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

Atılım Güneş Baydin,[1] **Lukas Heinrich,**[2] **Wahid Bhimji,**[3] **Lei Shao,**[4]
Saeid Naderiparizi,[5] **Andreas Munk,**[5] **Jialin Liu,**[3] **Bradley Gram-Hansen,**[1] **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 [47], supernova shock waves in astrophysics [34], market movements in economics [69], and blood flow in biology [68]. 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 [46].

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 [54, 66, 71, 77] and density ratio estimation [28, 33]. Alternatively, approximate Bayesian computation (ABC) [75, 79] 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 [21] 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 [42, 78] and inference compilation [61] 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, left) through which an inference engine can

Preprint. Under review.

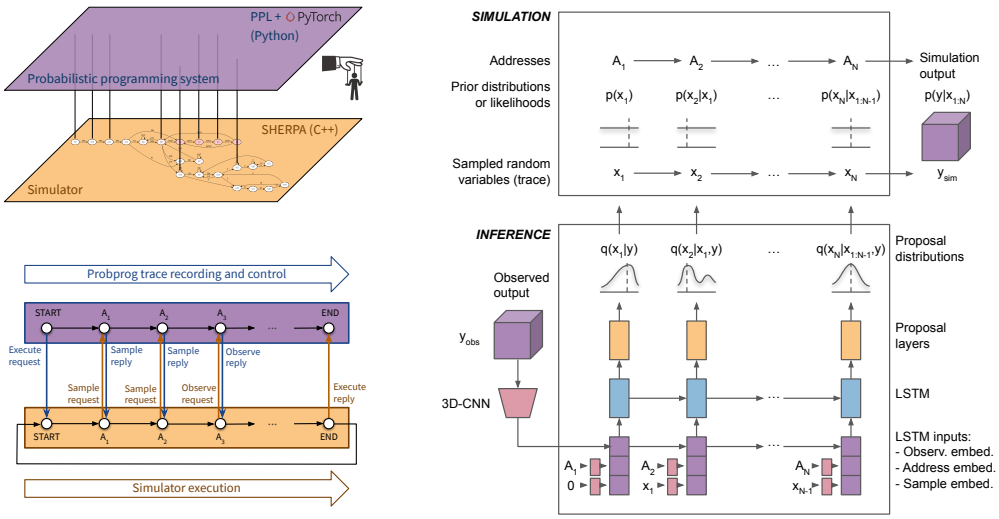


Figure 1: *Top left:* overall framework where the PPS is controlling the simulator. *Bottom left:* probabilistic execution of a single trace. *Right:* LSTM proposals conditioned on an observation.

control simulators in a language-agnostic way. We implement a range of general-purpose inference engines from the Markov chain Monte Carlo (MCMC) [24] and importance sampling [32] 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, 27] at CERN. This is achieved by coupling our framework with SHERPA[2] [40], 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 [40], Geant4 [5], Pythia8 [73], 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 these simulators are essentially very accurate generative models implementing the Standard Model of

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

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 [56]. 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, 27] through its decay to τ particles [12, 31, 44, 45]. 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 [78]. 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 [41, 81, 82], in contrast to languages such as Stan [37] that target the more restricted model class of probabilistic graphical models [56]. Inference engines available in PPLs range from MCMC-based lightweight Metropolis Hastings (LMH) [81] and random-walk Metropolis Hastings (RMH) [60] to importance sampling (IS) [11] and sequential Monte Carlo [32]. Modern PPLs such as Pyro [19] and TensorFlow Probability [30, 76] use gradient-based inference engines including variational inference [50, 55] and Hamiltonian Monte Carlo [51, 65] that benefit from modern deep learning hardware and automatic differentiation [18] features provided by PyTorch [67] 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 IC algorithm [61] that enables amortized inference [38].

3.2 Data Analysis in Particle Physics

Inference for an individual collision event in particle physics is often referred to as reconstruction [59]. 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, 22, 26, 35, 36, 43, 57, 63, 64, 72, 74]. 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 [23].

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, 53]. The field is increasingly employing deep learning techniques allowing these algorithms to process

high-dimensional, low-level data [14, 17, 29, 52, 70]. 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 [67] 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 [42, 78]. 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 a re-routing of 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. Because we would like to employ deep neural networks (NNs) in one of our inference engines, we base our PPL on PyTorch [67], whose automatic differentiation feature with support for dynamic computation graphs [18] has been crucial in our implementation. Our PPL currently has two families of inference engines: (1) MCMC of the lightweight Metropolis–Hastings (LMH) [81] and random-walk Metropolis–Hastings (RMH) [60] varieties, and (2) sequential importance sampling (IS) [11, 32] with its regular (i.e., sampling from the prior) and inference compilation (IC) [61] varieties. The IC technique, where a recurrent 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. 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,[5] 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

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

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 $\mathbf{y}$.

[4] `https://github.com/probprog/pyprob` [5] An "address" is a unique label representing the current execution point of the program. In our system it is based on a concatenation of stack frames leading up to the point of each random number draw.

Inference engines of the MCMC family, designed to work in the space of probabilistic execution traces [60, 78, 81], constitute the gold standard for obtaining samples from the true posterior of a probabilistic program. 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 their joint $q(\mathbf{x}|\mathbf{y})$ is close to the true posterior $p(\mathbf{x}|\mathbf{y})$. This is achieved by using a Kullback–Leibler divergence training objective $\mathbb{E}_{p(\mathbf{y})}\left[D_{\mathrm{KL}}\left(p(\mathbf{x}|\mathbf{y}) \,||\, q(\mathbf{x}|\mathbf{y};\phi)\right)\right]$ 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 desired outcome likely (Figure 1, right).

The NN architecture in IC is based on a stacked LSTM [49] 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 $f^{\mathrm{obs}}(\mathbf{y})$, the current address $f^{\mathrm{addr}}(a_t, i_t)$, and the previously sampled value $f^{\mathrm{smp}}_{a_{t-1},i_{t-1}}(x_{t-1})$. 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 [81] 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 we use, respectively, categorical and mixture of truncated Gaussian distributions as proposals parameterized by the 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)[6] 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,[7] 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[8] [48] 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 our Python PPL, the protocol defines a very flexible way of coupling any PPL system to any 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 we present this protocol as a probabilistic programming analogue to the Open Neural Network Exchange (ONNX)[9] project for interoperability between deep learning frameworks. Note that, more than a serialization format, the protocol 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's Simulation of Fundamental Particle Physics

We demonstrate our framework with SHERPA [40], 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.[10] We couple SHERPA to the front end 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.

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 τ lepton decay channel. The modification of this address 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 [39] 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 [58] 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 where during 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. During inference, we use this same proposal distribution $q_{a_t, i_{\text{last}}}$ in each loop execution. In other words, this corresponds to training the inference NN with the state that concludes the loop (i.e., satisfies the acceptance criterion), effectively learning 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.

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

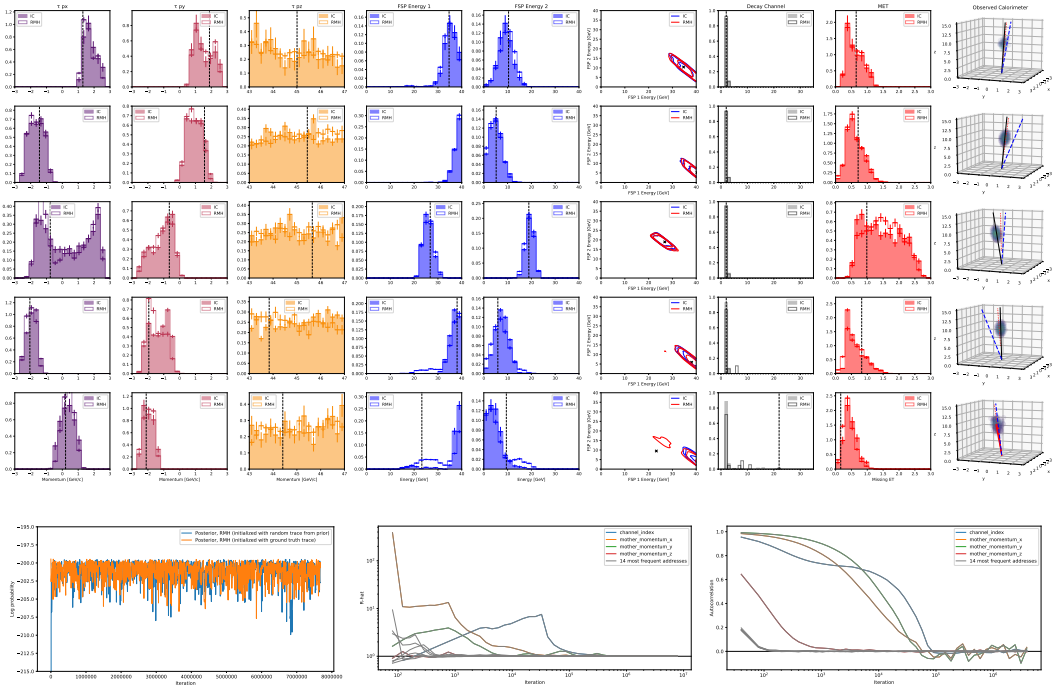


Figure 2: RMH and IC results where a Channel 2 decay event ($\tau \rightarrow \pi^{-}\nu_{\tau}$) is the mode of the posterior distribution. Note that the eight variables shown are just a subset of the full latent state of several thousand addresses (Figure 5, appendix). Vertical lines indicate the point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

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 $20\times35\times35$ 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 ($\tau \rightarrow \pi^{-}\nu_{\tau}$). 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 [25, 80] 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.[11] As an example, in Figure 2 (bottom) we show the joint log-probability, GR

[11] 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.

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 ($\tau \rightarrow K^+K^-K^+\nu_\tau$), a type of decay producing calorimeter depositions with similarity to Channel 2 decays and with extremely low probability in the prior (Figure 8), 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.

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, rejection sampling routines, addressing schemes, and IC network architectures, 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.

Acknowledgments

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] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] 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`.

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

[54] 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.

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

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

[57] 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.

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

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] G. Papamakarios, T. Pavlakou, and I. Murray. Masked autoregressive flow for density estimation. *arXiv preprint arXiv:1705.07057*, 2017.

[67] 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.

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

[69] 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.

[70] 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.

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

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

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

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

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

[76] 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.

[77] 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.

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

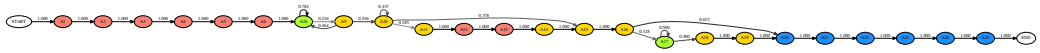
[79] R. D. Wilkinson. Approximate Bayesian computation (ABC) gives exact results under the assumption of model error. *Statistical Applications in Genetics and Molecular Biology*, 12(2):129–141.

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

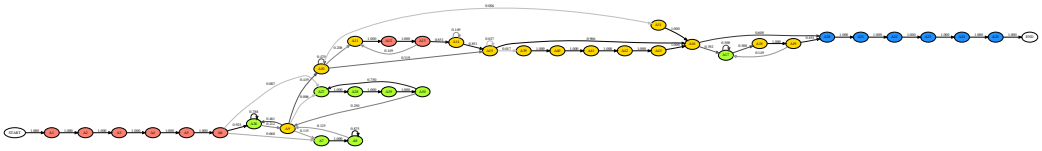
[81] D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

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

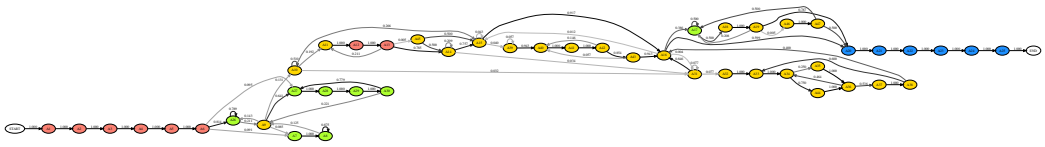
Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model (Supplementary Material)



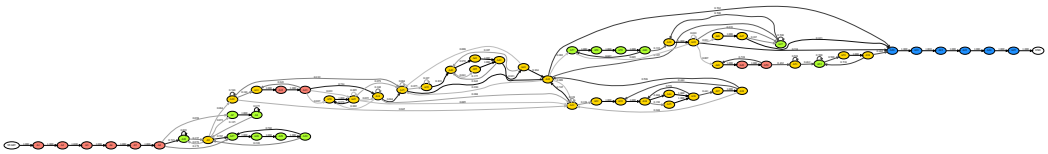
(a) Latent probabilistic structure of the 10 most frequent trace types.



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

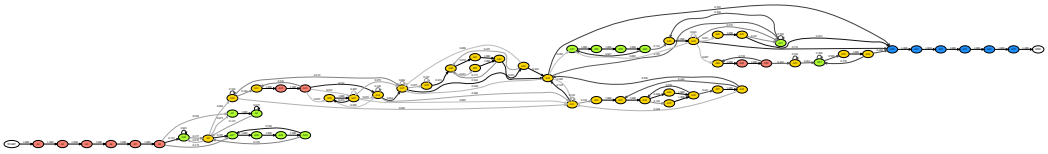


(c) Latent probabilistic structure of the 100 most frequent traces types.

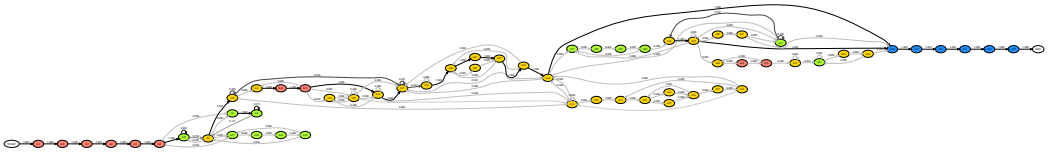


(d) Latent probabilistic structure of the 250 most frequent traces types.

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*: controlled with replacement ("rejection sampling mode"); *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*: controlled with replacement ("rejection sampling mode"); *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.

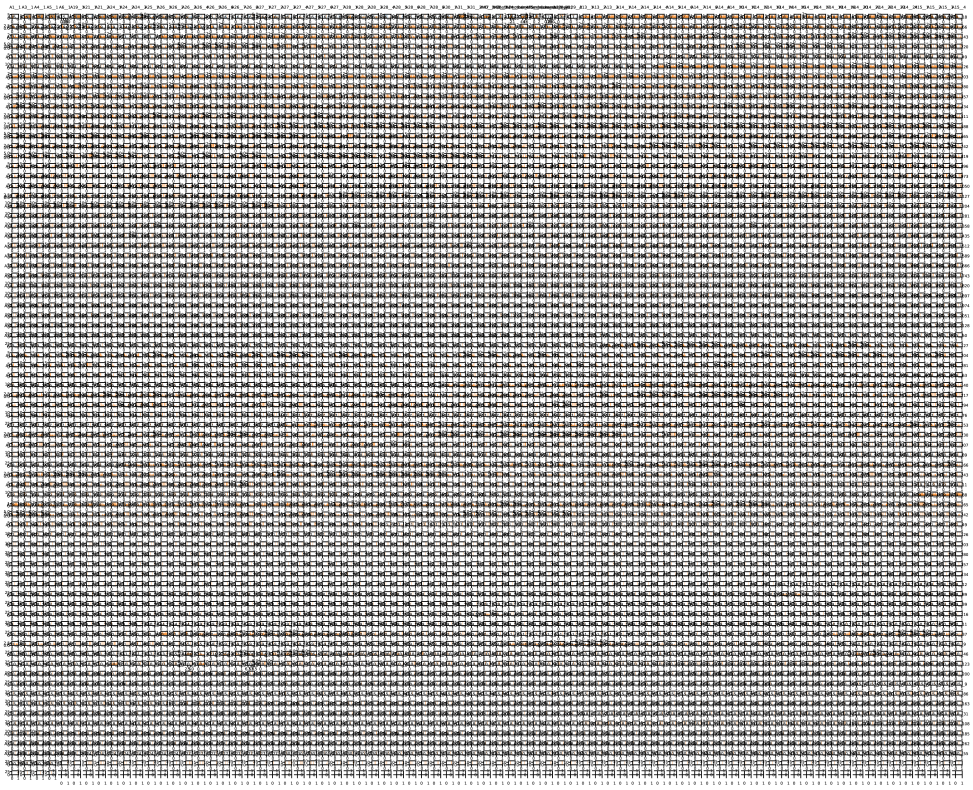


Figure 5: Example posterior over the entire latent state of the SHERPA simulator, conditioned on a single observed calorimeter. For the observation used, the posteriors presented in this figure contain approximately 6k addresses out of a total of approximately 24k addresses in the whole simulator. The 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.

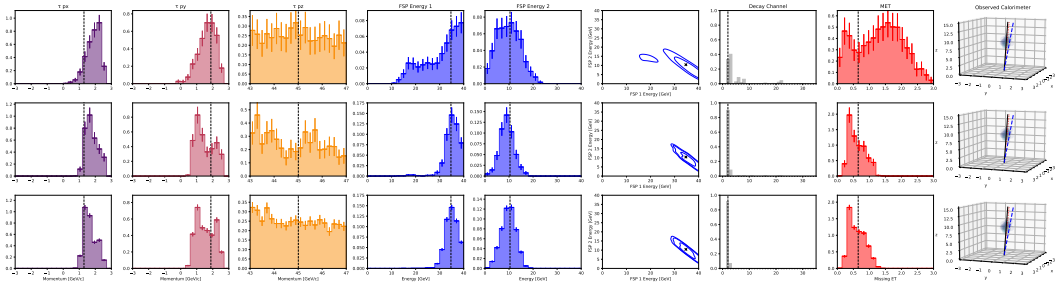


Figure 6: Steps of constructing the IC posterior for a Channel 2 GT event (first test case in Figure 2). The IC proposal (top row) is produced by the trained inference network. It is then weighted using Equation 3, giving IC posterior (middle row). The corresponding true posterior from RMH (MCMC) baseline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

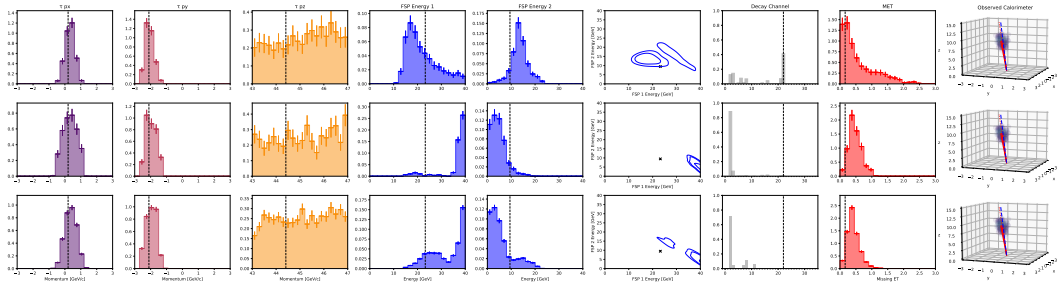


Figure 7: Steps of constructing the IC posterior for a Channel 22 GT event (last test case in Figure 2). The IC proposal (top row) is produced by the trained inference network. It is then weighted using Equation 3, giving IC posterior (middle row). The corresponding true posterior from RMH (MCMC) baseline 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.

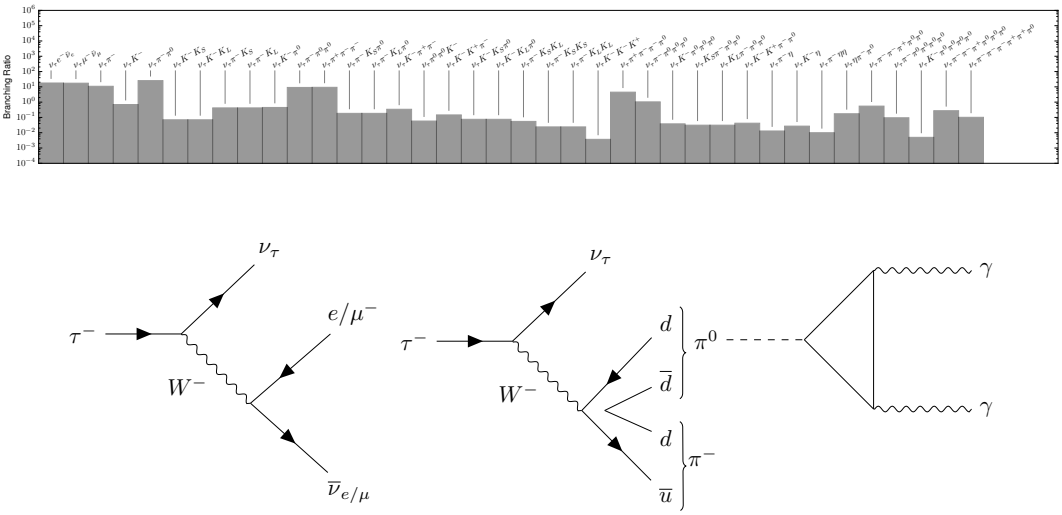


Figure 8: *Top:* branching ratios of the τ lepton, effectively the prior distribution of the decay channels in SHERPA. Note that the scale is logarithmic. *Bottom:* Feynman diagrams for τ decays illustrating that these can produce multiple detected particles.

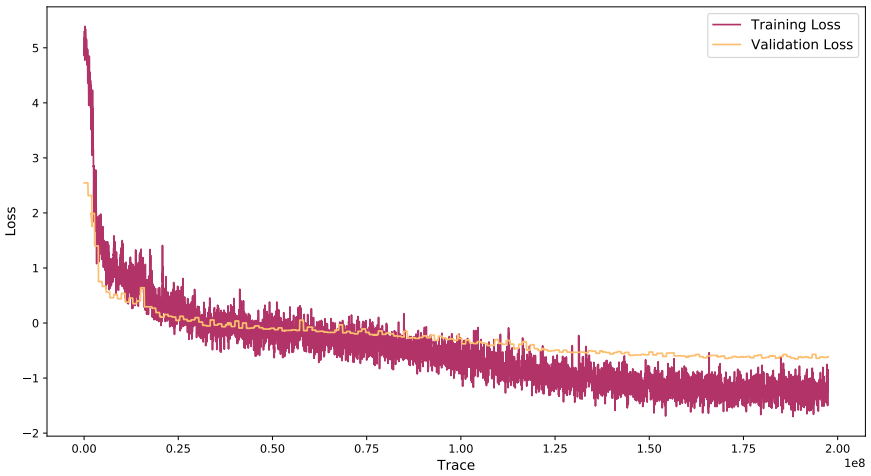


Figure 9: Training and validation losses of the IC inference NNs used for the results presented in Section 5. The network has 143,485,048 parameters and has been trained for 40 epochs. Network configuration: an LSTM with 512 hidden units; an observation embedding of size 256, encoded with a 3D convolutional NN (CNN) [62] 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 [20] (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