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 Gilles Louppe, 4 Lei Shao, 5 Prabhat, 3 Kyle Cranmer, 2 Frank Wood 6

1University of Oxford; 2New York University; 3Lawrence Berkeley National Lab
4University of Liège; 5Intel Corporation; 6University of British Columbia
{gunes,bradley}@robots.ox.ac.uk; {lukas.heinrich,g.louppe,cranmer}@cern.ch
{wbhimji,prabhat}@lbl.gov; lei.shao@intel.com; fwood@cs.ubc.ca

Abstract

We present a novel framework that enables efficient probabilistic inference in large-scale scientific models by allowing the execution of existing domain-specific simulators as probabilistic programs, resulting in highly interpretable posterior inference. Our framework is general purpose and scalable, and is based on a cross-platform probabilistic execution protocol through which an inference engine can control simulators in a language-agnostic way. We demonstrate the technique in particle physics, on a scientifically accurate simulation of the $\tau$ (tau) lepton decay, which is a key ingredient in establishing the properties of the Higgs boson. High-energy physics has a rich set of simulators based on quantum field theory and the interaction of particles in matter. We show how to use probabilistic programming to perform Bayesian inference in these existing simulator codebases directly, in particular conditioning on observable outputs from a simulated particle detector to directly produce an interpretable posterior distribution over decay pathways. 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 Markov chain Monte Carlo sampling.

1 Introduction

Complex simulators are used to express stochastic generative models of data across a wide segment of the scientific community, with applications as diverse as hazard analysis in seismology [1], supernova shock waves in astrophysics [2], market movements in economics [3], and blood flow in biology [4]. 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 [5].

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 [6–9] and density ratio estimation [10, 11]. Alternatively, approximate Bayesian computation (ABC) [12, 13] 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 [14] algorithms are often used when the posterior is intractable, they are not directly applicable when the likelihood of the data generating process is unknown.
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 [15] and inference compilation [16] 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-platform probabilistic execution protocol (Figure 1) 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) [17] and importance sampling [18] 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, and it can be used by any other language 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) [19, 20] at CERN. This is achieved by coupling our framework with SHERPA\textsuperscript{1} [21], a state-of-the-art Monte Carlo event generator of high-energy reactions of particles, which is commonly used with GEANT\textsuperscript{2} [22], a toolkit for the simulation of the passage of the resulting particles through detectors. In particular, we perform inference in the case of $\tau$ (tau) lepton particle decay in a realistic detector, controlling the simulation within the standard SHERPA software with minimal modification and extracting posterior distributions in agreement with ground truths. To our knowledge this is the first time that universal probabilistic programming has been applied in this domain and in this scale, controlling a codebase of nearly one million lines of code. Our approach is readily 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 primarily motivated by applications in high-energy physics (HEP), which studies elementary particles and their interactions using energetic events created in particle accelerators such as the LHC at CERN. In this setting, the observed data are the result of interactions of particles generated in a collision event and observed through particle detectors. From these observations, we would like to infer the properties of the particles and interactions that generated them. Collisions happen millions of times per second, creating cascading particle decays in 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 inference techniques. However, they have developed sophisticated simulator

---

\textsuperscript{1}Simulation of High-Energy Reactions of Particles. \url{https://sherpa.hepforge.org/}

\textsuperscript{2}Geometry and Tracking. \url{https://geant4.web.cern.ch/}
packages such as SHERPA [21], GEANT [22], PYTHIA [23], Herwig++ [24], and MadGraph [25] to model physical processes and the interactions of particles with detectors. This is interesting from a probabilistic programming point of view, because HEP simulators are essentially very accurate probabilistic algorithms implementing the Standard Model and the passage of particles through matter (i.e., particle detectors). These simulators are coded in languages with unbounded recursion, 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 [26]. 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 models using general-purpose probabilistic inference algorithms.

The $\tau$ Lepton Decay. The specific HEP setting we focus on in depth in this paper is the decay of a $\tau$ lepton particle inside an LHC-like detector. This is a real use case in particle physics currently under active study by LHC physicists [27] and it is also of interest due to its importance to establishing the properties of the recently discovered Higgs boson [19, 20] through its decay to $\tau$ particles. Once produced, the $\tau$ decays to further particles observed within the detector according to certain decay channels. The probabilities of these decays or “branching ratios” are shown in Figure 2, which have been measured by other experiments and provide prior estimations for inference.

3 Related Work

3.1 Probabilistic Programming

Our work belongs in the family of sampling-based approximate inference techniques, which have been conventionally based on importance sampling [28] and Markov chain Monte Carlo (MCMC) methods [17] such as the Metropolis–Hastings (MH) algorithm [29]. These are computationally inefficient on large-scale models, due to the difficulty in choosing correct proposal distributions and handling increasing model dimensionality. Recent developments in inference algorithms, such as variational methods [30], extensions that combine deep learning [31, 8], MCMC samplers based on physical dynamics such as the No-U-Turn Sampler (NUTS) [32] and Stochastic Gradient Langevin Dynamics [33], and methods that use deep neural networks to amortize the cost of inference [34] such as inference compilation (IC) [16], have been targeting fast and scalable inference.

Probabilistic programming languages (PPLs) attempt to decouple inference algorithms from model building, by creating a simple, yet expressive, syntax that allows one to take advantage of these powerful inference algorithms on any probabilistic generative model expressed as a regular computer program. Universal PPLs allow the expression of unrestricted probability models in a Turing-complete fashion [35–37], and there is a recent trend in combining these with variational inference and deep learning, leading to tools such as Pyro [38], ProbTorch [39], and Edward [40]. This is in...
contrast to languages such as Stan [41] that target the more restricted model class of probabilistic graphical models [26].

3.2 Data Analysis in High-Energy Physics

Inference for an individual collision event in HEP is often referred to as reconstruction [42]. 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 HEP 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 [43–58]. 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 [59].

Currently HEP 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. The field is increasingly employing deep learning techniques allowing these algorithms to process high-dimensional, low-level data [60–62]. 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, which consists of (1) *pyprob*, a PyTorch-based [63] PPL and associated inference engines in Python, (2) PPX, a probabilistic programming execution protocol that defines a cross-platform interface for connecting models and inference engines implemented in different programming languages and executed in separate processes, (3) *pyprob_cpp*, a lightweight C++ front end that allows the execution of models written in C++ under the control of *pyprob*.

4.1 Designing a PPL for Existing Large-Scale Simulators

A shortcoming of the current state-of-the-art in PPLs is that they are not designed to directly support existing codebases, severely limiting their applicability to a very large body of existing probabilistic 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 [15]. Domain-specific simulators in HEP and other fields are commonly probabilistic in nature, thus satisfying the behavior random *sampling*, albeit generally from simplistic distributions such as the continuous uniform. By automatically “reinterpreting” these existing codebases with a proper rewiring of the (pseudo-)random number generator and introducing a construct for *conditioning*, we can execute existing simulators under the control of general-purpose inference engines designed for probabilistic programming. This enables the application of Bayesian inference techniques in these simulators, essentially treating the existing simulator as a joint prior distribution of latent and observed variables of a model, and obtaining posterior distributions over latent variables conditioned on realizations of observed variables.

To realize our framework, we implement *pyprob*, a universal PPL specifically designed to control models written not only in Python but also in other languages. Because the main inference technique we use in this PPL is based on deep neural networks, we base our PPL on PyTorch [63], whose automatic differentiation (AD) [64] feature with support for dynamic computation graphs has been

---

3https://github.com/probprog/pyprob
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, and instance (counter) of a sample, the execution of which describes a joint probability distribution between latent (unobserved) random variables \(x := (x_t)_{t=1}^T\) and observed random variables \(y := (y_n)_{n=1}^N\) given by

\[
p(x, y) := \prod_{t=1}^T f_{a_t}(x_t | x_{1:t-1}) \prod_{n=1}^N g_{n}(y_n | x_{<n}) ,
\]

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 \(g_{n}(\cdot | x_{<n})\) is the likelihood density given the sample values \(x_{<n}\) preceding observation \(y_n\). A PPL is a regular programming language equipped with sample and observe statements \([15]\) for sampling random variables with given prior probability distributions and conditioning random variables upon particular observed values.

Once a model \(p(x, y)\) is expressed as a probabilistic program, we are interested in performing inference in order to get posterior distributions \(p(x | y)\) of latent variables \(x\) conditioned on observed variables \(y\). In the sequential IS scheme, a weighted set of samples \(\{(w^k, x^k)\}_{k=1}^K\) is used to construct an empirical approximation of the posterior distribution \(\hat{p}(x | y) = \sum_{k=1}^K w^k (x^k - x) / \sum_{j=1}^K w^j\), where \(\delta\) is the Dirac delta function. The importance weights for a probabilistic program are expressed as

\[
w^k = \prod_{n=1}^N g_{n}(y_n | x^k_{1:T}) \prod_{t=1}^T \frac{f_{a_t}(x^k_t | x^k_{1:t-1})}{g_{a_t,i_t}(x^k_t | x^k_{1:t-1})} ,
\]

where \(g_{a_t,i_t}(\cdot | x^k_{1:t-1})\) 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 are training a deep neural network to receive the observed values \(y\) and return a set of adapted proposals \(g_{a_t,i_t}(x_{1:t-1}, y)\) such that their joint \(q(x | y)\) is close to the true posterior \(p(x | y)\). This is achieved by using a Kullback–Leibler divergence training objective

\[
\mathcal{L}(\phi) := \mathbb{E}_{p(y)} \left[ D_{\text{KL}} (p(x | y) \parallel q(x | y; \phi)) \right] \\
= \int_y p(y) \int_x p(x | y) \log \frac{p(x | y)}{q(x | y; \phi)} \, dx \, dy \\
= \mathbb{E}_{p(x,y)} \left[ -\log q(x | y; \phi) \right] + \text{const. ,}
\]

where \(\phi\) represents the neural network weights. The neural network weights \(\phi\) are optimized to minimize this objective by continually drawing training pairs \((x, y) \sim p(x, y)\) from the probabilistic program (i.e., the generative model, or the simulator). To simplify the task of training, only a subset of all addresses \((a_t, i_t)\) are handled by the neural network, and the remaining addresses are left to use the prior \(f_{a_t}\) as proposal during inference. The IC controlling of an address is exposed as a boolean flag called control, which can be applied to individual sample statements or delimited regions of the codebase. Expressed in simple terms, taking a desired outcome \(y\) from the probabilistic program as its input, the neural network learns to control the random number draws of latents \(x\) during the execution in such a way that makes the desired outcome likely.

The neural network architecture in IC is based on a stacked LSTM [67] 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 \(f_{\text{obs}}(y)\), the previously sampled value \(f_{a_{t-1},i_{t-1}}^\text{smp}(x_{t-1})\), the current distribution type \(f_{\text{type}}(a_t)\), and the current address \(f_{\text{addr}}(a_t, i_t)\). \(f_{\text{obs}}\) is a neural network specific to the domain (such as a 3D convolutional neural network for volumetric
inputs), \( f_{\text{amp}} \) are feed-forward modules, \( f_{\text{type}} \) are one-hot vectors denoting a prior distribution type from the set of supported distributions, \( f_{\text{addr}} \) are learned address embeddings optimized via backpropagation for each \((a_i, t_i)\) pair encountered in the program execution. The addressing scheme \( a_i \) is the main link between semantic locations in the probabilistic program and the inputs to the neural network. The addressing scheme in Python is based on an analysis of Python bytecode of the location where the PPL \texttt{sample} or \texttt{observe} statement is called, and in the PPX protocol (Section 4.2) the addresses \( a_i \) are produced and supplied by the side hosting and executing the model.

The joint proposal distribution of the neural network \( q(x|y) \) is factorized into proposals in each time step \( q_{a_i,t_i} \), whose type depends on the type of the prior \( f_{a_i} \). In the experiments presented in this paper (Section 5) the system uses categorical and continuous uniform distributions in the prior, for which we use, respectively, categorical and mixture of Kumaraswamy [68, 69] distributions as proposals parameterized by the neural network.

A common challenge for inference in real-world scientific models, such as those in HEP, is the presence of large dynamic ranges of prior probabilities for various outcomes. For instance, some particle decays are much more probable than others (Figure 2), 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 neural network 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 technique called “prior inflation” for automatically adjusting 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 neural network, and the unmodified original model is used during inference, ensuring that the importance weights (Eq. 2) and therefore the empirical posterior are correct under the unmodified real 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 (PPX) protocol 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) \texttt{sample} statements, and (3) \texttt{observe} statements. The implementation is based on flatbuffers, which is an efficient cross-platform serialization library through which we compile PPX 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 \texttt{sample} and \texttt{observe} statements over the protocol. We exchange these flatbuffers-encoded messages over ZeroMQ sockets, which allow seamless communication between separate processes in the same machine (using inter-process sockets) or across a network (using TCP).

Besides its use with \texttt{pyprob}, the PPX protocol defines a very flexible way of coupling any PPL system and model so that they can be (1) implemented in different programming languages and (2) executed in separate processes and on separate machines across networks. Thus PPX is similar in spirit to, and indeed inspired by, the Open Neural Network Exchange (ONNX) project for interoperability between machine learning frameworks. Note that, more than a serialization format, PPX enables runtime execution of probabilistic models under the control of inference engines in separate processes. We are releasing this language-agnostic protocol as a separately maintained project, together with the rest of our work in Python and C++.

### 4.3 Controlling SHERPA and the Standard Model

In this paper our target simulator is SHERPA [21], a Monte Carlo event generator of high-energy reactions of particles, which is a state-of-the-art simulator of the Standard Model of particle physics developed as an international effort within the HEP community. SHERPA, like many other large-scale

---

7. https://onnx.ai/
We couple SHERPA to pyprob_cpp 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. 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 using probabilistic programming techniques.

Differing from the conventions in the probabilistic programming community, 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 for choosing the $\tau$ lepton decay channel. The modification of this address to use the proper categorical prior allows an effortless application of the prior inflation technique (Section 4.1) to generate training data equally representing each channel.

\footnote{https://github.com/probprog/pyprobCpp}
We address this problem by implementing a novel trace evaluation scheme where during training we only consider the last (thus accepted) instance \( t_{last} \) of any address \((a_t, i_t)\) that fall within a rejection sampling loop. During inference, we use the same proposal distribution \( q_{a_t, t_{last}} \) in each loop execution. In other words, this corresponds to training the inference network with the state that concludes the loop (i.e., satisfies the acceptance criterion), effectively selecting proposal distributions such that the rejection loop is concluded in as few iterations as possible. This scheme works by annotating the sample statements within long-running rejection sampling loops with a boolean flag called replace, which, when set true, enables the behavior described for the given sample address.

5 Experiments

An important decay of the Higgs boson is to \( \tau \) 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 selects a set of particles to which the initial \( \tau \) 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 GEANT is typically used to model the interactions in a detector, for our initial studies we implement a fast, approximate detector simulation for a calorimeter with longitudinal and transverse segmentation (with resolution \( 20 \times 35 \times 35 \)). The fast detector simulation deposits most of the energy for electrons and \( \pi^0 \) into the first layers and charged hadrons (e.g., \( \pi^\pm \)) deeper into the calorimeter with larger fluctuations. Given raw 3D calorimeter observations, we would like to infer primarily the decay channel that the \( \tau \) lepton followed and the initial momenta \( p_x, p_y, \) and \( p_z \). Using our framework, we compute posterior distributions for the decay channel, initial momenta, and other latent quantities in the model conditioning on various simulated observations with known ground truth. The discrete variable for decay channel has a known prior distribution (Figure 2) given by the branching ratio of the \( \tau \) into 38 possible decay channels [73].

In Figure 3 we show inference results obtained from the IC and RMH MCMC engines, for a single observation \( y \) sampled from the model joint prior \( p(x, y) \) by running the simulation. The IC proposals are generated by an inference network trained with 1.6 million execution traces, and the IC engine controls (i.e., makes proposals different from the prior for) 47 addresses, of these in replacement (rejection sampling) mode, out of a total of 24,429 addresses. The RMH engine, by its very nature, controls all addresses encountered in the simulation. During IC network training, 440 trace types are encountered (Table 1), which represent the reoccurrence of the same sequence of addresses with different actual sample values. Traces reach lengths up to 7,514 and 1,190 respectively when looking at all and controlled-only samples (Figure 4).

As can be seen in Figure 3 (a), the network proposes values in agreement with the ground truth values. Figure 3 (b) shows the posterior after importance sampling guided by these proposals: this shows the correct posterior over particle decays was identified and also that related \( \tau \) decays are shown as possible alternatives with correct uncertainty, in agreement with RMH samples from the correct posterior in Figure 3 (c). Furthermore, correlations between the final state particle momenta are well reproduced. In order to make the RMH results for the test observation tractable, we start the chain from the ground truth trace (i.e., eliminating the need for burn in), which would not be possible for inference on real experimental data. This shows good agreement with the decay channel and event composition posteriors obtained from the IC engine that has access to observation \( y \) only (i.e., did not start from ground truth), and that would be used for fast inference with real experimental data.

The ability to connect posterior samples to the simulator code is a key advantage of our method in scientific applications. This connection enables inference results to be interpretable in the context of the physically-motivated latent process encoded by the simulator. Note that the posterior distributions

\[9\text{The controlled addresses are those that fall within sections of the codebase deemed fundamental in the solution of the } \tau \text{ decay problem, based on domain knowledge.}\]
presented in Figure 3 do not show all of the posterior information this technique encodes. Probabilistic programming gives us posteriors over the full space of execution traces covering the entire latent structure of the simulator, which we show at different levels of detail in Figure 5. Table 2 provides several examples of how the actual addresses \((a_i, i)\) look like within C++, allowing us to pinpoint all individual nodes in the codebase where the model behaves probabilistically. For instance, this approach gives us the ability to inspect aspects such as the chain of particle decays and interactions within the detector that led to particular posterior predictions, mirroring the standard task of event reconstruction [74]. This capability is not present in inference techniques that do not have access to the simulator, such as those solely based on neural networks.

6 Conclusions

Our work is the first step in subsuming the vast existing body of scientific simulators, which are essentially accurate generative models with decades of development behind them in many instances, into a universal probabilistic programming framework. The ability to scale probabilistic programming 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–inference engine interface, handling of priors with long tails and rejection sampling routines, addressing schemes, and IC network architecture, which make it difficult to cover in depth in a single paper. A main limitation of the introduced technique, currently, is the need for domain expert decisions in marking regions of codebase as controlled (only needed for the IC engine, and not needed for MCMC), which can potentially be automated in future work.

Our advancement allows one to perform model-based machine learning with interpretability, meaning that we understand the exact processes behind how the predictions are produced and the uncertainty in each prediction. With this novel framework providing a clearly defined interface between existing scientific simulators and probabilistic machine learning techniques, we expect to influence both communities to perform research at the intersection of science and machine learning.

Acknowledgments

Baydin and Wood are supported under DARPA PPAML through the U.S. AFRL under Cooperative Agreement FA8750-14-2-0006, Sub Award number 61160290-111668. Gram-Hansen is supported by the UK EPSRC CDT in Autonomous Intelligent Machines and Systems. Cranmer, Louppe, and Heinrich are supported through NSF ACI-1450310, PHY-1505463, PHY-1205376, and the Moore-Sloan Data Science Environment at NYU. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S.Department of Energy under Contract No. DE-AC02-05CH11231.

References


### Appendix

Table 1: Trace types encountered in the $\tau$ lepton decay model, identified according to the address sequence contained in each trace. Only the first 36 most frequent trace types are shown out of a total of 440 types encountered over 1,602,880 executions. Note that even when the address sequence is the same, the sampled values in each trace of the same type would be different.

<table>
<thead>
<tr>
<th>Freq.</th>
<th>Length</th>
<th>Addresses (showing controlled only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.106</td>
<td>72</td>
<td>A1, A2, A3, A5, A6, A32, A33, A31</td>
</tr>
<tr>
<td>0.105</td>
<td>41</td>
<td>A1, A2, A3, A5, A6, A499, A31</td>
</tr>
<tr>
<td>0.078</td>
<td>1,780</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A31</td>
</tr>
<tr>
<td>0.053</td>
<td>188</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.039</td>
<td>56</td>
<td>A1, A2, A3, A5, A6, A499, A17, A18, A26, A31</td>
</tr>
<tr>
<td>0.039</td>
<td>592</td>
<td>A1, A2, A3, A5, A6, A499, A17, A18, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.038</td>
<td>162</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A500, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.030</td>
<td>240</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A26, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.029</td>
<td>836</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.027</td>
<td>643</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.023</td>
<td>135</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A26, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.023</td>
<td>485</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.023</td>
<td>608</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.019</td>
<td>316</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.018</td>
<td>298</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.018</td>
<td>283</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.018</td>
<td>608</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.013</td>
<td>1,053</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>50</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>204</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>252</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>234</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>58</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.010</td>
<td>502</td>
<td>A1, A2, A3, A5, A6, A499, A17, A18, A20, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.009</td>
<td>216</td>
<td>A1, A2, A3, A5, A6, A499, A17, A500, A20, A21, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.009</td>
<td>1,053</td>
<td>A1, A2, A3, A5, A6, A499, A17, A18, A20, A41, A42, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.009</td>
<td>800</td>
<td>A1, A2, A3, A5, A6, A499, A17, A500, A20, A21, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.007</td>
<td>92</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A500, A26, A31</td>
</tr>
<tr>
<td>0.007</td>
<td>32</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A26, A31</td>
</tr>
<tr>
<td>0.007</td>
<td>78</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A99, A100, A101, A102, A510, A511, A898, A31</td>
</tr>
<tr>
<td>0.006</td>
<td>120</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A20, A508, A99, A100, A101, A102, A31</td>
</tr>
<tr>
<td>0.005</td>
<td>553</td>
<td>A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A500, A20, A21, A41, A42, A99, A100, A101, A102, A26, A31</td>
</tr>
</tbody>
</table>
(a) Distribution of trace lengths (all addresses). Min: 13, max: 7,514, mean: 383.58.

(b) Distribution of trace lengths (controlled addresses only). Min: 6, max: 1,190, mean: 13.61.

(c) Distribution of trace types, sorted in decreasing frequency.

Figure 4: Probabilistic trace statistics in the $\tau$ decay model automatically extracted from SHERPA executions via PPX.
Table 2: Addresses in the $\tau$ lepton decay problem (C++). Only the first 6 addresses are shown out of a total of 24,382 addresses encountered over 1,602,880 executions.

<table>
<thead>
<tr>
<th>Address ID</th>
<th>Full address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x45f; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0x9f]_Uniform_1</td>
</tr>
<tr>
<td>A2</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x477; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0x9f]_Uniform_1</td>
</tr>
<tr>
<td>A3</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x48f; ATOOLS:: Random:: Get(bool, bool)+0x1d5; probprog_RNG:: Get(bool, bool)+0x9f]_Uniform_1</td>
</tr>
<tr>
<td>A4</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x48f; ATOOLS:: Particle:: SetTime()+0xd; ATOOLS:: Flavour:: GenerateLifeTime() const+0x35; ATOOLS:: Random:: Get()+0x18b; probprog_RNG:: Get()+0xdef]_Uniform_1</td>
</tr>
<tr>
<td>A5</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x49d; SHERPA:: Event_Handler:: IterateEventPhases(SHERPA:: eventtype:: code&amp;, double&amp;, 0x1d2; SHERPA:: Hadron_Decays:: Treat(ATOLS:: Blob_List*, double&amp;, 0x1d2; SHERPA:: Decay_Handler_Base:: TreatInitialBlob(ATOLS:: Blob*, METOOLS:: Amplitude2_Tensor*, std:: vector&lt;ATOOLS:: Particle*, std:: allocator&lt;ATOOLS:: Particle*&gt; &gt; const&amp;)+0x1ab1; SHERPA:: Hadron_Decay_Handler:: CreateDecayBlob(ATOLS:: Particle*)+0x4cd; PHASIC:: Decay_Table:: Select() const+0x76e; ATOLS:: Random:: Get()+0x1d5; probprog_RNG:: Get()+0x9f]_Uniform_1</td>
</tr>
<tr>
<td>A6</td>
<td>[forward(x:: xarray_container&lt;x:: uvector&lt;double, std:: allocator&lt;double&gt; &gt;, (xt:: layout_type)1, xt:: svector&lt;unsigned long, 4ul, std:: allocator&lt;unsigned long&gt;, true&gt;, xt:: tensor_expression_tag&gt;)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: GenerateEvent(SHERPA:: eventtype:: code)+0x44d; SHERPA:: Event_Handler:: GenerateHadronDecayEvent(SHERPA:: eventtype:: code&amp;)+0x49d; SHERPA:: Hadron_Decays:: Treat(ATOLS:: Blob_List*, double&amp;)+0x1d2; SHERPA:: Hadron_Decays:: Treat(ATOLS:: Blob_List*, double&amp;)+0x4cd; PHASIC:: Decay_Table:: Select() const+0x76e; ATOLS:: Random:: Get()+0x1d5; probprog_RNG:: Get()+0x9f]_Uniform_1</td>
</tr>
</tbody>
</table>
Figure 5: Interpretability of the latent structure of the $\tau$ lepton decay process, automatically extracted from SHERPA executions via PPX. Showing model structure with increasing detail by taking an increasing number of most common trace types into account. Note that the flow is probabilistic at the shown nodes and deterministic along edges. Node labels denote address IDs (A1, A2, etc.) that correspond to uniquely identifiable parts of model execution such as those in Table 2. Addresses A1, A2, A3 correspond to momenta $p_x, p_y, p_z$, and A6 corresponds to the decay channel in Figure 3. Edge labels denote the overall frequency an edge is taken. Red: controlled node; green: controlled node with replacement (“rejection sampling mode”); blue: observed node; yellow: uncontrolled node.