NUTS with Metropolis-within-Gibbs, and show that the performance between the two is comparable for the same computation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

6 Example Inference Engine: DHMC

We shall now demonstrate an example inference algorithm that is compatible with LF-PPL. Specifically, we provide an implementation of discontinuous HMC (DHMC)[8], 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 C for further details. on DHMC theory and for the generalized DHMC for a PPS algorithm.

6.1 Gaussian Mixture Model (GMM)

In our first example, we demonstrate how a classic model, namely a Gaussian mixture model, can be encoded in LF-PPL. The density of the GMM contains a

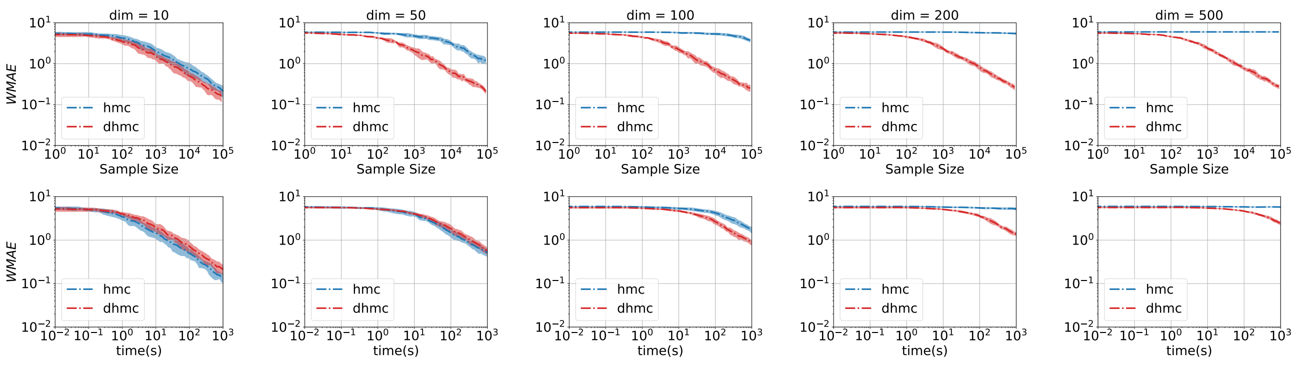


Figure 3: We compare DHMC against HMC on the worst mean absolute error (dashed lines) with the 20%/80% confidence intervals (shaded regions) over 20 independent runs for dimensions $D = 10, 50, 100, 200, 500$ (left to right). We demonstrate how the sample efficiency decreases with respect to sample size (*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.

mixture of continuous and discrete variables, where the discrete variables lead to discontinuities in the density. We construct the GMM as follows:

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

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 D. For our experiments, we considered a simple case with $\mu_0 = 0$, $\sigma_0 = 2$, $\sigma_{z_{1:N}} = 1$ and $p_0 = [0.5, 0.5]$, 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 [22], 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 2 as a function of the number of samples.

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[7] 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 D.

The results in Figure 3 provide a comparison between the DHMC and the standard HMC on the worst mean absolute error [7] as a function of the number of iterations and time, $\text{WMAE}(N) = \frac{1}{N} \max_{d=1,\ldots,D} \left|\sum_{n=1}^{N} \boldsymbol{x}_d^{(n)}\right|$. 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 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.

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.

Acknowledgements

Yuan Zhou is sponsored by China Scholarship Council (CSC). Bradley Gram-Hansen is supported by the UK EPSRC CDT in Autonomous Intelligent Machines and Systems (CDT in AIMS). Tom Rainforth's research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) ERC grant agreement no. 617071. Yang was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177). Kohn and Wood were supported by DARPA D3M, Intel as part of the NERSC Big Data Center, and NSERC under its Discovery grant and accelerator programs.

References

[1] H. Mohasel Afshar *et al.*, "Probabilistic inference in piecewise graphical models," 2016.

[2] A. Gelman, H. S. Stern, J. B. Carlin, D. B. Dunson, A. Vehtari, and D. B. Rubin, *Bayesian data analysis.* Chapman and Hall/CRC, 2013.

[3] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the on Future of Software Engineering*, pp. 167–181, ACM, 2014.

[4] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A Language for Generative Models," in *In UAI*, pp. 220–229, 2008.

[5] F. Wood, J. W. Meent, and V. Mansinghka, "A New Approach to Probabilistic Programming Inference," in *Artificial Intelligence and Statistics*, 2014.

[6] A. Gelman, D. Lee, and J. Guo, "Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization," *Journal of Educational and Behavioral Statistics*, vol. 40, no. 5, pp. 530–543, 2015.

[7] H. M. Afshar and J. Domke, "Reflection, Refraction, and Hamiltonian Monte Carlo," in *Advances in Neural Information Processing Systems*, pp. 3007–3015, 2015.

[8] A. Nishimura, D. Dunson, and J. Lu, "Discontinuous Hamiltonian Monte Carlo for Sampling Discrete Parameters," *arXiv preprint arXiv:1705.08510*, 2017.

[9] W. Lee, H. Yu, and H. Yang, "Reparameterization gradient for non-differentiable models," in *NIPS*, 2018.

[10] V. Dinh, A. Bilge, C. Zhang, I. Matsen, and A. Frederick, "Probabilistic path hamiltonian monte carlo," *arXiv preprint arXiv:1702.07814*, 2017.

[11] K. Yi and F. Doshi-Velez, "Roll-back hamiltonian monte carlo," *arXiv preprint arXiv:1709.02855*, 2017.

[12] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1303–1347, 2013.

[13] R. Ranganath, S. Gerrish, and D. Blei, "Black box variational inference," in *Artificial Intelligence and Statistics*, pp. 814–822, 2014.

[14] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[15] A. Kucukelbir, R. Ranganath, A. Gelman, and D. Blei, "Automatic variational inference in stan," in *Advances in neural information processing systems*, pp. 568–576, 2015.

[16] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Physics letters B*, 1987.

[17] R. M. Neal, "MCMC Using Hamiltonian dynamics," *Handbook of Markov Chain Monte Carlo*, 2011.

[18] Y. Zhang, Z. Ghahramani, A. J. Storkey, and C. A. Sutton, "Continuous relaxations for discrete hamiltonian monte carlo," in *Advances in Neural Information Processing Systems*, pp. 3194–3202, 2012.

[19] A. Pakman and L. Paninski, "Auxiliary-variable exact hamiltonian monte carlo samplers for binary distributions," in *Advances in neural information processing systems*, pp. 2490–2498, 2013.

[20] A. Pakman and L. Paninski, "Exact hamiltonian monte carlo for truncated multivariate gaussians," *Journal of Computational and Graphical Statistics*, vol. 23, no. 2, pp. 518–542, 2014.

[21] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey.,” *Journal of machine learning research*, vol. 18, no. 153, pp. 1–153.

[22] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, “Probabilistic Programming in Python Using PyMC3,” *PeerJ Computer Science*, vol. 2, p. e55, 2016.

[23] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei, “Deep probabilistic programming,” *arXiv preprint arXiv:1701.03757*, 2017.

[24] H. Ge, K. Xu, and Z. 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*, pp. 1682–1690, 2018.

[25] UberLabs, “Pyro, a universal probabilistic programming language.” `https://github.com/uber/pyro`, 2017.

[26] C. J. Maddison, A. Mnih, and Y. W. Teh, “The concrete distribution: A continuous relaxation of discrete random variables,” *arXiv preprint arXiv:1611.00712*, 2016.

[27] D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks, “BUGS 0.5: Bayesian Inference Using Gibbs Sampling Manual (version ii),” *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pp. 1–59, 1996.

[28] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood, “An introduction to probabilistic programming,” *arXiv preprint arXiv:1809.10756*, 2018.

[29] T. Rainforth, *Automating Inference, Learning, and Design using Probabilistic Programming.* PhD thesis.

[30] B. Mityagin, “The Zero Set of a Real Analytic Function,” *arXiv preprint arXiv:1512.07276*, 2015.

A Proof of Theorem 1

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

$$\mathcal{P}(x) = \sum_{i=1}^{N} \prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0] \cdot h_i(x) \tag{3}$$

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 [30]. 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 (3). 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

$$R_i = \{x \mid p_{i,j} \geq 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l\} \tag{4}$$

to

$$\begin{aligned} R_i' &= \{x \mid p_{i,j} > 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l\} \\ R_i'' &= R_i \setminus R_i'. \end{aligned} \tag{5}$$

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. □

B Proof of Theorem 2

Proof. As shown in Equation 1,

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

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 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 $\Big(\sum_{i=1}^{N_D} \eta_i \cdot k_i \Big)$ and $\Big(\sum_{j=1}^{N_F} \zeta_j \cdot l_j \Big)$ are non-negative functions on variables in 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 $\Big(\sum_{i=1}^{N_D} \eta_i \cdot k_i \Big) \cdot \Big(\sum_{j=1}^{N_F} \zeta_j \cdot l_j \Big)$ is a function on variables in 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. □

C Discontinuous Hamiltonian Monte Carlo

The discontinuous HMC (DHMC) algorithm was proposed by [8]. It uses a coordinate-wise integrator, Algorithm 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 [7], 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 exactly, even if the density is discontinuous. See Lemma 1 and Theorems 2-3 in [8]. This yields a rejection-free proposal.

Algorithm 1 Coordinate-wise Integrator. A random permutation ϕ on $\{1, \dots, B\}$ is appropriate if the induced random sequences $(\phi(1), \dots, \phi(|B|))$ and $(\phi(|B|), \dots, \phi(1))$ have the same distribution

```
 1: function CoordinateWise($\mathbf{x}, \mathbf{p}, \epsilon, U$)
 2:     pick an appropriate random permutation $\phi$ on $B$
 3:     for $i = 1, \dots, B$ do
 4:         $b \leftarrow \phi(i)$
 5:         $\mathbf{x}^* \leftarrow \mathbf{x}$
 6:         $\mathbf{x}_b^* \leftarrow \mathbf{x}_b^* + \epsilon m_b \cdot sign(\mathbf{p}_b)$
 7:         $\Delta U \leftarrow U(\mathbf{x}^*) - U(\mathbf{x})$
 8:         if $K(\mathbf{p}_b) = m_b|\mathbf{p}_b| > \Delta U$ then
 9:             $\mathbf{x}_b \leftarrow \mathbf{x}_b^*$
10:             $\mathbf{p}_b \leftarrow \mathbf{p}_b - m_b \cdot sign(\mathbf{p}_b) \cdot \Delta U$
11:         else
12:             $\mathbf{p}_b \leftarrow -\mathbf{p}_b$
13:         end if
14:     end for
15:     return $\mathbf{x}_b, \mathbf{p}_b$
16: end function
```

The DHMC algorithm [8] adapted for LF-PPL and our compilation scheme is as follows:

Algorithm 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.

```
1: function DHMC-LFPPL($\Delta, \Gamma, D, F, \mathbf{x}, \mathbf{p}, H, \epsilon, L$)
2:     $B = \Gamma$;   $A = \Delta \setminus \Gamma$
3:     for $a \in A$ do                                        ▷ $a$ represents the set of continuous variables
4:         $\mathbf{x}_a^0 \leftarrow \mathbf{x}_a$;   $\mathbf{p}_a \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$
5:     end for
6:     for $b \in B$ do
7:         $\mathbf{x}_b^0 \leftarrow \mathbf{x}_b$;   $\mathbf{p}_b \sim Laplace(\mathbf{0}, \mathbf{1})$          ▷ $b$ represents the set of discontinuous variables
8:     end for
9:     $\forall a \in A,\ \mathbf{x}_a^0 \leftarrow \mathbf{x}_a$;   $\mathbf{p}_a \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$              ▷ $A$ represents the set of continuous variables
10:    $\forall b \in B,\ \mathbf{x}_b^0 \leftarrow \mathbf{x}_b$;   $\mathbf{p}_b \sim Laplace(\mathbf{0}, \mathbf{1})$          ▷ $B$ represents the set of discontinuous variables
11:    $U \leftarrow -\text{LogJointDensity}(D, F)$
12:    for $i = 1$ to $L$ do
13:        $U_A \leftarrow U$ with names in $B$ replaced by their values in $\mathbf{x}_B^i$
14:        $(\mathbf{x}_A^i, \mathbf{p}_A^i) \leftarrow \text{Halfstep1}(\mathbf{x}_A^{i-1}, \mathbf{p}_A^{i-1}, \epsilon, U_A)$
15:        $U_B \leftarrow U$ with names in $A$ replaced by their values in $\mathbf{x}_A^i$
16:        $(\mathbf{x}_B^i, \mathbf{p}_B^i) \leftarrow \text{Coordinate-wise}(\mathbf{x}_B^{i-1}, \mathbf{p}_B^{i-1}, \epsilon, U_B)$
17:        $U_A \leftarrow U$ with names in $B$ replaced by their values in $\mathbf{x}_B^i$
18:        $(\mathbf{x}_A^i, \mathbf{p}_A^i) \leftarrow \text{Halfstep2}(\mathbf{x}_A^i, \mathbf{p}_A^i, \epsilon, U_A)$
19:    end for
20:    $\mathbf{x}^L \leftarrow \mathbf{x}_A^L \cup \mathbf{x}_B^L,\ \mathbf{p}^L \leftarrow \mathbf{p}_A^L \cup \mathbf{p}_B^L$;
21:    $\mathbf{x}^*, \mathbf{p}^* \leftarrow \text{Evaluate}(F,\ \mathbf{x}^L, \mathbf{p}^L)$
22:    $\alpha \sim Uniform(0, 1)$
23:    if $\alpha > \min\{1, \exp(H(\mathbf{x}, \mathbf{p}) - H(\mathbf{x}^*, \mathbf{p}^*))\}$ then
24:        return $\mathbf{x}^*, \mathbf{p}^*$
25:    else
26:        return $\mathbf{x}, \mathbf{p}$
27:    end if
28: end function
29: function HALFSTEP1($\mathbf{x}, \mathbf{p}, \epsilon, U$)
30:    $\mathbf{p}' \leftarrow \mathbf{p} - \frac{\epsilon}{2} \nabla_{\mathbf{x}} U(\mathbf{x})$
31:    $\mathbf{x}' \leftarrow \mathbf{x} + \frac{\epsilon}{2} \nabla_{\mathbf{p}'} K(\mathbf{p}')$
32:    return $(\mathbf{x}', \mathbf{p}')$
33: end function
34: function HALFSTEP2($\mathbf{x}, \mathbf{p}, \epsilon, U$)
35:    $\mathbf{x}' \leftarrow \mathbf{x} + \frac{\epsilon}{2} \nabla_{\mathbf{p}} K(\mathbf{p})$
36:    $\mathbf{p}' \leftarrow \mathbf{p} - \frac{\epsilon}{2} \nabla_{\mathbf{x}'} U(\mathbf{x}')$
37:    return $(\mathbf{x}', \mathbf{p}')$
38: end function
```

D 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 4: The LF-PPL version of the Gaussian mixture model detailed in Section 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 5: The LF-PPL version of the heavy-tailed model detailed in Section 6.

Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models
Publication Status	[✓ Published] ◻ Accepted for Publication Submitted for Publication ◻Unpublished and unsubmitted work written in a manuscript style
Publication Details	Yuan Zhou*, **Bradley J Gram-Hansen*,** Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. * **Equal contribution**, LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models, In Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS), 2019

Student Confirmation

Student Name:	Bradley Gram-Hansen		
Contribution to the Paper	The overall idea was developed jointly between myself, Zhou, Rainforth, Wood, Kohn and Yang. In particular, the compilation rules and the theoretical properties were developed by myself, Zhou and Yang, The experiments for the GMM and RRHMC model were generated by Zhou and myself, I generated the data and Zhou the figures. The implementation of the PyLFPPL source code was done by me and Tobias ~ 14,000 lines of code. I, Zhou, Yang, and Rainforth all contributed to writing the paper.		
Signature [illegible]		Date	12/02/2020

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Philip H.S. Torr		
Supervisor comments		
Signature Philip Torr	Date	26/04/2021

This completed form should be included in the thesis, at the end of the relevant chapter.

5
Real-world Simulators and Probabilistic Programming

Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model

Atılım Güneş Baydin,[1] **Lukas Heinrich,**[2] **Wahid Bhimji,**[3] **Bradley Gram-Hansen,**[1] **Lei Shao,**[4]
Saeid Naderiparizi,[5] **Andreas Munk,**[5] **Jialin Liu,**[3] **Gilles Louppe**[6]
Lawrence Meadows,[4] **Philip Torr,**[1] **Victor Lee,**[4] **Prabhat,**[3] **Kyle Cranmer,**[7] **Frank Wood**[5]
[1]University of Oxford, [2]CERN, [3]Lawrence Berkeley National Lab, [4]Intel Corporation
[5]University of British Columbia, [6]University of Liege, [7]New York University

Abstract

We present a novel probabilistic programming framework that couples directly to existing large-scale simulators through a cross-platform probabilistic execution protocol, which allows general-purpose inference engines to record and control random number draws within simulators in a language-agnostic way. The execution of existing simulators as probabilistic programs enables highly interpretable posterior inference in the structured model defined by the simulator code base. We demonstrate the technique in particle physics, on a scientifically accurate simulation of the τ (tau) lepton decay, which is a key ingredient in establishing the properties of the Higgs boson. Inference efficiency is achieved via inference compilation where a deep recurrent neural network is trained to parameterize proposal distributions and control the stochastic simulator in a sequential importance sampling scheme, at a fraction of the computational cost of a Markov chain Monte Carlo baseline.

1 Introduction

Complex simulators are used to express causal generative models of data across a wide segment of the scientific community, with applications as diverse as hazard analysis in seismology [49], supernova shock waves in astrophysics [36], market movements in economics [73], and blood flow in biology [72]. In these generative models, complex simulators are composed from low-level mechanistic components. These models are typically non-differentiable and lead to intractable likelihoods, which renders many traditional statistical inference algorithms irrelevant and motivates a new class of so-called likelihood-free inference algorithms [48].

There are two broad strategies for this type of likelihood-free inference problem. In the first, one uses a simulator indirectly to train a surrogate model endowed with a likelihood that can be used in traditional inference algorithms, for example approaches based on conditional density estimation [56, 70, 77, 85] and density ratio estimation [30, 35]. Alternatively, approximate Bayesian computation (ABC) [81, 87] refers to a large class of approaches for sampling from the posterior distribution of these likelihood-free models, where the original simulator is used directly as part of the inference engine. While variational inference [22] algorithms are often used when the posterior is intractable, they are not directly applicable when the likelihood of the data generating process is unknown [84].

The class of inference strategies that directly use a simulator avoids the necessity of approximating the generative model. Moreover, using a domain-specific simulator offers a natural pathway for inference algorithms to provide interpretable posterior samples. In this work, we take this approach, extend previous work in universal probabilistic programming [44, 86] and inference compilation [63, 65] to large-scale complex simulators, and demonstrate the ability to execute existing simulator codes under the control of general-purpose inference engines. This is achieved by creating a cross-

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

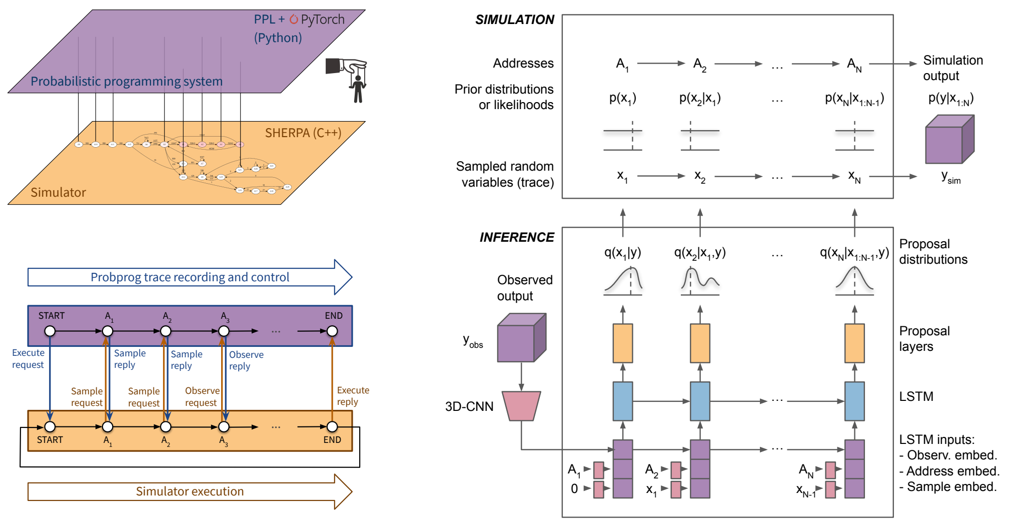


Figure 1: *Top left:* overall framework where the PPS is controlling the simulator. *Bottom left:* probabilistic execution of a single trace. *Right:* LSTM proposals conditioned on an observation.

platform probabilistic execution protocol (Figure 1, left) through which an inference engine can control simulators in a language-agnostic way. We implement a range of general-purpose inference engines from the Markov chain Monte Carlo (MCMC) [25] and importance sampling [34] families. The execution framework we develop currently has bindings in C++ and Python, which are languages of choice for many large-scale projects in science and industry. It can also be used by any other language that support flatbuffers[1] pending the implementation of a lightweight front end.

We demonstrate the technique in a particle physics setting, introducing probabilistic programming as a novel tool to determine the properties of particles at the Large Hadron Collider (LHC) [1, 29] at CERN. This is achieved by coupling our framework with SHERPA[2] [42], a state-of-the-art Monte Carlo event generator of high-energy reactions of particles, which is commonly used with Geant4[3] [5], a toolkit for the simulation of the passage of the resulting particles through detectors. In particular, we perform inference on the details of the decay of a τ (tau) lepton measured by an LHC-like detector by controlling the SHERPA simulation (with minimal modifications to the standard software), extract posterior distributions, and compare to ground truth. To our knowledge this is the first time that universal probabilistic programming has been applied in this domain and at this scale, controlling a code base of nearly one million lines of code. Our approach is scalable to more complex events and full detector simulators, paving the way to its use in the discovery of new fundamental physics.

2 Particle Physics and Probabilistic Inference

Our work is motivated by applications in particle physics, which studies elementary particles and their interactions using high-energy collisions created in particle accelerators such as the LHC at CERN. In this setting, collision events happen many millions of times per second, creating cascading particle decays recorded by complex detectors instrumented with millions of electronics channels. These experiments then seek to filter the vast volume of (petabyte-scale) resulting data to make discoveries that shape our understanding of fundamental physics.

The complexity of the underlying physics and of the detectors have, until now, prevented the community from employing likelihood-free inference techniques for individual collision events. However, they have developed sophisticated simulator packages such as SHERPA [42], Geant4 [5], Pythia8 [79], Herwig++ [16], and MadGraph5 [6] to model physical processes and the interactions of particles with detectors. This is interesting from a probabilistic programming point of view, because

[1] `https://google.github.io/flatbuffers/` [2] **S**imulation of **H**igh-**E**nergy **R**eactions of **Pa**rticles. `https://sherpa.hepforge.org/` [3] **Ge**ometry **an**d **T**racking. `https://geant4.web.cern.ch/`

these simulators are essentially very accurate generative models implementing the Standard Model of particle physics and the passage of particles through matter (i.e., particle detectors). These simulators are coded in Turing-complete general-purpose programming languages, and performing inference in such a setting *requires* using inference techniques developed for universal probabilistic programming that cannot be handled via more traditional inference approaches that apply to, for example, finite probabilistic graphical models [58]. Thus we focus on creating an infrastructure for the interpretation of existing simulator packages as probabilistic programs, which lays the groundwork for running inference in scientifically-accurate probabilistic models using general-purpose inference algorithms.

The τ Lepton Decay. The specific physics setting we focus on in this paper is the decay of a τ lepton particle inside an LHC-like detector. This is a real use case in particle physics currently under active study by LHC physicists [2] and it is also of interest due to its importance to establishing the properties of the recently discovered Higgs boson [1, 29] through its decay to τ particles [12, 33, 46, 47]. Once produced, the τ decays to further particles according to certain decay channels. The prior probabilities of these decays or "branching ratios" are shown in Figure 8 (appendix).

3 Related Work

3.1 Probabilistic Programming

Probabilistic programming languages (PPLs) extend general-purpose programming languages with constructs to do sampling and conditioning of random variables [86]. PPLs decouple model specification from inference: a model is implemented by the user as a regular program in the host programming language, specifying a model that produces samples from a generative process at each execution. In other words, the program produces samples from a joint prior distribution $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ that it implicitly defines, where $\mathbf{x}$ and $\mathbf{y}$ denote latent and observed random variables, respectively. The program can then be executed using a variety of general-purpose inference engines available in the PPL to obtain $p(\mathbf{x}|\mathbf{y})$, the posterior distribution of latent variables $\mathbf{x}$ conditioned on observed variables $\mathbf{y}$. *Universal* PPLs allow the expression of unrestricted probability models in a Turing-complete fashion [43, 89, 90], in contrast to languages such as Stan [28, 39] that target the more restricted model class of probabilistic graphical models [58]. Inference engines available in PPLs range from MCMC-based lightweight Metropolis Hastings (LMH) [89] and random-walk Metropolis Hastings (RMH) [62] to importance sampling (IS) [11] and sequential Monte Carlo [34]. Modern PPLs such as Pyro [20] and Edward2 [32, 82, 83] use gradient-based inference engines including variational inference [52, 57] and Hamiltonian Monte Carlo [53, 69] that benefit from modern deep learning hardware and automatic differentiation [18] features provided by PyTorch [71] and TensorFlow [3] libraries. Another way of making use of gradient-based optimization is to combine IS with deep-learning-based proposals trained with data sampled from the probabilistic program, resulting in the inference compilation (IC) algorithm [63] that enables amortized inference [40].

3.2 Data Analysis in Particle Physics

Inference for an individual collision event in particle physics is often referred to as reconstruction [61]. Reconstruction algorithms can be seen as a form of structured prediction: from the raw event data they produce a list of candidate particles together with their types and point-estimates for their momenta. The variance of these estimators is characterized by comparison to the ground truth values of the latent variables from simulated events. Bayesian inference on the latent state of an individual collision is rare in particle physics, given the complexity of the latent structure of the generative model. Until now, inference for the latent structure of an individual event has only been possible by accepting a drastic simplification of the high-fidelity simulators [4, 7–10, 15, 23, 27, 37, 38, 45, 59, 66, 67, 78, 80]. In contrast, inference for the fundamental parameters is based on hierarchical models and probed at the population level. Recently, machine learning techniques have been employed to learn surrogates for the implicit densities defined by the simulators as a strategy for likelihood-free inference [24].

Currently particle physics simulators are run in forward mode to produce substantial datasets that often exceed the size of datasets from actual collisions within the experiments. These are then reduced to considerably lower dimensional datasets of a handful of variables using physics domain knowledge, which can then be directly compared to collision data. Machine learning and statistical approaches for classification of particle types or regression of particle properties can be trained on these large pre-generated datasets produced by the high-fidelity simulators developed over many decades [13, 55].

The field is increasingly employing deep learning techniques allowing these algorithms to process high-dimensional, low-level data [14, 17, 31, 54, 74]. However, these approaches do not estimate the posterior of the full latent state nor provide the level of interpretability our probabilistic inference framework enables by directly tying inference results to the latent process encoded by the simulator.

4 Probabilistic Inference in Large-Scale Simulators

In this section we describe the main components of our probabilistic inference framework, including: (1) a novel PyTorch-based [71] PPL and associated inference engines in Python, (2) a probabilistic programming execution protocol that defines a cross-platform interface for connecting models and inference engines implemented in different languages and executed in separate processes, (3) a lighweight C++ front end allowing execution of models written in C++ under the control of our PPL.

4.1 Designing a PPL for Existing Large-Scale Simulators

A shortcoming of the state-of-the-art PPLs is that they are not designed to directly support *existing* code bases, requiring one to implement any model from scratch in each specific PPL. This limitation rules out their applicability to a very large body of existing models implemented as domain-specific simulators in many fields across academia and industry. A PPL, by definition, is a programming language with additional constructs for *sampling* random values from probability distributions and *conditioning* values of random variables via observations [44, 86]. Domain-specific simulators in particle physics and other fields are commonly stochastic in nature, thus they satisfy the behavior random *sampling*, albeit generally from simplistic distributions such as the continuous uniform. By interfacing with these simulators at the level of random number *sampling* (via capturing calls to the random number generator) and introducing a construct for *conditioning*, we can execute existing stochastic simulators as probabilistic programs. Our work introduces the necessary framework to do so, and makes these simulators, which commonly represent the most accurate models and understanding in their corresponding fields, subject to Bayesian inference using general-purpose inference engines. In this setting, a simulator is no longer a black box, as all predictions are directly tied into the fully-interpretable structured model implemented by the simulator code base.

To realize our framework, we implement a universal PPL called pyprob,[4] specifically designed to execute models written not only in Python but also in other languages. Our PPL currently has two families of inference engines:[5] (1) MCMC of the lightweight Metropolis–Hastings (LMH) [89] and random-walk Metropolis–Hastings (RMH) [62] varieties, and (2) sequential importance sampling (IS) [11, 34] with its regular (i.e., sampling from the prior) and inference compilation (IC) [63] varieties. The IC technique, where a recurrent neural network (NN) is trained in order to provide amortized inference to guide (control) a probabilistic program conditioning on observed inputs, forms our main inference method for performing efficient inference in large-scale simulators. Because IC training and inference uses dynamic reconfiguration of NN modules [63], we base our PPL on PyTorch [71], whose automatic differentiation feature with support for dynamic computation graphs [18] has been crucial in our implementation. The LMH and RMH engines we implement are specialized for sampling in the space of execution traces of probabilistic programs, and provide a way of sampling from the true posterior and therefore provide a baseline—at a high computational cost.

A probabilistic program can be expressed as a sequence of random samples $(x_t, a_t, i_t)_{t=1}^T$, where x_t, a_t, and i_t are respectively the value, address,[6] and instance (counter) of a sample, the execution of which describes a joint probability distribution between latent (unobserved) random variables $\mathbf{x} := (x_t)_{t=1}^T$ and observed random variables $\mathbf{y} := (y_n)_{n=1}^N$ given by

$$p(\mathbf{x}, \mathbf{y}) := \prod_{t=1}^{T} f_{a_t}(x_t | x_{1:t-1}) \prod_{n=1}^{N} g_n(y_n | x_{\prec n}) \ , \qquad (1)$$

[4] `https://github.com/pyprob/pyprob` [5] The selection of these families was motivated by working with existing simulators through an execution protocol (Section 4.2) precluding the use of gradient-based inference engines. We plan to extend this protocol in future work to incorporate differentiability. [6] An "address" is a label uniquely identifying each sampling or conditioning event in the execution of the program. In our system it is based on a concatenation of stack frames (Table 1) leading up to the point of each random number draw, and it also includes a suffix identifying the type of associated probability distribution.

where $f_{a_t}(\cdot|x_{1:t-1})$ denotes the prior probability distribution of a random variable with address a_t conditional on all preceding values $x_{1:t-1}$, and $g_n(\cdot|x_{\prec n})$ is the likelihood density given the sample values $x_{\prec n}$ preceding observation y_n. Once a model $p(\mathbf{x}, \mathbf{y})$ is expressed as a probabilistic program, we are interested in performing inference in order to get posterior distributions $p(\mathbf{x}|\mathbf{y})$ of latent variables $\mathbf{x}$ conditioned on observed variables $\mathbf{y}$.

Inference engines of the MCMC family, designed to work in the space of probabilistic execution traces, constitute the gold standard for obtaining samples from the true posterior of a probabilistic program [62, 86, 89]. Given a current sequence of latents $\mathbf{x}$ in the trace space, these work by making proposals $\mathbf{x}'$ according to a proposal distribution $q(\mathbf{x}'|\mathbf{x})$ and deciding whether to move from $\mathbf{x}$ to $\mathbf{x}'$ based on the Metropolis–Hasting acceptance ratio of the form

$$\alpha = \min\{1, \frac{p(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p(\mathbf{x})q(\mathbf{x}'|\mathbf{x}))}\} . \tag{2}$$

Inference engines in the IS family use a weighted set of samples $\{(w^k, \mathbf{x}^k)_{k=1}^K\}$ to construct an empirical approximation of the posterior distribution: $\hat{p}(\mathbf{x}|\mathbf{y}) = \sum_{k=1}^K w^k \delta(\mathbf{x}^k - \mathbf{x}) / \sum_{j=1}^K w^j$, where δ is the Dirac delta function. The importance weight for each execution trace is

$$w^k = \prod_{n=1}^{N} g_n(y_n|x^k_{1:\tau_k(n)}) \prod_{t=1}^{T^k} \frac{f_{a_t}(x^k_t|x^k_{1:t-1})}{q_{a_t,i_t}(x^k_t|x^k_{1:t-1})} , \tag{3}$$

where $q_{a_t,i_t}(\cdot|x^k_{1:t-1})$ is known as the proposal distribution and may be identical to the prior f_{a_t} (as in regular IS). In the IC technique, we train a recurrent NN to receive the observed values $\mathbf{y}$ and return a set of adapted proposals $q_{a_t,i_t}(x_t|x_{1:t-1}, \mathbf{y})$ such that the approximate posterior $q(\mathbf{x}|\mathbf{y})$ is close to the true posterior $p(\mathbf{x}|\mathbf{y})$. This is achieved by using a Kullback–Leibler divergence training objective $\mathbb{E}_{p(\mathbf{y})}\left[D_{\text{KL}}\left(p(\mathbf{x}|\mathbf{y}) \mid\mid q(\mathbf{x}|\mathbf{y};\phi)\right)\right]$ as

$$\mathcal{L}(\phi) := \int_{\mathbf{y}} p(\mathbf{y}) \int_{\mathbf{x}} p(\mathbf{x}|\mathbf{y}) \log \frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\mathbf{y};\phi)} \, \mathrm{d}\mathbf{x} \, \mathrm{d}\mathbf{y} = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q(\mathbf{x}|\mathbf{y};\phi)\right] + \text{const.} , \tag{4}$$

where ϕ represents the NN weights. The weights ϕ are optimized to minimize this objective by continually drawing training pairs $(\mathbf{x}, \mathbf{y}) \sim p(\mathbf{x}, \mathbf{y})$ from the probabilistic program (the simulator). In IC training, we may designate a subset of all addresses (a_t, i_t) to be "controlled" (learned) by the NN, leaving all remaining addresses to use the prior f_{a_t} as proposal during inference. Expressed in simple terms, taking an observation $\mathbf{y}$ (an observed event that we would like to recreate or explain with the simulator) as input, the NN learns to control the random number draws of latents $\mathbf{x}$ during the simulator's execution in such a way that makes the observed outcome likely (Figure 1, right).

The NN architecture in IC is based on a stacked LSTM [51] recurrent core that gets executed for as many time steps as the probabilistic trace length. The input to this LSTM in each time step is a concatenation of embeddings of the observation $f^{\text{obs}}(\mathbf{y})$, the current address $f^{\text{addr}}(a_t, i_t)$, and the previously sampled value $f^{\text{smp}}_{a_{t-1},i_{t-1}}(x_{t-1})$. f^{obs} is a NN specific to the domain (such as a 3D convolutional NN for volumetric inputs), f^{smp} are feed-forward modules, and f^{addr} are learned address embeddings optimized via backpropagation for each (a_t, i_t) pair encountered in the program execution. The addressing scheme a_t is the main link between semantic locations in the probabilistic program [89] and the inputs to the NN. The address of each `sample` or `observe` statement is supplied over the execution protocol (Section 4.2) at runtime by the process hosting and executing the model. The joint proposal distribution of the NN $q(\mathbf{x}|\mathbf{y})$ is factorized into proposals in each time step q_{a_t,i_t}, whose type depends on the type of the prior f_{a_t}. In our experiments in this paper (Section 5) the simulator uses categorical and continuous uniform priors, for which IC uses, respectively, categorical and mixture of truncated Gaussian distributions as proposals parameterized by the NN. The creation of IC NNs is automatic, i.e., an open-ended number of NN modules are generated by the PPL on-the-fly when a simulator address a_t is encountered for the first time during training [63]. These modules are reused (either for inference or undergoing further training) when the same address is encountered in the lifetime of the same trained NN.

A common challenge for inference in real-world scientific models, such as those in particle physics, is the presence of large dynamic ranges of prior probabilities for various outcomes. For instance, some particle decays are $\sim 10^4$ times more probable than others (Figure 8, appendix), and the prior distribution for a particle momentum can be steeply falling. Therefore some cases may be much more likely to be seen by the NN during training relative to others. For this reason, the proposal parameters and the quality of the inference would vary significantly according to the frequency of the observations in the prior. To address this issue, we apply a "prior inflation" scheme to automatically adjust the measure of the prior distribution during training to generate more instances of these unlikely outcomes. This applies only to the training data generation for the IC NN, and the unmodified original model prior is used during inference, ensuring that the importance weights (Eq. 3) and therefore the empirical posterior are correct under the original model.

4.2 A Cross-Platform Probabilistic Execution Protocol

To couple our PPL and inference engines with simulators in a language-agnostic way, we introduce a probabilistic programming execution protocol (PPX)[7] that defines a schema for the execution of probabilistic programs. The protocol covers language-agnostic definitions of common probability distributions and message pairs covering the call and return values of (1) program entry points (2) `sample` statements, and (3) `observe` statements (Figure 1, left). The implementation is based on flatbuffers,[8] which is an efficient cross-platform serialization library through which we compile the protocol into the officially supported languages C++, C#, Go, Java, JavaScript, PHP, Python, and TypeScript, enabling very lightweight PPL front ends in these languages—in the sense of requiring only an implementation to call `sample` and `observe` statements over the protocol. We exchange these flatbuffers-encoded messages over ZeroMQ[9] [50] sockets, which allow seamless communication between separate processes in the same machine (using inter-process sockets) or across a network (using TCP).

Connecting any stochastic simulator in a supported language involves only the redirection of calls to the random number generator (RNG) to call the `sample` method of PPX using the corresponding probability distribution as the argument, which is facilitated when a simulator-wide RNG interface is defined in a single code file as is the case in SHERPA (Section 4.3). Conditioning is achieved by either providing an observed value for any `sample` at inference time (which means that the sample will be fixed to the observed value) or adding manual `observe` statements, similar to Pyro [20].

Besides its use with our Python PPL, the protocol defines a very flexible way of coupling any PPL system to any model so that these two sides can be (1) implemented in different programming languages and (2) executed in separate processes and on separate machines across networks. Thus we present this protocol as a probabilistic programming analogue to the Open Neural Network Exchange (ONNX)[10] project for interoperability between deep learning frameworks, in the sense that PPX is an interoperability project between PPLs allowing language-agnostic exchange of existing models (simulators). Note that, more than a serialization format, the protocol enables runtime execution of probabilistic models under the control of inference engines in different PPLs. We are releasing this protocol as a separately maintained project, together with the rest of our work in Python and C++.

4.3 Controlling SHERPA's Simulation of Fundamental Particle Physics

We demonstrate our framework with SHERPA [42], a Monte Carlo event generator of high-energy reactions of particles, which is a state-of-the-art simulator of the Standard Model developed by the particle physics community. SHERPA, like many other large-scale scientific projects, is implemented in C++, and therefore we implement a C++ front end for our protocol.[11] We couple SHERPA to the front end by a system-wide rerouting of the calls to the RNG, which is made easy by the existence of a third-party RNG interface (External_RNG) already present in SHERPA. Through this setup, we can repurpose, with little effort, any stochastic simulation written in SHERPA as a probabilistic generative model in which we can perform inference.

Random number draws in C++ simulators are commonly performed at a lower level than the actual prior distribution that is being simulated. This applies to SHERPA where the only samples are from the standard uniform distribution $U(0, 1)$, which subsequently get used for different purposes using transformations or rejection sampling. In our experiments (Section 5) we work with all uniform samples except for a problem-specific single address that we know to be responsible for sampling from a categorical distribution representing particle decay channels. The modification of this address

[7] `https://github.com/pyprob/ppx` [8] `http://google.github.io/flatbuffers/`
[9] `http://zeromq.org/` [10] `https://onnx.ai/` [11] `https://github.com/pyprob/pyprob_cpp`

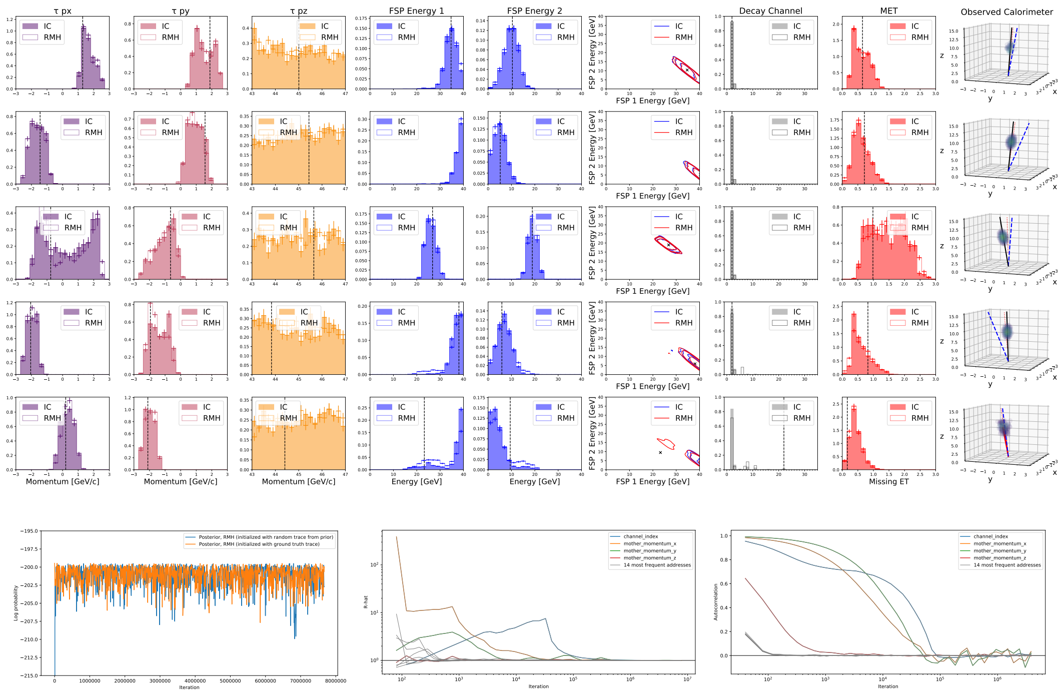


Figure 2: *Top histograms:* RMH and IC posterior results where a Channel 2 decay event ($\tau \to \nu_\tau \pi^-$) is the mode of the posterior distribution. Note that the eight variables shown are just a subset of the full latent state of several thousand addresses (Figure 5, appendix). Vertical lines indicate the point sample of the single GT trace supplying the calorimeter observation in each row. *Bottom plots:* trace joint log-probability, Gelman–Rubin diagnostic, autocorrelation results belonging to the posterior in the first row.

to use the proper categorical prior allows an effortless application of prior inflation (Section 4.1) to generate training data equally representing each channel.

Rejection sampling [41] sections in the simulator pose a challenge for our approach, as they define execution traces that are a priori unbounded; and since the IC NN has to backpropagate through every sampled value, this makes the training significantly slower. Rejection sampling is key to the application of Monte Carlo methods for evaluating matrix elements [60] and other stages of event generation in particle physics; thus an efficient treatment of this construction is primal. We address this problem by implementing a novel trace evaluation scheme which works by annotating the `sample` statements within long-running rejection sampling loops with a boolean flag called `replace`, which, when set true, enables a rejection-sampling-specific behavior for the given sample address. The simplest correct approach is to exclude these `replace` addresses from IC inference (i.e., proposing for these from the prior) and treat them as regular raw addresses in MCMC. Other approaches include amortization schemes where during IC NN training we only consider the last (thus accepted) instance i_{last} of any address (a_t, i_t) that fall within a rejection sampling loop. The results presented in this paper use the former simple mode. Efficient handling of rejection sampling in universal PPLs [68], and nested inference in general [75, 76], constitute an active area of research with several alternative approaches currently being formulated with varying degrees of complexity and sample efficiency that are beyond the scope of this paper.

5 Experiments

An important decay of the Higgs boson is to τ leptons, whose subsequent decay products interact in the detector. This constitutes a rich and realistic case to simulate, and directly connects to an important line of current research in particle physics. During simulation, SHERPA stochastically generates a set of particles to which the initial τ lepton will decay—a “decay channel”—and samples the momenta of these particles according to a joint density obtained from underlying physical theory. These particles then interact in the detector leading to observations in the raw sensor data. While Geant4 is typically used to model the interactions in a detector, for our initial studies we implement a fast, approximate, stochastic detector simulation for a calorimeter with longitudinal and transverse segmentation (with $20 \times 35 \times 35$ voxels). The detector deposits most of the energy for electrons and π^0 into the first layers and charged hadrons (e.g., $\pi^{\pm}$) deeper into the calorimeter with larger fluctuations.

Figure 2 presents posterior distributions of a selected subset of random variables in the simulator for five different test cases where the mode of the posterior is a channel-2 decay ($\tau \to \nu_\tau \pi^-$). Test cases are generated by sampling an execution trace from the simulator prior, giving us a "ground truth trace" (GT trace), from which we extract the simulated raw 3D calorimeter as a test observation. We run our inference engines taking only these calorimeter data as input, giving us posteriors over the entire latent state of the simulator, conditioned on the observed calorimeter using a physically-motivated Poisson likelihood. We show RMH (MCMC) and IC inference results, where RMH serves as a baseline as it samples from the true posterior of the model, albeit at great computational cost. For each case, we establish the convergence of the RMH posterior to the true posterior by computing the Gelman–Rubin (GR) convergence diagnostic [26, 88] between two MCMC chains conditioned on the same observation, one starting from the GT trace and one starting from a random trace sampled from the prior.[12] As an example, in Figure 2 (bottom) we show the joint log-probability, GR diagnostic, and autocorrelation plots of the RMH posterior (with 7.7M traces) belonging to the test case in the first row. The GR result indicates that the chains converged around 10^6 iterations, and the autocorrelation result indicates that we need approximately 10^5 iterations to accumulate each new effectively independent sample from the true posterior. These RMH baseline results incur significant computational cost due to the sequential nature of the sampling and the large number of iterations one needs to accumulate statistically independent samples. The example we presented took 115 compute hours on an Intel E5-2695 v2 @ 2.40GHz CPU node.

We present IC posteriors conditioned on the same observations in Figure 2 and plot these together with corresponding RMH baselines, showing good agreement in all cases. These IC posteriors were obtained in less than 30 minutes in each case, representing a significant speedup compared with the RMH baseline. This is due to three main strengths of IC inference: (1) each trace executed by the IC engine gives us a statistically independent sample from the learned proposal approximating the true posterior (Equation 4) (cf. the autocorrelation time of 10^5 in RMH); following from this independence, (2) IC inference does not necessitate a burn-in period (cf. 10^6 iterations to convergence in GR for RMH); and (3) IC inference is embarrassingly parallelizable. These features represent the main motivation to incorporate IC in our framework to make inference in large-scale simulators computationally efficient and practicable. The results presented were obtained by running IC inference in parallel on 20 compute nodes of the type used for RMH inference, using a NN with 143,485,048 parameters that has been trained for 40 epochs with a training set of 3M traces sampled from the simulator prior, lasting two days on 32 CPU nodes. This time cost for NN training needs to be incurred only once for any given simulator setup, resulting in a trained inference NN that enables fast, repeated inference in the model specified by the simulator—a concept referred to as "amortized inference". Details of the 3DCNN–LSTM architecture used are in Figure 9 (appendix).

In the last test case in Figure 2 we show posteriors corresponding to a calorimeter observation of a Channel 22 event ($\tau \to \nu_\tau K^- K^- K^+$), a type of decay producing calorimeter depositions with similarity to Channel 2 decays and with extremely low probability in the prior (Figure 8, appendix), therefore representing a difficult case to infer. We see the posterior uncertainty in the true (RMH) posterior of this case, where Channel 2 is the mode of the posterior with a small probability mass on Channel 22 among other channels. We see that the IC posterior is able to reproduce this small probability mass on Channel 22 with success, thanks to the "prior inflation" scheme with which we train IC NNs. This leads to a proposal where Channel 22 is the mode, which later gets adjusted by importance weighting (Equation 3) to match the true posterior result (Figure 7, appendix). Our results demonstrate the feasibility of Bayesian inference in the whole latent space of this existing simulator defining a potentially unbounded number of addresses, of which we encountered approximately 24k during our experiments (Table 1 also Figure 5, appendix). To our knowledge, this is the first time a PPL system has been used with a model expressed by an existing state-of-the-art simulator at this scale.

[12] The GR diagnostic compares estimated between-chains and within-chain variances, summarized as the $\hat{R}$ metric which approaches unity as the chains converge on the target distribution.

6 Conclusions

We presented the first step in subsuming the vast existing body of scientific simulators, which are causal, generative models that often reflect the most accurate understanding in their respective fields, into a universal probabilistic programming framework. The ability to scale probabilistic inference to large-scale simulators is of fundamental importance to the field of probabilistic programming and the wider modeling community. It is a hard problem that requires innovations in many areas such as model–PPL interface, handling of priors with long tails, amortization of rejection sampling routines [68], addressing schemes, IC network architectures, and distributed training and inference [19] which make it difficult to cover in depth in a single paper.

Our work allows one to use existing simulator code bases to perform model-based machine learning with interpretability, where the simulator is no longer used as a black box to generate synthetic training data, but as a highly structured generative model that the simulator's code already specifies. Bayesian inference in this setting gives results that are highly interpretable, where we get to see the exact locations and processes in the model that are associated with each prediction and the uncertainty in each prediction. With this novel framework providing a clearly defined interface between domain-specific simulators and probabilistic machine learning techniques, we expect to enable a wide range of applied work straddling machine learning and fields of science and engineering. In the particle physics setting, our ultimate aim is to run the inference stage of this approach on collision data from real detectors by implementing a full LHC physics analysis together with the full posterior, so that it can be exploited for discovery of new physics via simulations that contain processes beyond the current Standard Model.

Acknowledgments

We thank the anonymous reviewers for their constructive comments that helped us improve this paper significantly. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This work was partially supported by the NERSC Big Data Center; we acknowledge Intel for their funding support. KC, LH, and GL were supported by the National Science Foundation under the awards ACI-1450310. Additionally, KC was supported by the National Science Foundation award OAC-1836650. BGH is supported by the EPRSC Autonomous Intelligent Machines and Systems grant. AGB and PT are supported by EPSRC/MURI grant EP/N019474/1 and AGB is also supported by Lawrence Berkeley National Lab. FW is supported by DARPA D3M, under Cooperative Agreement FA8750-17-2-0093, Intel under its LBNL NERSC Big Data Center, and an NSERC Discovery grant.

References

[1] G. Aad, T. Abajyan, B. Abbott, J. Abdallah, S. Abdel Khalek, A. A. Abdelalim, O. Abdinov, R. Aben, B. Abi, M. Abolins, and et al. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716:1–29, Sept. 2012.

[2] G. Aad et al. Reconstruction of hadronic decay products of tau leptons with the ATLAS experiment. *Eur. Phys. J.*, C76(5):295, 2016.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] V. M. Abazov et al. A precision measurement of the mass of the top quark. *Nature*, 429:638–642, 2004.

[5] J. Allison, K. Amako, J. Apostolakis, P. Arce, M. Asai, T. Aso, E. Bagli, A. Bagulya, S. Banerjee, G. Barrand, B. Beck, A. Bogdanov, D. Brandt, J. Brown, H. Burkhardt, P. Canal, D. Cano-Ott, S. Chauvie, K. Cho, G. Cirrone, G. Cooperman, M. Cortés-Giraldo, G. Cosmo, G. Cuttone,

G. Depaola, L. Desorgher, X. Dong, A. Dotti, V. Elvira, G. Folger, Z. Francis, A. Galoyan, L. Garnier, M. Gayer, K. Genser, V. Grichine, S. Guatelli, P. Guèye, P. Gumplinger, A. Howard, I. Hřivnáčová, S. Hwang, S. Incerti, A. Ivanchenko, V. Ivanchenko, F. Jones, S. Jun, P. Kaitaniemi, N. Karakatsanis, M. Karamitros, M. Kelsey, A. Kimura, T. Koi, H. Kurashige, A. Lechner, S. Lee, F. Longo, M. Maire, D. Mancusi, A. Mantero, E. Mendoza, B. Morgan, K. Murakami, T. Nikitina, L. Pandola, P. Paprocki, J. Perl, I. Petrović, M. Pia, W. Pokorski, J. Quesada, M. Raine, M. Reis, A. Ribon, A. R. Fira, F. Romano, G. Russo, G. Santin, T. Sasaki, D. Sawkey, J. Shin, I. Strakovsky, A. Taborda, S. Tanaka, B. Tomé, T. Toshito, H. Tran, P. Truscott, L. Urban, V. Uzhinsky, J. Verbeke, M. Verderi, B. Wendt, H. Wenzel, D. Wright, D. Wright, T. Yamashita, J. Yarba, and H. Yoshida. Recent developments in GEANT4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 835(Supplement C):186 – 225, 2016.

[6] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H.-S. Shao, T. Stelzer, P. Torrielli, and M. Zaro. The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. *Journal of High Energy Physics*, 2014(7):79, 2014.

[7] J. Alwall, A. Freitas, and O. Mattelaer. The Matrix Element Method and QCD Radiation. *Phys. Rev.*, D83:074010, 2011.

[8] J. R. Andersen, C. Englert, and M. Spannowsky. Extracting precise Higgs couplings by using the matrix element method. *Phys. Rev.*, D87(1):015019, 2013.

[9] P. Artoisenet, P. de Aquino, F. Maltoni, and O. Mattelaer. Unravelling $t\bar{t}h$ via the Matrix Element Method. *Phys. Rev. Lett.*, 111(9):091802, 2013.

[10] P. Artoisenet and O. Mattelaer. MadWeight: Automatic event reweighting with matrix elements. *PoS*, CHARGED2008:025, 2008.

[11] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[12] A. Askew, P. Jaiswal, T. Okui, H. B. Prosper, and N. Sato. Prospect for measuring the CP phase in the $h\tau\tau$ coupling at the LHC. *Phys. Rev.*, D91(7):075014, 2015.

[13] L. Asquith et al. Jet Substructure at the Large Hadron Collider : Experimental Review. 2018.

[14] A. Aurisano, A. Radovic, D. Rocco, A. Himmel, M. Messier, E. Niner, G. Pawloski, F. Psihas, A. Sousa, and P. Vahle. A convolutional neural network neutrino event classifier. *Journal of Instrumentation*, 11(09):P09001, 2016.

[15] P. Avery et al. Precision studies of the Higgs boson decay channel $H \to ZZ \to 4l$ with MEKD. *Phys. Rev.*, D87(5):055006, 2013.

[16] M. Bähr, S. Gieseke, M. A. Gigg, D. Grellscheid, K. Hamilton, O. Latunde-Dada, S. Plätzer, P. Richardson, M. H. Seymour, A. Sherstnev, et al. Herwig++ physics and manual. *The European Physical Journal C*, 58(4):639–707, 2008.

[17] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308, 2014.

[18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research (JMLR)*, 18(153):1–43, 2018.

[19] A. G. Baydin, L. Shao, W. Bhimji, L. Heinrich, L. F. Meadows, J. Liu, A. Munk, S. Naderiparizi, B. Gram-Hansen, G. Louppe, M. Ma, X. Zhao, P. Torr, V. Lee, K. Cranmer, Prabhat, and F. Wood. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19), November 17–22, 2019*, 2019.

[20] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 2018.

[21] C. M. Bishop. Mixture density networks. Technical Report NCRG/94/004, Neural Computing Research Group, Aston University, 1994.

[22] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[23] S. Bolognesi, Y. Gao, A. V. Gritsan, K. Melnikov, M. Schulze, N. V. Tran, and A. Whitbeck. On the spin and parity of a single-produced resonance at the LHC. *Phys. Rev.*, D86:095031, 2012.

[24] J. Brehmer, K. Cranmer, G. Louppe, and J. Pavez. A Guide to Constraining Effective Field Theories with Machine Learning. *Phys. Rev.*, D98(5):052004, 2018.

[25] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

[26] S. P. Brooks and A. Gelman. General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*, 7(4):434–455, 1998.

[27] J. M. Campbell, R. K. Ellis, W. T. Giele, and C. Williams. Finding the Higgs boson in decays to $Z\gamma$ using the matrix element method at Next-to-Leading Order. *Phys. Rev.*, D87(7):073005, 2013.

[28] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, A. Riddell, et al. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(i01), 2017.

[29] S. Chatrchyan, V. Khachatryan, A. M. Sirunyan, A. Tumasyan, W. Adam, E. Aguilo, T. Bergauer, M. Dragicevic, J. Erö, C. Fabjan, and et al. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Physics Letters B*, 716:30–61, Sept. 2012.

[30] K. Cranmer, J. Pavez, and G. Louppe. Approximating likelihood ratios with calibrated discriminative classifiers. *arXiv preprint arXiv:1506.02169*, 2015.

[31] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman, and A. Schwartzman. Jet-images – deep learning edition. *Journal of High Energy Physics*, 2016(7):69, 2016.

[32] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

[33] A. Djouadi. The Anatomy of electro-weak symmetry breaking. I: The Higgs boson in the standard model. *Phys. Rept.*, 457:1–216, 2008.

[34] A. Doucet and A. M. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of Nonlinear Filtering*, 12(656-704):3, 2009.

[35] R. Dutta, J. Corander, S. Kaski, and M. U. Gutmann. Likelihood-free inference by penalised logistic regression. *arXiv preprint arXiv:1611.10242*, 2016.

[36] E. Endeve, C. Y. Cardall, R. D. Budiardja, S. W. Beck, A. Bejnood, R. J. Toedte, A. Mezzacappa, and J. M. Blondin. Turbulent magnetic field amplification from spiral SASI modes: implications for core-collapse supernovae and proto-neutron star magnetization. *The Astrophysical Journal*, 751(1):26, 2012.

[37] J. S. Gainer, J. Lykken, K. T. Matchev, S. Mrenna, and M. Park. The Matrix Element Method: Past, Present, and Future. In *Proceedings, 2013 Community Summer Study on the Future of U.S. Particle Physics: Snowmass on the Mississippi (CSS2013): Minneapolis, MN, USA, July 29-August 6, 2013*, 2013.

[38] Y. Gao, A. V. Gritsan, Z. Guo, K. Melnikov, M. Schulze, and N. V. Tran. Spin determination of single-produced resonances at hadron colliders. *Phys. Rev.*, D81:075022, 2010.

[39] A. Gelman, D. Lee, and J. Guo. Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015.

[40] S. J. Gershman and N. D. Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, 2014.

[41] W. R. Gilks and P. Wild. Adaptive rejection sampling for Gibbs sampling. *Applied Statistics*, pages 337–348, 1992.

[42] T. Gleisberg, S. Hoeche, F. Krauss, M. Schonherr, S. Schumann, F. Siegert, and J. Winter. Event generation with SHERPA 1.1. *Journal of High Energy Physics*, 02:007, 2009.

[43] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[44] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the Future of Software Engineering*, pages 167–181. ACM, 2014.

[45] A. V. Gritsan, R. Röntsch, M. Schulze, and M. Xiao. Constraining anomalous Higgs boson couplings to the heavy flavor fermions using matrix element techniques. *Phys. Rev.*, D94(5):055023, 2016.

[46] B. Grzadkowski and J. F. Gunion. Using decay angle correlations to detect CP violation in the neutral Higgs sector. *Phys. Lett.*, B350:218–224, 1995.

[47] R. Harnik, A. Martin, T. Okui, R. Primulando, and F. Yu. Measuring CP violation in $h \to \tau^+\tau^-$ at colliders. *Phys. Rev.*, D88(7):076009, 2013.

[48] F. Hartig, J. M. Calabrese, B. Reineking, T. Wiegand, and A. Huth. Statistical inference for stochastic simulation models–theory and application. *Ecology Letters*, 14(8):816–827, 2011.

[49] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, et al. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 3–14. IEEE Press, 2014.

[50] P. Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[51] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[52] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

[53] M. D. Hoffman and A. Gelman. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

[54] B. Hooberman, A. Farbin, G. Khattak, V. Pacela, M. Pierini, J.-R. Vlimant, M. Spiropulu, W. Wei, M. Zhang, and S. Vallecorsa. Calorimetry with Deep Learning: Particle Classification, Energy Regression, and Simulation for High-Energy Physics, 2017. Deep Learning in Physical Sciences (NIPS workshop). `https://dl4physicalsciences.github.io/files/nips_dlps_2017_15.pdf`.

[55] G. Kasieczka. Boosted Top Tagging Method Overview. In *10th International Workshop on Top Quark Physics (TOP2017) Braga, Portugal, September 17-22, 2017*, 2018.

[56] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4743–4751. Curran Associates, Inc., 2016.

[57] D. P. Kingma and M. Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[58] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[59] K. Kondo. Dynamical Likelihood Method for Reconstruction of Events With Missing Momentum. 1: Method and Toy Models. *J. Phys. Soc. Jap.*, 57:4126–4140, 1988.

[60] F. Krauss. Matrix elements and parton showers in hadronic interactions. *Journal of High Energy Physics*, 2002(08):015, 2002.

[61] W. Lampl, S. Laplace, D. Lelas, P. Loch, H. Ma, S. Menke, S. Rajagopalan, D. Rousseau, S. Snyder, and G. Unal. Calorimeter Clustering Algorithms: Description and Performance. Technical Report ATL-LARG-PUB-2008-002. ATL-COM-LARG-2008-003, CERN, Geneva, Apr 2008.

[62] T. A. Le. Inference for higher order probabilistic programs. *Masters Thesis, University of Oxford*, 2015.

[63] T. A. Le, A. G. Baydin, and F. Wood. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

[64] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[65] M. Lezcano Casado, A. G. Baydin, D. Martinez Rubio, T. A. Le, F. Wood, L. Heinrich, G. Louppe, K. Cranmer, W. Bhimji, K. Ng, and Prabhat. Improvements to inference compilation for probabilistic programming in large-scale scientific simulators. In *Neural Information Processing Systems (NIPS) 2017 workshop on Deep Learning for Physical Sciences (DLPS), Long Beach, CA, US, December 8, 2017*, 2017.

[66] T. Martini and P. Uwer. Extending the Matrix Element Method beyond the Born approximation: Calculating event weights at next-to-leading order accuracy. *JHEP*, 09:083, 2015.

[67] T. Martini and P. Uwer. The Matrix Element Method at next-to-leading order QCD for hadronic collisions: Single top-quark production at the LHC as an example application. 2017.

[68] S. Naderiparizi, A. Ścibior, A. Munk, M. Ghadiri, A. G. Baydin, B. Gram-Hansen, C. S. de Witt, R. Zinkov, P. H. Torr, T. Rainforth, Y. W. Teh, and F. Wood. Amortized rejection sampling in universal probabilistic programming. *arXiv preprint arXiv:1910.09056*, 2019.

[69] R. M. Neal. MCMC Using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2011.

[70] G. Papamakarios, T. Pavlakou, and I. Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

[71] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017*, 2017.

[72] P. Perdikaris, L. Grinberg, and G. E. Karniadakis. Multiscale modeling and simulation of brain blood flow. *Physics of Fluids*, 28(2):021304, 2016.

[73] M. Raberto, S. Cincotti, S. M. Focardi, and M. Marchesi. Agent-based simulation of a financial market. *Physica A: Statistical Mechanics and its Applications*, 299(1):319 – 327, 2001. Application of Physics in Economic Modelling.

[74] E. Racah, S. Ko, P. Sadowski, W. Bhimji, C. Tull, S.-Y. Oh, P. Baldi, et al. Revealing fundamental physics from the daya bay neutrino experiment using deep neural networks. In *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on*, pages 892–897. IEEE, 2016.

[75] T. Rainforth. Nesting probabilistic programs. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2018.

[76] T. Rainforth, R. Cornish, H. Yang, A. Warrington, and F. Wood. On nesting Monte Carlo estimators. In *International Conference on Machine Learning (ICML)*, 2018.

[77] D. J. Rezende and S. Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[78] D. Schouten, A. DeAbreu, and B. Stelzer. Accelerated Matrix Element Method with Parallel Computing. *Comput. Phys. Commun.*, 192:54–59, 2015.

[79] T. Sjöstrand, S. Mrenna, and P. Skands. Pythia 6.4 physics and manual. *Journal of High Energy Physics*, 2006(05):026, 2006.

[80] D. E. Soper and M. Spannowsky. Finding physics signals with shower deconstruction. *Phys. Rev.*, D84:074002, 2011.

[81] M. Sunnåker, A. G. Busetto, E. Numminen, J. Corander, M. Foll, and C. Dessimoz. Approximate Bayesian computation. *PLoS Computational Biology*, 9(1):e1002803, 2013.

[82] D. Tran, M. W. Hoffman, D. Moore, C. Suter, S. Vasudevan, and A. Radul. Simple, distributed, and accelerated probabilistic programming. In *Advances in Neural Information Processing Systems*, pages 7598–7609, 2018.

[83] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[84] D. Tran, R. Ranganath, and D. Blei. Hierarchical implicit models and likelihood-free variational inference. In *Advances in Neural Information Processing Systems*, pages 5523–5533, 2017.

[85] B. Uria, M.-A. Côté, K. Gregor, I. Murray, and H. Larochelle. Neural autoregressive distribution estimation. *Journal of Machine Learning Research*, 17(205):1–37, 2016.

[86] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An Introduction to Probabilistic Programming. *arXiv e-prints*, Sep 2018.

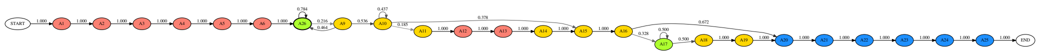
[87] R. D. Wilkinson. Approximate Bayesian computation (ABC) gives exact results under the assumption of model error. *Statistical Applications in Genetics and Molecular Biology*, 12(2):129–141.

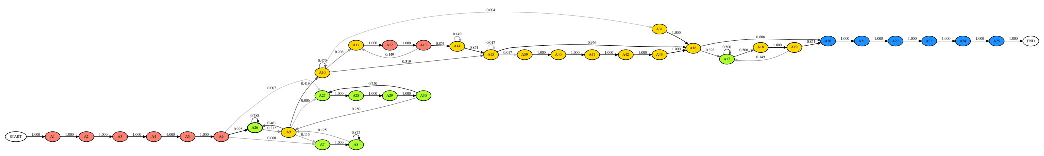
[88] D. Williams. *Probability with Martingales*. Cambridge University Press, 1991.

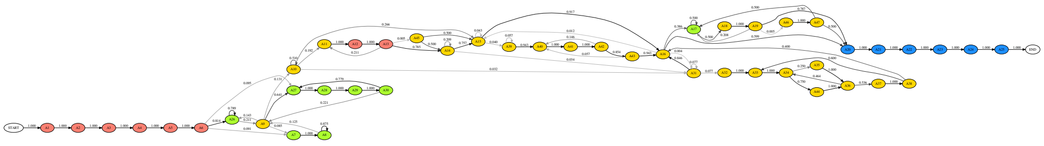
[89] D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

[90] F. Wood, J. W. Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model (Supplementary Material)


(a) Latent probabilistic structure of the 10 most frequent trace types.


(b) Latent probabilistic structure of the 25 most frequent traces types.


(c) Latent probabilistic structure of the 100 most frequent traces types.

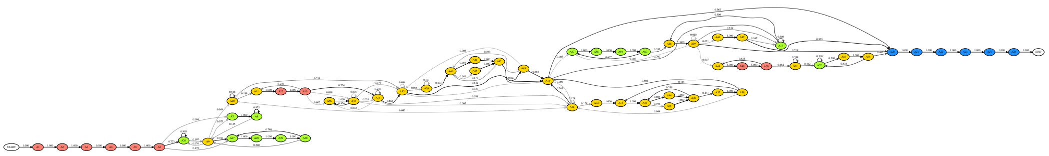
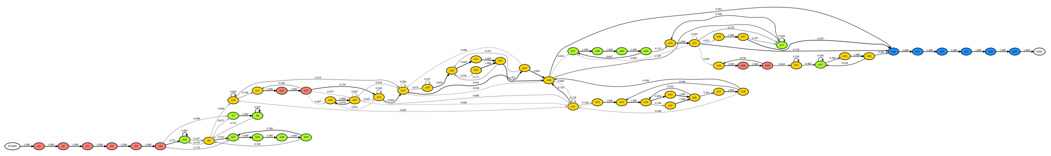

(d) Latent probabilistic structure of the 250 most frequent traces types.

Figure 3: Interpretability of the latent structure of the τ lepton decay process, automatically extracted from SHERPA executions via the probabilistic execution protocol. Showing model structure with increasing detail by taking an increasing number of most common trace types into account. Node labels denote address IDs (A1, A2, etc.) that correspond to uniquely identifiable parts of model execution such as those in Table 1. Addresses A1, A2, A3 correspond to momenta p_x, p_y, p_z, and A6 corresponds to the decay channel. Edge labels denote the frequency an edge is taken, normalized per source node. *Red:* controlled; *green*: rejection sampling; *blue*: observed; *yellow*: uncontrolled. *Note*: the addresses in these graphs are "aggregated", meaning that we collapse all instances i_t of addresses (a_t, i_t) into the same node in the graph, i.e., representing loops in the execution as cycles in the graph, in order to simplify the presentation. This gives us $\leq$60 aggregated addresses representing the transitions between a total of approximately 24k addresses (a_t, i_t) in the simulator.


(a) Prior execution $p(\mathbf{x}, \mathbf{y})$.

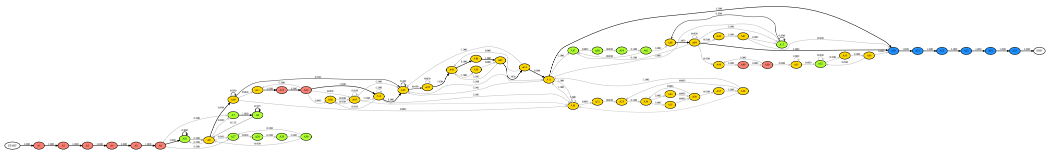

(b) Posterior execution $p(\mathbf{x}|\mathbf{y})$ conditioned on a given calorimeter observation $\mathbf{y}$.

Figure 4: Interpretability of the latent probabilistic structure of the τ lepton decay simulator code, automatically extracted from 10,000 SHERPA executions via the probabilistic execution protocol. The flow is probabilistic at the shown nodes and deterministic along the edges. Edge labels denote the frequency an edge is taken, normalized per source node. *Red:* controlled; *green*: rejection sampling; *blue*: observed; *yellow*: uncontrolled. *Note*: the addresses in these graphs are "aggregated", meaning that we collapse all instances i_t of addresses (a_t, i_t) into the same node in the graph, i.e., representing loops in the execution as cycles in the graph, in order to simplify the presentation. This gives us $\leq$60 aggregated addresses representing the transitions between a total of approximately 24k addresses (a_t, i_t) in the simulator.

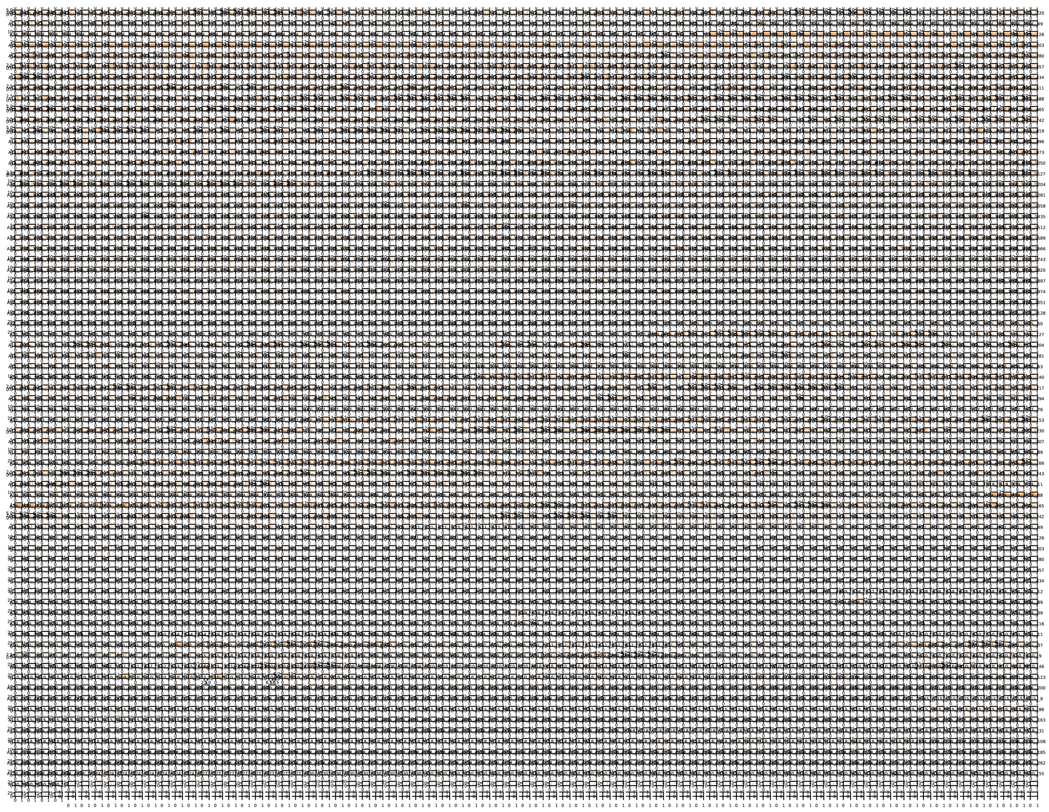


Figure 5: Example posterior over the entire latent state of the SHERPA simulator, conditioned on a single observed calorimeter. For the observation used, the posteriors presented in this figure contain approximately 6k addresses out of a total of approximately 24k addresses in the whole simulator. The histograms shown in Figure 2 are only a subset of this collection. *Note*: presenting this many plots in a single figure is challenging and a better plotting code is pending.

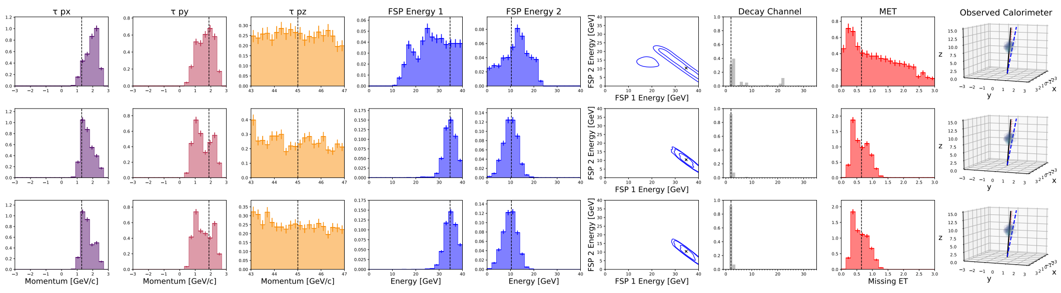


Figure 6: Steps of constructing the IC posterior for a Channel 2 GT event ($\tau \to \nu_\tau \pi^-$, first test case in Figure 2). The IC proposal (top row) is produced by the trained inference network. It is then weighted using Equation 3, giving IC posterior (middle row). The corresponding true posterior from RMH (MCMC) baseline is given below (bottom row). Note that the shown variables are just a subset of the full latent variables available in each case.

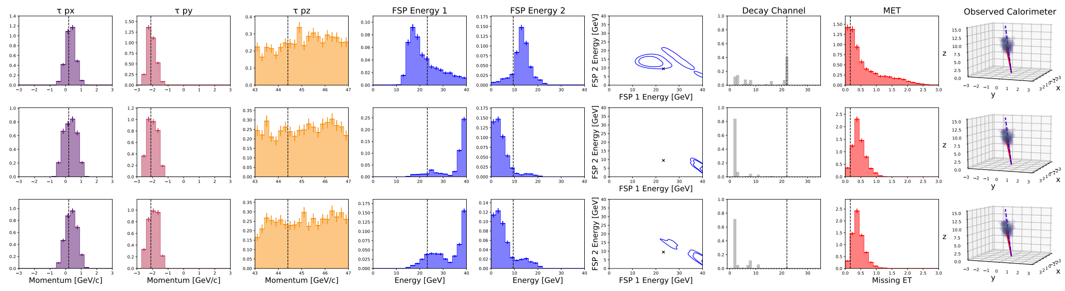
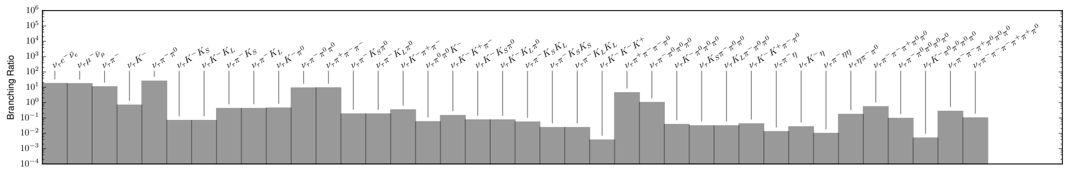


Figure 7: Steps of constructing the IC posterior for a Channel 22 GT event ($\tau \to \nu_\tau K^- K^- K^+$, last test case in Figure 2). The IC proposal (top row) is produced by the trained inference network. It is then weighted using Equation 3, giving IC posterior (middle row). The corresponding true posterior from RMH (MCMC) baseline is given below (bottom row). Note that the shown variables are just a subset of the full latent variables available in each case. The effect of "prior inflation" can be seen in the proposal mode of Channel 22 which the NN proposes as the most likely (i.e., mode of the proposal). However after importance weighting the IC posterior matches the true posterior from RMH (MCMC) where Channel 22 has very low (but non-zero) posterior probability due to the prior model.



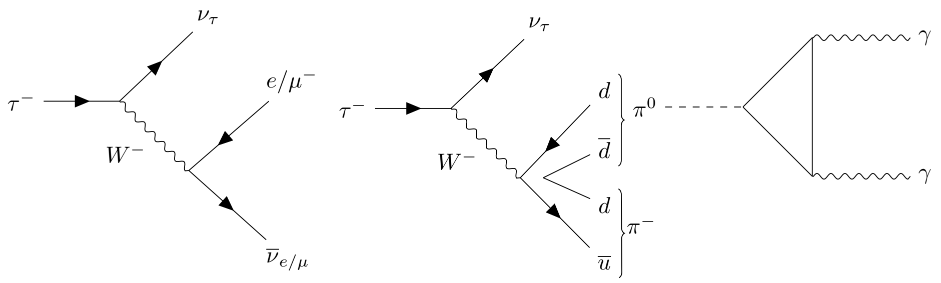


Figure 8: *Top:* branching ratios of the τ lepton, effectively the prior distribution of the decay channels in SHERPA. Note that the scale is logarithmic. *Bottom:* Feynman diagrams for τ decays illustrating that these can produce multiple detected particles.

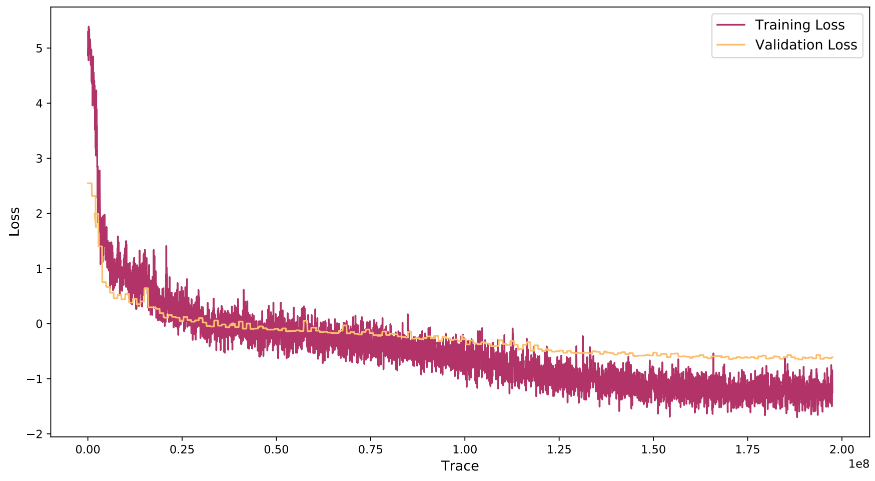


Figure 9: Training and validation losses of the IC inference NNs used for the results presented in Section 5. The network has 143,485,048 parameters and has been trained for 40 epochs. Network configuration: an LSTM with 512 hidden units; an observation embedding of size 256, encoded with a 3D convolutional NN (CNN) [64] with layer configuration Conv3D(1, 64, 3)–Conv3D(64, 64, 3)–MaxPool3D(2)–Conv3D(64, 128, 3)–Conv3D(128, 128, 3)–Conv3D(128, 128, 3)– MaxPool3D(2)–FC(2048, 256). We use previous sample embeddings of size 4 given by single-layer NNs, and address embeddings of size 64. The proposal layers are two-layer NNs, the output of which are either a mixture of ten truncated normal distributions [21] (for uniform continuous priors) or a categorical distribution (for categorical priors). We use ReLU nonlinearities in all NN components.

Table 1: Examples of addresses in the τ lepton decay problem in SHERPA (C++). Only the first 6 addresses are shown out of a total of 24,382 addresses encountered over 1,602,880 executions to collect statistics.

Address ID	Full address
A1	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x45f; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0xf9]_Uniform_1
A2	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x477; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0xf9]_Uniform_1
A3	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x48f; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0xf9]_Uniform_1
A4	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x8f4; ATOOLS:: Particle:: SetTime()+0xd; ATOOLS:: Flavour:: GenerateLifeTime() const+0x35; ATOOLS:: Random:: Get()+0x18b; probprog_RNG:: Get()+0xde]_Uniform_1
A5	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x982; SHERPA:: Event_Handler:: IterateEventPhases(SHERPA:: eventtype:: code&, double&)+0x1d2; SHERPA:: Hadron_Decays:: Treat(ATOOLS:: Blob_List*, double&)+0x975; SHERPA:: Decay_Handler_Base:: TreatInitialBlob(ATOOLS:: Blob*, METOOLS:: Amplitude2_Tensor*, std:: vector<ATOOLS:: Particle*, std:: allocator<ATOOLS:: Particle*> > const&)+0x1ab1; SHERPA:: Hadron_Decay_Handler:: CreateDecayBlob(ATOOLS:: Particle*)+0x4cd; PHASIC:: Decay_Table:: Select() const+0x76e; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0xf9]_Uniform_1
A6	[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&)+0x982; SHERPA:: Event_Handler:: IterateEventPhases(SHERPA:: eventtype:: code&, double&)+0x1d2; SHERPA:: Hadron_Decays:: Treat(ATOOLS:: Blob_List*, double&)+0x975; SHERPA:: Decay_Handler_Base:: TreatInitialBlob(ATOOLS:: Blob*, METOOLS:: Amplitude2_Tensor*, std:: vector<ATOOLS:: Particle*, std:: allocator<ATOOLS:: Particle*> > const&)+0x1ab1; SHERPA:: Hadron_Decay_Handler:: CreateDecayBlob(ATOOLS:: Particle*)+0x4cd; PHASIC:: Decay_Table:: Select() const+0x9d7; ATOOLS:: Random:: GetCategorical(std:: vector<double, std:: allocator<double> > const&, bool, bool)+0x1a5; probprog_RNG:: GetCategorical(std:: vector<double, std:: allocator<double> > const&, bool, bool)+0x111]_Categorical(length_categories:38)_1

Statement of Authorship for joint/multi-authored papers for PGR thesis
To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model
Publication Status	[✓ Published] ◻ Accepted for Publication ◻Submitted for Publication ◻Unpublished and unsubmitted work written in a manuscript style
Publication Details	Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, **Bradley Gram-Hansen**, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, Frank Wood, Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model, In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS), 2019.

Student Confirmation

Student Name:	Bradley Gram-Hansen		
Contribution to the Paper	The project was a large collaborative project between many different organisations due to the large-scale nature of the project. In regards to my contribution to these papers, I, along with Atlium Gunes Baydin developed the probabilistic programming execution (PPX) protocols that allowed arbitrary simulators to be connected to a probabilistic programming system, PyProb, central to the success of both papers. I participated in the general discussions regarding the construction of the idea and contributed to the writing of the papers and figures 3 and 4. I also contributed to the writing of the paper and a small part of the PyProb source code – this was already an established project.		
Signature [illegible]		Date	12/02/21

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Philip H.S. Torr		
Supervisor comments		
Signature Philip Torr	Date	26/04/2021

This completed form should be included in the thesis, at the end of the relevant chapter.

Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale

Atılım Güneş Baydin
University of Oxford

Lei Shao
Intel Corporation

Wahid Bhimji
Lawrence Berkeley National Laboratory

Lukas Heinrich
CERN

Bradley Gram-Hansen
University of Oxford

Lawrence Meadows
Intel Corporation

Jialin Liu
Lawrence Berkeley National Laboratory

Andreas Munk
University of British Columbia

Saeid Naderiparizi
University of British Columbia

Gilles Louppe
University of Liège

Mingfei Ma
Intel Corporation

Xiaohui Zhao
Intel Corporation

Philip Torr
University of Oxford

Victor Lee
Intel Corporation

Kyle Cranmer
New York University

Prabhat
Lawrence Berkeley National Laboratory

Frank Wood
University of British Columbia

ABSTRACT

Probabilistic programming languages (PPLs) are receiving widespread attention for performing Bayesian inference in complex generative models. However, applications to science remain limited because of the impracticability of rewriting complex scientific simulators in a PPL, the computational cost of inference, and the lack of scalable implementations. To address these, we present a novel PPL framework that couples directly to existing scientific simulators through a cross-platform probabilistic execution protocol and provides Markov chain Monte Carlo (MCMC) and deep-learning-based inference compilation (IC) engines for tractable inference. To guide IC inference, we perform distributed training of a dynamic 3DCNN–LSTM architecture with a PyTorch-MPI-based framework on 1,024 32-core CPU nodes of the Cori supercomputer with a global minibatch size of 128k: achieving a performance of 450 Tflop/s through enhancements to PyTorch. We demonstrate a Large Hadron Collider (LHC) use-case with the C++ Sherpa simulator and achieve the largest-scale posterior inference in a Turing-complete PPL.

KEYWORDS

probabilistic programming, simulation, inference, deep learning

ACM Reference Format:
Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Bradley Gram-Hansen, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3295500.3356180

1 INTRODUCTION

Probabilistic programming [71] is an emerging paradigm within machine learning that uses general-purpose programming languages to express probabilistic models. This is achieved by introducing statistical conditioning as a language construct so that inverse problems can be expressed. Probabilistic programming languages (PPLs) have semantics [67] that can be understood as Bayesian inference [13, 24, 26]. The major challenge in designing useful PPL systems is that language evaluators must solve arbitrary, user-provided inverse problems, which usually requires general-purpose inference algorithms that are computationally expensive.

In this paper we report our work that enables, for the first time, the use of *existing* stochastic simulator code as a probabilistic program in which one can do fast, repeated (amortized) Bayesian inference; this enables one to predict the distribution of input parameters and all random choices in the simulator from an observation of its output. In other words, given a simulator of a generative process in the forward direction (inputs→outputs), our technique can provide the reverse (outputs→inputs) by predicting the whole latent state of the simulator that could have given rise to an observed instance of its output. For example, using a particle physics simulation we can get distributions over the particle properties and decays within the simulator that can give rise to a collision event observed in a detector, or, using a spectroscopy simulator we can determine the elemental matter composition and dispersions within the simulator explaining an observed spectrum. In fields where accurate simulators of real-world phenomena exist, our technique enables the interpretable explanation of real observations under the structured model defined by the simulator code base.

We achieve this by defining a probabilistic programming execution protocol that interfaces with existing simulators at the sites of random number draws, without altering the simulator's structure and execution in the host system. The random number draws are routed through the protocol to a PPL system which treats these as samples from corresponding prior distributions in a Bayesian setting, giving one the capability to record or guide the execution of the simulator to perform inference. Thus we generalize existing simulators as probabilistic programs and make them subject to inference under general-purpose inference engines.

Inference in the probabilistic programming setting is performed by sampling in the space of execution traces, where a single sample (an execution trace) represents a full run of the simulator. Each execution trace itself is composed of a potentially unbounded sequence of addresses, prior distributions, and sampled values, where an address is a unique label identifying each random number draw. In other words, we work with empirical distributions over simulator executions, which entails unique requirements on memory, storage, and computation that we address in our implementation. The addresses comprising each trace give our technique the unique ability to provide direct connections to the simulator code base for any predictions at test time, where the simulator is no longer used as a black box but as a highly structured and interpretable probabilistic generative model that it implicitly represents.

Our PPL provides inference engines from the Markov chain Monte Carlo (MCMC) and importance sampling (IS) families. MCMC inference guarantees closely approximating the true posterior of the simulator, albeit with significant computational cost due to its sequential nature and the large number of iterations one needs to accumulate statistically independent samples. Inference compilation (IC) [47] addresses this by training a dynamic neural network to provide proposals for IS, leading to fast amortized inference.

We name this project "Etalumis", the word "simulate" spelled backwards, as a reference to the fact that our technique essentially inverts a simulator by probabilistically inferring all choices in the simulator given an observation of its output. We demonstrate this by inferring properties of particles produced at the Large Hadron Collider (LHC) using the Sherpa[1] [29] simulator.

1.1 Contributions

Our main contributions are:

- A novel PPL framework that enables execution of existing stochastic simulators under the control of general-purpose inference engines, with HPC features including handling multi-TB data and distributed training and inference.
- The largest scale posterior inference in a Turing-complete PPL, where our experiments encountered approximately 25,000 latent variables[2] expressed by the existing Sherpa simulator code base of nearly one million lines of code in C++ [29].
- Synchronous data parallel training of a dynamic 3DCNN–LSTM neural network (NN) architecture using the PyTorch [61] MPI framework at the scale of 1,024 nodes (32,768 CPU cores) with a global minibatch size of 128k. To our knowledge this is the largest scale use of PyTorch's builtin MPI functionality,[3] and the largest minibatch size used for this form of NN model.

[1]https://gitlab.com/sherpa-team/sherpa

2 PROBABILISTIC PROGRAMMING FOR PARTICLE PHYSICS

Particle physics seeks to understand particles produced in collisions at accelerators such at the LHC at CERN. Collisions happen millions of times per second, creating cascading particle decays, observed in complex instruments such as the ATLAS detector [2], comprising millions of electronics channels. These experiments analyze the vast volume of resulting data and seek to reconstruct the initial particles produced in order to make discoveries including physics beyond the current Standard Model of particle physics [28][73][63][72].

The Standard Model has a number of parameters (e.g., particle masses), which we can denote θ, describing the way particles and fundamental forces act in the universe. In a given collision at the LHC, with initial conditions denoted E, we observe a cascade of particles interact with particle detectors. If we denote *all* of the random "choices" made by nature as $\mathbf{x}$, the Standard Model describes, generatively, the conditional probability $p(\mathbf{x}|E,\theta)$, that is, the distribution of all choices $\mathbf{x}$ as a function of initial conditions E and model parameters θ. Note that, while the Standard Model can be expressed symbolically in mathematical notation [32, 62], it can also be expressed computationally as a stochastic simulator [29], which, given access to a random number generator, can draw samples from $p(\mathbf{x})$.[4] Similarly, a particle detector can be modeled as a stochastic simulator, generating samples from $p(\mathbf{y}|\mathbf{x})$, the likelihood of observation $\mathbf{y}$ as a function of $\mathbf{x}$.

In this paper we focus on a real use-case in particle physics, performing experiments on the decay of the τ (tau) lepton. This is under active investigation by LHC physicists [4] and important to uncovering properties of the Higgs boson. We use the state-of-the-art Sherpa simulator [29] for modeling τ particle creation in LHC collisions and their subsequent decay into further particles (the stochastic events $\mathbf{x}$ above), coupled to a fast 3D detector simulator for the detector observation $\mathbf{y}$.

Current methods in the field include performing classification and regression using machine learning approaches on low dimensional distributions of derived variables [4] that provide point-estimates without the posterior of the full latent state nor the deep interpretability of our approach. Inference of the latent structure has only previously been used in the field with drastically simplified models of the process and detector [43] [3].

PPLs allow us to express inference problems such as: given an actual particle detector observation $\mathbf{y}$, what sequence of choices $\mathbf{x}$ are likely to have led to this observation? In other words, we would

[2]Note that the simulator defines an unlimited number of random variables because of the presence of rejection sampling loops.
[3]Personal communication with PyTorch developers.
[4]Dropping the dependence on E and θ because everything in this example is conditionally dependent on these quantities.

like to find $p(\mathbf{x}|\mathbf{y})$, the distribution of $\mathbf{x}$ as a function of $\mathbf{y}$. To solve this inverse problem via conditioning requires invoking Bayes rule

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}, \mathbf{x})}{p(\mathbf{y})} = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{\int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x}}$$

where the posterior distribution of interest, $p(\mathbf{x}|\mathbf{y})$, is related to the composition of the two stochastic simulators in the form of the joint distribution $p(\mathbf{y}, \mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ renormalized by the marginal probability, or evidence of the data, $p(\mathbf{y}) = \int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x}$. Computing the evidence requires summing over all possible paths that the simulation can take. This is a large number of possible paths; in most models this is a quantity that is impossible to compute in polynomial time. In practice PPLs approximate the posterior $p(\mathbf{x}|\mathbf{y})$ using sampling-based inference engines that sidestep the integration problem but remain computationally intensive. This specifically is where probabilistic programming meets, for the first time in this paper, high-performance computing.

3 STATE OF THE ART

3.1 Probabilistic programming

Within probabilistic programming, recent advances in computational hardware have made it possible to distribute certain types of inference processes, enabling inference to be applied to problems of real-world relevance [70]. By parallelizing computation over several cores, PPLs have been able to perform large-scale inference on models with increasing numbers of observations, such as the cause and effect analysis of 1.6×10^9 genetic measurements [30, 70], spatial analysis of 1.5×10^4 shots from 308 NBA players [18], exploratory analysis of 1.7×10^6 taxi trajectories [36], and probabilistic modeling for processing hundreds-of-thousands of Xbox live games per day to rank and match players fairly [33, 55].

In all these large-scale programs, despite the number of observations being large, model sizes in terms of the number of latent variables have been limited [36]. In contrast, to perform inference in a complex scientific model such as the Standard Model encoded by Sherpa requires handling thousands of latent variables, all of which need to be controlled within the program to perform inference in a scalable manner. To our knowledge, no existing PPL system has been used to run inference at the scale we are reporting in this work, and instances of distributed inference in existing literature have been typically restricted to small clusters [19].

A key feature of PPLs is that they decouple model specification from inference. A model is implemented by the user as a stand-alone regular program in the host programming language, specifying a generative process that produces samples from the joint prior distribution $p(\mathbf{y}, \mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ in each execution, that is, a forward model going from choices $\mathbf{x}$ to outcomes (observations) $\mathbf{y}$. The same program can then be executed using a variety of general-purpose inference engines available in the PPL system to obtain $p(\mathbf{x}|\mathbf{y})$, the inverse going from observations $\mathbf{y}$ to choices $\mathbf{x}$. Inference engines available in PPLs range from MCMC-based lightweight Metropolis Hastings (LMH) [74] and random-walk Metropolis Hastings (RMH) [46] algorithms to importance sampling (IS) [8] and sequential Monte Carlo [22]. Modern PPLs such as Pyro [11] and TensorFlow Probability [19, 70] use gradient-based inference engines including variational inference [36, 42] and Hamiltonian Monte Carlo [37, 57] that benefit from modern deep learning hardware and automatic differentiation [9] features provided by PyTorch [61] and TensorFlow [5] libraries. Another way of making use of gradient-based optimization is to combine IS with deep-learning-based proposals trained with data sampled from the probabilistic program, resulting in the IC algorithm [47, 49] in an amortized inference setting [25].

3.2 Distributed training for deep learning

To perform IC inference in Turing-complete PPLs in general, we would like to support the training of dynamic NNs whose runtime structure changes in each execution of the probabilistic model by rearranging NN modules corresponding to different addresses (unique random number draws) encountered [47] (Section 4.3). Moreover, depending on probabilistic model complexity, the NNs may grow in size if trained in an online setting, as a model can represent a potentially unbounded number of random number draws. In addition to these, the volume of training data required is large, as the data keeps track of all execution paths within the simulator. To enable rapid prototyping, model evaluation, and making use of HPC capacity, scaling deep learning training to multiple computation units is highly desirable [38, 44, 45, 51, 52].

In this context there are three prominent parallelism strategies: data- and model-parallelism, and layer pipelining. In this project we work in a data-parallel setting where different nodes train the same model on different subsets of data. For such training, there are synchronous- and asynchronous-update approaches. In synchronous update [16, 59], locally computed gradients are summed across the nodes at the same time with synchronization barriers for parameter update. In asynchronous update [17, 58, 77], one removes the barrier so that nodes can independently contribute to a parameter server. Although synchronous update can entail challenges due to straggler effects [15, 69], it has desirable properties in terms of convergence, reproducibility, and ease of debugging. In this work, given the novelty of the probabilistic techniques we are introducing and the need to fully understand and compare trained NNs without ambiguity, we employ synchronous updates.

In synchronous updates, large global minibatches can make convergence challenging and hinder test accuracy. Keskar et al. [39] pointed out large-minibatch training can lead to sharp minima and a generalization gap. Other work [31, 76] argues that the difficulties in large-minibatch training are optimization related and can be mitigated with learning rate scaling [31]. You et al. [76] apply layer-wise adaptive rate scaling (LARS) to achieve large-minibatch-size training of a Resnet-50 architecture without loss of accuracy, and Ginsburg et al. [27] use layer-wise adaptive rate control (LARC) to improve training stability and speed. Smith et al. [65] have proposed to increase the minibatch size instead of decaying the learning rate, and more recent work [53, 64] showed relationships between gradient noise scale (or training steps) and minibatch size. Through such methods, distributed training has been scaled to many thousands of CPUs or GPUs [44, 45, 51, 54]. While we take inspiration from these recent approaches, our dynamic NN architecture and training data create a distinct training setting which requires appropriate innovations, as discussed in Section 4.3.

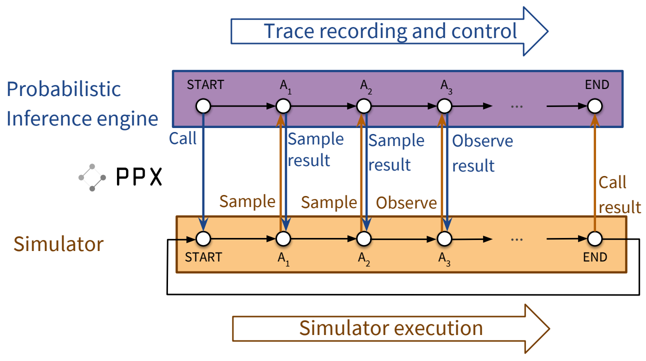


Figure 1: The probabilistic execution protocol (PPX). *Sample* and *observe* statements correspond to random number draws and conditioning, respectively.

4 INNOVATIONS

4.1 PPX and pyprob: executing existing simulators as probabilistic programs

One of our main contributions in Etalumis is the development of a probabilistic programming execution protocol (PPX), which defines a cross-platform API for the execution and control of stochastic simulators[5] (Figure 1). The protocol provides language-agnostic definitions of common probability distributions and message pairs covering the call and return values of: (1) program entry points; (2) *sample* statements for random number draws; and (3) *observe* statements for conditioning. The purpose of this protocol is twofold:

- It allows us to record execution traces of a stochastic simulator as a sequence of *sample* and *observe* (conditioning) operations on random numbers, each associated with an address A_t. We can use these traces for tasks such as inspecting the probabilistic model implemented by the simulator, computing likelihoods, learning surrogate models, and generating training data for IC NNs.
- It allows us to control the execution of the simulator, at inference time, by making intelligent choices for each random number draw as the simulator keeps requesting random numbers. General-purpose PPL inference guides the simulator by making random number draws not from the prior $p(\mathbf{x})$ but from proposal distributions $q(\mathbf{x}|\mathbf{y})$ that depend on observed data $\mathbf{y}$ (Section 2).

PPX is based on flatbuffers,[6] a streamlined version of Google protocol buffers, providing bindings into C++, C#, Go, Java, JavaScript, PHP, Python, and TypeScript, enabling lightweight PPL front ends in these languages—in the sense of requiring the implementation of a simple intermediate layer to perform *sample* and *observe* operations over the protocol. We exchange PPX messages over ZeroMQ[7] [34] sockets, which allow communication between separate processes in the same machine (via inter-process sockets) or across a network (via TCP). PPX is inspired by the Open Neural Network Exchange (ONNX) project[8] allowing interoperability between major deep learning frameworks, and it allows the execution of any stochastic simulator under the control of any PPL system, provided that the necessary bindings are incorporated on both sides.

Using the PPX protocol as the interface, we implement two main components: (1) pyprob, a PyTorch-based PPL[9] in Python and (2) a C++ binding to the protocol to route the random number draws in Sherpa to the PPL and therefore allow probabilistic inference in this simulator. Our PPL is designed to work with models written in Python and other languages supported through PPX. This is in contrast to existing PPLs such as Pyro [11] and TensorFlow Probability [19, 70] which do not provide a way to interface with existing simulators and require one to implement any model from scratch in the specific PPL.[10] We develop pyprob based on PyTorch [61], to utilize its automatic differentiation [9] infrastructure with support for dynamic computation graphs for IC inference.

4.2 Efficient Bayesian inference

Working with existing simulators as probabilistic programs restricts the class of inference engines that we can put to use. Modern PPLs commonly use gradient-based inference such as Hamiltonian Monte Carlo [57] and variational inference [36, 42] to approximate posterior distributions. However this is not applicable in our setting due to the absence of derivatives in general simulator codes. Therefore in pyprob we focus our attention on two inference engine families that can control Turing-complete simulators over the PPX protocol: MCMC in the RMH variety [46, 74], which provides a high-compute-cost sequential algorithm with statistical guarantees to closely approximate the posterior, and IS with IC [47], which does not require derivatives of the simulator code but still benefits from gradient-based methods by training proposal NNs and using these to significantly speed up IS inference.

It is important to note that the inference engines in pyprob work in the space of execution traces of probabilistic programs, such that a single sample from the inference engine corresponds to a full run of the simulator. Inference in this setting amounts to making adjustments to the random number draws, re-executing the simulator, and scoring the resulting execution in terms of the likelihood of the given observation. Depending on the specific observation and the simulator code involved, inference is computationally very expensive, requiring up to millions of executions in the RMH engine. Despite being very costly, RMH provides a way of sampling from the true posterior [56, 57], which is needed in initial explorations of any new simulator to establish correct posteriors serving as reference to confirm that IC inference can work correctly in the given setting. To establish the correctness of our inference results, we implement several MCMC convergence diagnostics. Autocorrelation measures the number of iterations one needs to get effectively independent samples in the same MCMC chain, which allows us to estimate how long RMH needs to run to reach a target effective sample size. The Gelman–Rubin metric, given multiple independent MCMC chains sampled from the same posterior, compares the variance of each chain to the pooled variance of all chains to statistically establish that we converged on the true posterior [24].

RMH comes with a high computational cost. This is because it requires a large number of initial samples to be generated that are

[5]https://github.com/probprog/ppx
[6]http://google.github.io/flatbuffers/
[7]http://zeromq.org/
[8]https://onnx.ai/
[9]https://github.com/probprog/pyprob
[10]We are planning to provide PPX bindings for these PPLs in future work.

then discarded, of the order $\sim 10^6$ for the Sherpa model we present in this paper. This is required to find the posterior density, which, as the model begins from an arbitrary point of the prior, can be very far from the starting region. Once this “burn-in” stage is completed the MCMC chain should be sampling from within the region containing the posterior. In addition to this, the sequential nature of each chain limits our ability to parallelize the computation, again creating computational inefficiencies in the high-dimensional space of simulator execution traces that we work with in our technique.

In order to provide fast, repeated inference in a distributed setting, we implement the IC algorithm, which trains a deep recurrent NN to provide proposals for an IS scheme [47]. This works by running the simulator many times and therefore sampling a large set of execution traces from the simulator prior $p(\mathbf{x}, \mathbf{y})$, and using these to train a NN that represents $q(\mathbf{x}|\mathbf{y})$, i.e., informed proposals for random number draws $\mathbf{x}$ given observations $\mathbf{y}$, by optimizing the loss $\mathcal{L}(\phi) = \mathbb{E}_{p(\mathbf{y})}\left[D_{\mathrm{KL}}(p(\mathbf{x}|\mathbf{y})||q_\phi(\mathbf{x}|\mathbf{y}))\right] = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q_\phi(\mathbf{x}|\mathbf{y})\right] +$ const., where ϕ are NN parameters (Algorithm 1 and Figure 3) [47]. This phase of sampling the training data and training the NN is costly, but it needs to be performed only once for any given model. Once the proposal NN is trained to convergence, the IC inference engine becomes competitive in performance, which allows us to achieve a given effective sample size in the posterior $p(\mathbf{x}|\mathbf{y})$ using a fraction of the RMH computational cost. IC inference is embarrassingly parallel, where many instances of the same trained NN can be executed to run distributed inference on a given observation.

To further improve inference performance, we make several low-level improvements in the code base. The C++ front end of PPX uses concatenated stack frames of each random number draw as a unique address identifying a latent variable in the corresponding PPL model. Stack traces are obtained with the `backtrace(3)` function as instruction addresses and then converted to symbolic names using the `dladdr(3)` function [50]. The conversion is quite expensive, which prompted us to add a hash map to cache `dladdr` results, giving a 5x improvement in the production of address strings that are essential in our inference engines. The particle detector simulator that we use was initially coded to use the xtensor library[11] to implement the probability density function (PDF) of multivariate normal distributions in the general case, but was exclusively called on 3D data. This code was replaced by a scalar-based implementation limited to the 3D case, resulting in a 13x speed-up in the PDF, and a 1.5x speed-up of our simulator pipeline in general. The bulk of our further optimizations focus on the NN training for IC inference and are discussed in the next sections.

4.3 Dynamic neural network architecture

The NN architecture used in IC inference is based on a LSTM [35] recurrent core that gets executed as many time steps as the simulator’s probabilistic trace length (Figure 3). To this core NN, various other NN components get attached according to the series of addresses A_t executed in the simulator. In other words, we construct a dynamic NN whose runtime structure changes in each execution trace, implemented using the dynamic computation graph infrastructure in PyTorch. The input to this LSTM in each time step is a concatenation of embeddings of the observation, the current address in

[11]https://xtensor.readthedocs.io

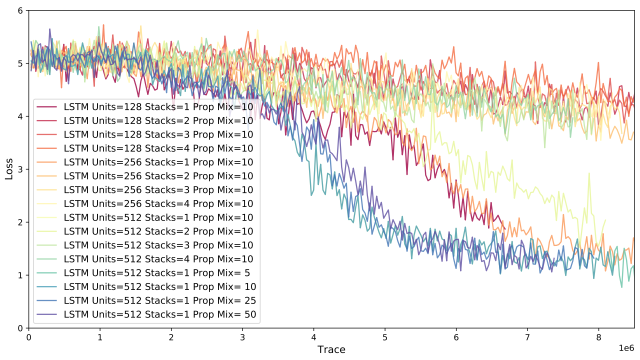


Figure 2: Loss curves for NN architectures considered in the hyperparameter search detailed in the text.

the simulator, and the previously sampled value. The observation embedding is a NN specific to the observation domain. Address embeddings are learned vectors representing the identity of random choices A_t in the simulator address space. Sample embeddings are address-specific layers encoding the value of the random draw in the previous time step. The LSTM output, at each time step, is fed into address-specific proposal layers that provide the final output of the NN for IC inference: proposal distributions $q(\mathbf{x}|\mathbf{y})$ to use for each address A_t as the simulator keeps running and requesting new random numbers over the PPX protocol (Section 4.1).

For the Sherpa experiments reported in this paper, we work with 3D observations of size 35x35x20, representing particle detector voxels. To tune NN architecture hyperparameters, we search a grid of LSTM stacks in range {1, 4}, LSTM hidden units in the set {128, 256, 512}, and number of proposal mixture components in the set {5, 10, 25, 50} (Figure 2). We settle on the following architecture: an LSTM with 512 hidden units; an observation embedding of size 256, encoded with a 3D convolutional neural network (CNN) [48] acting as a feature extractor, with layer configuration Conv3D(1, 64, 3)–Conv3D(64, 64, 3)–MaxPool3D(2)–Conv3D(64, 128, 3)–Conv3D(128, 128, 3)–Conv3D(128, 128, 3)– MaxPool3D(2)–FC(2048, 256); previous sample embeddings of size 4 given by single-layer NNs; and address embeddings of size 64. The proposal layers are two-layer NNs, the output of which are either a mixture of ten truncated normal distributions [12] (for uniform continuous priors) or a categorical distribution (for categorical priors). We use ReLU nonlinearities in all NN components. All of these NN components except the LSTM and the 3DCNN are dependent on addresses A_t in the simulator, and these address-specific layers are created at the first encounter with a random number draw at a given address. Thus the number of trainable parameters in an IC NN is dependent on the size of the training data, because the more data gets used, the more likely it becomes to encounter new addresses in the simulator.

The pyprob framework is capable of operating in an “online” fashion, where NN training and layer generation happens using traces sampled by executing the simulator on-the-fly and discarding traces after each minibatch, or “offline”, where traces are sampled from the simulator and saved to disk as a dataset for further reuse (Algorithm 2). In our experiments, we used training datasets of 3M and 15M traces, resulting in NN sizes of 156,960,440 and 171,732,688

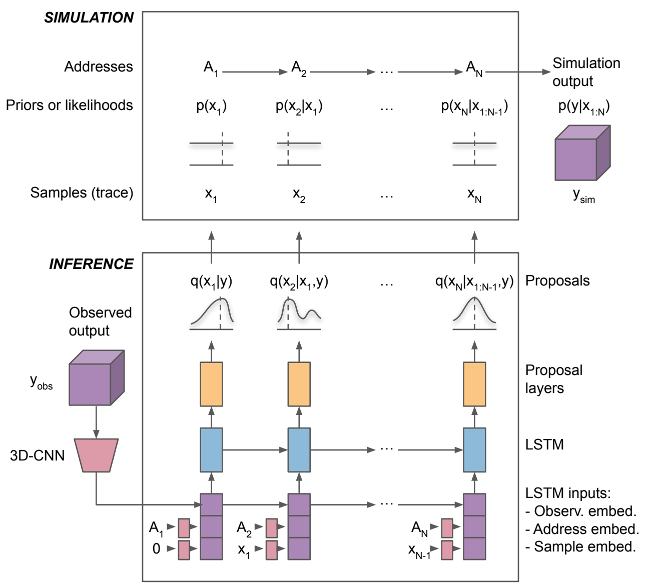


Figure 3: Simulation and inference. *Top:* model addresses, priors and samples. *Bottom:* IC inference engine proposals and NN architecture.

parameters respectively. All timing and scaling results presented in Sections 6.1 and 6.2 are performed with the larger network.

4.4 Training of dynamic neural networks

Scalable training of dynamic NNs we introduced in Section 4.3 pose unique challenges. Because of the address-dependent nature of the embedding and proposal layers of the overall IC NN, different nodes/ranks in a distributed training setting will work with different NN configurations according to the minibatch of training data they process at any given time. When the same NN is not shared across all nodes/ranks, it is not possible to rely on a generic allreduce operation for gradient averaging which is required for multi-node synchronous SGD. Inspired by neural machine translation (NMT) [75], in the offline training mode with training data saved on the disk, we implemented the option of pre-processing the whole dataset to pre-generate all embedding and proposal layers that a given dataset would imply to exist. Once layer pre-generation is done, the collection of all embedding and proposal layers are shared on each node/rank. In this way, for offline training, we have a globally shared NN representing the superset of all NN components each node needs to handle in any given minibatch, thus making it possible to scale training of the NN on multiple nodes.

Our allreduce-based training algorithm can also work in the online training setting, where training data is sampled from the simulator on-the-fly, if we freeze a globally shared NN and discard any subsequently encountered traces that contain addresses unknown at the time of NN architecture freezing. In future work, we intend to add a distributed open-ended implementation for online training to allow running without discarding, that will require the NN instances in each node/rank to grow with newly seen addresses.

Algorithm 1 Computing minibatch loss $\mathcal{L}_n$ of NN parameters ϕ

```
Require: Minibatch $\mathcal{D}_n$
  $L \leftarrow$ number of unique trace types found in $\mathcal{D}_n$
  Construct sub-minibatches $\mathcal{D}_n^l$, for $l = 1, \dots, L$
  $\mathcal{L}_n \leftarrow 0$
  for $l \in \{1, \dots, L\}$ do
    $\mathcal{L}_n \leftarrow \mathcal{L}_n - \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}_n^l} \log q_\phi(\boldsymbol{x}|\boldsymbol{y})$
  end for
  return $\mathcal{L}_n$
```

Algorithm 2 Distributed training with MPI backend. $p(\boldsymbol{x}, \boldsymbol{y})$ is the simulator and $\hat{G}(\boldsymbol{x}, \boldsymbol{y})$ is an offline dataset sampled from $p(\boldsymbol{x}, \boldsymbol{y})$

```
Require: OnlineData {True/False value}
Require: $B$ {Minibatch size}
  Initialize inference network $q_\phi(\boldsymbol{x}|\boldsymbol{y})$
  $N \leftarrow$ number of processes
  for all $n \in \{1, \dots, N\}$ do
    while Not Stop do
      if OnlineData then
        Sample $\mathcal{D}_n = \{(\boldsymbol{x},\boldsymbol{y})_1, \dots, (\boldsymbol{x},\boldsymbol{y})_B\}$ from $p(\boldsymbol{x},\boldsymbol{y})$
      else
        Get $\mathcal{D}_n = \{(\boldsymbol{x},\boldsymbol{y})_1, \dots, (\boldsymbol{x},\boldsymbol{y})_B\}$ from $\hat{G}(\boldsymbol{x},\boldsymbol{y})$
      end if
      Synchronize parameters ($\phi$) across all processes
      $\mathcal{L}_n \leftarrow -\frac{1}{B} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}_n} \log q_\phi(\boldsymbol{x}|\boldsymbol{y})$
      Calculate $\nabla_\phi \mathcal{L}_n$
      Call all_reduce s.t. $\nabla_\phi \mathcal{L} \leftarrow \frac{1}{N} \sum_{n=1}^N \nabla_\phi \mathcal{L}_n$
      Update $\phi$ using $\nabla_\phi \mathcal{L}$ with e.g. ADAM, SGD, LARC, etc.
    end while
  end for
```

4.4.1 Single node improvements to Etalumis. We profiled the Etalumis architecture with vtune, Cprofiler, and PyTorch autograd profiler, identifying data loading and 3D convolution as the primary computational hot-spots on which we focused our optimization efforts. We provide details on data loading and 3D convolution in the subsequent sections. In addition to these, execution traces from the Sherpa simulator have many different trace types (a unique sequence of addresses A_t, with different sampled values) with different rates of occurrence: in a given dataset, some trace types can be encountered thousands of times while others are seen only once. This is problematic because at training time we further divide each minibatch into "sub-minibatches" based on trace type, where each sub-minibatch can be processed by the NN in a single forward execution due to all traces being of the same type, i.e., sharing the same sequence of addresses A_t and therefore requiring the same NN structure (Algorithm 1). Therefore minibatches containing more than one trace type do not allow for effective parallelization and vectorization. In other words, unlike conventional NNs, the effective minibatch size is determined by the average size of sub-minibatches, and the more trace types we have within a minibatch, the slower the computation. To address this, we explored multiple methods to enlarge effective minibatch size, such as sorting traces, multi-bucketing, and selectively batching traces from the same trace type together in each minibatch. These options and their trade offs are described in more detail in Section 7.

4.4.2 Single node improvements to PyTorch. The flexibility of dynamic computation graphs and competitive speed of PyTorch have been crucial for this project. Optimizations were performed on code belonging to Pytorch stable release v1.0.0 to better support this project on Intel® Xeon® CPU platforms, focused in particular on 3D convolution operations making use of the MKL-DNN open source math library. MKL-DNN uses a direct convolution algorithm and for a 5-dimensional input tensor with layout {N, C, D, H, W}, it is reordered into a layout of {N, C, D, H, W, 8c} which is more amenable for SIMD vectorization.[12] The 3D convolution operator is vectorized on the innermost dimension which matches the 256-bit instruction length on AVX2, and parallelized on the outer dimensions. We also performed cache optimization to further improve performance. With these improvements we found the heavily used 3D convolution kernel achieved an 8x improvement on the Cori HSW platform.[13] The overall improvement on single node training time is given in Section 6.1. These improvements are made available in a fork of the official PyTorch repository.[14]

4.4.3 I/O optimization. I/O is challenging in many deep learning workloads partly due to random access patterns, such as those induced by shuffling, disturbing any pre-determined access order. In order to reduce the number of random access I/O operations, we developed a parallel trace sorting algorithm and pre-sorted the 15M traces according to trace type (Section 4.4.1). We further grouped the small trace files into larger files, going from 750 files with 20k traces per file to 150 files with 100k traces per file. With this grouping and sorting, we ensured that I/O requests follow a sequential access onto a contiguous file region which further improved the I/O performance. Metadata operations are also costly, so we enhanced the Python shelve module's file open/close performance with a caching mechanism, which allows concurrent access from different ranks to the same file.

Specific to our PPL setting, training data consists of execution traces that have a complex hierarchy, with each trace file containing many trace objects that consist of variable sequences of sample objects representing random number draws, which further contain variable length tensors, strings, integers, booleans, and other generic Python objects. PyTorch serialization with pickle is used to handle the complex trace data structure, but the pickle and unpickle overhead are very high. We developed a "pruning" function to shrink the data by removing non-necessary structures. We also designed a dictionary of simulator addresses A_t, which accumulates the fairly long address strings and assigns shorthand IDs that are used in serialization. This brought a 40% memory consumption reduction as well as large disk space saving.

For distributed training, we developed distributed minibatch sampler and dataset classes conforming to the PyTorch training API. The sampler first splits the sorted trace indices into minibatch-sized chunks, so that all traces in each minibatch are highly likely to be of the same type, then optionally groups these chunks into several buckets (Section 7.2). Within each bucket, the chunks are assigned with a round-robin algorithm to different ranks, such that each rank has roughly same distribution of workload. The distributed sampler enables us to scale the training at 1,024 nodes.

The sorting of traces and their grouping into minibatch chunks significantly improves the training speed (up to 50× in our experiments) by enabling all traces in a minibatch to be propagated through the NN in the same forward execution, in other words, decreasing the need for "sub-minibatching" (Section 4.4.1). This sorting and chunking scheme generates minibatches that predominantly contain a single trace type. However, the minibatches used at each iteration are sampled randomly without replacement from different regions of the sorted training set, and therefore contain different trace types, resulting in a gradient unbiased in expectation during any given epoch.

In our initial profiling, the cost of I/O was more than 50% of total run time. With these data re-structuring and parallel I/O optimizations, we reduced the I/O to less than 5%, achieving 10x speedup at different scales.

4.4.4 Distributed improvements to PyTorch MPI CPU code. PyTorch has a torch.distributed backend,[15] which allows scalable distributed training with high performance on both CPU and GPU clusters. Etalumis uses the MPI backend as appropriate for the synchronous SGD setting that we implement (Algorithm 2) and the HPC machines we utilize (Section 5). We have made various improvements to this backend to enable the large-scale distributed training on CPU systems required for this project. The call `torch.distributed.all_reduce` is used to combine the gradient tensors for all distributed MPI ranks. In Etalumis, the set of non-null gradient tensors differs for each rank and is a small fraction of the total set of tensors. Therefore we first perform an allreduce to obtain a map of all the tensors that are present on all ranks; then we create a list of the tensors, filling in the ones that are not present on our rank with zero; finally, we reduce all of the gradient tensors in the list. PyTorch `all_reduce` does not take a list of tensors so normally a list comprehension is used, but this results in one call to `MPI_Allreduce` for each tensor. We modified PyTorch `all_reduce` to accept a list of tensors. Then, in the PyTorch C++ code for allreduce, we concatenate small tensors into a buffer, call `MPI_Allreduce` on the buffer, and copy the results back to the original tensor. This eliminates almost all the allreduce latency and makes the communication bandwidth-bound.

We found that changing Etalumis to reduce only the non-null gradients gives a 4x improvement in allreduce time. Tensor concatenation improves overall performance by an additional 4% on one node which increases as nodes are added. With these improvements, the load balance effects discussed in Sections 6.2 and 7.2 are dominant and so are our primary focus of further distributed optimizations. Other future work could include performing the above steps for each backward layer with an asynchronous allreduce to overlap the communications for the previous layer with the computation for the current layer.

5 SYSTEMS AND SOFTWARE

5.1 Cori

We use the "data" partition of the Cori system at the National Energy Research Scientific Computing Center (NERSC) at Lawrence

[12]https://intel.github.io/mkl-dnn/understanding_memory_formats.html
[13]https://docs.nersc.gov/analytics/machinelearning/benchmarks/
[14]Intel-optimized PyTorch: https://github.com/intel/pytorch
[15]https://pytorch.org/docs/stable/distributed.html

Table 1: Intel®Xeon® CPU models and codes

Model	Code
E5-2695 v2 @ 2.40GHz (12 cores/socket)	IVB
E5-2698 v3 @ 2.30GHz (16 cores/socket)	HSW
E5-2697A v4 @ 2.60GHz (16 cores/socket)	BDW
Platinum 8170 @ 2.10GHz (26 cores/socket)	SKL
Gold 6252 @ 2.10GHz (24 cores/socket)	CSL

Berkeley National Laboratory. Cori is a Cray XC40 system, and the data partition features 2,388 nodes. Each node has two sockets and each socket is populated with a 16-core 2.3 GHz Intel® Xeon® E5-2698 v3 CPU (referred to as HSW from now on), with peak single-precision (SP) performance of 1.2 Tflop/s and 128 GB of DDR4-2133 DRAM. Nodes are connected via the Cray Aries low-latency, high-bandwidth interconnect utilizing the dragonfly topology. In addition, the Cori system contains 288 Cray DataWarp nodes (also known as the "Burst Buffer") which house the input datasets for the Cori experiments presented here. Each DataWarp node contains 2 × 3.2 TB SSDs, giving a system total of 1.8 PB of SSD storage, with up to 1.7 TB/sec read/write performance and over 28M IOP/s. Cori also has a Sonnexion 2000 Lustre filesystem, which consists of 248 Object Storage Targets (OSTs) and 10,168 disks, giving nearly 30 PB of storage and a maximum of 700 GB/sec IO performance. This filesystem is used for output files (networks and logs) for Cori experiments and both input and output for the Edison experiments.

5.2 Edison

We also make use of the Edison system at NERSC. Edison is a Cray XC30 system with 5,586 nodes. Each node has two sockets, each socket is populated with a 12-core 2.4 GHz Intel® Xeon® E5-2695 v2 CPU (referred to as IVB from now on), with peak performance of 460.8 SP Gflop/s, and 64 GB DDR3-1866 memory. Edison mounts the Cori Lustre filesystem described above.

5.3 Diamond cluster

In order to evaluate and improve the performance on newer Intel® processors we make use of the Diamond cluster, a small heterogeneous cluster maintained by Intel Corporation. The interconnect uses Intel® Omni-Path Architecture switches and host adapters. The nodes used for the results in this paper are all two socket nodes. Table 1 presents the CPU models used and the three letter abbreviations used in this paper.

5.4 Particle physics simulation software

In our experiments we use Sherpa version 2.2.3, coupled to a fast 3D detector simulator that we configure to use 20x35x35 voxels. Sherpa is implemented in C++, and therefore we use the C++ front end for PPX. We couple to Sherpa by a system-wide rerouting of the calls to the random number generator, which is made easy by the existence of a third-party random number generator interface (External_RNG) already present in Sherpa.

For this paper, in order to facilitate reproducible experiments, we run in the offline training mode and produce a sample of 15M traces that occupy 1.7 TB on disk. Generation of this 15M dataset was completed in 3 hours on 32 IVB nodes of Edison. The traces are stored using Python shelve[16] serialization, allowing random access to all entries contained in the collection of files with 100k traces in each file. These serialized files are accessed via the Python dbm module using the gdbm backend.

Table 2: Single node training throughput in traces/sec and flop rate (Gflop/s). 1-socket throughput and flop rate are for a single process while 2-socket is for 2 MPI processes on a single node.

Platform	1-socket traces/s	2-socket traces/s	1-socket Gflop/s (% peak)
IVB (Edison)	13.9	25.6	196 (43%)
HSW (Cori)	32.1	56.5	453 (38%)
BDW (Diamond)	30.5	57.8	430 (32%)
SKL (Diamond)	49.9	82.7	704 (20%)
CSL (Diamond)	51.1	93.1	720 (22%)

6 EXPERIMENTS AND RESULTS

6.1 Single node performance

We ran single node tests with one rank per socket for one and two ranks on the IVB nodes on Edison, the HSW partition of Cori and the BDW, SKL, and CSL nodes of the Diamond cluster. Table 2 shows the throughput and single socket flop rate and percentage of peak theoretical flop rate. We find that the optimizations described in Section 4.4.2 provide an improvement of 7x on the overall single socket run throughput (measured on HSW) relative to a default PyTorch version v1.0.0 installed via the official conda channel. We achieve **430 SP Gflop/s** on a single socket of the BDW system, measured using the available hardware counters for 256-bit packed SIMD single precision operations. This includes IO and is averaged over an entire 300k trace run. This can be compared to a theoretical peak flop rate for that BDW socket of 1,331 SP Gflop/s. Flop rates for other platforms are scaled from this measurement and given in Table 2. For further profiling we instrument the code with timers for each phase of the training (in order): minibatch read, forward, backward, and optimize. Figure 4 shows a breakdown of the time spent on a single socket after the optimizations described in Sections 4.4.1 and 4.4.2.[17]

6.2 Multi-node performance

In addition to the single socket operations, we time the two synchronization (allreduce) phases (gradient and loss). This information is recorded for each rank and each minibatch. Postprocessing finds the rank with the maximum work time (sum of the four phases mentioned in Section 6.1) and adds the times together. This gives the actual execution time. Further, we compute the *average* time across ranks for each work phase for each minibatch and add those together, giving the best time assuming no load imbalance. The results are shown in Figure 4. Comparing single socket results with 2 and 64 socket results shows the increased impact of load imbalance

[16]https://docs.python.org/3/library/shelve.html
[17]See disclaimers section after conclusions.

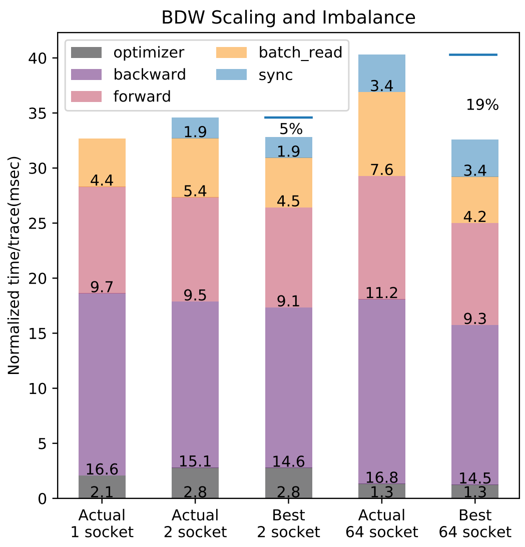


Figure 4: Actual and estimated best times for 1, 2, and 64 sockets. Horizontal bars at the top are to aid comparison between columns.

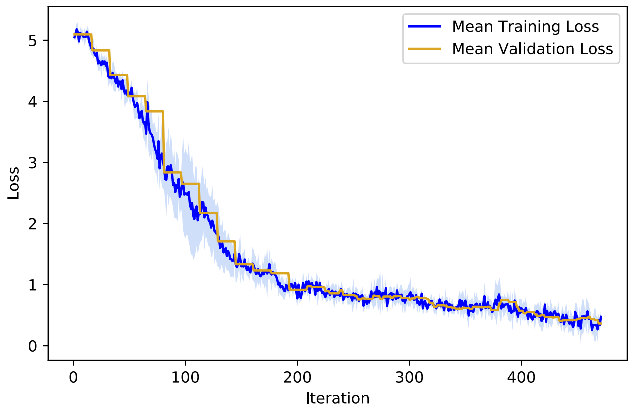


Figure 5: Mean loss and standard deviation (shaded) for five experiments with 128k minibatch size.

as nodes are added. This demonstrates a particular challenge of this project where the load on each node depends on the random minibatch of traces sampled for training on that node. The challenge and possible mitigation approaches we explored are discussed further in Section 7.2.

6.3 Large scale training on Cori and Edison

In order to choose training hyperparameters, we explored global minibatch sizes of {64, 2k, 32k, 128k}, and learning rates in the range $\left[10^{-7}, 10^{-1}\right]$ with grid search and compared the loss values after one epoch to find the optimal learning rate for different global minibatch sizes separately. For global minibatch sizes of 2k, 32k, and 128k, we trained with both Adam and Adam-LARC optimizers and compared loss value changes as a function of iterations. For training at 1,024 nodes we choose to use 32k and 128k global minibatch sizes. For the 128k minibatch size, best convergence was found using the Adam-LARC optimizer with a polynomial decay (order=2) learning rate schedule [76] that decays from an initial global learning rate of 5.70×10^{-4} to final 2×10^{-5} after completing 12 epochs for the dataset with 15M traces. In Figure 5, we show the mean and standard-deviation for five training runs with this 128k minibatch size and optimizer, demonstrating stable convergence.

Figure 6 shows weak scaling results obtained for distributed training to over a thousand nodes on both the Cori and Edison systems. We use a fixed local minibatch size of 64 per rank with 2 ranks per node, and plot the mean and standard deviation throughput for each iteration in terms of traces/s (labeled “average” in the plot). We also show the fastest iteration (labeled “peak”). The average scaling efficiency at 1,024 nodes is 0.79 on Edison and 0.5 on Cori. The throughput at 1,024 nodes on Cori and Edison is 28,000 and 22,000 traces/s on average, with the peak as 42,000 and 28,000 traces/s respectively. One can also see that there is some variation in this performance due to the different compute times taken to process execution traces of different length and the related load imbalance as discussed in Sections 6.2 and 7.2. We determine the maximum sustained performance over a 10-iteration sliding window to be **450 Tflop/s** on Cori and 325 Tflop/s on Edison.[18]

We have performed distributed training with global minibatch sizes of 32k and 128k at 1,024-node scale for extended periods to achieve convergence on both Cori and Edison systems. This is illustrated in Figure 7 where we show the loss for training and validation datasets as a function of iteration for an example run on Edison.

6.4 Inference and science results

Using our framework and the NNs trained using distributed resources at NERSC as described previously, we perform inference on test τ observation data that has not been used for training. As the approach of applying probabilistic programming in the setting of large-scale existing simulators is completely novel, there is no direct baseline in literature that provides the full posterior in each of these variables. Therefore we use our own MCMC (RMH)-based posterior as a baseline for the validation of the IC approach. We establish the convergence of the RMH posterior by running two independent MCMC chains with different initializations and computing the the Gelman–Rubin convergence metric [24] to confirm that they converge onto the same posterior distribution (Section 4.2).

Figure 8 shows a comparison of inference results from the RMH and IC approaches. We show selected latent variables (addresses) that are a small subset of the more than 24k addresses that were encountered in the prior space of the Sherpa experimental setup, but are of physics interest in that they correspond to properties of the τ particle. It can be seen that there is close agreement between the RMH and IC posterior distributions validating that our network has been adequately trained. We have made various improvements to the RMH inference processing rate but this form of inference is compute intensive and takes **115 hours** on a Edison IVB node to produce the 7.68M trace result shown. The corresponding 2M trace IC result completed in **30 mins** (achieving a 230× speedup

[18]See disclaimers section after conclusions.

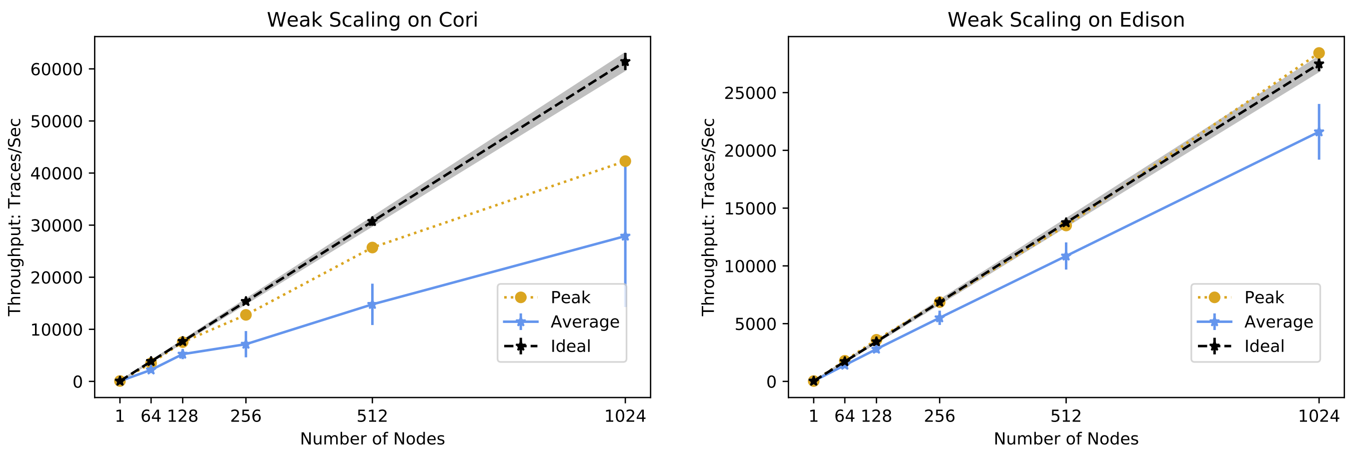


Figure 6: Weak scaling on Cori and Edison systems, showing throughput for different node counts with a fixed local minibatch size of 64 traces per MPI rank with 2 ranks per node. Average (mean) over all iterations and the peak single iteration are shown. Ideal scaling is derived from the mean single-node rate with a shaded uncertainty from the standard deviation in that rate.

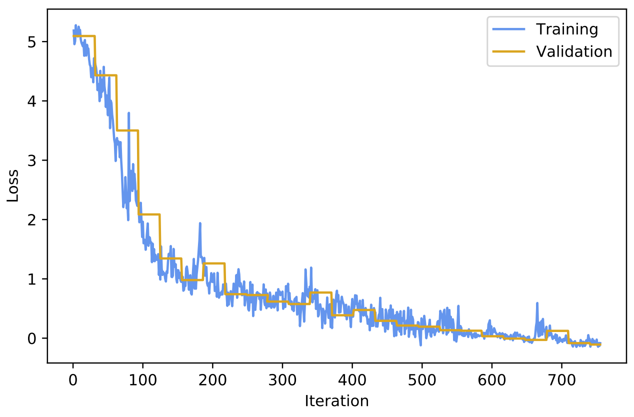


Figure 7: Training and validation loss for a 128k minibatch size experiment with the configuration described in the text run on 1,024 nodes of the Edison system.

for a comparable posterior result) on 24 HSW nodes, enabled by the parallelism of IC inference.

In addition to parallelization, a significant advantage of the IC approach is that it is amortized. This means that once the proposal NN is trained for any given model, it can be readily applied to large volumes of new collision data. Moreover IC inference runs with high effective sample sizes in comparison to RMH: each sample from the IC NN is an independent sample from the proposal distribution, which approaches the true posterior distribution with more training, whereas our autocorrelation measurements in the RMH posterior indicate that a very large number of iterations are needed to get statistically independent traces (on the order of $\sim 10^5$ for the type of decay event we use as the observation). These features of IC inference combine to provide a tractable approach for fast Bayesian inference in complex models implemented by large-scale simulators.

7 DISCUSSION

The dynamic NN architecture employed for this project has presented a number of unique challenges for distributed training, which we covered in Section 4.4. Our innovations proved successful in enabling the training of this architecture at scale, and in this section we capture some of the lessons learned and unresolved issues encountered in our experiments.

7.1 Time to solution: trade-off between throughput and convergence

7.1.1 Increasing effective local minibatch size. As mentioned in Section 4.4.1, the distributed SGD scheme given in Algorithms 1 and 2 uses random traces sampled from the simulator, and can suffer from slow training throughput if computation cannot be efficiently parallelized across the full minibatch due to the presence of different trace types. Therefore, we explored the following methods to improve effective minibatch size: sorting the traces before batching, online batching of the same trace types together, and multi-bucketing. Each of these methods can improve throughput, but risk introducing bias into SGD training or increasing the number of iterations to converge, so we compare the wall clock time for convergence to determine the relative trade-off. The impact of multi-bucketing is described and discussed in section 7.2 below. Sorting and batching traces from the same trace type together offers considerable throughput increase with a relatively small impact on convergence, when combined with shuffling of minibatches for randomness, so we used these throughput optimizations for the results in this paper.

7.1.2 Choice of optimizers, learning rate scaling and scheduling for convergence. There is considerable recent literature on techniques

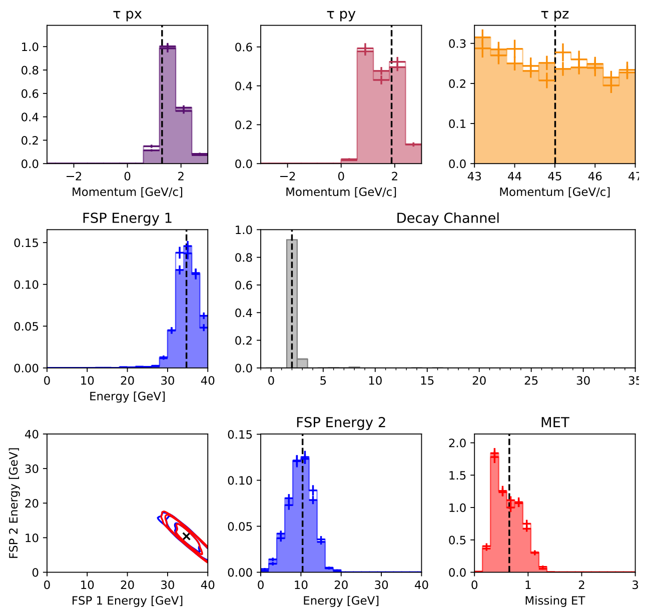


Figure 8: A comparison of posterior distributions obtained by RMH (filled histograms) and IC (outline histograms) and the ground truth values (dashed vertical lines) for a test τ decay observation. We show an illustrative subset of the latent variables, including x, y and z components of the τ-lepton momentum (top row), the energies of the two highest energy particles produced by the τ decay (middle left and bottom center), a contour plot showing correlation between these (bottom left), the τ decay channel ($\tau \to \pi\ \nu_\tau$ as mode) (middle right), and the missing transverse energy (bottom right).

for distributed and large-minibatch training, including scaling learning rates with number of nodes [31], alternative optimizers for large-minibatch-size training, and learning rate decay [27, 41, 76]. The network employed in this project presents a very different use-case to those considered in literature, prompting us to document our experiences here. For learning rate scaling with node count, we found sub-sqrt learning rate scaling works better than linear for an Adam-based optimizer [41]. We also compared Adam with the newer Adam-LARC optimizer [27, 76] for convergence performance and found the Adam-LARC optimizer to perform better for the very large global minibatch size 128k in our case. For smaller global minibatch sizes of 32K or lower, both plain Adam and Adam-LARC performed equally well. Finally, for the learning rate decay scheduler, we explored the following options: no decay, multi-step decay (per epoch), and polynomial decay of order 1 or 2 (calculated per iteration) [76]. We found that learning-rate decay can improve training performance and polynomial decay of order 2 provided the most effective schedule.

7.2 Load balancing

As indicated in Section 6.2, our work involves distinct scaling challenges due to the variation in compute time depending on execution trace length, address-dependent proposal and embedding layers, and representation of trace types inside each minibatch. These factors contribute to load imbalance. The trace length variation bears similarity to varying sequence lengths in NMT; however, unlike that case it is not possible to truncate the execution traces, and padding would introduce a cost to overall number of operations.

To resolve this load imbalance issue, we have explored a number of options, building on those from NMT, including a *multi-bucketing* scheme and a novel *dynamic batching* approach.

In *multi-bucketing* [10, 20, 40], traces are grouped into several buckets based on lengths, and every global minibatch is solely taken from a randomly picked bucket for every iteration. Multi-bucketing not only helps to balance the load among ranks for the same iteration, but also increases the effective minibatch size as traces from the same trace type have a higher chance to be in the same minibatch than in the non-bucketing case, achieving higher throughput. For a local minibatch-size of 16 with 10 buckets we measured throughput increases in the range of 30–60% at 128–256 node scale on Cori. However, our current multi-bucketing implementation does multiple updates in the same bucket continuously. When this implementation is used together with batching the traces from the same trace type together (as discussed above in Section 7.1.1), it negatively impacts convergence behavior. We believe this to be due to the fact that training on each specific bucket for multiple updates introduces over-fitting onto that specific subset of networks so moving to a new bucket for multiple updates causes information of the progress made with previous buckets to be lost. As such we did not employ this configuration for the results reported in this paper.

With *dynamic batching*, we replaced the requirement of fixed minibatch size per rank with a desired number of "tokens" per rank (where a token is a unit of random number draws in each trace), so that we can, for instance, allocate many short traces (with smaller number of tokens each) for one rank but only a few long traces for another rank, in order to balance the load for the LSTM network due to length variation. While an equal-token approach has been used in NMT, this did not offer throughput gains for our model, which has an additional 3DCNN component in which the compute time depends on the number of traces within the local minibatch, so if dynamic batching only considers total tokens per rank for the LSTM it can negatively impact the 3DCNN load.

Through these experiments we found that our current optimal throughput and convergence performance came from not employing these load-balancing schemes although we intend to explore modifications to these schemes as ongoing work.

8 SCIENCE IMPLICATIONS AND OUTLOOK

We have provided a common interface to connect PPLs with simulators written in arbitrary code in a broad range of programming languages. This opens up possibilities for future work in all applied fields where simulators are used to model real-world systems, including epidemiology modeling such as disease transmission and prevention models [66], autonomous vehicle and reinforcement learning environments [21], cosmology [7], and climate science

[68]. In particular, the ability to control existing simulators at scale and to generate interpretable posteriors is relevant to scientific domains where interpretability in model inference is critical.

We have demonstrated both MCMC- and IC-based inference of detector data originating from τ-decays simulated with the Sherpa Monte Carlo generator at scale. This offers, for the first time, the potential of Bayesian inference on the full latent structure of the large numbers of collision events produced at accelerators such as the LHC, enabling deep interpretation of observed events. For instance, ambiguity in the decay of a particle can be related exactly to the physics processes in the simulator that would give rise to that ambiguity. In order to fully realize this potential, future work will expand this framework to more complex particle decays (such as the Higgs decay to τ leptons) and incorporate a more detailed detector simulation (e.g., Geant4 [6]). We will demonstrate this on a full LHC physics analysis, reproducing the efficiency of point-estimates, together with the full posterior and intpretability, so that this can be exploited for discovery of new fundamental physics.

The IC objective is designed so that the NN proposal $q(\mathbf{x}|\mathbf{y})$ approximates the posterior $p(\mathbf{x}|\mathbf{y})$ asymptotically closely with more training. This costly training phase needs to be done *only once* for a given simulator-based model, giving us a NN that can provide samples from the model posterior in parallel for any new observed data. In this setting where have a fast, amortized $q(\mathbf{x}|\mathbf{y}) \approx p(\mathbf{x}|\mathbf{y})$, our ultimate goal is to add the machinery of Bayesian inference to the toolbox for critical tasks such as triggering [1] and event reconstruction by conditioning on potentially interesting events (e.g., $q(\text{ParticleType}|\cdot) \geq \epsilon$). Recent activity exploring the use of FPGAs for NN inference for particle physics [23] will help implementation of these approaches, and HPC systems will be crucial in the training and inference phases of such frameworks.

9 CONCLUSIONS

Inference in simulator-based models remains a challenging problem with potential impact across many disciplines [14, 60]. In this paper we present the first probabilistic programming implementation capable of controlling existing simulators and running at large-scale on HPC platforms. Through the PPX protocol, our framework successfully couples with large-scale scientific simulators leveraging thousands of lines of existing simulation code encoding domain-expert knowledge. To perform efficient inference we make use of the inference compilation technique, and we train a dynamic neural network involving LSTM and 3DCNN components, with a large global minibatch size of 128k. IC inference achieved a 230× speedup compared with the MCMC baseline. We optimize the popular PyTorch framework to achieve a significant single-socket speedup for our network and 20–43% of peak theoretical flop rate on a range of current CPUs.[19] We augment and develop PyTorch's MPI implementation to run it at the unprecedented scale of 1,024 nodes (32,768 and 24,576 cores) of the Cori and Edison supercomputers with a sustained flop rate of 0.45 Pflop/s. We demonstrate we can successfully train this network to convergence at these large scales, and use this to perform efficient inference on LHC collision events. The developments described here open the door for exploiting HPC resources and existing detailed scientific simulators to perform rapid Bayesian inference in very complex scientific settings.

[19]See disclaimers section after conclusions.

DISCLAIMERS

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more complete information visit www.intel.com/benchmarks.

Performance results are based on testing as of March 22 and March 28, 2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product or component can be absolutely secure.

Configurations: Testing on Cori and Edison (see §6.1) was performed by NERSC. Testing on Diamond cluster was performed by Intel.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Intel, VTune, Xeon are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

ACKNOWLEDGMENTS

The authors would like to acknowledge valuable discussions with Thorsten Kurth on scaling aspects, Quincey Koziol on I/O; Steve Farrell on NERSC PyTorch, Holger Schulz on Sherpa, and Xiaoming Cui, from Intel AIPG team, on NMT. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This work was partially supported by the NERSC Big Data Center; we acknowledge Intel for their funding support. KC, LH, and GL were supported by the National Science Foundation under the awards ACI-1450310. Additionally, KC was supported by the National Science Foundation award OAC-1836650. BGH is supported by the EPRSC Autonomous Intelligent Machines and Systems grant. AGB and PT are supported by EPSRC/MURI grant EP/N019474/1 and AGB is also supported by Lawrence Berkeley National Lab. FW is supported by DARPA D3M, under Cooperative Agreement FA8750-17-2-0093, Intel under its LBNL NERSC Big Data Center, and an NSERC Discovery grant.

REFERENCES

[1] Morad Aaboud et al. 2017. Performance of the ATLAS Trigger System in 2015. *European Physical Journal* C77, 5 (2017), 317. https://doi.org/10.1140/epjc/s10052-017-4852-3

[2] G. Aad et al. 2008. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST* 3 (2008), S08003. https://doi.org/10.1088/1748-0221/3/08/S08003

[3] G. Aad et al. 2015. Search for the Standard Model Higgs boson produced in association with top quarks and decaying into bb in pp collisions at sqrt s=8 TeV with the ATLAS detector. *The European Physical Journal C* 75, 7 (29 Jul 2015), 349. https://doi.org/10.1140/epjc/s10052-015-3543-1

[4] G. Aad et al. 2016. Reconstruction of hadronic decay products of tau leptons with the ATLAS experiment. *The European Physical Journal C* 76, 5 (25 May 2016), 295. https://doi.org/10.1140/epjc/s10052-016-4110-0

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[6] Sea Agostinelli, John Allison, K al Amako, J Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G Barrand, et al. 2003. GEANT4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506, 3 (2003), 250–303.

[7] Joël Akeret, Alexandre Refregier, Adam Amara, Sebastian Seehars, and Caspar Hasner. 2015. Approximate Bayesian computation for forward modeling in cosmology. *Journal of Cosmology and Astroparticle Physics* 2015, 08 (2015), 043.

[8] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. 2002. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing* 50, 2 (2002), 174–188.

[9] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research (JMLR)* 18, 153 (2018), 1–43. http://jmlr.org/papers/v18/17-468.html

[10] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 41–48.

[11] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2018. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research* (2018).

[12] Christopher M Bishop. 1994. *Mixture density networks*. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.

[13] Christopher M Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[14] Johann Brehmer, Gilles Louppe, Juan Pavez, and Kyle Cranmer. 2018. Mining gold from implicit models to improve likelihood-free inference. *arXiv preprint arXiv:1805.12244* (2018).

[15] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[16] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey. 2016. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent. *ArXiv e-prints* (Feb. 2016). arXiv:cs.DC/1602.06709

[17] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

[18] Adji Bousso Dieng, Dustin Tran, Rajesh Ranganath, John Paisley, and David Blei. 2017. Variational Inference via $\backslash chi$ Upper Bound Minimization. In *Advances in Neural Information Processing Systems*. 2732–2741.

[19] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. TensorFlow distributions. *arXiv preprint arXiv:1711.10604* (2017).

[20] Patrick Doetsch, Pavel Golik, and Hermann Ney. 2017. A comprehensive study of batch construction strategies for recurrent neural networks in mxnet. *arXiv preprint arXiv:1705.02414* (2017).

[21] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[22] Arnaud Doucet and Adam M Johansen. 2009. A tutorial on particle filtering and smoothing: Fifteen years later. (2009).

[23] Javier Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *JINST* 13, 07 (2018), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027 arXiv:physics.ins-det/1804.06913

[24] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*. Chapman and Hall/CRC.

[25] Samuel Gershman and Noah Goodman. 2014. Amortized inference in probabilistic reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 36.

[26] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521, 7553 (2015), 452.

[27] B. Ginsburg, I. Gitman, and O. Kuchaiev. 2018. Layer-Wise Adaptive Rate Control for Training of Deep Networks. *in preparation* (2018).

[28] Sheldon L Glashow. 1961. Partial-symmetries of weak interactions. *Nuclear Physics* 22, 4 (1961), 579–588.

[29] Tanju Gleisberg, Stefan Höche, F Krauss, M Schönherr, S Schumann, F Siegert, and J Winter. 2009. Event generation with SHERPA 1.1. *Journal of High Energy Physics* 2009, 02 (2009), 007.

[30] Prem Gopalan, Wei Hao, David M Blei, and John D Storey. 2016. Scaling probabilistic models of genetic variation to millions of humans. *Nature Genetics* 48, 12 (2016), 1587.

[31] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677v1* (2017).

[32] David Griffiths. 2008. *Introduction to elementary particles*. John Wiley & Sons.

[33] Ralf Herbrich, Tom Minka, and Thore Graepel. 2007. TrueSkill™: a Bayesian skill rating system. In *Advances in Neural Information Processing Systems*. 569–576.

[34] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc.

[35] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[36] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *The Journal of Machine Learning Research* 14, 1 (2013), 1303–1347.

[37] Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.

[38] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer. 2015. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *ArXiv e-prints* (Oct. 2015). arXiv:cs.CV/1511.00175

[39] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[40] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. 2016. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 100–103.

[41] D. P. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. *ArXiv e-prints* (Dec. 2014). arXiv:cs.LG/1412.6980

[42] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114* (2013).

[43] Kunitaka Kondo. 1988. Dynamical likelihood method for reconstruction of events with missing momentum. I. Method and toy models. *Journal of the Physical Society of Japan* 57, 12 (1988), 4126–4140.

[44] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. 2018. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 51, 12 pages. http://dl.acm.org/citation.cfm?id=3291656.3291724

[45] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. 2017. Deep learning at 15PF: supervised and semi-supervised classification for scientific data, In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. *ArXiv e-prints*, 7. arXiv:1708.05256

[46] Tuan Anh Le. 2015. Inference for higher order probabilistic programs. *Masters thesis, University of Oxford* (2015).

[47] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research)*, Vol. 54. PMLR, Fort Lauderdale, FL, USA, 1338–1348.

[48] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[49] Mario Lezcano Casado, Atılım Güneş Baydin, David Martinez Rubio, Tuan Anh Le, Frank Wood, Lukas Heinrich, Gilles Louppe, Kyle Cranmer, Wahid Bhimji, Karen Ng, and Prabhat. 2017. Improvements to Inference Compilation for Probabilistic Programming in Large-Scale Scientific Simulators. In *Neural Information Processing Systems (NIPS) 2017 workshop on Deep Learning for Physical Sciences (DLPS), Long Beach, CA, US, December 8, 2017.*

[50] Linux man-pages project. 2019. *Linux Programmer's Manual*. http://man7.org/linux/man-pages/index.html

[51] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Karna, Daina Moise, Simon J. Pennycook, Kristyn Maschoff, Jason Sewall, Nalini Kumar, Shirley Ho, Mike Ringenburg, Prabhat, and Victor Lee. 2018. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 65, 11 pages. https://dl.acm.org/citation.cfm?id=3291743

[52] Amrita Mathuriya, Thorsten Kurth, Vivek Rane, Mustafa Mustafa, Lei Shao, Debbie Bard, Victor W Lee, et al. 2017. Scaling GRPC Tensorflow on 512 nodes of Cori Supercomputer. In *Neural Information Processing Systems (NIPS) 2017 workshop on Deep Learning At Supercomputer Scale, Long Beach, CA, US, December 8, 2017.*

[53] Sam McCandlish, Jared Kaplan, and et.al Amodei, Dario. 2018. An Empirical Model of Large-Batch Training. *arXiv preprint arXiv:1812.06162v1* (2018).

[54] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. *CoRR* abs/1811.05233 (2018). arXiv:1811.05233 http://arxiv.org/abs/1811.05233

[55] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. /Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[56] Radford M Neal. 1993. *Probabilistic inference using Markov chain Monte Carlo methods*. Technical Report CRG-TR-93-1. Dept. of Computer Science, University of Toronto.

[57] Radford M. Neal. 2011. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng (Eds.). Vol. 2. 2.

[58] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., USA, 693–701. http://dl.acm.org/citation.cfm?id=2986459.2986537

[59] X. Pan, J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. 2017. Revisiting Distributed Synchronous SGD. *ArXiv e-prints* (Feb. 2017). arXiv:cs.DC/1702.05800

[60] George Papamakarios and Iain Murray. 2016. Fast ϵ-free Inference of Simulation Models with Bayesian Conditional Density Estimation. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 1028–1036.

[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017*.

[62] Michael E Peskin. 2018. *An introduction to quantum field theory*. CRC Press.

[63] A Salam. 1968. Proceedings of the Eighth Nobel Symposium on Elementary Particle Theory, Relativistic Groups, and Analyticity, Stockholm, Sweden, 1968. (1968).

[64] Christopher J. Shallue, Jaehoom Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network training. *arXiv preprint arXiv:1811.03600v2* (2018).

[65] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489* (2017).

[66] T. Smith, N. Maire, A. Ross, M. Penny, N. Chitnis, A. Schapira, A. Studer, B. Genton, C. Lengeler, F. Tediosi, and et al. 2008. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology* 135, 13 (2008), 1507–1516. https://doi.org/10.1017/S0031182008000371

[67] Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–10.

[68] John Sterman, Thomas Fiddaman, Travis Franck, Andrew Jones, Stephanie McCauley, Philip Rice, Elizabeth Sawin, and Lori Siegel. 2012. Climate interactive: the C-ROADS climate policy model. *System Dynamics Review* 28, 3 (2012), 295–305.

[69] Michael Teng and Frank Wood. 2018. Bayesian Distributed Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*. 6380–6390.

[70] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

[71] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *arXiv e-prints*, Article arXiv:1809.10756 (Sep 2018). arXiv:stat.ML/1809.10756

[72] Martinus Veltman et al. 1972. Regularization and renormalization of gauge fields. *Nuclear Physics B* 44, 1 (1972), 189–213.

[73] S Weinberg. 1967. *Phys. Rev. Lett* 19 (1967), 1264.

[74] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 770–778.

[75] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[76] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).

[77] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. 2016. Asynchronous Stochastic Gradient Descent with Delay Compensation. *ArXiv e-prints* (Sept. 2016). arXiv:cs.LG/1609.08326

Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale
Publication Status	☒ Published ☐ Accepted for Publication ☐Submitted for Publication ☐Unpublished and unsubmitted work written in a manuscript style
Publication Details	Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, **Bradley Gram-Hansen**, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Frank Wood, In Proceedings of the International Conference for High Performance Computing (SC), 2019.

Student Confirmation

Student Name:	Bradley Gram-Hansen		
Contribution to the Paper	The project was a large collaborative project between many different organisations due to the large-scale nature of the project. In regards to my contribution to these papers, I, along with Atlium Gunes Baydin developed the probabilistic programming execution (PPX) protocols that allowed arbitrary simulators to be connected to a probabilistic programming system, PyProb, central to the success of both papers. I participated in the general discussions regarding the construction of the idea and contributed to the writing of the papers. I also contributed to a small part of the PyProb source code – this was already an established project, and to the writing of the papers. The majority of writing for this paper was done by Atlium Gunes Baydin & Lei Shao. I contributed to the writing about probabilistic programming, the PPX protocols, and the efficient Bayesian inference subsections.		
Signature [illegible]		Date	12/02/2021

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Philip H.S. Torr		
Supervisor comments		
Signature [illegible]	Date	26/04/2021

This completed form should be included in the thesis, at the end of the relevant chapter.

6
Creating Efficient, General-Purpose Inference Strategies for Nested Simulators

Effective Approximate Inference for Nested Simulators

Bradley J. Gram-Hansen[1] **Adam Golinski**[1] **Christian Schroeder de Witt**[1]
Saeid Naderiparizi[2] **Adam Scibior**[2] **Andreas Munk**[2]
Frank Wood[2] **Philip Torr**[1] **Yee Whye Teh**[1]
Atilim Gunes Baydin[1] **Tom Rainforth**[1]
[1]University of Oxford [2]University of British Columbia

Abstract

We introduce two approaches for conducting effective approximate inference in stochastic simulators containing nested stochastic sub-procedures, i.e. internal procedures for which the density cannot be calculated directly such as rejection sampling loops and nested inferences. The resulting class of simulators are used extensively throughout the sciences, but fall outside the standard class of Bayesian models: they are doubly intractable. Drawing inferences from them poses a substantial challenge due to the inability to evaluate even their unnormalized density, preventing the use of standard procedures. In particular, the small number of specialized existing methods that can deal with such models are based around forward sampling and thus scale catastrophically poorly in the dimensionality. To address this, we introduce algorithms based on a two-step approach that first approximates the conditional densities of the individual sub-procedures, before using these approximations to run an MCMC sampler on the full simulator. Because the sub-procedures can be dealt with separately and are lower-dimensional than the overall problem, this two-step process allows them to be isolated, and thus tractably dealt with, without placing restrictions on the overall dimensionality. We show empirically that our approaches provide effective inference in settings that cannot be practically handled by existing methodology.

Preliminary work. Under review by UAI 2021. Do not distribute.

1 Introduction

Stochastic simulators are used in a myriad of scientific and industrial settings, such as epidemiology (Patlolla et al., 2004; Ferguson et al., 2006; Smith et al., 2008), physics (Heermann, 1990; Gleisberg et al., 2009), engineering (Hangos and Cameron, 2001) and climate modelling (Held, 2005). They can be complex and high-dimensional, often incorporating domain-specific expertise accumulated over many years of development.

As shown by the probabilistic programming (Goodman et al., 2012; Gordon et al., 2014; van de Meent et al., 2018) and approximate Bayesian computation (ABC) (Csilléry et al., 2010; Marin et al., 2012) literatures, these simulators can be interpreted as probabilistic generative models, implicitly defining a probability distribution over their internal variables and outputs. As such, they form valid targets for drawing Bayesian inferences. In particular, by constraining selected internal variables or outputs to take on specific values, we implicitly define a conditional distribution, or posterior, over the remaining variables. This effectively allows us to, amongst other things, run the simulator in *reverse*, fixing the outputs to some observed values and inferring what parameter values might have led to them. For example, given a simulator for modeling high-energy physics (Gleisberg et al., 2009), we can run inference on the simulator with an observed energy deposit to infer what decay patterns might have led to them (Baydin et al., 2019a).

Though recent advances in probabilistic programming systems (PPSs) (Le et al., 2017; Tran et al., 2017; Rainforth, 2018; Bingham et al., 2019) have provided convenient mechanisms for encoding and reasoning about such simulators, performing the necessary inference is still often extremely challenging, particularly for complex or high-dimensional problems.

In this paper, we consider a scenario where this inference is particularly challenging to perform: when the simulator makes calls to nested stochastic sub-procedures (NSSPs). These can take several different forms, such as internal rejection sampling loops (Gleisberg et al., 2009; Di Pasquale et al., 2015), nested inference procedures (Smith et al., 2008; Stuhlmüller and Goodman, 2014; Bershteyn et al., 2018; Rainforth, 2018), external sub-simulators we have no control over, or even real-world experiments (Foster et al., 2019).

Their unifying common feature is that the density of their outputs cannot be evaluated up to an input-independent normalizing constant in closed form. This, in turn, means the normalized density of the overall simulator cannot be evaluated. They therefore fall outside the standard class of Bayesian inference problems; they are doubly intractable (Murray, 2007). This prevents one from using most common inference methods, such as conventional Markov Chain Monte Carlo (MCMC) and variational inference approaches. In some cases, such as nested probabilistic programs (Stuhlmüller and Goodman, 2014; Rainforth, 2018), one cannot even directly construct consistent Monte Carlo estimator at all, having to instead turn to *nested* estimation approaches (Rainforth et al., 2018), such as nested importance sampling (Rainforth, 2018; Naderiparizi et al., 2019). These have fundamentally slower convergence rates than standard Monte Carlo approaches (Fort et al., 2017; Rainforth et al., 2018) and thus want to be avoided at all costs.

Even for the subclass of NSSPs where these issues with nesting can be avoided, existing applicable approaches rely on *forward sampling* to get around the lack of a closed form density (Goodman et al., 2012; Wood et al., 2014; Rainforth, 2017). This forward sampling typically suffers acutely from the curse of dimensionality and thus circumvents the use of such approaches on all but simplest of problems.

To address these issues, we introduce two new approaches for performing inference in such models. Both are based around approximating the individual NSSPs. The first, generally applicable, approach directly approximates the conditional density of the NSSP outputs using an amortized inference artefact. This then forms a surrogate density for the NSSP, which, once trained, is used to replace it.

Our second approach focuses on the specific case where the *unnormalized* density of the NSSP can be evaluated in isolation, such as a nested probabilistic program or rejection sampling loop, where but its normalizing constant depends on the NSSP inputs. Here, we train a regressor to approximate the normalizing constant of the NSSP as a function of its inputs. Once learnt, this allows the NSSP to be collapsed into the outer program: the ratio of the known unnormalized density and the approximated normalizing constant can be directly used as a factor in the overall density.

Both approaches lead to an *approximate* version of the overall unnormalized density, which can then be used as a target for conventional inference methods like MCMC and variational inference. Because these approximations can be calculated separately for each NSSP, this allows the approach to scale to higher dimensional overall simulators far more gracefully than existing approaches, opening the door to tractably running inference for much more complex problems. Further, once trained, the approximations can be reused for different datasets and configurations of the outer simulator, thereby amortizing the cost of running multiple different inferences for no extra cost. The approaches themselves are also amenable to automation, making them suitable candidates for PPS inference engines.

We confirm the utility of our approaches using a conceptually simple, but numerically challenging, artificial simulator that has been carefully constructed to allow analytic calculation of a ground truth. Namely, we show that while existing approaches completely break down in more than a few dimensions, our approach is able to gracefully scale with increasing dimensionality and produce effective inference.

2 Background and Problem Formulation

NSSPs arise naturally in many real-world systems. Sometimes they are inherent to the model itself, such as in *nested inference* settings (Rainforth, 2018), whereby restrictions in the flow of information cause a double intractability (Murray et al., 2006), e.g., because we are modeling two agents reasoning about each other (Stuhlmüller and Goodman, 2014). They can also occur because we only have access to a sampler for part of the model and not its density—e.g., because the simulator relies on rejection sampling loops (Gleisberg et al., 2009; Di Pasquale et al., 2015) or comprises of a complex external simulator of its own (Marin et al., 2012). Of particular note, recent breakthroughs in universal probabilistic programming to large-scale scientific simulators (Lezcano Casado et al., 2017; Baydin et al., 2019a; Gram-Hansen et al., 2019), have provided automated ways to translate existing large scale stochastic simulators into probabilistic programming systems, without having to re-write the existing simulator inside the given PPS. However, many of these simulators contain NSSPs.

We now formalize the problem of models containing NSSPs before providing some background on existing strategies for coping with them.

2.1 Problem Formulation

For any simulator or *program*, we can define the program density over valid program traces $x_{1:n_x}$ as:

$$p(x_{1:n_x}) \propto \gamma(x_{1:n_x}) = \prod_{j=1}^{n_x} f_{a_j}(x_j|\phi_j) \prod_{k=1}^{n_y} g_{b_k}(y_k|\psi_k) \tag{1}$$

Here n_x is the length of the trace. Each $f_{a_j}(x_j|\phi_j)$ represents the density of the j^{th} random draw, which is made at location a_j and takes in parameters ϕ_j. n_y is a number of *observations*, each of which factor the trace density by $g_{b_k}(y_k|\psi_k)$, where b_k is the location of this observation statement, y_k is the observed value, and ψ_k are parameters of the factorization. All terms—i.e., x_j, n_x, a_j, ϕ_j, n_y, b_k, y_k, ψ_k—may be random variables, but each is deterministically calculable from the trace $x_{1:n_x}$ (see, e.g., Rainforth (2017, Section 4.3.2)).

A NSSP can now be formally defined as a $f_{a_j}(x_j|\phi_j)$ term which cannot be directly evaluated exactly, but where for a given ϕ_j either **[Case A]** we can draw samples from $f_{a_j}(x_j|\phi_j)$ directly or **[Case B]** we have access to an unnormalized form of the density $\gamma^{in}_{a_j}(x_j|\phi_j)$,[1] but do not know the corresponding *input dependent* normalization constant $I_{a_j}(\phi_j)$. In some cases, NSSPs can correspond to both cases (i.e. we can sample and have the unnormalized density), but all NSSPs must conform to at least one of them as otherwise the density would be undefined.

We can now denote the *unnormalized* density for a program containing NSSPs as:

$$\gamma(x_{1:n_x}) = P_{pr}(x_{1:n_x}) \prod_{k=1}^{n_y} g_{b_k}(y_k|\psi_k) \quad \text{where} \tag{2}$$

$$P_{pr}(x_{1:n_x}) := \prod_{\{j \in 1:n_x | a_j \notin S_r\}} f_{a_j}(x_j|\phi_j) \prod_{\{j \in 1:n_x | a_j \in S_r\}} P^{in}_{a_j}(x_j|\phi_j) \tag{3}$$

is a representation of the *forward* or *prior* program which ignores all conditioning statements; $S_r = \{a_1, \dots, a_n\}$ represents the set of addresses that produce intractable densities; and we use $P^{in}_{a_j}(x_j|\phi_j)$ to distinguish the NSSPs from tractable sampling terms. We explain how to addresses $a_i \in S_r$ can be extracted automatically from real-world simulators in Appendix A (see also Gram-Hansen et al. (2019); Baydin et al. (2019b)), but for now we will just assume they are known, which is often the case in practice anyway.

[1]More typically, we actually only have access to some pre-image of $\gamma^{in}_{a_j}(x_j|\phi_j)$, which turns out to be sufficient. See Section 3.2.

2.2 Forward Sampling

The issues imposed by NSSPs of type Case A are immediately apparent: we have no direct characterization of the density of the NSSP at all and must somehow leverage our ability to generate samples from the NSSP to run our overall inference. The simplest way to do this is to simply forward sample from the full program (Goodman et al., 2012), that is draw samples from $P_{pr}(x_{1:n_x})$, before weighting these samples using $\prod_{k=1}^{n_y} g_{b_k}(y_k|\psi_k)$ (Rainforth, 2017). This likelihood weighting approach equates to importance sampling using the prior as our proposal and thus inevitably scales catastrophically poorly as the dimension increases.

Unfortunately, it transpires to be surprisingly difficult to improve on this naive approach. One setting in which improvements can be made is if that if the number of observations n_y is fixed and these observations are interleaved with the sampling statements. Here we can employ particle based inference methods (Wood et al., 2014), like sequential Monte Carlo (Doucet et al., 2001), to exploit the intermediary information provided by these observations. However, many, if not most, models do not possess such interleaving, for which such methods regress back to simple likelihood weighting.

In principle, one can also use Approximate Bayesian Computation (ABC) methods in such settings (Tavare et al., 1997; Pritchard et al., 1999; Sunnåker et al., 2013). However, these will generally be inferior to simply likelihood weighting: the density of the *likelihood* of our overall program is actually directly evaluable, it is the density of our *prior* that is intractable, meaning that such approximations are not generally necessary.

2.3 Nested Monte Carlo and Nested Inference

The issues imposed by NSSPs of type Case B are arguably more subtle. To understand these, we observe that they are equivalent to *nested probabilistic programs* (Stuhlmüller and Goodman, 2014; Rainforth, 2018): they define their own normalized density and require separate, *nested*, inference procedures to be run—either to draw samples or to estimate their posterior density—for each sample realization of the outer program. Such problems are known as *nested inference* problems (Mantadelis and Janssens, 2011; Rainforth, 2018) and they correspond to a more general class of problems than conventional Bayesian inference. In essence, the nesting corresponds to a local normalization of the density, rendering the problem doubly intractable because we have to solve an intractable inference for *each* realization of the outer program.

Conducting inference in such problems requires the use of *nested estimators*, most typically nested Monte Carlo (NMC, Fort et al. (2017); Rainforth et al. (2018)); naïvely running conventional inference approaches like MCMC leads to inconsistent estimators with substantial asymptotic biases. Rainforth (2018) show how consistent nested estimators can be derived through a careful nesting of importance samplers wherein the estimators for the nested programs use more samples as the number of iterations of the outer program increases. Though this is arguably inevitable (Rainforth et al., 2018), the resulting convergence rates of these nested importance samplers are, unfortunately, fundamentally slower than conventional Monte Carlo. Namely the mean squared errors of their estimators converge at rate $\mathcal{O}(1/N^{2/3})$ rather than $\mathcal{O}(1/N)$ in the number of samples N. Moreover, because of the reliance on importance sampling, such approaches suffer acutely from the curse of dimensionality and cannot be used on anything but the most simple problems. Despite this, they remain the state-of-the-art approach for dealing with such problems (Naderiparizi et al., 2019).

3 Approximating NSSPs

We now introduce our approaches for approximating NSSPs and show how these approximations can, in turn, be used to produce efficient inference algorithms for the overall simulator. Both methods are based on replacing each of the intractable NSSP densities, i.e. $P^{in}_{a_j}(x_j|\phi_j)$ in (3), with an approximation. Once learned, these can then be used to construct a directly evaluable approximate target density $\hat{\gamma}(x_{1:n_x}) \approx \gamma(x_{1:n_x})$ by replacing each $P^{in}_{a_j}(x_j|\phi_j)$ in (3), then running an MCMC sampler (or some other conventional inference method) on $\hat{\gamma}(x_{1:n_x})$. The performance for the resulting estimators thus depends on both the accuracy of approximations $\hat{\gamma}(x_{1:n_x})$, in a manner akin to the error in variational inference but without requiring of mean field assumptions, and the efficiency of the MCMC sampling.

To be more specific, our approaches involve the gradient-based learning of a neural-network-based amortized approximation for each NSSP that takes in the NSSP inputs and either returns an approximation of the density of the outputs (Method 1) or the normalizing constant (Method 2).

Note the critical feature that learning these approximations does not require access to the observations of the outer program (i.e. the g_{b_k}); they operate only on the prior program $P_{pr}(x_{1:n_x})$. Moreover, the learning them can be done *independently* for each NSSP: the outer program is used only to provide typical example inputs to each NSSP and does not effect the relative optimality of the approximation of each NSSP for a given input. As such, the learning of these approximation should, at least in theory, scale only with the size of the individual NSSPs, not the number of NSSPs or the dimensionality of the overall problem.

3.1 Method 1: Surrogate Replacement

Our first method is based around learning an amortized variational approximation of each NSSP. The goal of amortization is to learn a parameterized function that can map from different sets of observations (Kingma and Welling, 2014; Rezende et al., 2014) to parameters that define an approximate posterior distribution under those observations. This means we learn a set of parameters once, during an offline training procedure, that can then be used as an approximation of the posterior for all possible inputs. In the typical setting these inputs would be data, but for us they will correspond to the NSSP inputs ϕ_j.

To be precise, our method replaces each $P^{in}_{a_j}(x_j|\phi_j)$ by an approximate surrogate $q^{in}_{a_j}(x_j|\phi_j;\eta_{a_j})$:

$$P_{pr}(x_{1:n_x}) \simeq q(x_{1:n_x};\kappa) := \prod_{\{j\in 1:n_x|a_j\notin S_r\}} f_{a_j}(x_j|\phi_j) \prod_{\{j\in 1:n_x|a_j\in S_r\}} q^{in}_{a_j}(x_j|\phi_j;\eta_{a_j})$$

where $\kappa = \{\eta_{a_j}; a_j \in S_r\}$ are the surrogate parameters. Following existing amortized variational approaches (Kingma and Welling, 2014; Rezende et al., 2014; Le et al., 2017; Ritchie et al., 2016; Paige and Wood, 2016), each $q^{in}_{a_j}(x_j|\phi_j;\eta_{a_j})$ is taken as a variational distribution parametrized by a deep neural network with weights η_{a_j} and takes ϕ_j as its input. Training of these networks is done by minimizing the Kullback–Leibler (KL) divergence from $P_{pr}(x_{1:n_x})$ to $q(x_{1:n_x};\kappa)$ (Paige and Wood, 2016)

$$\begin{aligned}\mathrm{KL}(P_{pr}(x_{1:n_x})||q(x_{1:n_x};\kappa)) \\ = \int P_{pr}(x_{1:n_x}) \log\left(\frac{P_{pr}(x_{1:n_x})}{q(x_{1:n_x};\kappa)}\right) dx.\end{aligned}$$

The optimal network parameters are now given by

$$\kappa^* = \operatorname*{argmin}_{\kappa} \mathrm{KL}\left(P_{pr}||q_\kappa\right)$$

$$= \operatorname*{argmin}_{\{\eta_{a_j};a_j\in S_r\}} \mathbb{E}_{P_{pr}}\left[\sum_{\{j\in 1:n_x|a_j\in S_r\}} -\log q^{in}_{a_j}(x_j|\phi_j;\eta_{a_j})\right] \tag{4}$$

$$\eta^*_r = \operatorname*{argmax}_{\eta_r} \mathbb{E}_{P_{pr}}\left[\sum_{j=1}^{n_x} \mathbb{I}(r=a_j)\log(q^{in}_r(x_j|\phi_j;\eta_r))\right] \tag{5}$$

$\forall r \in S_r$. Because each individual problem is low dimensional, we can efficiently learn an approximation to each NSSP in the set S_r, noting from (5) that these effectively break down into separate problems. As the expectations of equation (4) and (5) are with respect to the simulator density, the required minimization can be done using stochastic gradient descent. Namely, we can generate (potentially approximate) input-output pairs $\{\phi_j, x_j\}$ by running the forward simulator and then use the gradient estimator ($\forall r \in S_r$)

$$\nabla_{\eta_r}\text{KL} \approx -\frac{1}{N}\sum_{k=1}^{N}\sum_{j=1}^{n_x}\mathbb{I}(r = a_j^k)\nabla_{\eta_r}\log(q_r^{in}(x_j^k|\phi_j^i;\eta_r)).$$

If the given NSSP is of type Case A, drawing these samples is straightforward as, by assumption, we can then draw samples from each $P_{a_j}^{in}(x_j|\phi_j)$ and, in turn, samples from P_{pr}, which is computationally cheap. However, if our program contains NSSPs of type Case B, this will require us to run a separate nested inference (Rainforth, 2018) on these NSSPs to generate the required x_j^k from the corresponding ϕ_j. Though this is potentially non-trivial, it is, crucially, far easier than running inference on the overall program: because P_{pr} itself does not include any conditioning statements, generating these samples does not require inference to be run for the outer program. As such, each nested inference problem constitutes its own isolated problem which is far simpler than the overall inference. In other words, the role of sampling from P_{pr} is only to generate example input-output pairs for each NSSP, with each surrogate then separately trained using its local pairs.

3.2 Method 2: Normalization Constant Approximation

If all of our NSSPs satisfy Case B, this implies that each has a known unnormalized density on its internal variables and unknown input-dependent normalizing constant that causes a double-intractability. If the functional forms for all these normalizing constants were known, this would be sufficient to collapse all the NSSPs into the outer program and produce a directly evaluable density for the overall program. Our second method thus looks to learn regressors to predict the normalizing constants and thereby facilitate this. Though in this approach we still use the NSSPs after the regressors have been learned, we no long need to perform a *nested inference* on them: we have converted the problem from a doubly-intractable inference, to a conventional inference.

To formalize this, let us for now assume that the x_j returned by each NSSP corresponds to its full set of internal random draws $z^j_{1:n_x^j}$, i.e., $x_j = z^j_{1:n_x^j}$, such that

$$P_{a_j}^{in}(x_j|\phi_j) = \frac{\gamma_{a_j}^{in}(x_j|\phi_j)}{I_{a_j}(\phi_j)} = \frac{\gamma_{a_j}^{in}\left(z^j_{1:n_x^j}\middle|\phi_j\right)}{I_{a_j}(\phi_j)} \qquad (6)$$

where $\gamma_{a_j}^{in}(z^j_{1:n_x^j}|\phi_j)$ can be evaluated directly, because it is itself an unnormalized probabilistic program density of the form (1), but $I_{a_j}(\phi_j)$ is an intractable normalization constant. If we now introduce a set of regressors $R_r(\phi_j;\tau_r)$, $\forall r \in S_r$, with parameters τ_r, to approximate each $I_r^{in}(\phi_j)$, we can approximate P_{pr} as

$$P_{pr}(x_{1:n_x}) \simeq$$

$$\prod_{\{j\in 1:n_x|a_j\notin S_r\}} f_{a_j}(x_j|\phi_j) \prod_{\{j\in 1:n_x|a_j\in S_r\}} \frac{\gamma_{a_j}^{in}\left(z^j_{1:n_x^j}\middle|\phi_j\right)}{R_{a_j}(\phi_j;\tau_{a_j})}.$$

We can extend this approach to the case where $x_j = \Omega(z^j_{1:n_x^j})$ for some deterministic function Ω, by instead defining our reference measure in the space of $\mathcal{X}_a := \{x_j\}_{j\in 1:n_x|a_j\notin S_r} \cup \{z^j_{1:n_x^j}\}_{j\in 1:n_x|a_j\in S_r}$ and using the pre-image of the prior program density: $P_{pr}(\mathcal{X}_a)$. We can then run inference in this pre-image space and rely on the law of the unconscious statistician to ensure the samples produced are from the desired posterior, see e.g., Rainforth (2017, Section 4.3.2).

Learning the regressors $R_r(\phi_j;\tau_r)$ is done in a similar vein to method 1. Namely we run the simulator forward to gather pairs $\{\phi_j, \hat{I}_r(\phi_j)\}$ for each NSSP, where $\hat{I}_r(\phi_j)$ is an unbiased approximation of $I_r(\phi_j)$, and then use this as a training dataset for learning the regressor. Specifically, for each NSSP we train a neural network regressor to minimize the expected squared error between $R_r(\phi_j;\tau_r)$ and $\hat{I}_r(\phi_j)$. Thus, our objective and gradient update are:

$$\mathcal{L}_r = \mathbb{E}\left[\left(R_r(\phi_j;\tau_r) - \hat{I}_r(\phi_j)\right)^2\right], \qquad (7)$$

$$\nabla_{\tau_r}\mathcal{L}_r = \mathbb{E}\left[\nabla_{\tau_r}\left(R_r(\phi_j;\tau_r) - \hat{I}_r(\phi_j)\right)^2\right] \qquad (8)$$

where the expectation is over both the estimates and the inputs ϕ_j, with the distribution of the latter defined by running P_{pr} forward and, if necessary, randomly selecting between the inputs that are passed to NSSP r if it is called more than once. This can further be Rao-Blackwellized by averaging over all the inputs passed to the NSSP instead of choosing between them. Thus, by running the simulator forward, collecting samples from the NSSPs generated from sampling the priors of each NSSP, we can make updates based on Monte Carlo estimates of $\nabla_{\tau_r}\mathcal{L}_r$.

This scheme results in $R_r(\phi_j;\tau_r) = I_r(\phi_j)$ in the limit of a large number of training samples if our neural network has sufficient capacity to exactly capture $I_r^{in}(\phi_j)$.

To see this note that

$$\mathbb{E}\left[\left(R_r(\phi_j;\tau_r)-\hat{I}_r(\phi_j)\right)^2\middle|\phi_j\right] = \\ (R_r(\phi_j;\tau_r)-I_r(\phi_j))^2+\mathbb{E}\left[\left(I_r(\phi_j)-\hat{I}_r(\phi_j)\right)^2\middle|\phi_j\right]$$

where the second term does not depend on τ_r and the first is minimized when $R_{a_j}(\phi_j;\tau_{a_j})$=$I_{a_j}(\phi_j)$ for all ϕ_j.

Once trained, we can run inference on the approximate, unnormalized, target:

$$\hat{\gamma}(x_{1:n_x}) = \prod_{i=1,a_i\notin S_R}^{n_x} f_{a_i}(x_i;\phi_i)\prod_{k=1,b_k}^{n_y} g_{b_k}(y_k;\phi_k) \\ \prod_{j=1,a_j\in S_R}^{n_x}\frac{\gamma^{in}_{a_j}(x_j|\phi_j)}{R_{a_j}(\phi_j;\tau)}. \tag{9}$$

It is important to note that this method never actually requires us to directly evaluate $\gamma^{in}_{a_j}(x_j|\phi_j)$, instead we introduce the variables z that implicitly produce the correct pushforward distribution on the x's. In essence we are actually defining a higher dimensional auxiliary variable problem that has the desired pushforward on the variables of interest.

3.2.1 An Adjusted Approach for Nested Rejection Samplers

In the case where a NSSP is given by a rejection sampler, then it is preferable to slightly adjust the procedure for Method 2 to utilize the ability of rejection samplers to unbiasedly estimate $1/I(\phi)$.

Here we have $I(\phi)=\mathbb{E}[\mathbb{I}(A(z,\phi))]$ where $A(z,\phi)$ represents the rejection criterion, returning true if the sample is to be accepted, and the expectation is with respect to running a single iteration of the rejection sampling loop. The naive Monte Carlo estimate

$$I(\phi)\approx\frac{1}{N}\sum_{n=1}^{N}\mathbb{I}(A(z_n,\phi)=1), \tag{10}$$

is only unbiased, if N is independent of the z_n, which is not the case for a standard rejection sampler.

Typically, one would like to instead run the rejection sampler in the standard manner: running the sampler until a sample is accepted, at which point we have generated N_a samples. Here N_a is not independent of the z_n, such that the naive estimator in (10) is now biased. However, instead fixing N upfront can return an estimate $\hat{I}(\phi)=0$ which will in turn can cause the estimate of the normalized density to become infinite, thereby triggering a failure in the overall inference.

This conundrum can be circumvented by instead trying to directly estimate $1/I(\phi)$ and use this as the basis for the regressor. This is possible because rejection samplers have the property $\mathbb{E}[N_a|\phi]=1/I(\phi)$ as follows:

$$\mathbb{E}[N_a|\phi]=\mathbb{E}\left[\sum_{n=1}^{N_a}1\middle|\phi\right]=\mathbb{E}\left[\sum_{n=1}^{\infty}\mathbb{I}(N_a\geq n)\middle|\phi\right] \\ =\sum_{n=1}^{\infty}\mathbb{E}\left[\mathbb{I}(N_a\geq n)|\phi\right]=\sum_{n=0}^{\infty}(1-I(\phi))^n=\frac{1}{I(\phi)}.$$

Therefore, in this setting we learn our regressor R_{a_j} to go from ϕ_j to $1/I(\phi)$, exploiting the fact that N_a is an unbiased estimate of the latter, and subsequently use

$$P^{in}_{a_j}(x_j|\phi_j)\approx\gamma^{in}_{a_j}(x_j|\phi_j)R_{a_j}(\phi_j;\tau_{a_j}) \tag{11}$$

to construct the approximate objective.

It is interesting to further note that

$$\mathbb{E}[\gamma^{in}_{a_j}(x_j|\phi_j)N_a|x_j,\phi_j]=P^{in}_{a_j}(x_j|\phi_j) \tag{12}$$

such that it should in principle also be possible to use this result to develop pseudo-marginal samplers.

4 Experiments

To confirm their ability to provide accurate and efficient inference, we now test our methods on an artificial model where we can easily calculate the ground truth, introduce NSSPs of both types, and adjust the difficulty of the problem by varying its dimensionality. Though simple, we will see that this model is beyond what can be tackled by previous approaches (assuming we do not exploit the analytic solutions). We emphasize here that because existing approaches for dealing with NSSPs are so limited, the only viable way of accurately asserting the performance of approaches is to manually construct a problem to permit analytic simplifications to allow ground truth calculations; hence the artificial nature of these experiments. Further experiments are given in the supplement.

To be more specific, our model comprises of a multivariate Gaussian unknown mean problem, but with a twist as we write the model using NSSPs. This model is chosen for two reasons. First, via conjugacy relationships we can analytically calculate the posterior mean of the outer program. Second, it allows us to easily replace individual variables in the problem with NSSPs of either type Case A or Case B (or both) without changing the ground truth posterior.

4.1 Model Definition

Let $\boldsymbol{x}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})$, $\boldsymbol{x}\in\mathbb{R}^{n_x}$. We set $\boldsymbol{\mu}$ to a fixed value, randomly generated upfront. $\boldsymbol{\Sigma}$ is also randomly

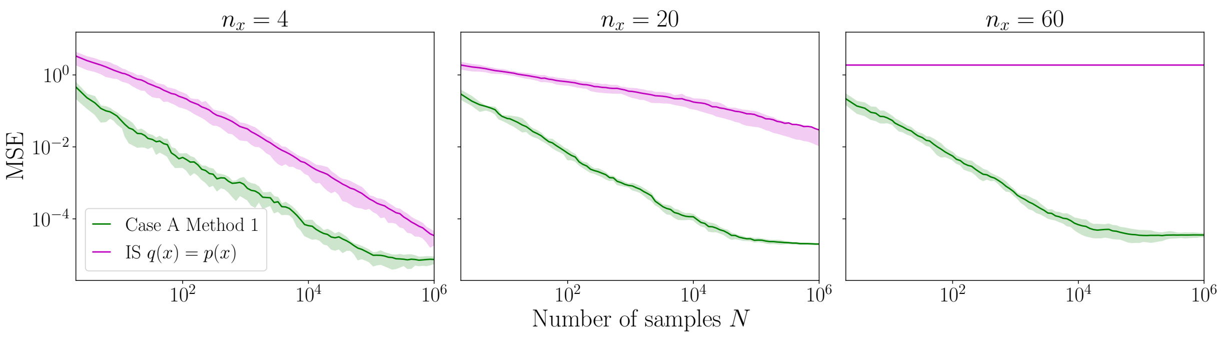


Figure 1: Convergence of the MSE for posterior mean calculated across all latent dimensions, for different problem dimensionalities for the Nested Gaussian model Case A. Shown is importance sampling from the prior (magenta) and our surrogate method (i.e., Method 1, green). The 25%-75% quantiles are shown as shaded regions, formed using 100 independently run chains of 10^6 samples per chain (after burn-in). We see that our method produces significantly lower error than the baseline, particularly as the dimensionality increases.

generated, but in a particular manner that ensures that the following depending structure holds

$$p(\boldsymbol{x}_{1:n_x}) = p(x_1)p(x_2|x_1)\dots p(x_{n_x}|x_{n_x-1}).$$

Details on how this is done are given in the supplement. Given this induced dependency structure, the forward call of the simulator can be written out as

$$\begin{aligned} p(x_1) &= f_{a_1}(x_1|\phi_1) = \mathcal{N}(x_1; \mu_1, \Sigma_{1,1}) \\ p(x_i|x_{i-1}) &= f_{a_i}(x_i|\phi_i) \\ &= \mathcal{N}(x_i; \mu_{i|i-1}(x_{i-1}), \Sigma_{i|i-1}(x_{i-1})), \end{aligned}$$

where $\mu_{i|i-1}(x_{i-1})$ and $\Sigma_{i|i-1}(x_{i-1})$ are conditional means and covariances calculated using standard Gaussian identities (Petersen and Pedersen, 2012). Finally, we introduce a single Gaussian distributed observation $\boldsymbol{y} \in \mathbb{R}^{n_x}$ of the form $g_{b_1}(\boldsymbol{y}|x_{1:n_x}) = \mathcal{N}(\boldsymbol{y}; x_{1:n_x}, I)$, where I is an identity matrix.

To induce nested structures into this model, for each of even addresses we replace $f_{a_i}(x_i|\phi_i)$ with an NSSPs (i.e., if i is even then $a_i \in S_r$, else, $a_i \notin S_r$). These NSSPs implicitly define the same density, but we either only provide a black–box sampler (Case A), or in an input–dependent unnormalized form (Case B).

For the former case, we simply make an external call to a function that returns samples according to $\mathcal{N}(x_i; \mu_{i|i-1}(x_{i-1}), \Sigma_{i|i-1}(x_{i-1}))$, but for which we cannot directly evaluate the density. For the latter case, we define a set of nested probabilistic programs

```
def NSSP_i($\phi_i = x_{i-1}$)
    $z \sim \mathcal{N}(\mu_i, \Sigma_{i,i})$
    $\mu_{i-1|i} = \mu_{i-1} + (z - \mu_i)\Sigma^2_{i-1,i}/\Sigma_{i,i}$
    $\Sigma_{i-1|i} = \Sigma_{i-1,i-1} - \Sigma^2_{i-1,i}/\Sigma_{i,i}$
    factor($\mathcal{N}(\phi_i; \mu_{i-1|i}, \Sigma_{i-1|i})$)
return  $z$
```

where `factor` represents a factoring of the density, i.e. this is a likelihood term of density $\mathcal{N}(\phi_i; \mu_{i-1|i}, \Sigma_{i-1|i})$. Drawing from this NSSP requires us to run separate inference procedures, because the nested model involves a local normalization (Rainforth, 2018). We can consider the nested sub-procedure in isolation and its inputs as fixed variables when calculating the form of its local posterior, but this posterior must be estimated separately for each possible instance of the inputs. Critically, we can also estimate the marginal likelihood of this nested program for input ϕ_i by drawing samples of z and then taking the average of the likelihood evaluations (i.e. the average of the exponential of the factor statements). This thus lets us carry out Method 2.

4.2 Evaluation

We now consider running Method 1 on the Case A variation of our model and Method 2 on the Case B variation. Note that Method 2 cannot be run for problems that are only of type Case A, while Method 1 will generally be inferior to Method 2 when the latter can be run (such that we do not generally recommend doing this) because it requires us to regress from the inputs to a full distribution, rather than just the scalar normalizing constant.[2] All the same, results for running Method 1 on Case B are given in the supplement. We note that for both methods the real time spent training the approximations was comparable to that for running the subsequent MCMC sampling.

Recall that in the training phase of Method 1 we require only the input–output pairs from the NSSPs; these can

[2]A potential exception to this is if the individual NSSPs contain a large number of internal random variables z (Section 3.2), as here the final MCMC sampling for Method 2 is on a much higher dimensional space than for Method 1.

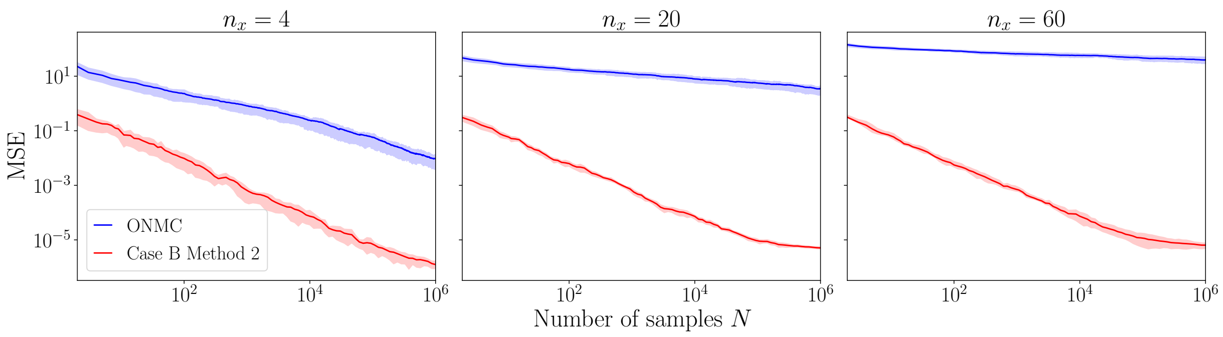


Figure 2: Convergence of the MSE for posterior mean calculated across all latent dimensions, for different problem dimensionalities for the Nested Gaussian model Case B. Shown is an online nested importance sampling estimate (blue) and our normalization approximation method (i.e., Method 2, red). Conventions as per Fig. 1.

all be directly generated by forward simulation. This can also be straightforwardly batched so training is efficient. Once trained we run Metropolis-Hastings (MH) over the whole simulator with the NSSPs replaced by their approximations. Full details of the NN architectures and training procedure are provided in Appendix B. We compare to a baseline of importance sampling from the prior as described in Section 2.2, noting that neither particle methods nor ABC approaches are helpful in this setting; despite the crudeness of approach, we are not aware of any stronger baselines in the literature because of the unusual nature of the problem. The results are given in Figure 1, where we use our ground truth for the posterior mean of $\boldsymbol{x}$ (see Appendix C) to calculate a mean squared error (MSE) for variants of the model with different dimensionality. We use a burn in period of 10^4 MCMC samples (not included in the plot). We see that our approach significantly outperforms the baseline, while this improvement becomes more noticeable as the dimensionality increases. In fact, by the time $n_x = 60$, the baseline effectively fails to generate any good samples of note, while our approach maintains its performance.

Training of Method 2, on the other hand, requires each NSSP to return both marginal likelihood estimate for training its regressor, and an approximate sample of its posterior so that the forward sampling of the program can continue. For the former, we draw a number of samples the NSSP's local prior (i.e. numerous z) and return the average likelihood evaluations resulting from the factor statements, giving an unbiased estimate of the marginal likelihood. We then draw one of these samples in proportion to its "weight" (i.e. its likelihood evaluation) as the returned sample, in a manner akin to self-normalized importance sampling with resampling. We note here that these returned samples do not need to be exact posterior samples for effective training (and indeed they are not): we are only doing this sampling to generate example inputs for training the regressors of later NSSPs. Again estimation steps can be batched, so training is efficient. Full training details are again provided in Appendix B.2.

Figure 2 shows the results for this case, where we now instead compare to the online nested importance sampling (ONMC) approach of Rainforth (2018), noting that, to the best of our knowledge, nesting importance sampling approaches (or slight variations therein, e.g. Naderiparizi et al. (2019)) are the only general–purpose consistent estimators for this class of problems currently present in the literature. We again see that our approach provides substantial improvements, with these again becoming more pronounced as the dimensionality increases.

It is noticeable that both our methods converge to biased solutions as we increase the number of samples at inference time. However, this is to be expected as they are based on approximating the density of the NSSP. Critically though, this error can be reduced by performing more training of the approximation networks, increasing their capacity, or using more accurate estimates in the training procedure (e.g. more importance samples for each NSSP when training Method 2). The key is that the methods are providing effective estimates, even for the higher dimensional problems where the baselines break down completely.

5 Conclusions and Future Work

We have introduced two approaches for performing effective inference in simulators containing nested stochastic sub-procedures (NSSPs). These are both based on first separately approximating each individual NSSP—either using an amortized inference surrogate on its outputs or a combination of the original program and a regressed approximation of its marginal likelihood from its inputs—and then using these approximations to form an approximation for the overall program that can be used as a target for scalable inference algorithms like MCMC and variational inference. We have shown that this provides substantial gains over existing baselines on a synthetic model where we can analytically derive the ground truth. In particular, our approaches have allowed us to tackle problems of a much higher dimension that those which can be efficiently tackled by existing approaches.

Our work provides a promising pathway to perform statistically principled and computationally efficient inference in large scale simulators. Indeed there are a host of current simulator-based inference problems currently actively being researched in the literature where the bottleneck is the restrictions that NSSPs impose on our ability to run inference (Baydin et al., 2019a; Gram-Hansen et al., 2019). We hope that this work will substantially increase the viability of such ventures, thereby leading to a large number of downstream applications.

References

Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, and Frank Wood. Efficient probabilistic inference in the quest for physics beyond the standard model. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2019a.

Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence F. Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356180.

Anna Bershteyn, Jaline Gerardin, Daniel Bridenbecker, Christopher W Lorton, Jonathan Bloedow, Robert S Baker, Guillaume Chabot-Couture, Ye Chen, Thomas Fischle, Kurt Frey, et al. Implementation and applications of emod, an individual-based multi-disease modeling platform. *Pathogens and Disease*, 76(5):fty059, 2018.

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. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

Katalin Csilléry, Michael GB Blum, Oscar E Gaggiotti, and Olivier François. Approximate Bayesian computation (ABC) in practice. *Trends in Ecology & Evolution*, 25(7):410–418, 2010.

Valentina Di Pasquale, Salvatore Miranda, Raffaele Iannone, and Stefano Riemma. A simulator for human error probability analysis. *Reliability Engineering & System Safety*, 139:17–32, 2015.

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.

Neil M Ferguson, Derek AT Cummings, Christophe Fraser, James C Cajka, Philip C Cooley, and Donald S Burke. Strategies for mitigating an influenza pandemic. *Nature*, 442(7101):448–452, 2006.

Gersende Fort, Emmanuel Gobet, and Eric Moulines. MCMC design-based non-parametric regression for rare event. application to nested risk computations. *Monte Carlo Methods and Applications*, 23(1):21–42, 2017.

Adam Foster, Martin Jankowiak, Elias Bingham, Paul Horsfall, Yee Whye Teh, Thomas Rainforth, and Noah Goodman. Variational Bayesian optimal experimental design. In *Advances in Neural Information Processing Systems*, pages 14036–14047, 2019.

Tanju Gleisberg, Stefan Höche, F Krauss, M Schönherr, S Schumann, F Siegert, and J Winter. Event generation with SHERPA 1.1. *Journal of High Energy Physics*, 2009(02):007, 2009.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

Bradley Gram-Hansen, Christian Schröder de Witt, Tom Rainforth, Philip HS Torr, Yee Whye Teh, and Atılım Güneş Baydin. Hijacking malaria simulators with probabilistic programming. *ICML Workshop on AI for Social Good*, 2019.

Katalin M Hangos and Ian T Cameron. *Process modelling and model analysis*, volume 4. Academic press London, 2001.

Dieter W Heermann. Computer-simulation methods. In *Computer Simulation Methods in Theoretical Physics*, pages 8–12. Springer, 1990.

Isaac M Held. The gap between simulation and understanding in climate modeling. *Bulletin of the American Meteorological Society*, 86(11):1609–1614, 2005.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

Mario Lezcano Casado, Atılım Güneş Baydin, David Martinez Rubio, Tuan Anh Le, Frank Wood, Lukas Heinrich, Gilles Louppe, Kyle Cranmer, Wahid Bhimji, Karen Ng, and Prabhat. Improvements to inference compilation for probabilistic programming in large-scale scientific simulators. In *Neural Information Processing Systems (NIPS) 2017 workshop on*

Deep Learning for Physical Sciences (DLPS), Long Beach, CA, US, December 8, 2017, 2017.

Theofrastos Mantadelis and Gerda Janssens. Nesting probabilistic inference. *arXiv preprint arXiv:1112.3785*, 2011.

Jean-Michel Marin, Pierre Pudlo, Christian P Robert, and Robin J Ryder. Approximate Bayesian computational methods. *Statistics and Computing*, 22(6): 1167–1180, 2012.

Iain Murray, Zoubin Ghahramani, and David JC MacKay. MCMC for doubly-intractable distributions. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, pages 359–366. AUAI Press, 2006.

Iain Andrew Murray. *Advances in Markov chain Monte Carlo methods*. PhD thesis, University of London, 2007.

Saeid Naderiparizi, Adam Ścibior, Andreas Munk, Mehrdad Ghadiri, Atılım Güneş Baydin, Bradley Gram-Hansen, Christian Schroeder de Witt, Robert Zinkov, Philip H. S. Torr, Tom Rainforth, Yee Whye Teh, and Frank Wood. Amortized rejection sampling in universal probabilistic programming. *arXiv:1910.09056 [cs, stat]*, 2019.

Brooks Paige and Frank Wood. Inference networks for sequential Monte Carlo in graphical models. In *International Conference on Machine Learning*, pages 3040–3049, 2016.

Padmavathi Patlolla, Vandana Gunupudi, Armin R Mikler, and Roy T Jacob. Agent-based simulation tools in computational epidemiology. In *International Workshop on Innovative Internet Community Systems*, pages 212–223. Springer, 2004.

Kaare Brandt Petersen and Michael Syskind Pedersen. The Matrix Cookbook. `https://archive.org/details/imm3274/mode/2up`, 2012.

J. K. Pritchard, M. T. Seielstad, A. Perez-Lezaun, and M. W. Feldman. Population growth of human y chromosomes: a study of y chromosome microsatellites. *Molecular Biology and Evolution*, 16(12):1791–1798, 1999. ISSN 0737-4038. doi: 10.1093/oxfordjournals.molbev.a026091.

Thomas William Gamlen Rainforth. *Automating inference, learning, and design using probabilistic programming*. PhD thesis, University of Oxford, 2017.

Tom Rainforth. Nesting probabilistic programs. *arXiv preprint arXiv:1803.06328*, 2018.

Tom Rainforth, Rob Cornish, Hongseok Yang, Andrew Warrington, and Frank Wood. On nesting Monte Carlo estimators. In *International Conference on Machine Learning*, pages 4267–4276, 2018.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on International Conference on Machine Learning-Volume 32*, pages II–1278, 2014.

Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

T Smith, N Maire, A Ross, M Penny, N Chitnis, A Schapira, A Studer, B Genton, C Lengeler, Fabrizio Tediosi, et al. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13):1507–1516, 2008.

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.

Mikael Sunnåker, Alberto Giovanni Busetto, Elina Numminen, Jukka Corander, Matthieu Foll, and Christophe Dessimoz. Approximate bayesian computation. *PLoS Comput Biol*, 9(1):e1002803, 2013.

S. Tavare, D. J. Balding, R. C. Griffiths, and P. Donnelly. Inferring coalescence times from DNA sequence data. *Genetics*, 145(2):505–518, 1997. ISSN 0016-6731.

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.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv:1809.10756 [cs, stat]*, 2018.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

A Locating Addresses in Simulators

Running the introduced inference approaches requires one to identify calls in the simulator corresponding to NSSPs. Often this can be done manually as these will be known ahead of time. However, in certain cases, particularly for large scale simulators or probabilistic programs containing rejection sampling loops, it may not actually be trivial to manually identify and then replace these NSSPs in the code to allow our approach to be used.

Thankfully, there has been substantial recent progress on how this can be done in the literature through the concept of *hijacking* a simulator. Namely, recent work by Gram-Hansen et al. (2019); Baydin et al. (2019a) described a hijacking process that takes an arbitrary stochastic simulator, defines a joint density, and hijacks the underlying random primitives of that simulator by passing their control to a probabilistic programming system (PPS). This backend PPS can then both manipulate the running of this simulator by overwriting calls to the random primitives and also track the control flows of the simulator to identify rejection sampling loops and other NSSPs. Thus, in turn, these methods should, at least in principle, be able to automate the running of our inference approaches.

Viewed from another perspective, these hijacking based PPS approaches are often used to tackle problems of the same class as our work, but at present their backend inference engines rely on simple importance sampling procedures. As such, they are not sufficiently powerful to satisfactorily perform inference in the types of simulators they are designed to target, such as the OpenMalaria simulator (Smith et al., 2008) and the prominent Covid-19 simulator of Ferguson et al. (2006). The techniques developed in this paper provide a pathway to running inference in models like this where this was previously completely infeasible, due to their ability to scale gracefully with the dimensionality of the model, while existing approaches scale exponentially poorly and quickly become redundant.

B Further Experiments and Experimental Details

B.1 Method 1 for Case B Problems

Method 2 (i.e. regressing the marginal likelihood) is typically preferable when all our NSSPs conform to Case B but not Case A (i.e. we have access to their unnormalized densities, but cannot generate samples directly). This is because a) it is simpler to regress to a one dimensional marginal likelihood that a set of variational parameters that approximate the output distribution, and b) it does not introduce additional approximations from running inference on the individual NSSPs during training (as our marginal likelihood estimates will generally be unbiased, even if they are high variance). However, this is not universally the case: Method 1 (i.e. learning an amortized surrogate density) can still sometimes be preferable in Case B scenarios, particular if in the individual NSSPs are high–dimensional themselves. This is because a) the final MCMC inference is lower dimensional because it does not include the *internal* random draws of the NSSPs, and b) sometimes producing approximate posterior samples from a Case B NSSP is easier than producing a reasonable marginal likelihood estimate (e.g. if the NSSP is internally high–dimensional it might be feasible to generate good approximate output samples by MCMC, but not low–variance marginal likelihood estimates through importance sampling).

In this section, we consider running Method 2 for NSSPs which are only of type Case B. For this, we use exactly the same Nested Gaussian model as the one described in Section 4. We assume that the model is specified using the nested probabilistic programs `NSSP_i`, but that we cannot draw samples directly from them. It is still possible to apply Method 1 here by treating the problem as a nested inference. That is, whenever we encounter a NSSP during training, we run a local inference to generate the required (approximate) output samples associated with the provided input, that is approximate samples from the posterior distribution of `NSSP_i`(ϕ_i) given input ϕ_i. Sampling from the posterior of each NSSP constitutes an separate, one-dimensional, non-nested inference problem, and we will need to run inference on each NSSP we encounter in the forward run of the simulator in the process of generating the samples for training of our conditional density estimators.

Among many inference methods we could use to solve such an inference problem, we decide to use Metropolis-Hastings. The details of the training procedure are given in Appendix B.2, and are analogous to both Method 1 for Case A and Method 2 for Case B experiments in the main paper in all aspects they share.

In Figure 3 we compare to the same ONMC baseline considered in the main paper. We see that the performance is similar to when we used Method 2, but with slightly higher final errors.

B.2 Training details for the Nested Gaussian

In this section we give the details of the training schemes used. We first give the details common for all the models we trained, and then describe model-specific details for individual models in subsections.

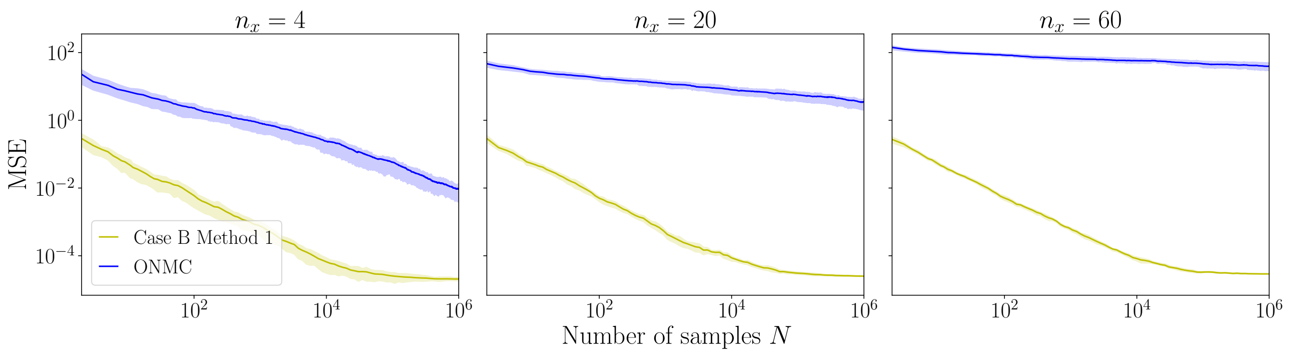


Figure 3: Convergence of the MSE for posterior mean calculated across all latent dimensions, for different problem dimensionalities for the Nested Gaussian model Case B. Shown is nested importance sampling (blue) and the surrogate method (i.e., Method 1 for Case B, yellow). The 25%-75% quantiles, shaded regions, are over 100 independently run chains of 10^6 samples per chain. As expected, we see that both our approach produces significantly lower error that the baseline as the dimensionality increases.

All the neural networks used for determining the parameters of the conditional density estimator for Method 1 (i.e., mapping from the value of ϕ_i to the parameters of the parametric density estimator approximating the posterior distribution of the NSSP) and regressing the marginal likelihood of the NSSP for Method 2 (i.e., mapping from the value of ϕ_i to the approximation of the marginal likelihood of the NSSP) were MLPs with 64 hidden units per layer and 4 hidden layers.

We trained a separate density estimator (Method 1) or regressor (Method 2) for each instance of the NSSP which means that we trained 2, 10, 30 separate neural networks for $d = 4, 20, 60$, respectively.

We trained for 10^4 gradient updates, each with newly generated batch of data from the forward run of the program. Each batch had 10^3 examples. The initial learning rate for all experiments was 10^{-3}. We used `ReduceLROnPlateau` learning rate scheduler with patience of 10^3 and decrease factor of 0.29.

Final inference on the program with the probability density of the NSSPs approximated using the learned components (for both Method 1 and Method 2) were run using Metropolis-Hastings with isotropic Gaussian proposal, with proposal standard deviations $1.0, 0.4, 0.2$ for $d = 4, 20, 60$, respectively. The MCMC chains were initiated at a vector of all zeros and used 10^4 samples for burn in (not included in plots).

B.2.1 Method 1 for Case A

The distribution family used for amortized surrogates was taken to just be a one dimensional Gaussian, $\mathcal{N}(m(\phi), s(\phi))$, where m and s are MLPs as described earlier that take in the NSSP input ϕ and return a mean and standard deviation respectively. As such, this approximation family is able to encapsulate the true NSSP conditional distributions exactly if sufficiently accurate MLPs can be learned. Training these conditional density estimators was done as explained in the main paper.

B.2.2 Method 2 for Case B

We obtain a Monte Carlo estimate of the marginal likelihood for each instance of the NSSP by taking 10^4 samples of z in the `NSSP_i(`ϕ`)` and forming an average over the evaluations of the likelihood (given by the exponentiation of the `factor`$(\cdot)$ statement). This estimation is trivially vectorized on a GPU, which means that taking this relatively large number of samples in the estimate is only nominally slower than it would be to only use a single sample. As such, the training phase of Method 2 is generally much quicker than one might expect.

B.2.3 Method 1 for Case B

The conditional density estimators used were the same as per Section B.2.1. As described in Section B.1, in this setting we run separate MCMC inference on each instance of the NSSP. We ran a separate chain for each element in the training batch. We ran Metropolis-Hastings with an isotropic Gaussian proposal, with proposal standard deviation of 3.0 which yielded an average acceptance rate around 44%. The chains were initialized by using the conditional density estimator. Each chain was run for 25 samples, with only the last sample presented to the density estimator as a training sample.

C Derivation of the Ground Truth

We now derive the ground truth posterior for the Gaussian model used in the main paper. The key here is to reverse engineer the prior covariance, $\mathbf{\Sigma}$, in such a way that the following Markov dependency structure will hold

$$p(\boldsymbol{x}_{1:n_x}) = p(x_1)p(x_2|x_1)\dots p(x_{n_x}|x_{n_x-1}).$$

This then allows us to replace any arbitrary $p(x_i|x_{i-1})$ with an NSSP for the form given in the main paper (or equivalently a direct sampler) without requiring any information from the $< i-1$ indices.

To show how this can be done, let us start by noting the standard results for the marginals of an arbitrary Gaussian (Petersen and Pedersen, 2012). Namely, if $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma})$ and

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_a \\ \boldsymbol{x}_b \end{bmatrix}, \ \boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix}, \ \mathbf{\Sigma} = \begin{bmatrix} \mathbf{\Sigma}_a & \mathbf{\Sigma}_c \\ \mathbf{\Sigma}_c^T & \mathbf{\Sigma}_b \end{bmatrix},$$

then $\boldsymbol{x}_b|\boldsymbol{x}_a \sim \mathcal{N}(\boldsymbol{\mu}_{b|a}, \mathbf{\Sigma}_{b|a})$, where

$$\begin{aligned} \boldsymbol{\mu}_{b|a} &= \boldsymbol{\mu}_b + \mathbf{\Sigma}_c^T \mathbf{\Sigma}_a^{-1}(\boldsymbol{x}_a - \boldsymbol{\mu}_a) \\ \mathbf{\Sigma}_{b|a} &= \mathbf{\Sigma}_b - \mathbf{\Sigma}_c^T \mathbf{\Sigma}_a^{-1} \mathbf{\Sigma}_c. \end{aligned}$$

Now taking $\boldsymbol{x}_b = \boldsymbol{x}_i$ and $\boldsymbol{x}_a = \boldsymbol{x}_{1:i-1}$, we thus have $\boldsymbol{x}_i|\boldsymbol{x}_{1:i-1} \sim \mathcal{N}(\boldsymbol{\mu}_{i|1:i-1}, \mathbf{\Sigma}_{i|1:i-1})$ where

$$\begin{aligned} \boldsymbol{\mu}_{i|1:i-1} &= \boldsymbol{\mu}_i + \mathbf{\Sigma}_{1:i-1,i}^T \mathbf{\Sigma}_{1:i-1,1:i-1}^{-1}(\boldsymbol{x}_{1:i-1} - \boldsymbol{\mu}_{1:i-1}) \\ \mathbf{\Sigma}_{i|1:i-1} &= \mathbf{\Sigma}_{i,i} - \mathbf{\Sigma}_{1:i-1,i}^T \mathbf{\Sigma}_{1:i-1,1:i-1}^{-1} \mathbf{\Sigma}_{1:i-1,i}. \end{aligned}$$

For our induced dependency structure to hold, we require $\boldsymbol{\mu}_{i|1:i-1} = \boldsymbol{\mu}_{i|i-1}$ and $\mathbf{\Sigma}_{i|1:i-1} = \mathbf{\Sigma}_{i|i-1}$, where $\boldsymbol{\mu}_{i|i-1}$ and $\mathbf{\Sigma}_{i|i-1}$ are derived by instead taking $\boldsymbol{x}_a = \boldsymbol{x}_{i-1}$ to yield

$$\begin{aligned} \boldsymbol{\mu}_{i|i-1} &= \boldsymbol{\mu}_i + (\boldsymbol{x}_{i-1} - \boldsymbol{\mu}_{i-1})\mathbf{\Sigma}_{i-1,i}/\mathbf{\Sigma}_{i-1,i-1} \\ \mathbf{\Sigma}_{i|i-1} &= \mathbf{\Sigma}_{i,i} - \mathbf{\Sigma}_{i-1,i}^2/\mathbf{\Sigma}_{i-1,i-1}. \end{aligned}$$

We thus need the following equations to hold for all $x_{1:i-2}$

$$\begin{aligned} &\mathbf{\Sigma}_{1:i-1,i}^T \mathbf{\Sigma}_{1:i-1,1:i-1}^{-1}(\boldsymbol{x}_{1:i-1} - \boldsymbol{\mu}_{1:i-1}) \\ &\qquad = (\boldsymbol{x}_{i-1} - \boldsymbol{\mu}_{i-1})\mathbf{\Sigma}_{i-1,i}/\mathbf{\Sigma}_{i-1,i-1} \end{aligned} \tag{13}$$

$$\mathbf{\Sigma}_{1:i-1,i}^T \mathbf{\Sigma}_{1:i-1,1:i-1}^{-1} \mathbf{\Sigma}_{1:i-1,i} = \mathbf{\Sigma}_{i-1,i}^2/\mathbf{\Sigma}_{i-1,i-1}. \tag{14}$$

Considering the first of these, we see that we must have

$$\mathbf{\Sigma}_{1:i-1,i}^T \mathbf{\Sigma}_{1:i-1,1:i-1}^{-1} = \left[\mathbf{0}, \frac{\mathbf{\Sigma}_{i-1,i}}{\mathbf{\Sigma}_{i-1,i-1}}\right],$$

noting that the $\mathbf{0}$ term originates from the fact that changing $x_{1:i-2}$ must not lead to changes in the left–hand side of (13). Rearranging gives

$$\mathbf{\Sigma}_{1:i-1,i}^T = \left[\mathbf{0}, \frac{\mathbf{\Sigma}_{i-1,i}}{\mathbf{\Sigma}_{i-1,i-1}}\right] \mathbf{\Sigma}_{1:i-1,1:i-1} \tag{15}$$

such that we can construct $\mathbf{\Sigma}_{1:i-1,i}^T$ from $\mathbf{\Sigma}_{1:i-1,1:i-1}$ and $\mathbf{\Sigma}_{i-1,i}$ using a matrix multiplication. Simple substitution into the left–hand size of (14) shows that this also provides a solution to our second required equality as well. Thus any $\mathbf{\Sigma}$ that satisfies (15) for all i will produce our desired dependency relationship and further ensure that the NSSP used in the main paper defines the desired target density.

We can construct such a $\mathbf{\Sigma}$ by the following process:

Algorithm 1 Generate $\mathbf{\Sigma}$

```
1: Generate $\mathbf{\Sigma}_{1,1}$
2: for $i = 2$ to $n_x$ do
3:     Generate $\mathbf{\Sigma}_{i,i}$ and $\mathbf{\Sigma}_{i-1,i}$
4:     $\mathbf{\Sigma}_{i,1:i-1} \leftarrow \left[\mathbf{0}, \frac{\mathbf{\Sigma}_{i-1,i}}{\mathbf{\Sigma}_{i-1,i-1}}\right] \mathbf{\Sigma}_{1:i-1,1:i-1}$
5:     $\mathbf{\Sigma}_{1:i-1,i} \leftarrow \mathbf{\Sigma}_{i,1:i-1}^T$
6: end for
```

Note that the reassignments do not change the values of $\mathbf{\Sigma}_{i-1,i}$ as there is a canceling that ensures this remains the same. Here the $\mathbf{\Sigma}_{i,i}$ can be generated in completely arbitrary manner—provided they are positive (e.g. we could sample them from a gamma distribution)—but the generation of $\mathbf{\Sigma}_{i-1,i}$ must be done carefully to ensure that the covariance matrix remains positive–definite. This is achieved by ensuring that $\mathbf{\Sigma}_{i-1,i}^2 < \mathbf{\Sigma}_{i,i}\mathbf{\Sigma}_{i-1,i-1}$ so that all the conditional covariances are positive.

We generated $\mathbf{\Sigma}_{i,i}$ and $\mathbf{\Sigma}_{i-1,i}$ as following: $\epsilon_i \sim \text{Uniform}[0,1]$, $\mathbf{\Sigma}_{i,i} \leftarrow 1 + \epsilon_i$, and $r_i \sim \text{Uniform}[0, 0.7]$, $\mathbf{\Sigma}_{i-1,i} \leftarrow r_i \cdot \mathbf{\Sigma}_{i-1,i-1}$, while the mean $\boldsymbol{\mu}$ of our joint distribution was generated from a standard multivariate normal $\boldsymbol{\mu} \sim \mathcal{N}(0, I)$. This generation process is performed only once for each dimensionality of the problem $n_x = 4, 20, 60$, and then those 3 matrices $\mathbf{\Sigma}$ are reused for all of our experiments.

To finish our derivation of the ground truth we now simply need to incorporate our likelihood term $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{x}, \mathbf{\Sigma}_{\boldsymbol{y}})$ (where we take $\mathbf{\Sigma}_{\boldsymbol{y}} = I$ in practice). This is just a standard Gaussian unknown mean problem, yielding $\boldsymbol{x}|\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_{\boldsymbol{x}|\boldsymbol{y}}, \mathbf{\Sigma}_{\boldsymbol{x}|\boldsymbol{y}})$ where (Murphy, 2012)

$$\boldsymbol{\mu}_{\boldsymbol{x}|\boldsymbol{y}} = \left(\mathbf{\Sigma}^{-1} + \mathbf{\Sigma}_{\boldsymbol{y}}^{-1}\right)^{-1}\left(\mathbf{\Sigma}^{-1}\boldsymbol{\mu} + \mathbf{\Sigma}_{\boldsymbol{y}}^{-1}\boldsymbol{y}\right) \tag{16}$$

$$\mathbf{\Sigma}_{\boldsymbol{x}|\boldsymbol{y}} = \left(\mathbf{\Sigma}^{-1} + \mathbf{\Sigma}_{\boldsymbol{y}}^{-1}\right)^{-1}. \tag{17}$$

References

Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, and Frank Wood. Efficient probabilistic inference in the quest for physics beyond the standard model. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2019.

Neil M Ferguson, Derek AT Cummings, Christophe Fraser, James C Cajka, Philip C Cooley, and Donald S Burke. Strategies for mitigating an influenza pandemic. *Nature*, 442(7101):448–452, 2006.

Bradley Gram-Hansen, Christian Schröder de Witt, Tom Rainforth, Philip HS Torr, Yee Whye Teh, and Atılım Güneş Baydin. Hijacking malaria simulators with probabilistic programming. *ICML Workshop on AI for Social Good*, 2019.

Kevin P Murphy. *Machine Learning: A Probabilistic Perspective.* MIT Press, 2012.

Kaare Brandt Petersen and Michael Syskind Pedersen. The Matrix Cookbook. *https://archive.org/details/imm3274/mode/2up*, 2012.

T Smith, N Maire, A Ross, M Penny, N Chitnis, A Schapira, A Studer, B Genton, C Lengeler, Fabrizio Tediosi, et al. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13):1507–1516, 2008.

Statement of Authorship for joint/multi-authored papers for PGR thesis
To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	Efficient Bayesian Inference for Nested Simulators
Publication Status	☒ Published ☐ Accepted for Publication ☒Submitted for Publication ☐Unpublished and unsubmitted work written in a manuscript style
Publication Details	This version is an extension of Bradley Gram-Hansen, Christian Schroeder de Witt, Robert Zinkov, Saeid Naderiparizi, Adam Scibior, Andreas Munk, Frank Wood, Mehrdad Ghadiri, Philip Torr, Yee Whye Teh, Atilim Gunes Baydin, Tom Rainforth, Efficient Bayesian Inference for Nested Simulators, In Proceedings of the Advances in Approximate Bayesian Inference, (AABI), 2019. The extension presented in the thesis is titled, Effective Approximate Inference for Nested Simulators, and is currently under review at the Conference on Uncertainty in Artificial Intelligence (UAI) 2021.

Student Confirmation

Student Name:	Bradley Gram-Hansen		
Contribution to the Paper	The overall idea and theoretical framework were developed by myself and Rainforth. Discussions surrounding the work took place with Golinski, Baydin, Zinkov, Schroeder de Witt, Torr, Teh, Rainforth, and myself. I generated all source code, experiments, and figures. Rainforth and I primarily did the writing of the paper.		
Signature [illegible]		Date	12/02/2020

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Philip H.S. Torr
Supervisor comments

Signature [illegible]	Date	26/04/2021

This completed form should be included in the thesis, at the end of the relevant chapter.

7
Conclusions

To recap, in this thesis we have presented three pieces of original work, through four papers, with the common theme being enhancing existing probabilistic programming systems by expanding the class of probabilistic models that a given probabilistic programming system can handle with the use of compilation schemes, coroutines and new inference algorithms, with a focus on real-world simulators and probabilistic models that had mixtures of continuous and discontinuous variables. In these concluding remarks, we will summarise our contributions and suggest directions for future work. The first work presented in this thesis was Chapter 4, where we developed a series of compilation rules to enable the extension of non-differentiable, first-order probabilistic models, to differentiable first-order probabilistic programming systems. By developing these compilation rules and the low-level FOPPL, we enabled the generation of a compilation target that could be leveraged by existing first-order differentiable systems, without them having to redesign the underlying language or create an entirely new probabilistic programming system, to perform inference in probabilistic models with mixtures of continuous and discontinuous variables utilising statistically correct and computationally efficient inference algorithms, provided the discontinuities generated are measure zero. A unique feature of our low-level PPL, and a key feature of the compilation scheme, was its ability to detect discontinuities caused by latent variables in `conditional` statements, that are typically missed in existing systems. The ability to track variables in this way, provides a path to performing inference efficiently in *what if?* probabilistic models, where we want to understand causal relationships, which arise in many healthcare related scenarios.

In Chapter 5 we developed a collection of coroutines called the Probabilistic Programming eXecution (PPX) protocols that enable the compilation of real-world stochastic simulators written in arbitrary program languages to probabilistic programming systems by making small incisions in the underlying code base. This work, as we demonstrated in [Baydin et al., 2019a], Paper one in Chapter 5, and in [Gram-Hansen et al., 2019a] and [de Witt et al., 2020], showed that it is possible to turn pre-existing, large scale, model-rich, real-world simulators into probabilistic programs, without having to rewrite the simulator in a probabilistic programming system, giving simulators added utility by enabling them to leverage inference algorithms, without having to spend additional time or financial resources on reimplementing a probabilistic model that requires highly specialised knowledge to implement. Furthermore, in [Baydin et al., 2019b], Paper two in Chapter 5, we showed that once a given simulator is converted we can utilise the PPX protocols and scale the inference procedure over twenty-five thousand latent variables, showing that probabilistic programming can be efficiently applied to real-world simulators and at scale in HPC settings. This work, although in its infancy, presents lots of exciting future opportunities, such as automating the recalibration of existing simulators, without having to do this procedure manually. Moreover, building probabilistic models is subjective and time-consuming, so leverage existing stochastic simulators and converting them into probabilistic programs enables us to think of new ways to combine existing models. This could be, for example, conditioning on the field observations of the number of infections from the malaria vector in a given demographic, on the output of the OpenMalaria intervention simulations [Smith et al., 2008] and also feeding information from climatic data, or a weather simulator [Held, 2005], as climatic factors play a crucial role in how the malaria vector spreads [Cameron et al., 2015]. By doing this, we could allow for near-real time updates for how a malaria vector may move through a given demographic, enabling health practitioners to be preemptive, rather than reactive. The PPX protocols help as the OpenMalaria is a stochastic simulator, but, in its current form not amenable to inference on external observations and so one cannot calibrate the latent variables in the simulator for climatic factors and new field observations in an automated way, which is problematic from a time and complexity perspective, as the code base is very large and this procedure has to be done manually, taking weeks and months. However, we would be remissed for not discussing the challenges of implementing such a system. Even though the PPX protocols allow for this conversion, you still have to make modifications to the simulator source-code, and this may not be trivial to do in all simulators. Also, gradient based inference schemes cannot be currently used with the PPX protocols as there is no way to pass gradient information through the simulators. Finally, simulators, such as the OpenMalaria simulator [Smith et al., 2008], contain internal, nested inference procedures, which are not directly amenable to MCMC inference and led to the new inference schemes presented in Chapter 6.

Finally, we saw in Chapter 6 that drawing inferences from nested probabilistic models poses a substantial challenge due to the inability to evaluate even their unnormalised density, preventing the use of many standard inference procedures like MCMC. To work around this challenge we developed two approaches for conducting efficient Bayesian inference in nested probabilistic models, where internal nested inference procedures inhibit out ability to calculate the log density. The inference algorithms introduced are based on a two-step approach that first approximates the conditional densities of the individual sub-procedures, before using these approximations to run MCMC methods on the full program. By developing these methods we can utilise scalable MCMC methods as we can construct an approximate representation of the log density, in method 1, and in method 2 we can exactly construct the log density, we can then leverage this log density for inference. Because the sub-procedures can be dealt with separately and are lower-dimensional than that of the overall problem, this two-step process allows them to be isolated and thus be tractably dealt with, without placing restrictions on the overall dimensionality of the problem. However, implementing these inference procedures in a probabilistic programming system to enable inference in an arbitrary stochastic simulator that is not written directly in the probabilistic programming system is challenging from an engineering perspective. The main difficulty is not in locating the nested sub-procedures, but being able to detect them automatically and pass all input and output data from that nested procedure to our probabilistic programming system for inference for the given nested sub-procedures. The PPX protocols give us the ability to extract the inputs and outputs from any procedure in the simulator, but it does not allow, yet, for the automatic detection of nested-procedures. If the simulator was written directly in a probabilistic programming system, the task at hand would be straight forward, since you have direct access to all the source code and stochastic primitives. To do this for an arbitrary simulator is future work as it requires the development of new coroutines.

7.1 Future challenges

A fundamental challenge to inference in both FOPPLs and UPPLs lies in the behavior when the program encounters control flow statements. As programmers, we are used to writing conditional statements of the form `if A, else B` and when a program is run, only one branch is executed, when these branches are initiated via stochastic choices, evaluating just one branch is insufficient.

In the case when the expressions e take a restricted form we can utilise efficient inference algorithms, Chapter 4, but doing this in a UPPL where you could have nested functions as an expression within the conditional is challenging. Work by Le et al. [2020] has focused on performing inference in probabilistic models with stochastic control-flow in UPPLs using the re-weighted wake-sleep algorithm [Hinton et al., 1995], but this work fundamentally relies on importance sampling which does not scale as the dimensionality of the problem increases.

Beyond challenges for specific inference algorithms, there are meta-challenges for implementing inference in universal languages since the program is not compiled into the familiar graphical model representation, as it is for FOPPLs, thus, it is an open question what underlying representation or abstraction is appropriate for our probabilistic models. The PPX protocols developed in Chapter 5 go some way to addressing this, but, code bases are messy and so there will always be challenges in transforming arbitrary stochastic simulators into the probabilistic programming paradigm and it will be difficult to resolve these issues without manual intervention. The optimal strategy is to write the simulator directly in a probabilistic programming system, which, in many instances is inefficient if the simulator consists of tens-of-thousands to millions-of-lines of code. However, even if we do this, developing inference procedures that are efficient, scalable to both high-dimensional data and latent spaces, while being automatable for all probabilistic models is still on-going research.

A
Appendix

A.1 The manual coin-flip

We are going to start on the assumption that we have a non-informative prior, that is the probability that the outcome of a coin-flip is biased follows a uniform distribution on the interval [0,1], thus the form of the prior is $p(\mathbf{x} = H) = U(0,1)$ and we sample from this prior during each forward execution of the model, $\mathbf{x} \sim U(0,1)$. We choose this prior because we suspect the coin may be bias; however, we are uncertain in which direction it is bias, as we ran an additional one-hundred flips and the majority of coin-flip outcomes were tails. If we believe that the coin prefers one outcome more than another, we can incorporate that belief with a prior that follows a `beta` distribution.

Once we have chosen our prior we construct the likelihood of our model. For the biased coin, we have a natural distribution that describes our problem set-up, the binomial distribution $Bin(n, \mathbf{x})$. The Binomial distribution is a discrete distribution that enables us to derive the probability of determining N successes, in this case, the number of heads, $n = n_h$, from $N = n_h + n_t$ independent runs of our model, flipping the coin, where the result of each forward run is either H, with probability $p(\mathbf{x} = H) = \mathbf{x}$, or T with probability $p(\neg\mathbf{x}) = 1 - p(\mathbf{x} = H)$.

Thus, the model has the following form

$$\mathbf{x} \sim U(0,1) \tag{A.1}$$

$$p(n_h|\mathbf{x}, N) = \binom{N}{n_h}\mathbf{x}^{n_h}(1-\mathbf{x})^{N-n_h} \tag{A.2}$$

Utilising Bayes' theorem we construct the form of the posterior:

$$p(\mathbf{x}|n_h, n_t) = \frac{p(n_h|\mathbf{x}, N)p(\mathbf{x})}{p(n_h) = \int_0^1 p(n_h|\mathbf{x}, N)p(\mathbf{x})d\mathbf{x}} \tag{A.3}$$

However, in order to calculate the marginal, $p(n_h)$, we have to do some work. We utilise induction and integration by parts, and start with the following observation when $n_h = 0$:

$$\int_0^1 (1-\mathbf{x})^N d\mathbf{x} = \frac{1}{N+1} = \frac{0!(N-0)!}{(N+1)!} \tag{A.4}$$

assuming the formula holds for some n_h, we find that integrating by parts yields:

$$\int_0^1 \mathbf{x}^{n_h+1}(1-\mathbf{x})^{N-(n_h+1)} d\mathbf{x} = \frac{-\mathbf{x}^{n_h+1}(1-\mathbf{x})^{N-n_h}}{N-n_h} + \int_0^1 \frac{(n_t+1)\mathbf{x}^{n_h}(1-\mathbf{x})^{N-n_h}}{N-n_h} d\mathbf{x} \tag{A.5}$$

$$= \frac{(n_h+1)}{(N-n_h)} \int_0^1 \mathbf{x}^{n_h}(1-\mathbf{x})^{N-n_h} d\mathbf{x} \tag{A.6}$$

By performing this procedure recursively we arrive at:

$$\int_0^1 \mathbf{x}^{n_h+1}(1-\mathbf{x})^{N-(n_h+1)} d\mathbf{x} = \frac{(n_h+1)(n_h)!(N-n_t)}{(N-n_h)(N+1)!} \tag{A.7}$$

$$= \frac{(n_h+1)!(N-(n_h+1))}{(N+1)!} = Beta(n_h+1, n_t+1) \tag{A.8}$$

which is the $Beta$ function, where $n_t = N - n_h$.

Once we have acquired our normalisation factor, the marginal, we can construct the posterior density as we know the from of the likelihood, $p(y|\mathbf{x})$, and prior, $p(\mathbf{x})$:

$$p(\mathbf{x}|n_h, n_t) = \frac{1}{Beta(n_h+1, n_t+1)} \mathbf{x}^{n_h}(1-\mathbf{x})^{N-n_h} \tag{A.9}$$

From the posterior we can then calculate the moments, in particular, we are interested in the first moment under the distribution $p(\mathbf{x}|n_h, n_t)$, $\mathbb{E}_{p(\mathbf{x}|n_h,n_t)}[\mathbf{x}] = \int \mathbf{x} p(\mathbf{x}|n_h, n_t) d\mathbf{x}$, the expected probability that we observe heads from the flip of a coin.

$$\mathbb{E}[\mathbf{x}] = \int_0^1 \mathbf{x} \frac{1}{Beta(n_h+1, n_t+1)} \mathbf{x}^{n_h}(1-\mathbf{x})^{N-n_h} d\mathbf{x} \tag{A.10}$$

In this instance we can calculate this expectation analytically by expanding the polynomials and using integration by parts, once we define the number of observed heads, n_h. In Figure A.1, we see how the uncertainty in our beliefs changes as we observe a differing numbers of heads. Initially, after $N = 10$ runs, we believe the coin is unbiased, but we are still uncertain in the range of values x can take, with the expected value being $\mathbb{E}[\mathbf{x}] = \frac{1}{2}$, a fair coin. As we run more experiments we become more certain in our prediction, as the variance of the posteriors reduces and the density begins to peak around a set of points. Indeed, by the time we have run

$N = 1000$ experiments our beliefs inform us that the coin is bias, and we are more certain about how bias the coin is, that is we expect to get a head $\mathbb{E}[\mathbf{x}] = \frac{2}{3}$ of the time. So we infer that the coin is biased, and because we have a posterior distribution we can see how confident we are in that belief. This is a not only a nice feature of Bayesian inference, but important for many real-world applications that care about the robustness of a prediction.

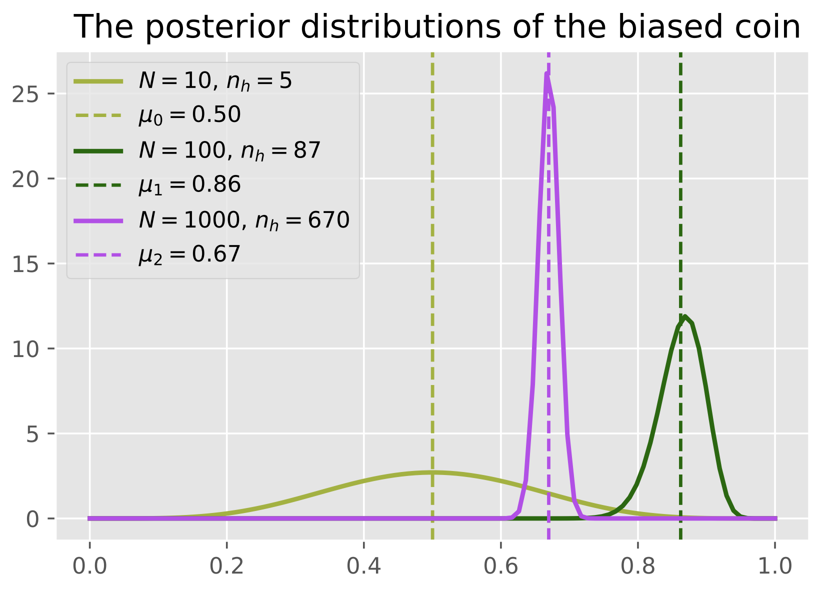


Figure A.1: As we generate more observations our posterior beliefs are updated. Here we plot the posteriors for a number of different runs of the experiment, $N = 10, 100, 1000$ and a varying number of observations during each experiment, $n_h = 5, 87, 670$.

Bibliography

N. L. Ackerman, C. E. Freer, and D. M. Roy. On the computability of conditional probability. *Journal of the ACM (JACM)*, 66(3):1–40, 2019.

H. M. Afshar and J. Domke. Reflection, Refraction, and Hamiltonian Monte Carlo. In *Advances in Neural Information Processing Systems*, pages 3007–3015, 2015.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–153, 2018.

A. G. Baydin, W. Bhimji, L. Heinrich, B. Gram-Hansen, S. Naderiparizi, A. Munk, J. Liu, L. Shao, G. Louppe, L. Meadows, et al. Efficient probabilistic inference in the quest for physics beyond the standard model. In *Advances in neural information processing systems*, pages 5459–5472, 2019a.

A. G. Baydin, L. Shao, W. Bhimji, L. Heinrich, B. Gram-Hansen, L. F. Meadows, J. Liu, A. Munk, S. Naderiparizi, G. Louppe, M. Ma, X. Zhao, P. Torr, V. Lee, K. Cranmer, Prabhat, and F. Wood. Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 978-1-4503-6229-0. doi: 10.1145/3295500.3356180. event-place: Denver, Colorado.

T. Bayes. LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, FRS communicated by Mr. Price, in a letter to John Canton, AMFR S. *Philosophical transactions of the Royal Society of London*, pages 370–418, 1763. Publisher: The Royal Society London.

G. Bhanot and A. D. Kennedy. Bosonic Lattice Gauge Theory With Noise. *Phys. Lett. B*, 157:70–76, 1985. doi: 10.1016/0370-2693(85)91214-6.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019. Publisher: JMLR. org.

D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017. Publisher: Taylor & Francis.

J. Borgström, U. Dal Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices*, 51(9):33–46, 2016.

A. Bouchard-Côté, S. J. Vollmer, and A. Doucet. The bouncy particle sampler: A nonreversible rejection-free markov chain monte carlo method. *Journal of the American Statistical Association*, 113(522): 855–867, 2018.

G. E. P. Box. Science and Statistics. *Journal of the American Statistical Association*, 71(356):791–799, Dec. 1976. ISSN 0162-1459. doi: 10.1080/01621459.1976.10480949. URL `https://www.tandfonline.com/doi/abs/10.1080/01621459.1976.10480949`.

E. Cameron, K. E. Battle, S. Bhatt, D. J. Weiss, D. Bisanzio, B. Mappin, U. Dalrymple, S. I. Hay, D. L. Smith, J. T. Griffin, and others. Defining the relationship between infection prevalence and clinical incidence of Plasmodium falciparum malaria. *Nature Communications*, 6:8170, 2015. Publisher: Nature Publishing Group.

O. Cappé, R. Douc, A. Guillin, J.-M. Marin, and C. P. Robert. Adaptive importance sampling in general mixture classes. *Statistics and Computing*, 18(4):447–459, 2008.

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017. Publisher: Columbia Univ., New York, NY (United States); Harvard Univ., Cambridge, MA ....

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. 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.

C. S. de Witt, B. Gram-Hansen, N. Nardelli, A. Gambardella, R. Zinkov, P. Dokania, N. Siddharth, A. B. Espinosa-Gonzalez, A. Darzi, P. Torr, et al. Simulation-based inference for global health decisions. In

ICML workshop on ML for Global health (ML4GH), 2020.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Physics letters B*, 1987. Publisher: Elsevier.

H. Ge, K. Xu, and Z. 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, 2018. URL `http://proceedings.mlr.press/v84/ge18b.html`.

A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis*. CRC press, 2013.

C. J. Geyer. Practical markov chain monte carlo. *Statistical science*, pages 473–483, 1992. Publisher: JSTOR.

W. R. Gilks, S. Richardson, and D. Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995.

T. Gleisberg, S. Hoeche, F. Krauss, M. Schönherr, S. Schumann, F. Siegert, and J. Winter. Event generation with SHERPA 1.1. *Journal of High Energy Physics*, 2009, Dec. 2008. doi: 10.1088/1126-6708/2009/02/007.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0-262-03561-8 978-0-262-03561-3.

N. D. Goodman and A. Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. 2014. URL `http://dippl.org`.

N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A Language for Generative Models. In *In UAI*, pages 220–229, 2008.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

B. Gram-Hansen, C. S. de Witt, T. Rainforth, P. H. Torr, Y. W. Teh, and A. G. Baydin. Hijacking Malaria Simulators with Probabilistic Programming. ICML, 2019a.

B. Gram-Hansen, C. S. de Witt, R. Zinkov, S. Naderiparizi, A. Scibior, A. Munk, F. Wood, M. Ghadiri, P. Torr, Y. W. Te, A. G. Baydin, and T. Rainforth. Efficient Bayesian Inference for Nested Simulators. In *Second Symposium on Advances in Approximate Bayesian Inference (AABI), Vancouver, Canada, 8 December 2019*, 2019b.

P. J. Green and S. Richardson. Hidden markov models and disease mapping. *Journal of the American Statistical Association*, 97:1055–1070, 2001.

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 1970. Publisher: Biometrika Trust.

I. M. Held. The gap between simulation and understanding in climate modeling. *Bulletin of the American Meteorological Society*, 86(11):1609–1614, 2005. Publisher: American Meteorological Society.

R. Hickey. The Clojure Programming Language. *Proceedings of the 2008 Symposium on Dynamic Languages*, 2008. Publisher: ACM.

G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

M. D. Hoffman and A. Gelman. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

C.-R. Hwang, S.-Y. Hwang-Ma, and S.-J. Sheu. Accelerating diffusions. *Ann. Appl. Probab.*, 15(2): 1433–1444, 05 2005. doi: 10.1214/105051605000000025. URL `https://doi.org/10.1214/105051605000000025`.

ISO. Programming languages c++. page 732, 1998.

P. Jäckel. *Monte Carlo Methods in Finance*. The Wiley Finance Series. Wiley, 2002. ISBN 978-0-471-49741-7. URL `https://books.google.co.uk/books?id=jG6BQgAACAAJ`.

D. Koller, N. Friedman, and F. Bach. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

T. A. Le. Inference for Higher Order Probabilistic Programs. *Masters Thesis, University of Oxford*, 2015.

T. A. Le, A. G. Baydin, and F. Wood. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

T. A. Le, A. R. Kosiorek, N. Siddharth, Y. W. Teh, and F. Wood. Revisiting reweighted wake-sleep for models with stochastic control flow. In *Uncertainty in Artificial Intelligence*, pages 1039–1049. PMLR, 2020.

L. Li and S. J. Russell. The BLOG Language Reference. Technical Report UCB/EECS-2013-51, EECS Department, University of California, Berkeley, May 2013. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-51.html`.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS—a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, 2000. Publisher: Springer.

A.-M. Lyne, M. Girolami, Y. Atchadé, H. Strathmann, and D. Simpson. On russian roulette estimates for bayesian inference with doubly-intractable likelihoods. *Statist. Sci.*, 30(4):443–467, 11 2015. doi: 10.1214/15-STS523. URL `https://doi.org/10.1214/15-STS523`.

D. J. MacKay. Introduction to Gaussian Processes. *NATO ASI Series F Computer and Systems Sciences*, 168:133–166, 1998. Publisher: Springer Verlag.

N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44 (247):335–341, 1949. ISSN 01621459. URL `http://www.jstor.org/stable/2280232`.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 1352–1359. Morgan Kaufmann Publishers Inc., 2005.

T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. *Infer.NET 2.6*. 2014.

J. Moller and R. P. Waagepetersen. *Statistical inference and simulation for spatial point processes*. CRC Press, 2003.

K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

I. Murray, Z. Ghahramani, and D. J. MacKay. MCMC for doubly-intractable distributions. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, pages 359–366. AUAI Press, 2006.

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

A. Nishimura, D. B. Dunson, and J. Lu. Discontinuous Hamiltonian Monte Carlo for discrete parameters and discontinuous likelihoods. *Biometrika*, 107(2):365–380, 03 2020. ISSN 0006-3444. doi: 10.1093/biomet/asz083. URL `https://doi.org/10.1093/biomet/asz083`.

A. B. Owen. *Monte Carlo theory, methods and examples*. 2013.

A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

A. Pfeffer. The design and implementation of IBAL: A generalpurpose probabilistic programming language. In *Harvard Univesity*. Citeseer, 2005.

M. Raberto, S. Cincotti, S. M. Focardi, and M. Marchesi. Agent-based simulation of a financial market. *Physica A: Statistical Mechanics and its Applications*, 299(1-2):319–327, 2001. Publisher: Elsevier.

T. Rainforth. *Automating Inference, Learning, and Design using Probabilistic Programming*. PhD Thesis, 2018a.

T. Rainforth. Nesting probabilistic programs. *arXiv preprint arXiv:1803.06328*, 2018b.

T. Rainforth, R. Cornish, H. Yang, A. Warrington, and F. Wood. On Nesting Monte Carlo Estimators. In *International Conference on Machine Learning*, pages 4267–4276, 2018.

D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic Programming in Python Using PyMC3. *PeerJ Computer Science*, 2:e55, 2016. Publisher: PeerJ Inc.

A. F. Smith and A. E. Gelfand. Bayesian statistics without tears: a sampling–resampling perspective. *The American Statistician*, 46(2):84–88, 1992.

T. Smith, N. Maire, A. Ross, M. Penny, N. Chitnis, A. Schapira, A. Studer, B. Genton, C. Lengeler, F. Tediosi, and others. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13):1507–1516, 2008. Publisher: Cambridge University Press.

T. A. Smith. Estimation of heterogeneity in malaria transmission by stochastic modelling of apparent deviations from mass action kinetics. *Malaria journal*, 7(1):12, 2008. Publisher: BioMed Central.

D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks. BUGS 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pages 1–59, 1996.

S. Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.

S. Staton, H. Yang, F. Wood, C. Heunen, and O. Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 525–534. ACM, 2016.

A. Stuhlmüller and N. D. Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014. Publisher: Elsevier.

D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

J.-W. Van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

G. Van Rossum and F. L. Drake Jr. Python reference manual. 1995.

D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.