Probabilistic Programming and Inference in Particle Physics

Atılım Güneş Baydin, Wahid Bhimji, Kyle Cranmer, Bradley Gram-Hansen, Lukas Heinrich, Victor Lee, Jialin Liu, Gilles Louppe, Larry Meadows, Andreas Munk, Saeid Naderiparizi, Prabhat, Lei Shao, Frank Wood

Atılım Güneş Baydin
gunes@robots.ox.ac.uk

International Centre for Theoretical Physics
Trieste, Italy, 9 April 2019

UNIVERSITY OF
OXFORD

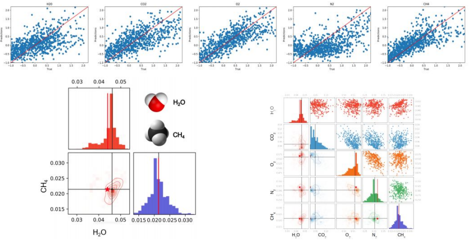
About me

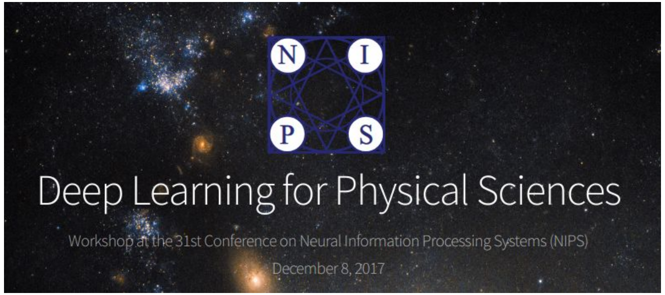
http://www.robots.ox.ac.uk/~gunes/

I work in probabilistic programming and machine learning for science

- High-energy physics
- Space sciences, NASA Frontier Development Lab, ESA Gaia collaboration
- Workshop in Deep Learning for Physical Sciences at NeurIPS conference

Other interests: automatic differentiation, hyperparameter optimization, evolutionary algorithms, computational physics





NASA FDL
frontierdevelopmentlab.org

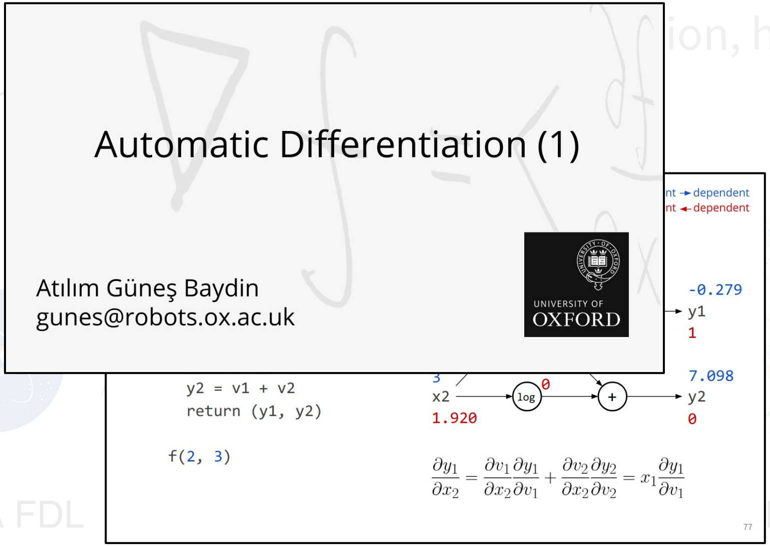
Exoplanetary atmospheres
https://arxiv.org/abs/1811.03390

https://dl4physicalsciences.github.io/

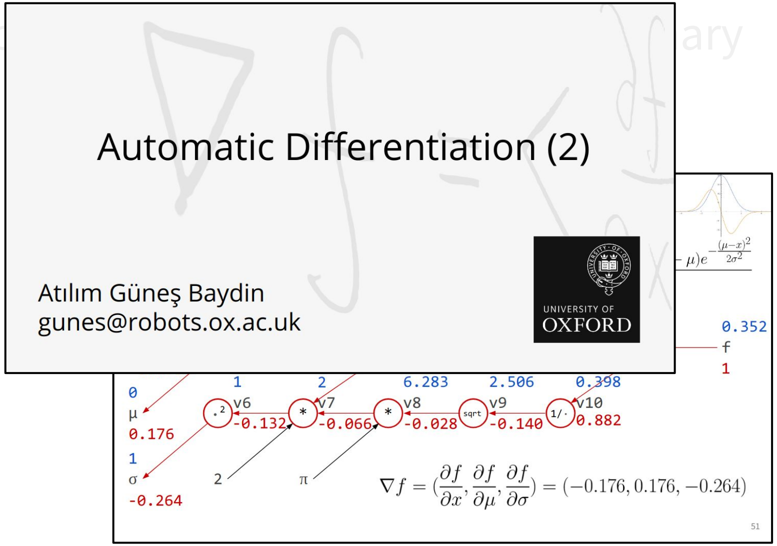
Automatic differentiation / differentiable programming

Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, *18*, pp.1-43. https://arxiv.org/abs/1502.05767

https://docs.google.com/presentation/d/1aBX-wgGmO8Gfl2bdZQBWdAlQjP_nj8_TLLceAbC-pKA/edit?usp=sharing



https://docs.google.com/presentation/d/1NTodzA0vp6zLljJ0v4vXpbz9z_Pe8mWaNDtD5QdK3v4/edit?usp=sharing



Probabilistic programming

Probabilistic programming

Probabilistic models define a set of random variables and their relationships

- Observed variables
- Unobserved (hidden, latent) variables

Probabilistic programming

Probabilistic models define a set of random variables and their relationships

- Observed variables
- ~~Unobserved (hidden, latent) variables~~ ***HEP: Monte Carlo truth***

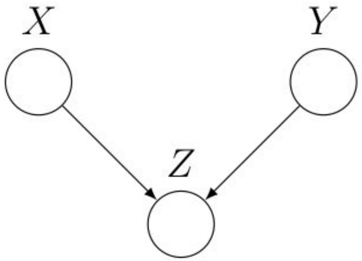
Probabilistic programming

Probabilistic models define a set of random variables and their relationships

- Observed variables
- ~~Unobserved (hidden, latent) variables~~ ***HEP: Monte Carlo truth***

Probabilistic graphical models use graphs to express conditional dependence

- Bayesian networks
- Markov random fields (undirected)



$$p(x, y, z) = p(x)p(y)p(z|x, y)$$

Probabilistic programming

Probabilistic models define a set of random variables and their relationships

- Observed variables
- ~~Unobserved (hidden, latent) variables~~ ***HEP: Monte Carlo truth***

Probabilistic programming extends this to *“ordinary programming with two added constructs”* (Gordon et al. 2014):

- **Sampling** from distributions
- **Conditioning** random variables by specifying observed values

```
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
return(c1, c2);
```

```
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
observe(c1 || c2);
return(c1, c2);
```

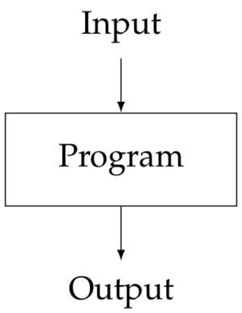
Inference

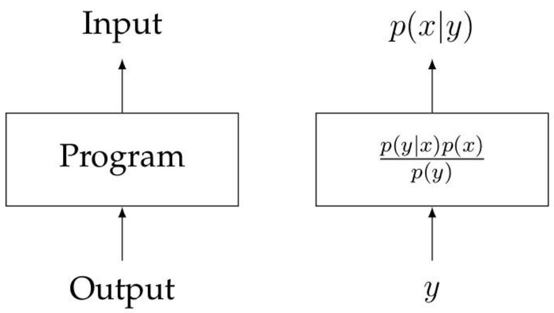
With a probabilistic program, we define a joint distribution of **unobserved** and **observed** variables $p(x, y)$

Inference engines give us distributions over **unobserved** variables, given **observed** variables (data)

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

Ordinary program





Probabilistic program

Inference engines

Model writing is decoupled from running inference

After writing the program, we execute it using an **inference engine**

- Exact (limited applicability)
    - Belief propagation
    - Junction tree algorithm
- Approximate (very common)
    - Deterministic
        - Variational methods
    - Stochastic (sampling-based)
        - Monte Carlo methods
            - Markov chain Monte Carlo (MCMC)
            - Sequential Monte Carlo (SMC)

Probabilistic programming languages (PPLs)

- Anglican (Clojure)
- Church (Scheme)
- **Edward, TensorFlow Probability (Python, TensorFlow)**
- **Pyro (Python, PyTorch)**
- Figaro (Scala)
- LibBi (C++ template library)
- PyMC3 (Python)
- Stan (C++)
- WebPPL (JavaScript)

For more, see http://probabilistic-programming.org

Large-scale simulators as probabilistic programs

Interpreting simulators as probprog

A stochastic simulator implicitly defines a probability distribution by **sampling** (pseudo-)random numbers
→ already satisfying one requirement for probprog

Idea:

- Interpret all RNG calls as **sampling** from a prior distribution
- Introduce **conditioning** functionality to the simulator
- Execute under the control of general-purpose inference engines
- Get posterior distributions over all simulator latents conditioned on observations

Interpreting simulators as probprog

A stochastic simulator implicitly defines a probability distribution by **sampling** (pseudo-)random numbers
→ already satisfying one requirement for probprog

Advantages:

- Vast body of existing scientific simulators (accurate generative models) with years of development: MadGraph, Sherpa, Geant4
- Enable model-based (Bayesian) machine learning in these
- Explainable predictions directly reaching into the simulator (simulator is not used as a black box)
- Results are still from the simulator and meaningful

Coupling probprog and simulators

Several things are needed:

- A PPL with with simulator control incorporated into design
- A language-agnostic interface for connecting PPLs to simulators
- Front ends in languages commonly used for coding simulators

Coupling probprog and simulators

Several things are needed:

- A PPL with with simulator control incorporated into design
  pyprob

- A language-agnostic interface for connecting PPLs to simulators
  PPX - the *P*robabilistic *P*rogramming e*X*ecution protocol

- Front ends in languages commonly used for coding simulators
  pyprob_cpp

pyprob

https://github.com/probprog/pyprob

A PyTorch-based PPL

Inference engines:

- Markov chain Monte Carlo
  - Lightweight Metropolis Hastings (LMH)
  - Random-walk Metropolis Hastings (RMH)
- Importance Sampling
  - Regular (proposals from prior)
  - Inference compilation (IC)

pyprob

https://github.com/probprog/pyprob

A PyTorch-based PPL

PyTorch

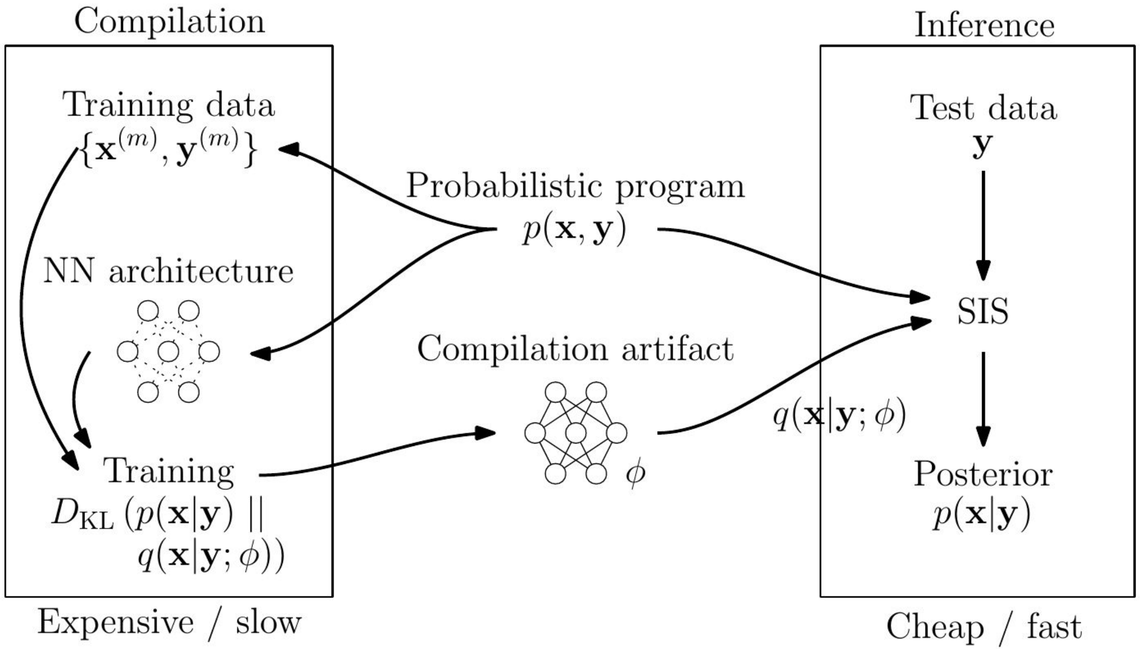
Inference engines:

- Markov chain Monte Carlo
  - Lightweight Metropolis Hastings (LMH)
  - Random-walk Metropolis Hastings (RMH)
- Importance Sampling
  - Regular (proposals from prior)
  - **Inference compilation (IC)**

Le, Baydin and Wood. Inference Compilation and Universal Probabilistic Programming. AISTATS 2017 *arXiv:1610.09900*.

Inference compilation

Transform a generative model implemented as a probabilistic program into a trained neural network artifact for performing inference



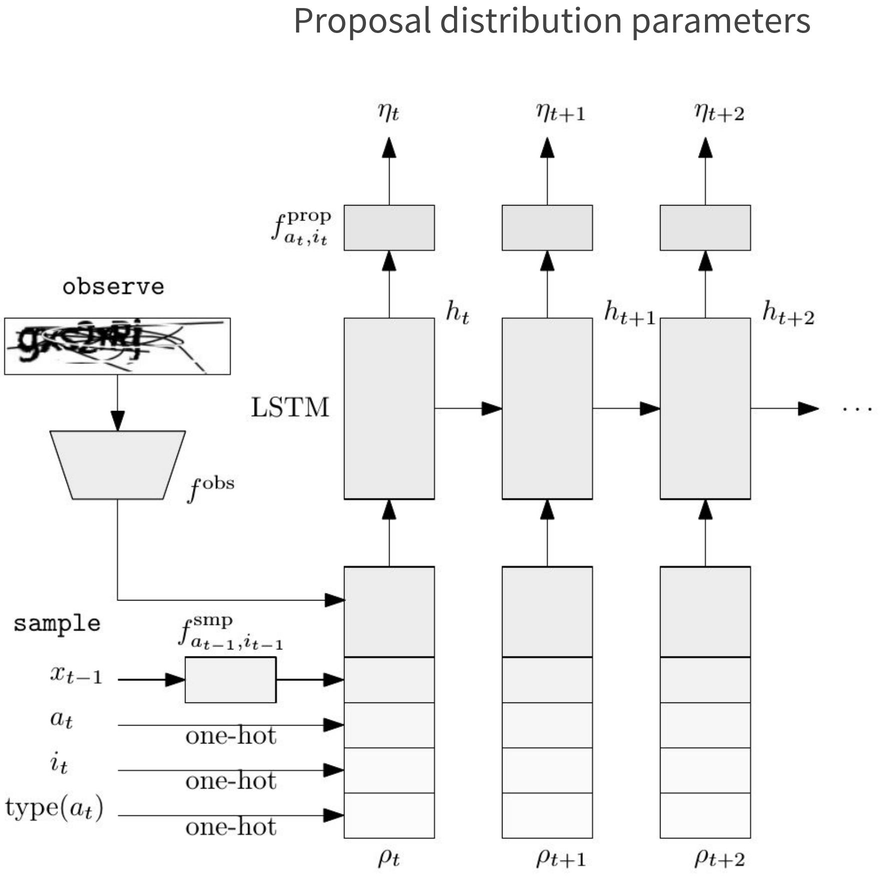
Inference compilation

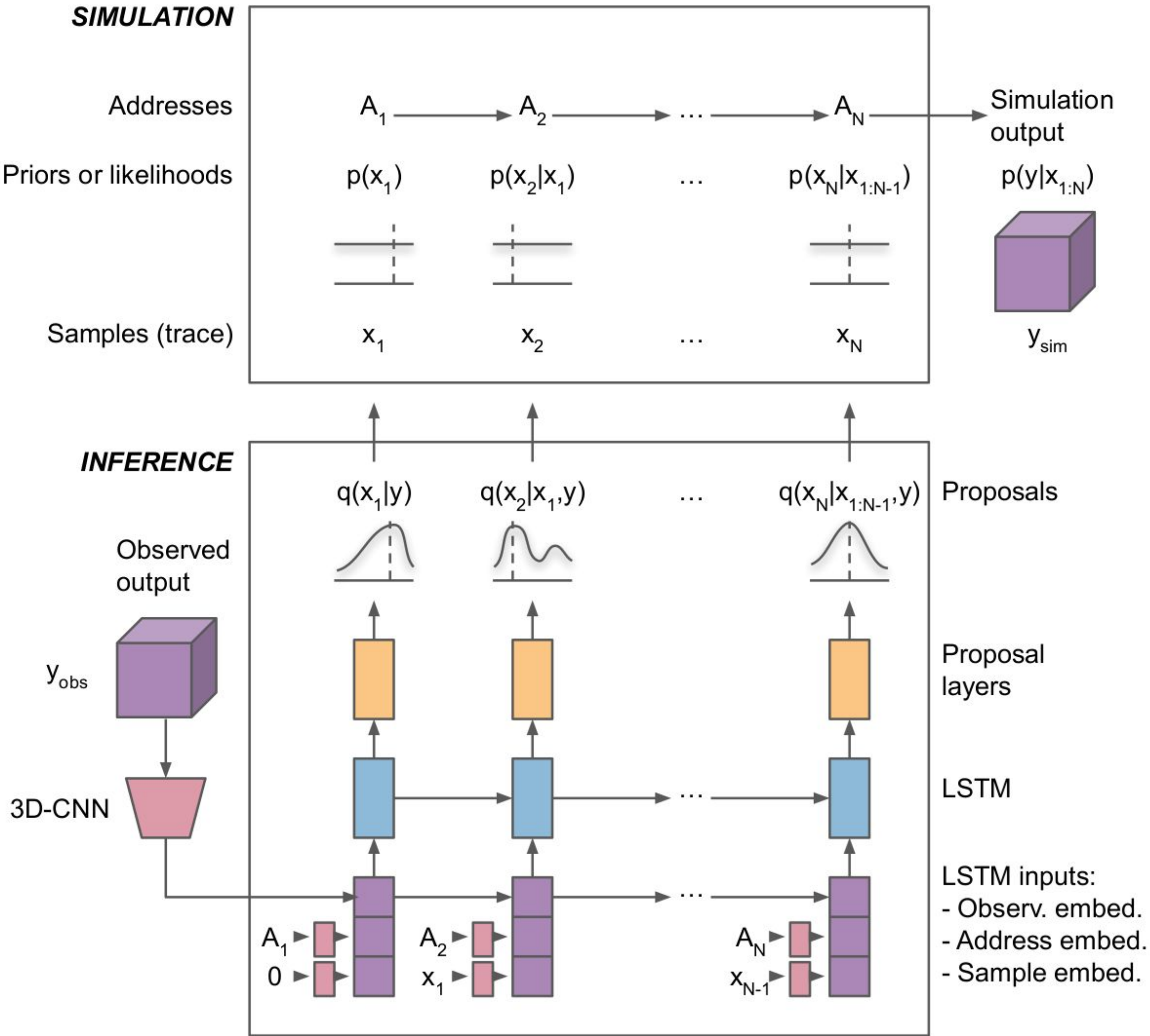
- A stacked LSTM core
- Observation embeddings, sample embeddings, and proposal layers specified by the probabilistic program

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

sample value | sample address | sample instance | trace length

$$(\boxed{x_t}, \boxed{a_t}, \boxed{i_t})_{t=1}^{\boxed{T}}$$




SIMULATION
Addresses
A1 A2 … AN
Simulation output
Priors or likelihoods
p(x1) p(x2|x1) … p(xN|x1:N-1)
p(y|x1:N)
Samples (trace)
x1 x2 … xN
ysim
INFERENCE
q(x1|y) q(x2|x1,y) … q(xN|x1:N-1,y)
Proposals
Observed output
yobs
Proposal layers
3D-CNN
LSTM
LSTM inputs:
- Observ. embed.
- Address embed.
- Sample embed.
A1 0 A2 x1 AN xN-1

PPX

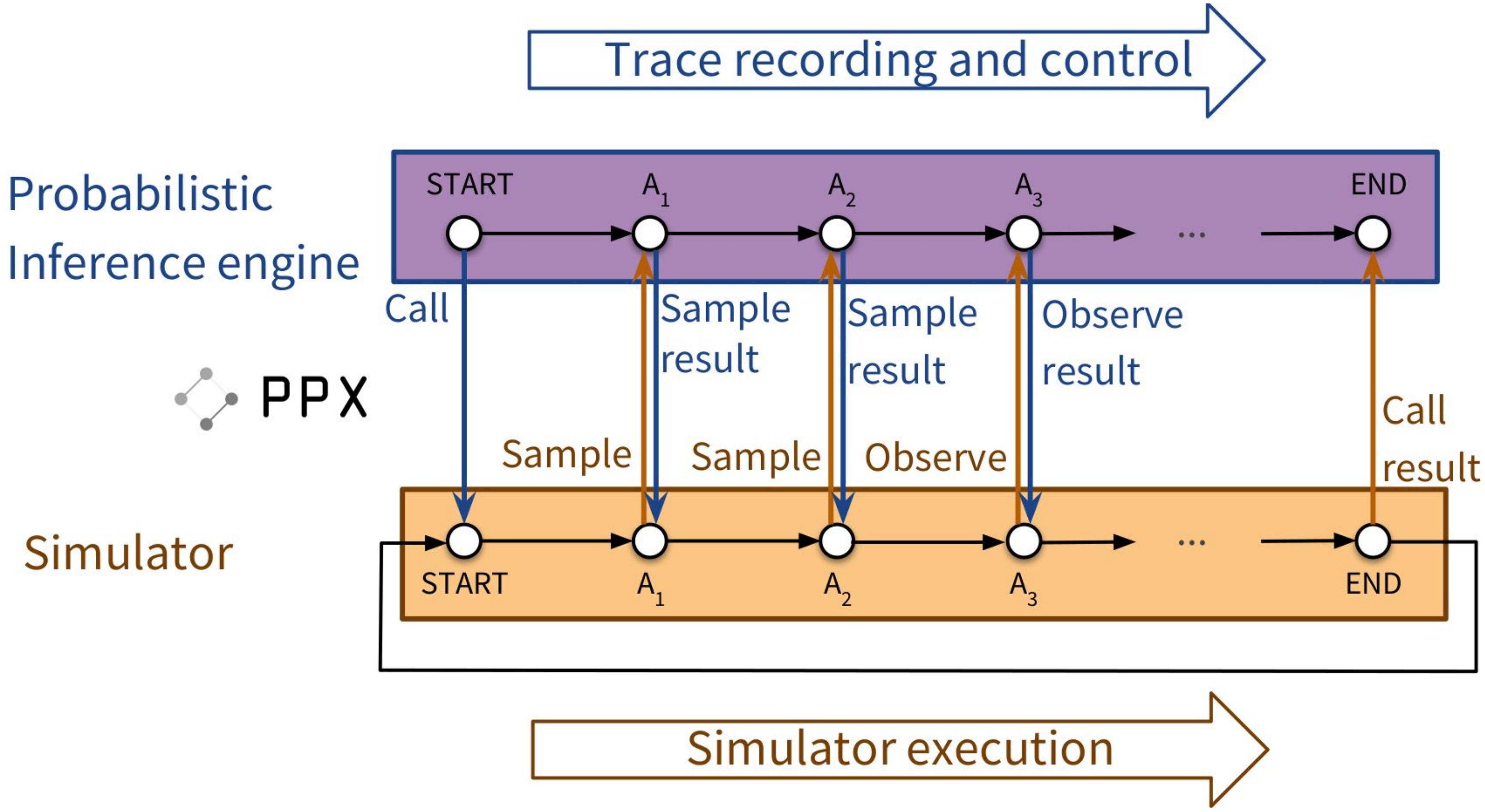
https://github.com/probprog/ppx

Probabilistic **P**rogramming e**X**ecution protocol

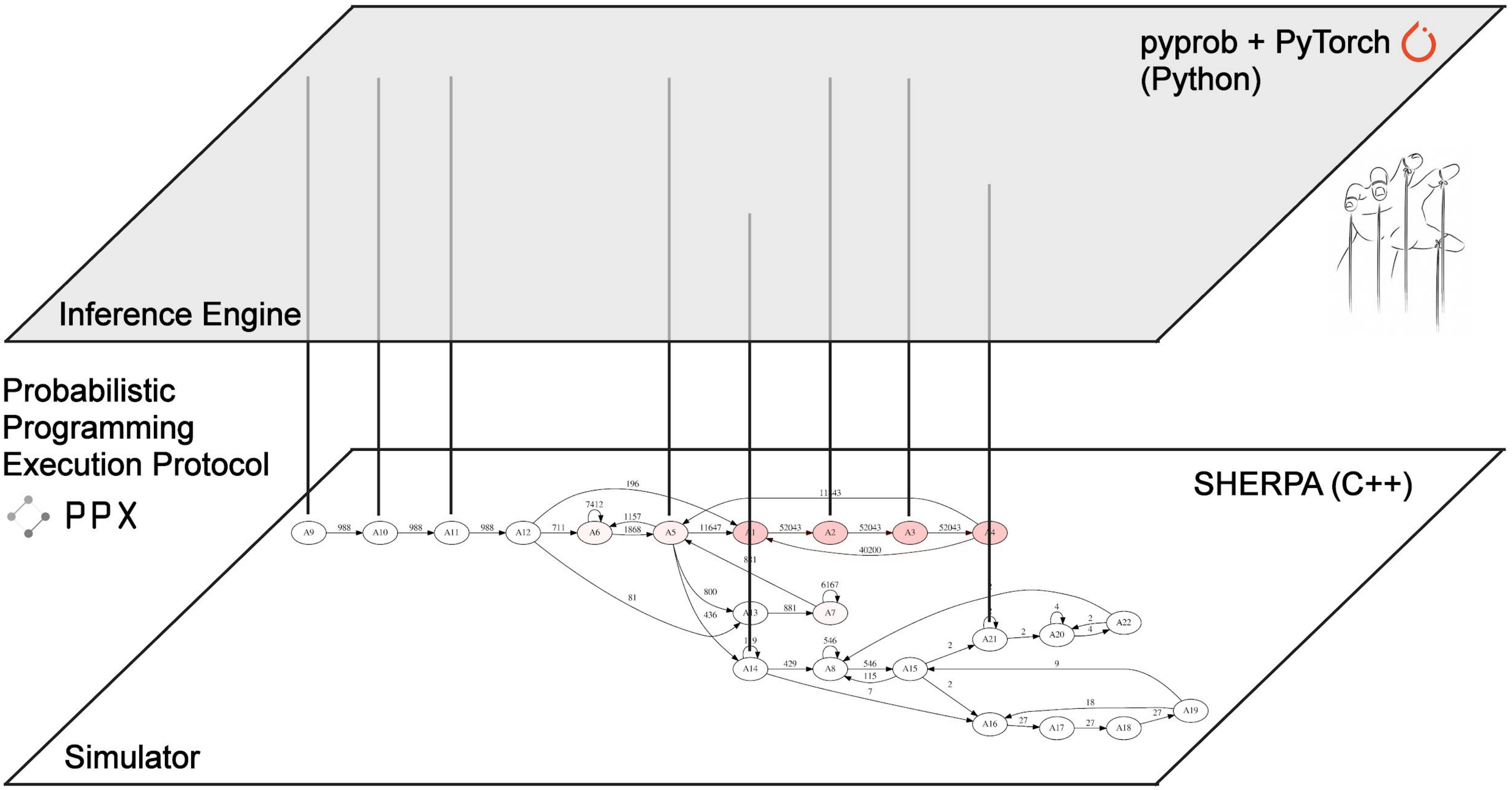
- Cross-platform, via flatbuffers: http://google.github.io/flatbuffers/
- Supported languages: C++, C#, Go, Java, JavaScript, PHP, Python, TypeScript, Rust, Lua
- Similar to Open Neural Network Exchange (ONNX) for deep learning

Enables inference engines and simulators to be

- implemented in different programming languages
- executed in separate processes, separate machines across networks


Trace recording and control
Probabilistic
Inference engine
START
A_1
A_2
A_3
...
END
Call
Sample
result
Sample
result
Observe
result
PPX
Sample
Sample
Observe
Call
result
Simulator
START
A_1
A_2
A_3
...
END
Simulator execution

PPX



pyprob + PyTorch
(Python)
Inference Engine
Probabilistic
Programming
Execution Protocol
PPX
SHERPA (C++)
Simulator

pyprob_cpp

https://github.com/probprog/pyprob_cpp

A lightweight C++ front end for PPX

```cpp
#include <pyprob_cpp.h>

// Gaussian with unkown mean
// http://www.robots.ox.ac.uk/~fwood/assets/pdf/Wood-AISTATS-2014.pdf

xt::xarray<double> forward(xt::xarray<double> observation)
{
  auto prior_mean = 1;
  auto prior_stddev = std::sqrt(5);
  auto likelihood_stddev = std::sqrt(2);

  auto prior = pyprob_cpp::distributions::Normal(prior_mean, prior_stddev);
  auto mu = pyprob_cpp::sample(prior);

  auto likelihood = pyprob_cpp::distributions::Normal(mu, likelihood_stddev);
  for (auto & o : observation)
  {
    pyprob_cpp::observe(likelihood, o);
  }

  return mu;
}
```

Probprog and high-energy physics
“etalumis”

etalumis simulate

Atılım Güneş Baydin
Bradley Gram-Hansen

Lukas Heinrich

Kyle Cranmer

Frank Wood
Andreas Munk
Saeid Naderiparizi

Wahid Bhimji
Jialin Liu
Prabhat

Gilles Louppe

Lei Shao
Larry Meadows
Victor Lee

pyprob_cpp and Sherpa

```
#include <pyprob_cpp.h>

xt::xarray<double> forward()
{
  int channel_index;
  std::vector<double> mother_momentum;
  std::vector<std::vector<double>> final_state_particles;
  std::tie(channel_index, mother_momentum, final_state_particles) = sherpa.Generate();
  pyprob_cpp::tag(xt::xarray<double>({(double)(channel_index)}), "channel_index");
  pyprob_cpp::tag(xt::xarray<double>(mother_momentum[0]), "mother_momentum_x");
  pyprob_cpp::tag(xt::xarray<double>(mother_momentum[1]), "mother_momentum_y");
  pyprob_cpp::tag(xt::xarray<double>(mother_momentum[2]), "mother_momentum_z");
  pyprob_cpp::tag(xt::adapt(flatten(final_state_particles), std::vector<std::size_t> { 30, 8 }), "final_state_particles");

  auto calo_histo = calorimeter.calo_simulation(final_state_particles);

  xt::xarray<double> mean_n_deposits = calo_histo / caloutils::minEnergyDeposit;
  //flatten
  mean_n_deposits.reshape({uint(caloutils::NBINX*caloutils::NBINY*caloutils::NBINZ)});
  auto likelihood = pyprob_cpp::distributions::Poisson(mean_n_deposits + 1E-19L);
  pyprob_cpp::observe(likelihood, "calorimeter_n_deposits");

  return xt::xarray<double>({(double)(channel_index)});
}


int main(int argc, char *argv[])
{
  auto serverAddress = (argc > 1) ? argv[1] : "ipc://@sherpa_tau_decay";
  pyprob_cpp::Model model = pyprob_cpp::Model(forward, "SHERPA tau lepton decay");
  model.startServer(serverAddress);
  return 0;
}
```

pyprob and Sherpa

jupyter sherpa_tau_decay (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help

Not Trusted | Python 3

Run Code

Connect to Sherpa

If you started this notebook using the following, SHERPA is already running:

```
docker run --rm -it --net=host etalumis/sherpa tmuxp load ./sherpa_tau_decay.yaml
```

If you are doing something else, and SHERPA is not already running, start it as follows:

```
docker run -it --rm --net=host etalumis/sherpa ./sherpa_tau_decay
```

Note: by default `sherpa_tau_decay` uses the Unix domain sockets (IPC) address `ipc://@sherpa_tau_decay`. You can run it with other addresses, e.g., TCP, using: `./sherpa_tau_decay tcp://*:5555`

In [3]:

```
model = ModelRemote('ipc://@sherpa_tau_decay')
```

```
ppx (Python): zmq.REQ socket connecting to server ipc://@sherpa_tau_decay
ppx (Python): This system        : pyprob 0.10.0
ppx (Python): Connected to system: pyprob_cpp 0.1.3 (master:8110b71)
ppx (Python): Model name         : SHERPA tau lepton decay
```

Inspect the prior

We construct an empirical distribution of the prior, using a given number of `traces`. Beware, large number of traces can take a long time. `traces=500` gives a reasonable-looking plot.

In [4]:

```
prior_dist = model.prior_distribution(num_traces=5)
```

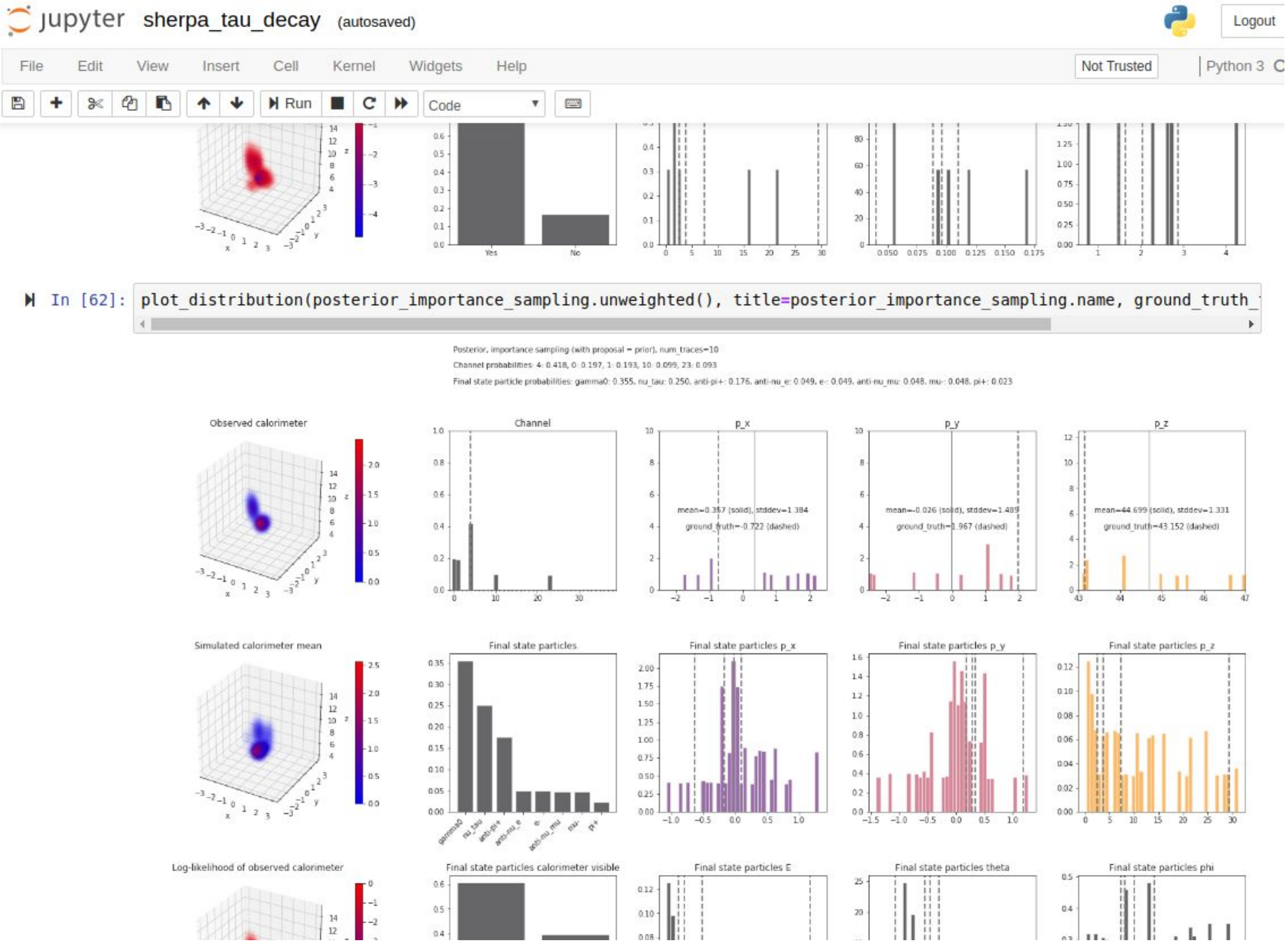
```
Time spent  | Time remain.| Progress             | Trace | Traces/sec
0d:00:00:01 | 0d:00:00:00 | #################### | 5/5 | 2.60
```

Separate (marginalize) the distribution into distributions over momenta and channel, by mapping a function.

In [5]:

```
prior_dist_px = prior_dist.map(lambda x: float(x[0]))
prior_dist_py = prior_dist.map(lambda x: float(x[1]))
prior_dist_pz = prior_dist.map(lambda x: float(x[2]))
prior_dist_channel = prior_dist.map(lambda x: int(x[3]))
```

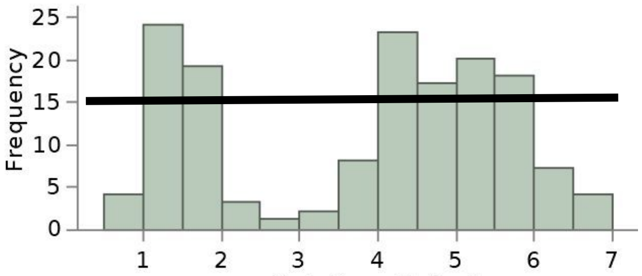
pyprob and Sherpa



Main challenges

Working with large-scale HEP simulators requires several innovations

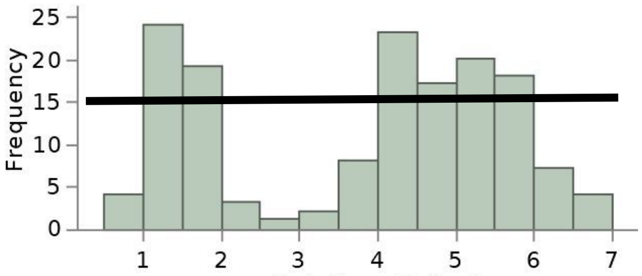
- Wide range of prior probabilities, some events highly unlikely and not learned by IC neural network
- Solution: “prior inflation”
  - Training: modify prior distributions to be uninformative
  - Inference: use the unmodified (real) prior for weighting proposals



Main challenges

Working with large-scale HEP simulators requires several innovations

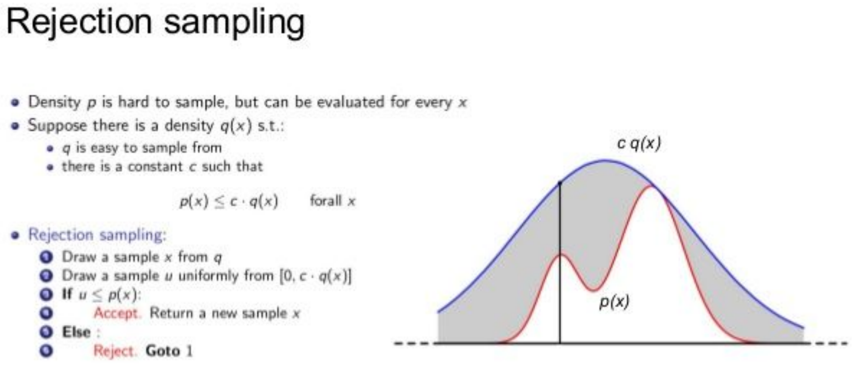
- Wide range of prior probabilities, some events highly unlikely and not learned by IC neural network
- Solution: “prior inflation”
    - Training: modify prior distributions to be uninformative
      HEP: sample according to phase space
    - Inference: use the unmodified (real) prior for weighting proposals
      HEP: differential cross-section = phase space * matrix element



Main challenges

Working with large-scale HEP simulators requires several innovations

- Potentially very long execution traces due to rejection sampling loops
- Solution: “replace” (or “rejection-sampling”) mode
  - Training: only consider the last (accepted) values within loops
  - Inference: use the same proposal distribution for these samples



Jaroszewicz, Bishop

Experiments

Tau lepton decay

Tau decay in Sherpa, 38 decay channels, coupled with an approximate calorimeter simulation in C++

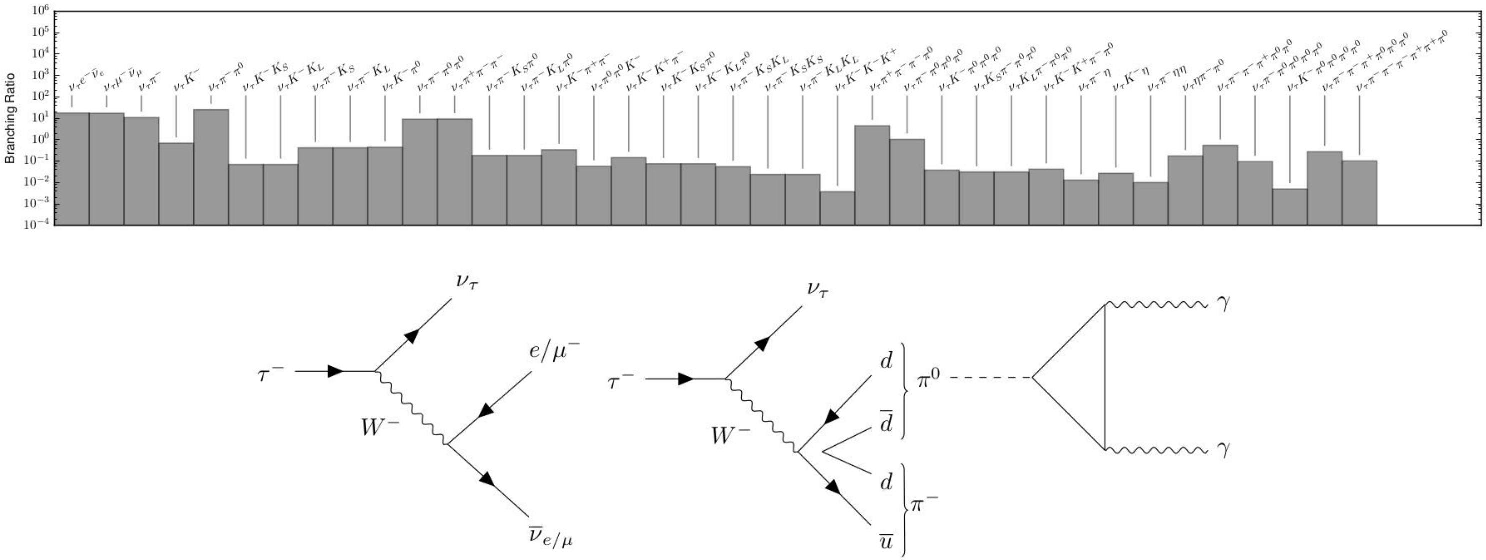


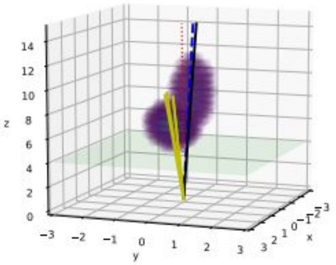
Figure 2: *Top:* branching ratios of the τ lepton, effectively the prior distribution of the decay channels in SHERPA. Note that the scale is logarithmic. *Bottom:* Feynman diagrams for τ decays illustrating that these can produce multiple detected particles.

Tau lepton decay

Tau decay in Sherpa, 38 decay channels, coupled with an approximate calorimeter simulation in C++

Observation: 3D calorimeter depositions (Poisson)

- Particle showers modeled as Gaussian blobs, deposited energy parameterizes a multivariate Poisson
- Shower shape variables and sampling fraction based on final state particle



Monte Carlo truth (latent variables) of interest:

- Decay channel (Categorical)
- px, py, pz momenta of tau particle (Continuous uniform)
- Final state momenta and IDs

Probabilistic addresses in Sherpa

Approximately 25,000 addresses encountered

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
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

...

Common trace types in Sherpa

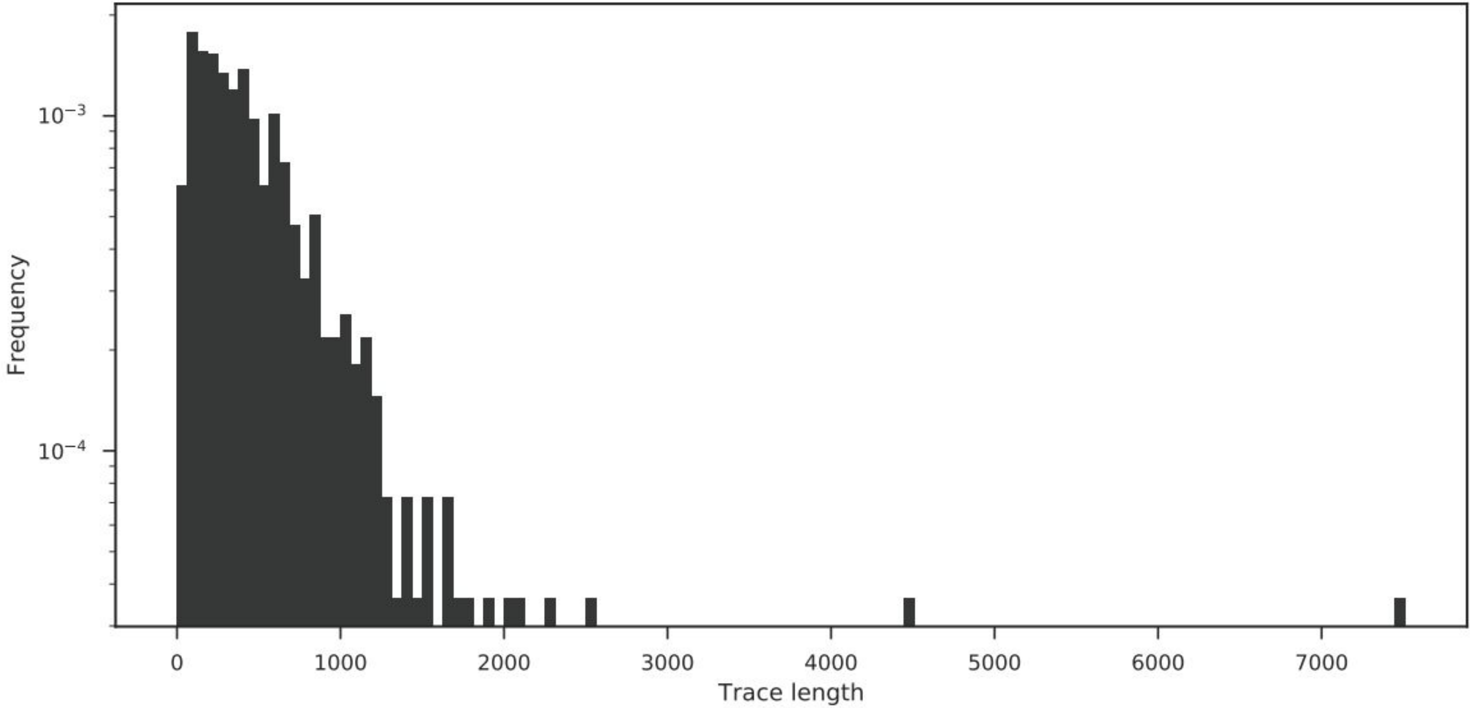
Approximately 450 trace types encountered

Trace type: unique sequencing of addresses (with different sampled values)

Freq.	Length	Addresses (showing controlled only)
0.106	72	A1, A2, A3, A5, A6, A32, A33, A31
0.105	41	A1, A2, A3, A5, A6, A499, A31
0.078	1,780	A1, A2, A3, A5, A6, A7, A8, A9, A10, A31
0.053	188	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A26, A31
0.053	100	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A99, A100, A101, A102, A31
0.039	56	A1, A2, A3, A5, A6, A499, A17, A18, A26, A31
0.039	592	A1, A2, A3, A5, A6, A499, A17, A18, A99, A100, A101, A102, A31
0.038	162	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A500, A99, A100, A101, A102, A31
0.030	240	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A26, A99, A100, A101, A102, A31
0.029	836	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A99, A100, A101, A102, A26, A31
0.027	643	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A507, A99, A100, A101, A102, A31
0.023	135	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A26, A99, A100, A101, A102, A31
0.023	485	A1, A2, A3, A5, A6, A7, A8, A9, A10, A17, A18, A20, A21, A41, A42, A44, A45, A99, A100, A101, A102, A26, A31
...		

Common trace types in Sherpa

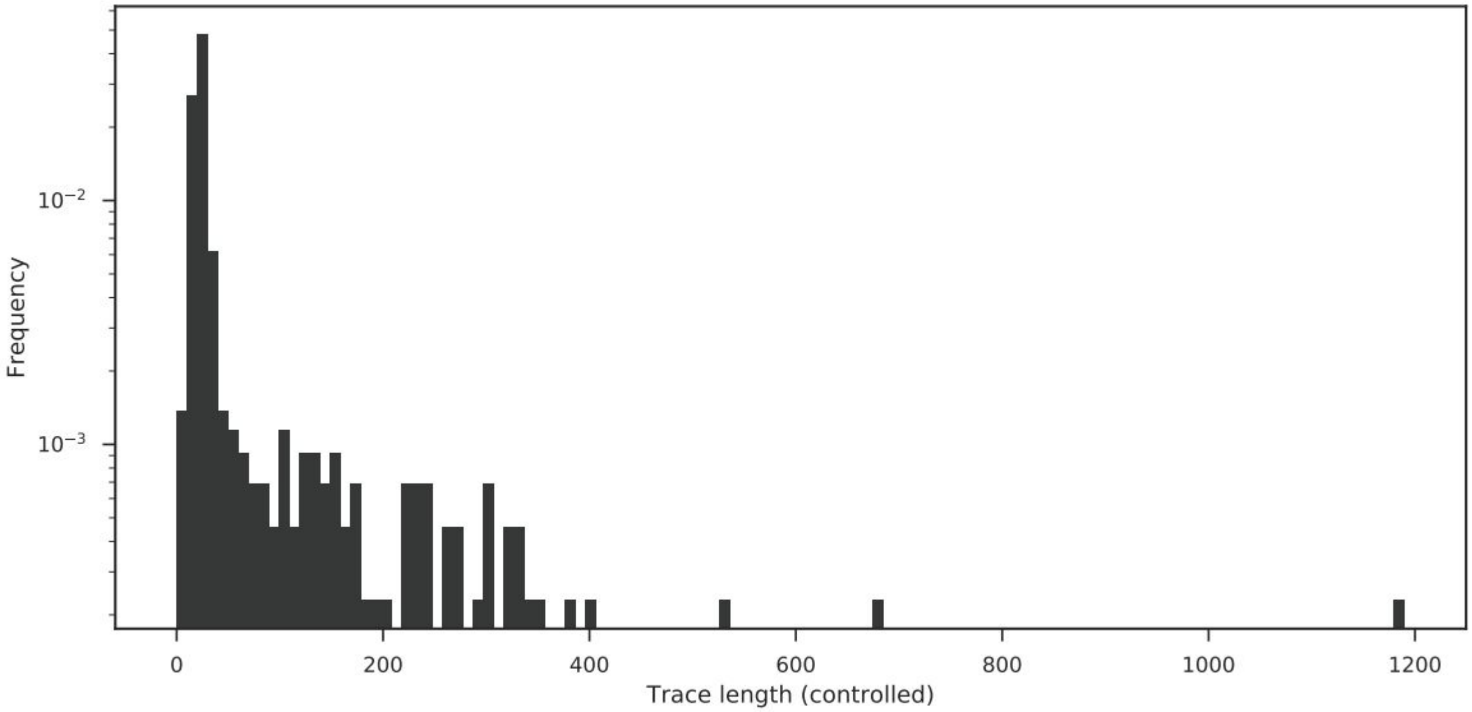
Approximately 450 trace types encountered



(a) Distribution of trace lengths (all addresses). Min: 13, max: 7,514, mean: 383.58.

Common trace types in Sherpa

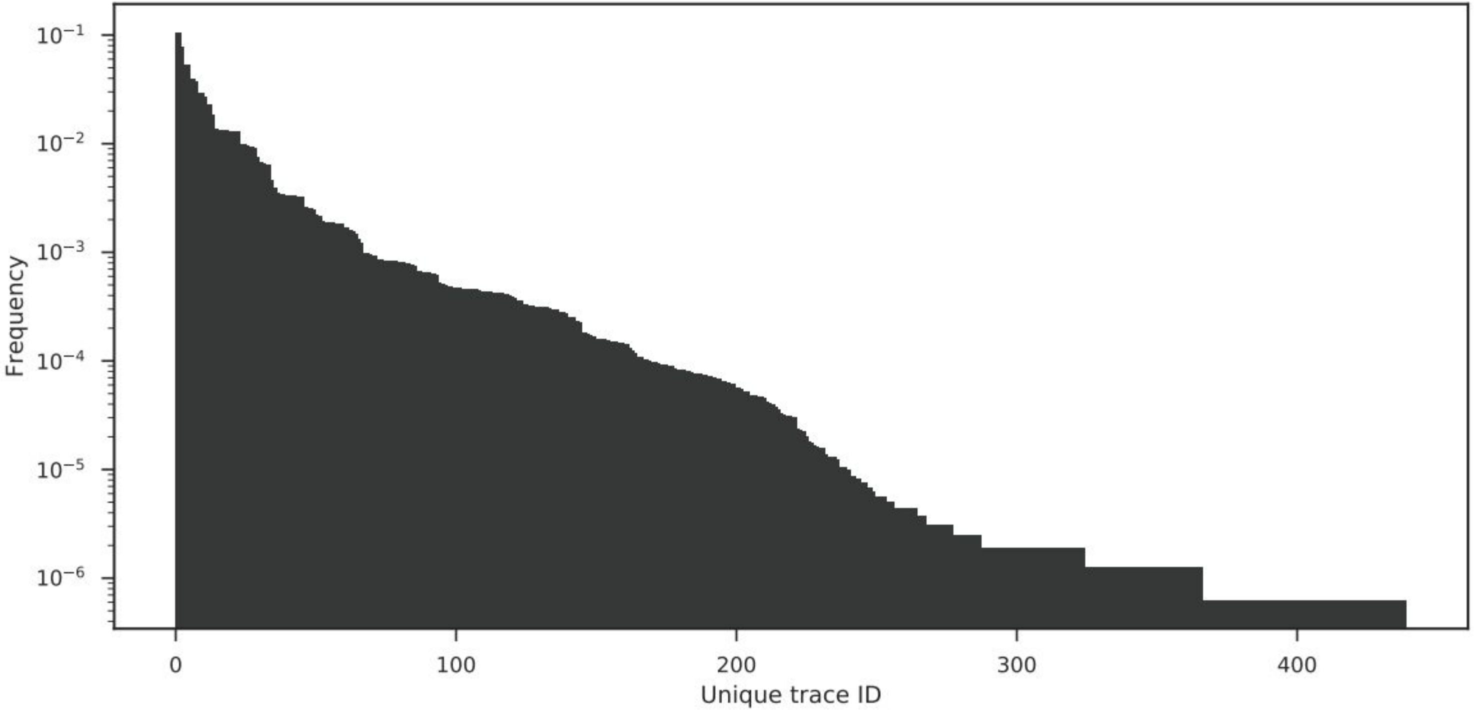
Approximately 450 trace types encountered



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

Common trace types in Sherpa

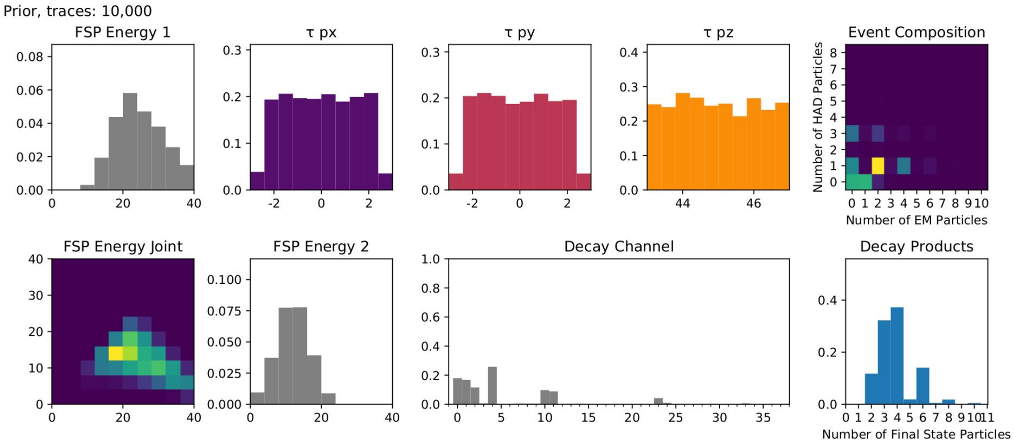
Approximately 450 trace types encountered



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

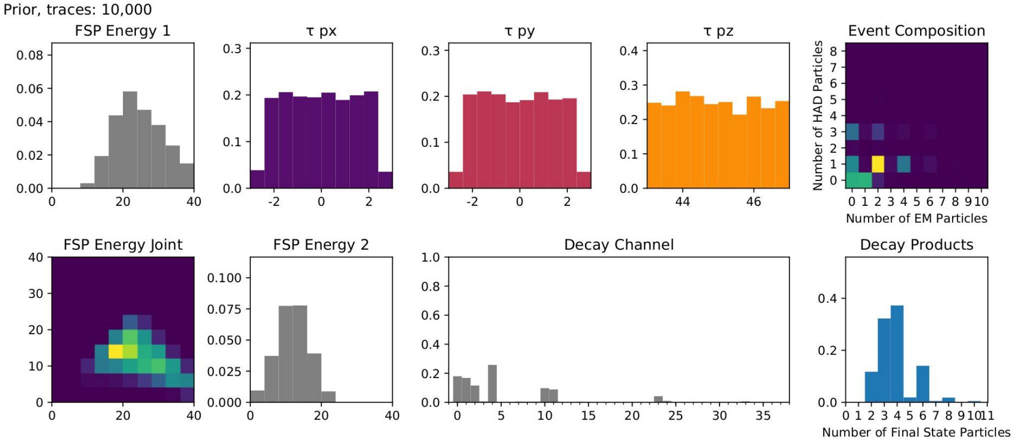
Inference results with MCMC engine

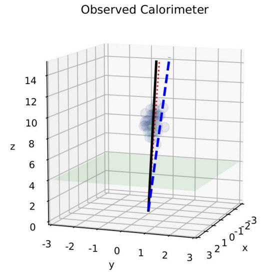
Prior



Inference results with MCMC engine

Prior

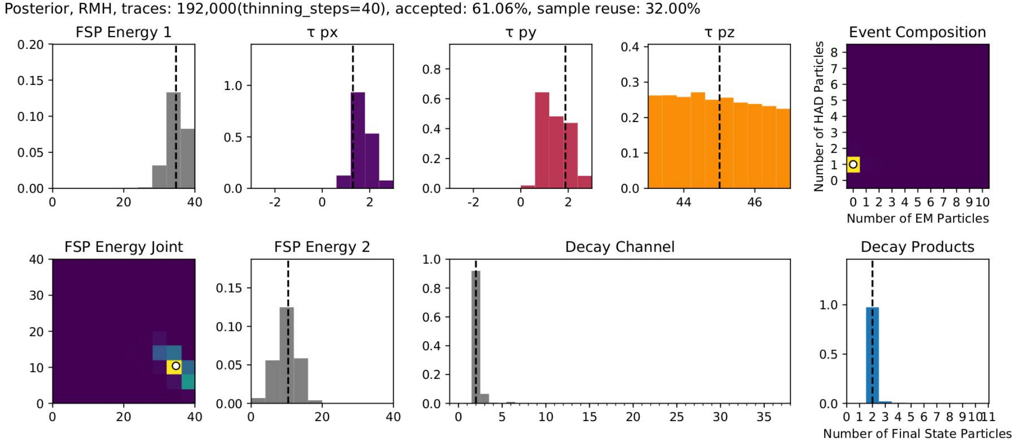




MCMC Posterior

conditioned on

calorimeter



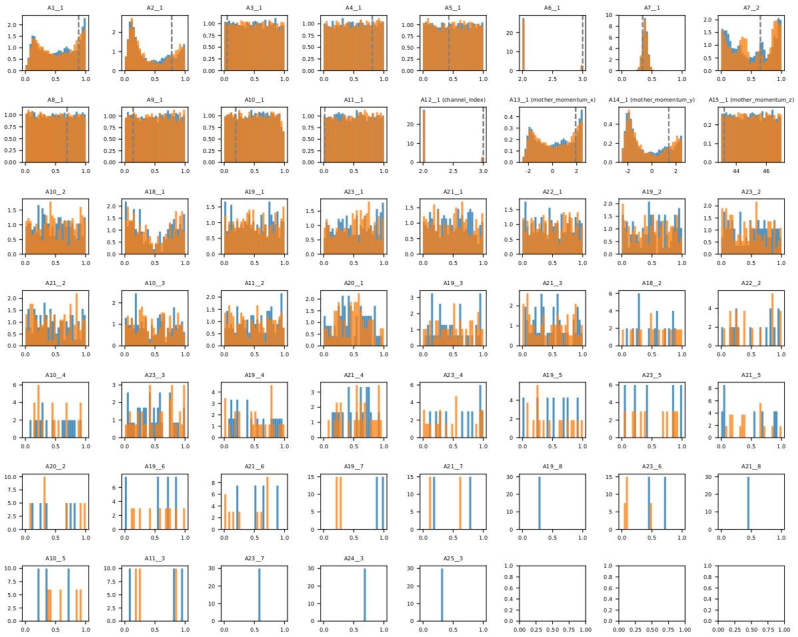
7,700,000 samples

Slow and has to run single node

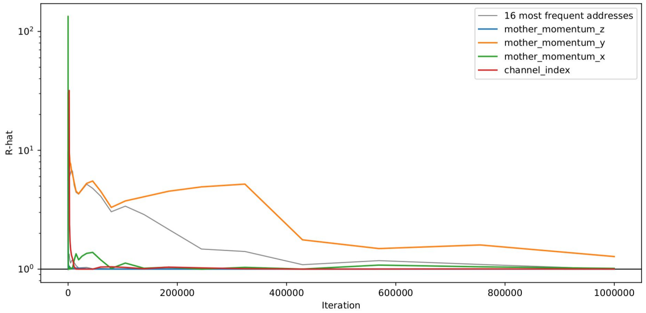
Convergence to true posterior

We establish that two independent RMH MCMC chains converge to the same posterior for all addresses in Sherpa

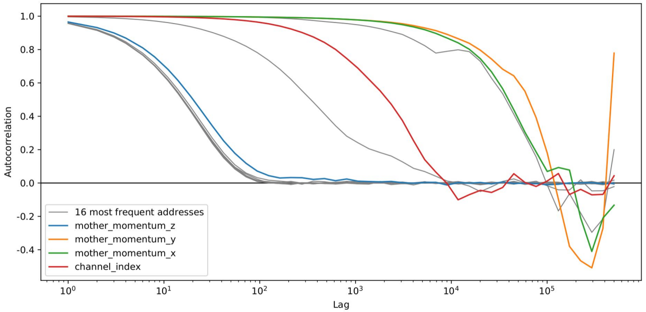
- Chain initialized with random trace from prior
- Chain initialized with known ground-truth trace



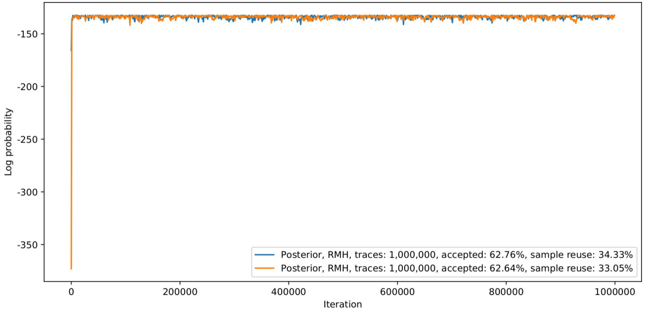
Gelman-Rubin convergence diagnostic



Autocorrelation



Trace log-probability

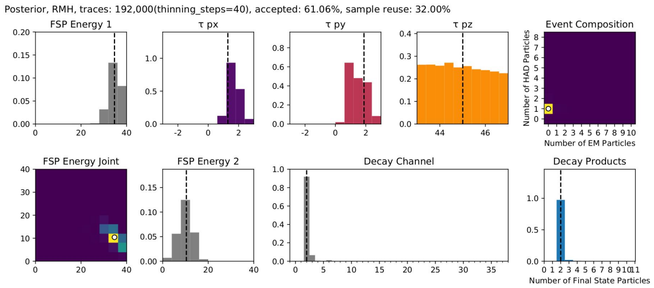


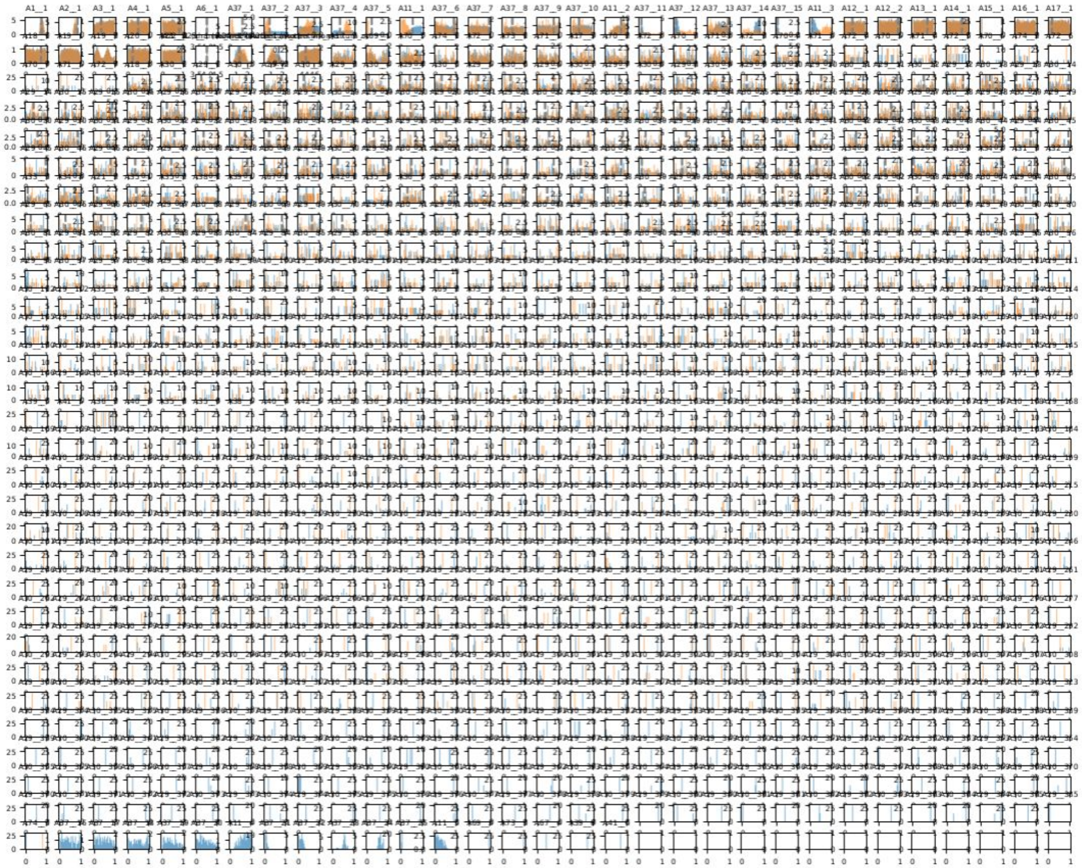
Convergence to true posterior

Important:

- We get **posteriors over the whole Sherpa address space, 1000s of addresses**
- Trace complexity varies depending on observed event

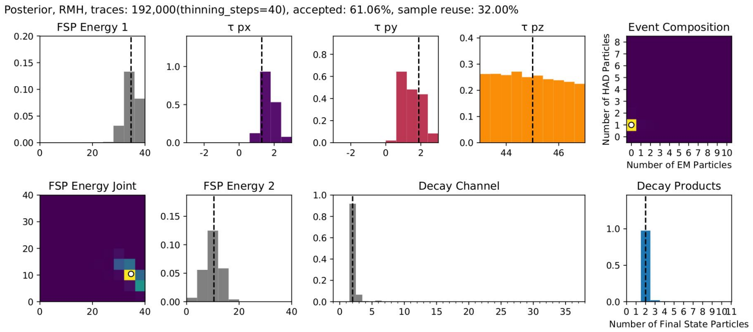
This is just a selected subset:



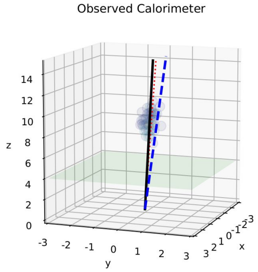


Inference results with IC engine

MCMC true posterior (7.7M single node)



Inference results with IC engine

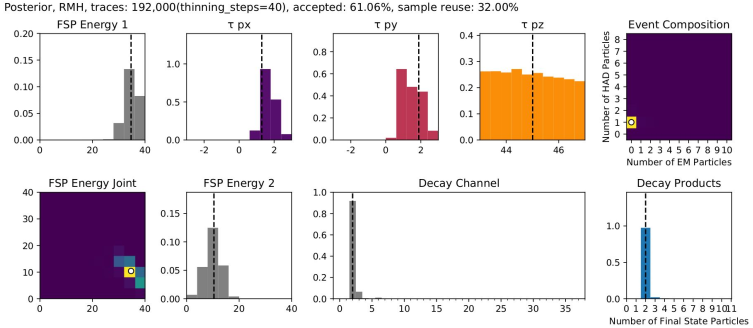


320,000 samples

Fast “embarrassingly” parallel multi-node

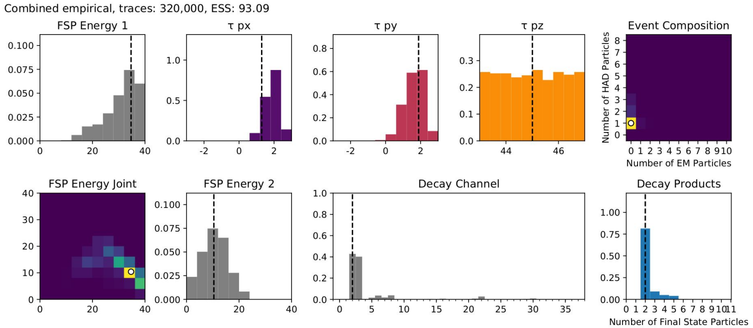
MCMC true posterior

(7.7M single node)



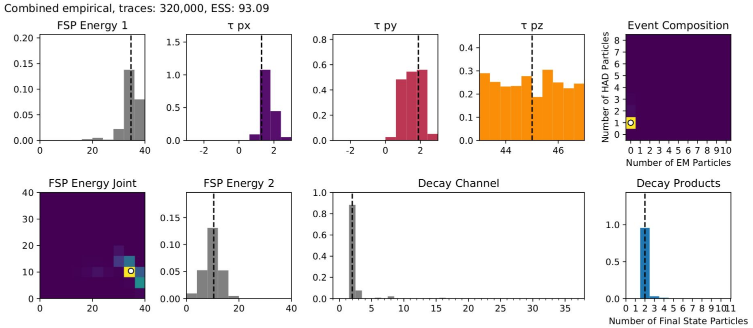
IC proposal

from trained NN



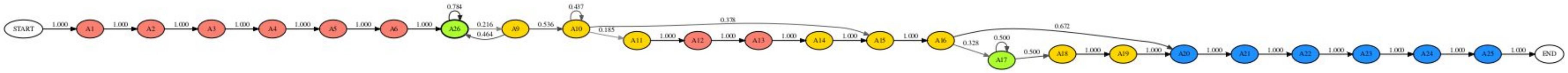
IC posterior

after importance weighting



Interpretability

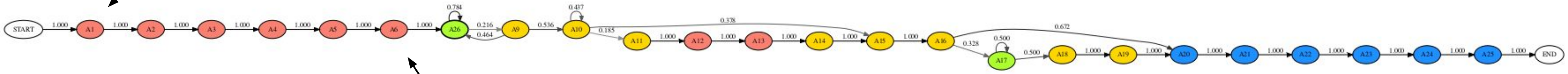
Latent probabilistic structure of 10 most frequent trace types



Interpretability

Latent probabilistic structure of 10 most frequent trace types

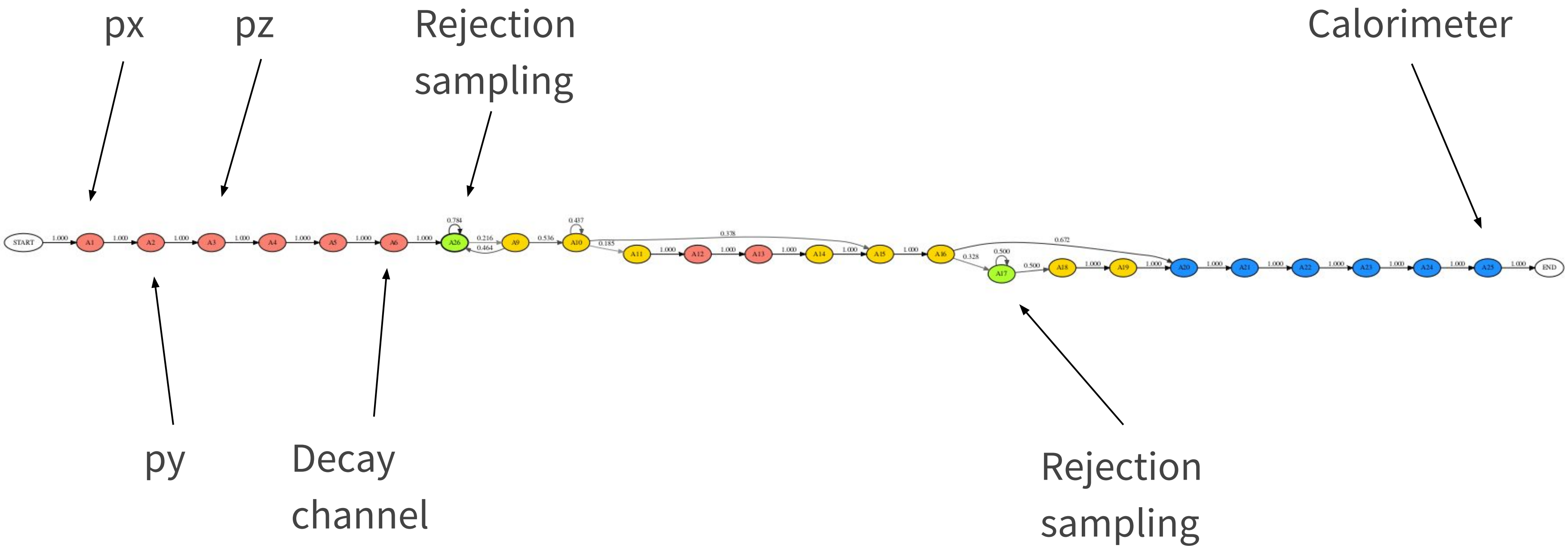
[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler:: 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



[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

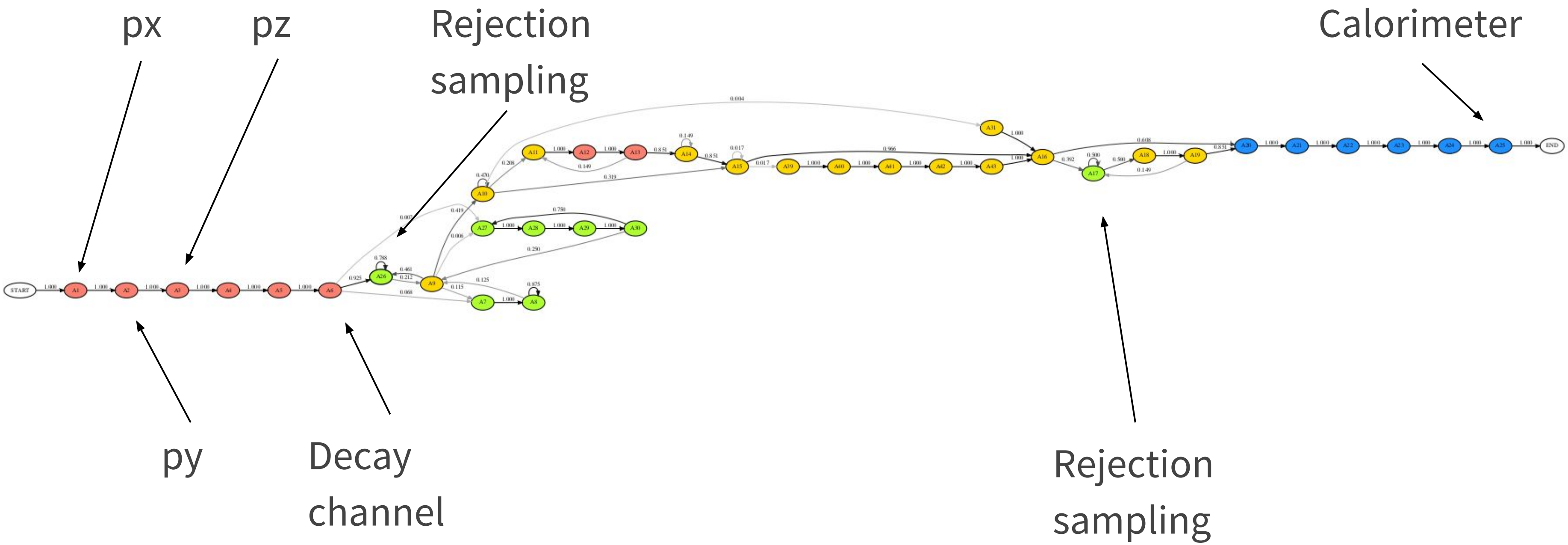
Interpretability

Latent probabilistic structure of 10 most frequent trace types



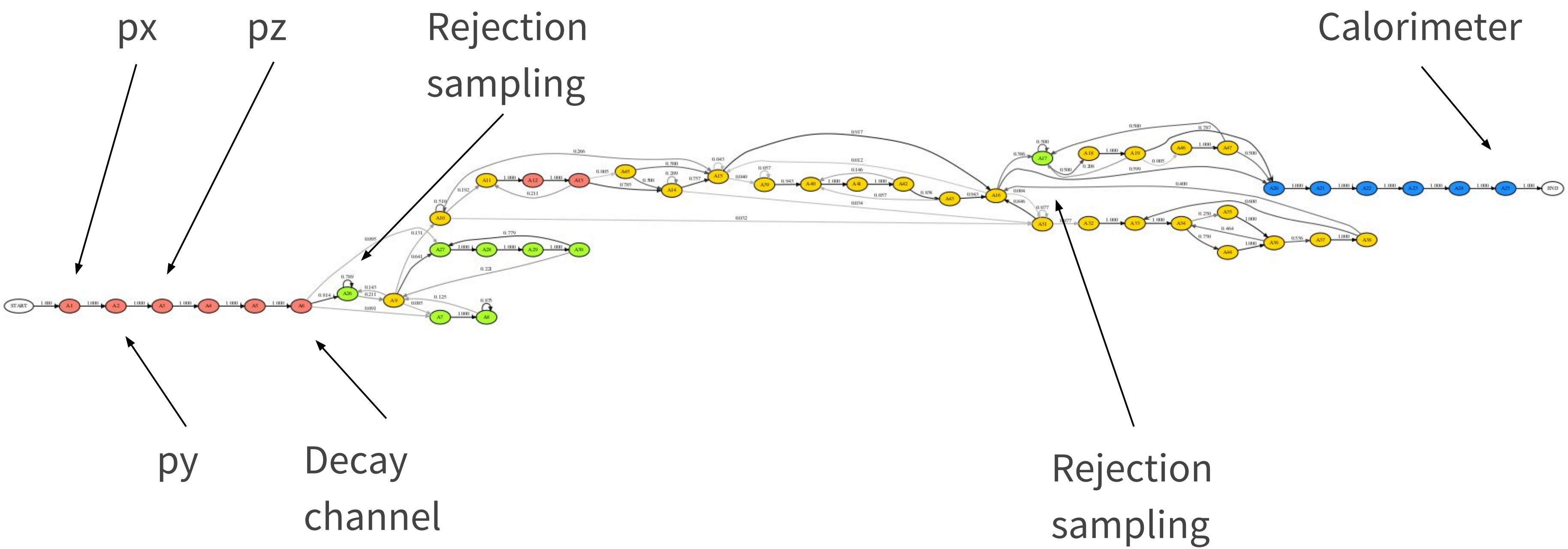
Interpretability

Latent probabilistic structure of 25 most frequent trace types



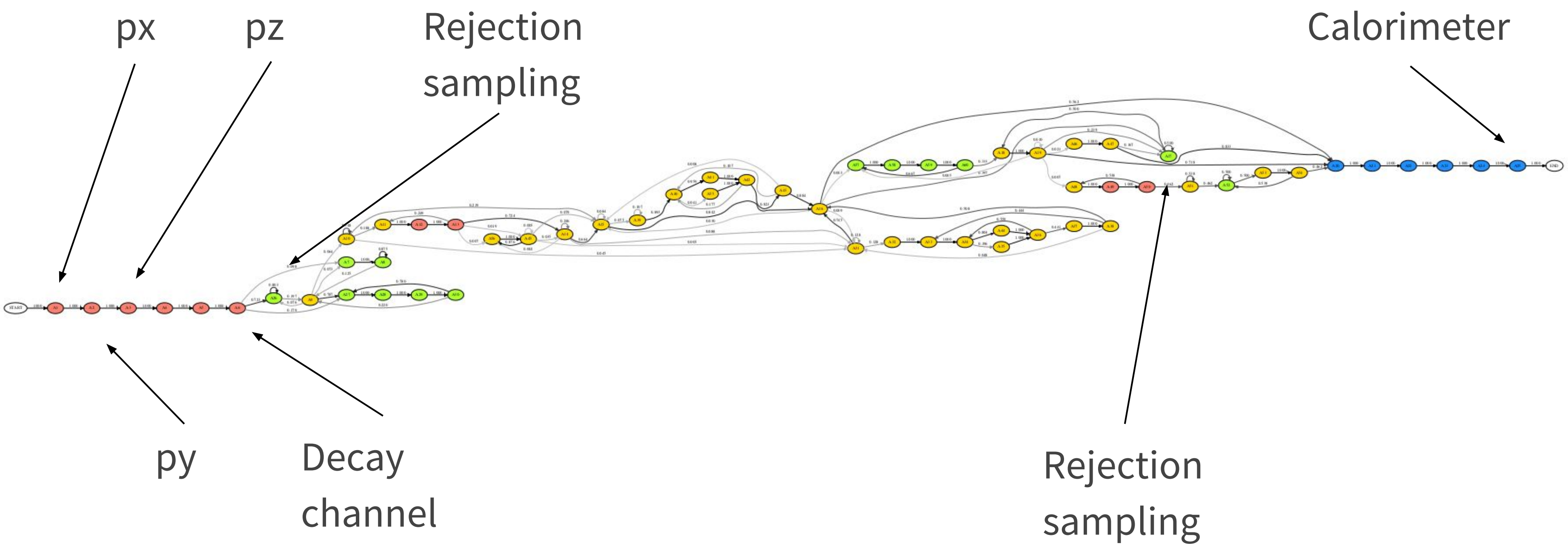
Interpretability

Latent probabilistic structure of 100 most frequent trace types

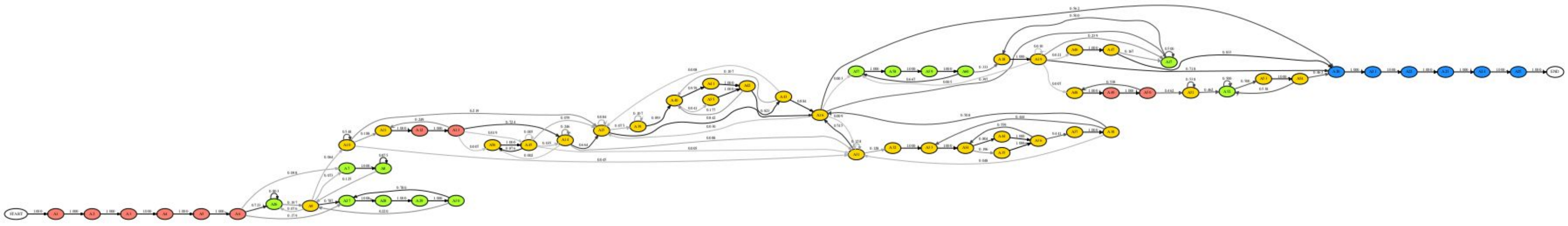


Interpretability

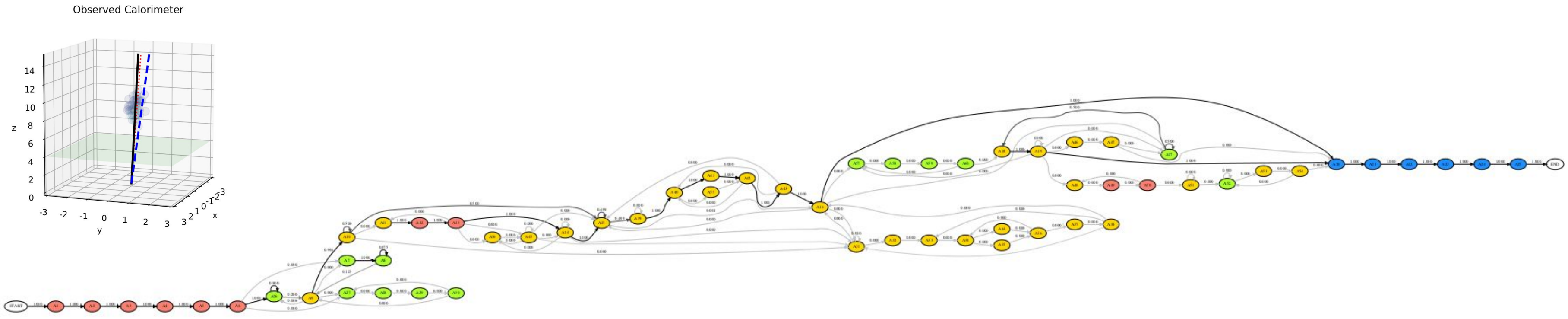
Latent probabilistic structure of 250 most frequent trace types



Interpretability



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



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

What’s next?

Current and upcoming work

- Science
  - Statistically measure distance between RMH and IC results
  - Uniform(0,1)-only control
  - Rare event simulation for compilation (“prior inflation”)
  - Control / not control
- Engineering
  - Batching of open-ended traces for NN training
  - Distributed training of dynamic networks (thanks to PyTorch)
  - Balancing distributed data generation and training nodes
  - User-friendly features: posterior code highlighting, etc.
  - Other simulators

Thank you for listening

International Centre for Theoretical Physics
Trieste, Italy, 9 April 2019

UNIVERSITY OF OXFORD

References

Baydin, A.G., Heinrich, L., Bhimji, W., Gram-Hansen, B., Louppe, G., Shao, L., Prabhat, Cranmer, K., Wood, F. 2018. Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model arXiv preprint arXiv:1807.07706.

Gershman, S. and Goodman, N., 2014, January. Amortized inference in probabilistic reasoning. In Proceedings of the Cognitive Science Society (Vol. 36, No. 36).

Gordon, A.D., Henzinger, T.A., Nori, A.V. and Rajamani, S.K., 2014, May. Probabilistic programming. In Proceedings of the on Future of Software Engineering (pp. 167-181). ACM.

Le, T.A., Baydin, A.G. and Wood, F., 2017, April. Inference Compilation and Universal Probabilistic Programming. In International Conference on Artificial Intelligence and Statistics (AISTATS) (pp. 1338-1348).

Le, Tuan Anh, Atılım Güneş Baydin, Robert Zinkov, and Frank Wood. 2017. “Using Synthetic Data to Train Neural Networks Is Model-Based Reasoning.” In 30th International Joint Conference on Neural Networks, May 14–19, 2017, Anchorage, AK, USA.

Le, Tuan Anh, Atılım Güneş Baydin, and Frank Wood. 2016. “Nested Compiled Inference for Hierarchical Reinforcement Learning.” In NIPS 2016 Workshop on Bayesian Deep Learning, Barcelona, Spain, December 10, 2016.

Extra slides

Calorimeter

For each particle in the final state coming from Sherpa:

1. Determine whether it interacts with the calorimeter at all (muons and neutrinos don't)
2. Calculate the total mean number and spatial distribution of energy depositions from the calorimeter shower (simulating combined effect of secondary particles)
3. Draw a number of actual depositions from the total mean and then draw that number of energy depositions according to the spatial distribution

Training objective and data for IC

- Minimize

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

- Using stochastic gradient descent with Adam
- Infinite stream of minibatches

$$\mathcal{D}_{\text{train}} = \left\{ \left(x_t^{(m)}, a_t^{(m)}, i_t^{(m)}\right)_{t=1}^{T^{(m)}}, \left(y_n^{(m)}\right)_{n=1}^{N} \right\}_{m=1}^{M}$$

sampled from the model

$$p(\mathbf{x},\mathbf{y})$$

Gelman-Rubin and autocorrelation formulae

Gelman-Rubin diagnostic ($\hat{R}$)

- Compute *m* independent Markov chains
- Compares variance of each chain to pooled variance
- If initial states (θ_{1j}) are overdispersed, then $\hat{R}$ approaches unity from above
- Provides estimate of how much variance could be reduced by running chains longer
- It is an *estimate!*

$$W = \frac{1}{m}\sum_{j=1}^{m} s_j^2 \qquad \bar{\bar{\theta}} = \frac{1}{m}\sum_{j=1}^{m}\bar{\theta}_j$$

$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\theta}_j - \bar{\bar{\theta}})^2 \qquad s_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\theta_{ij} - \bar{\theta}_j)^2$$

$$\hat{\mathrm{Var}}(\theta) = (1 - \frac{1}{n})W + \frac{1}{n}B \qquad \hat{R} = \sqrt{\frac{\hat{\mathrm{Var}}(\theta)}{W}}$$

From Eric B. Ford (Penn State): Bayesian Computing for Astronomical Data Analysis
http://astrostatistics.psu.edu/RLectures/diagnosticsMCMC.pdf

Gelman-Rubin and autocorrelation formulae

Check Autocorrelation of Markov chain

- Autocorrelation as a function of lag

$$\rho_{lag} = \frac{\sum_i^{N-lag}(\theta_i - \bar{\theta})(\theta_{i+lag} - \bar{\theta})}{\sum_i^N(\theta_i - \bar{\theta})^2}$$

- What is smallest lag to give an $\rho_{lag} \approx 0$?
- One of several methods for estimating how many iterations of Markov chain are needed for *effectively* independent samples

From Eric B. Ford (Penn State): Bayesian Computing for Astronomical Data Analysis
http://astrostatistics.psu.edu/RLectures/diagnosticsMCMC.pdf