Universal Probabilistic Programming in Simulators

“etalumis”

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

Institute for Pure & Applied Mathematics, UCLA
15 October 2019

Probabilistic programming

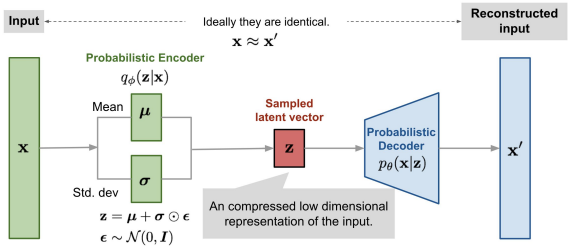
Deep learning

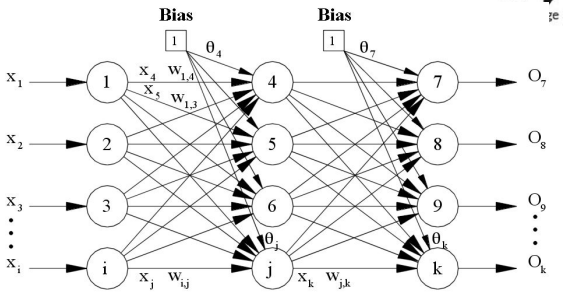
Model is learned from data as a differentiable transformation

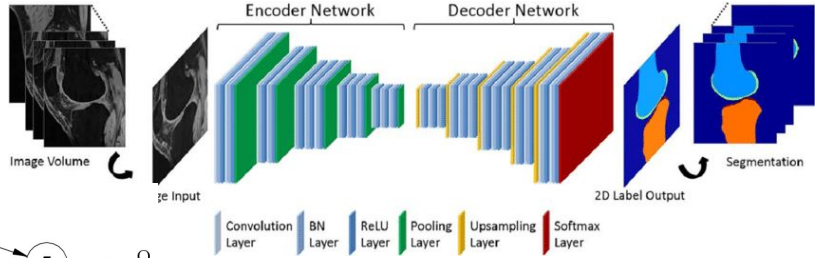
Deep learning

Model is learned from data as a differentiable transformation

Neural network **(differentiable program)**







Difficult to interpret the actual learned model

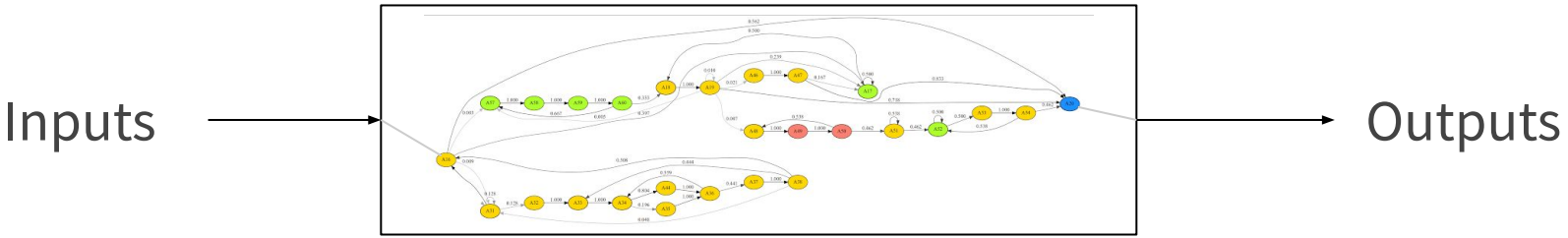
Deep learning

Model is learned from data as a differentiable transformation

Neural network (differentiable program)

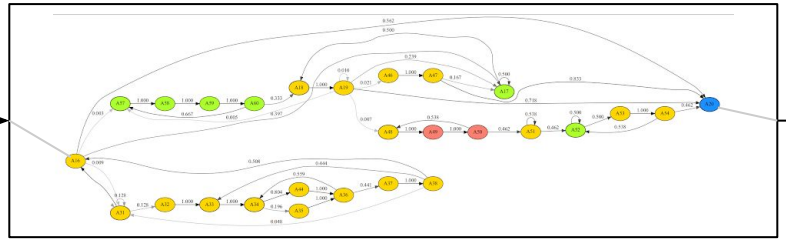
Probabilistic programming

Model is defined as a structured generative program



Model / probabilistic program / simulator

Probabilistic programming

Inputs →  → Outputs

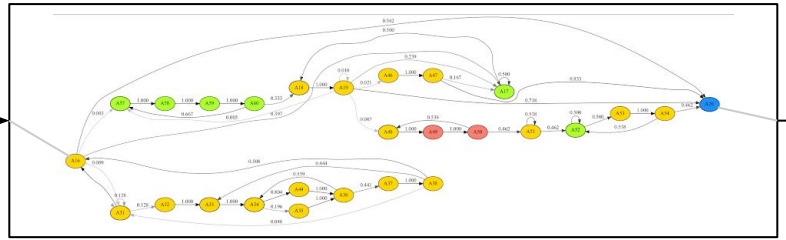
Model / probabilistic program / simulator

Probabilistic model: a joint distribution $p(\mathbf{x}, \mathbf{y})$ of random variables

- **Latent** (hidden, unobserved) variables $\mathbf{x}$
- **Observed** variables (data) $\mathbf{y}$

Probabilistic programming

Inputs



Outputs

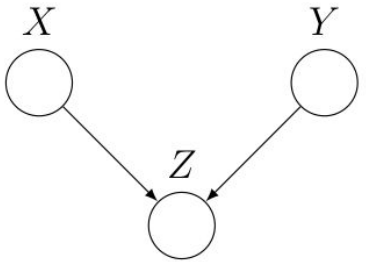
Model / probabilistic program / simulator

Probabilistic model: a joint distribution $p(\mathbf{x}, \mathbf{y})$ of random variables

- **Latent** (hidden, unobserved) variables $\mathbf{x}$
- **Observed** variables (data) $\mathbf{y}$

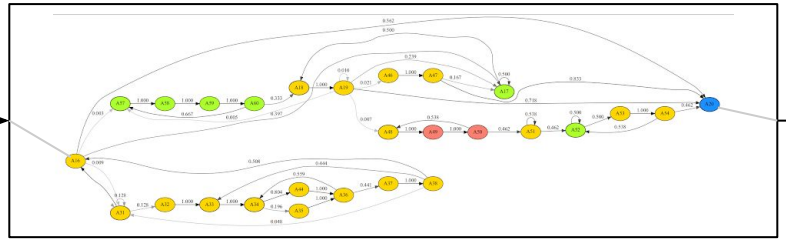
Probabilistic graphical models use graphs to express conditional dependence

- Bayesian networks
- Markov random fields (undirected)



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

Probabilistic programming

Inputs →  → Outputs

Model / probabilistic program / simulator

Probabilistic model: a joint distribution $p(\mathbf{x}, \mathbf{y})$ of random variables

- **Latent** (hidden, unobserved) variables $\mathbf{x}$
- **Observed** variables (data) $\mathbf{y}$

```
1: bool c1, c2;
2: c1 = Bernoulli(0.5);
3: c2 = Bernoulli(0.5);
4: return(c1, c2);
```

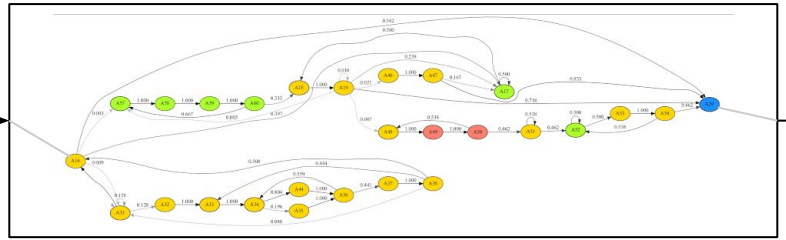
Probabilistic programming extends this to

"ordinary programming with two added constructs"

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

```
1: bool c1, c2;
2: c1 = Bernoulli(0.5);
3: c2 = Bernoulli(0.5);
4: observe(c1 || c2);
5: return(c1, c2);
```

Inference

Inputs →  → Outputs

Model / probabilistic program / simulator

Use your model $p(\mathbf{x}, \mathbf{y})$ to analyze (explain) some given data
as the posterior distribution of latents $\mathbf{x}$ conditioned on observations $\mathbf{y}$

Likelihood:
How do data depend on latents?

Posterior:
Distribution of latents
describing given data

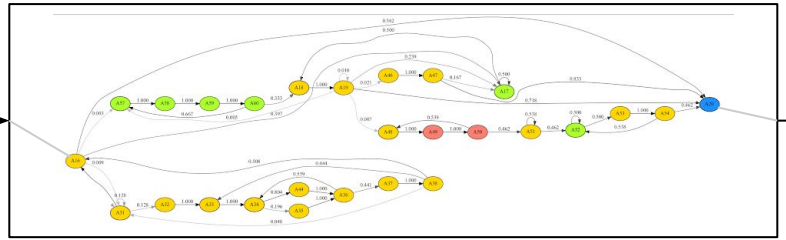
Prior, describes latents

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

See Edward tutorials for a good intro: http://edwardlib.org/tutorials/

Inference

Inputs



Model / probabilistic program / simulator

$\mathbf{y}$

Simulated data

- Run many times

$$\text{sample}(p(x_1)); \ldots; \text{sample}(p(x_N|\mathbf{x}_{1:N-1})); \text{observe}(p(\mathbf{y}|\mathbf{x}_{1:N}), \mathbf{y}_{\text{obs}})$$

- Record **execution traces** $\{\mathbf{x}^t, w^t\}_{t=1}^T, w^t = p(\mathbf{y}_{\text{obs}}|\mathbf{x}^t)$

- Approximate the posterior

$$\hat{p}(\mathbf{x}|\mathbf{y}) \propto \sum_{t=1}^T w^t \delta(\mathbf{x}^t - \mathbf{x})$$

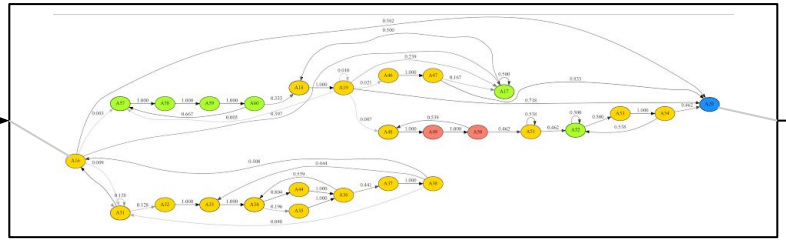
$$I_f = \int f(\mathbf{x}) p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x} \approx \sum_{t=1}^T w^t f(\mathbf{x}^t) / \sum_{t=1}^T w^t$$

$\mathbf{y}_{\text{obs}}$

Observed data

Inference

Inputs



Model / probabilistic program / simulator

$\mathbf{y}$

Simulated data

- Run many times

$$\text{sample}(p(x_1)); \ldots; \text{sample}(p(x_N|\mathbf{x}_{1:N-1})); \text{observe}(p(\mathbf{y}|\mathbf{x}_{1:N}), \mathbf{y}_{\text{obs}})$$

- Record **execution traces** $\{\mathbf{x}^t, w^t\}_{t=1}^T, w^t = p(\mathbf{y}_{\text{obs}}|\mathbf{x}^t)$

- Approximat

$$\hat{p}(\mathbf{x}|\mathbf{y}) \propto \sum$$

This is importance sampling, other inference engines run differently

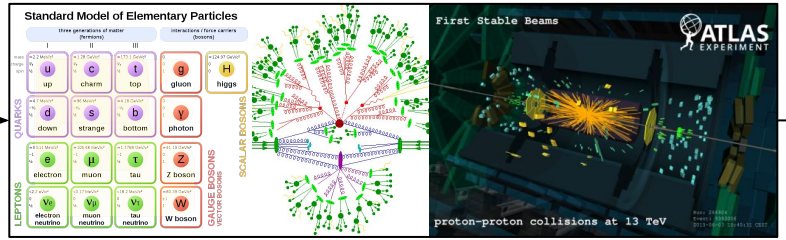
$$I_f = \int f(\mathbf{x})p(\mathbf{x}|\mathbf{y})\mathrm{d}\mathbf{x} \approx \sum_{t=1}^T w^t f(\mathbf{x}^t) / \sum_{t=1}^T w^t$$

$\mathbf{y}_{\text{obs}}$

Observed data

Inference reverses the generative process

Inputs



Generative model / simulator (e.g., Sherpa, Geant)

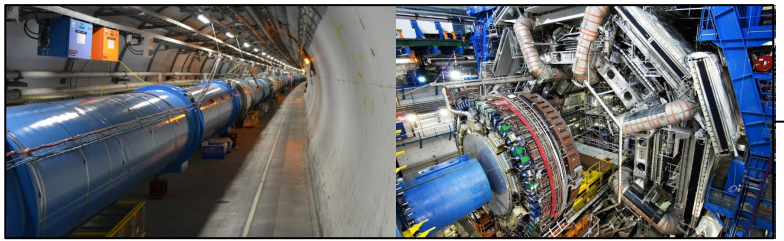
$\mathbf{y}$

Simulated data

(detector r

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

Inputs ← $p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ ← $\text{observe}(p(\mathbf{y}|\mathbf{x}), \mathbf{y}_{\text{obs}})$



Real world system

$\mathbf{y}_{\text{obs}}$

Observed data

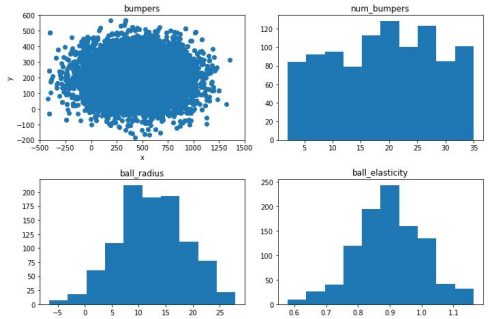
(detector

response)

Inference

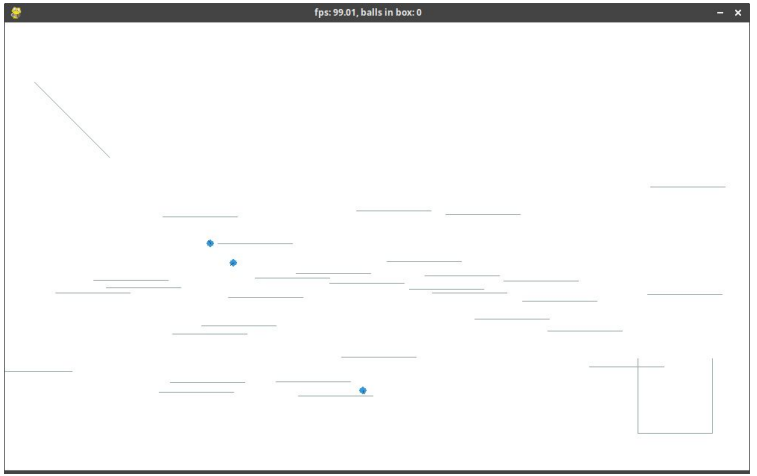
Live demo

```
In [11]: model = PhysicsModel(draw=False)
         prior = model.prior(num_traces=1000)

Time spent  | Time remain.| Progress             | Trace     | Traces/sec
0d:00:00:30 | 0d:00:00:00 | #################### | 1000/1000 | 32.77

In [12]: plot_dist(prior)

Resample, num_samples: 1000, min_index: 0, max_index: 1000
Time spent  | Time remain.| Progress             | Samples   | Samples/sec
0d:00:00:00 | 0d:00:00:00 | #################### | 1000/1000 | 127,134.76
```



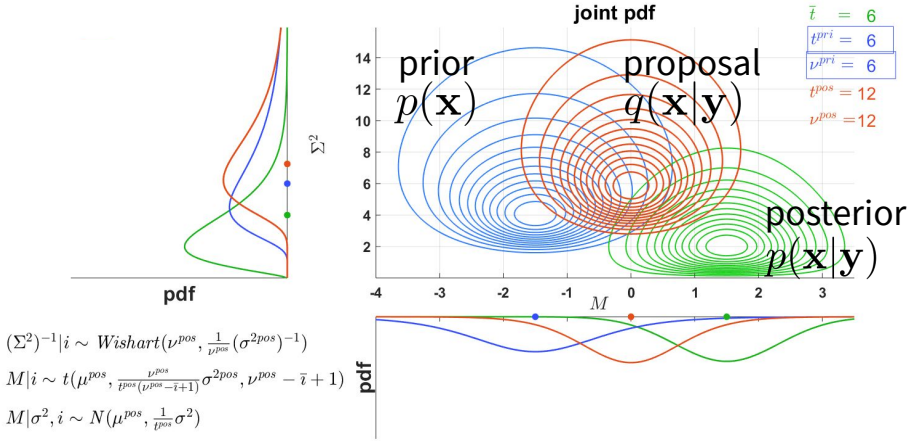

```
class PhysicsModel(Model):
    def __init__(self, draw=True, physics_steps_per_frame=5):
        super().__init__('Physics')
        self._draw = draw
        self._physics_steps_per_frame = physics_steps_per_frame

    def forward(self):
        ball_radius = max(5,int(pyprob.sample(Normal(12, 6), name='ball_radius')))
        ball_elasticity = float(pyprob.sample(Normal(0.9, 0.1), name='ball_elasticity'))
        num_bumpers = int(pyprob.sample(Uniform(2, 35), name='num_bumpers'))
        bumpers = []
        for i in range(num_bumpers):
            x = int(pyprob.sample(Normal(450, 250), name='bumper{}x'.format(i)))
            y = int(pyprob.sample(Normal(200, 100), name='bumper{}y'.format(i)))
            bumpers.append([x, y])
        p = PhysicsSim(bumpers=bumpers, ball_radius=ball_radius, ball_elasticity=ball_ela
        p.run()
        balls_in_box = len(p._balls_in_box)
        pyprob.observe(Normal(balls_in_box, 1), balls_in_box, name='balls_in_box')
```

```
model = PhysicsModel(draw=True, physics_steps_per_frame=2)
trace = model.get_trace()
```



Inference engines

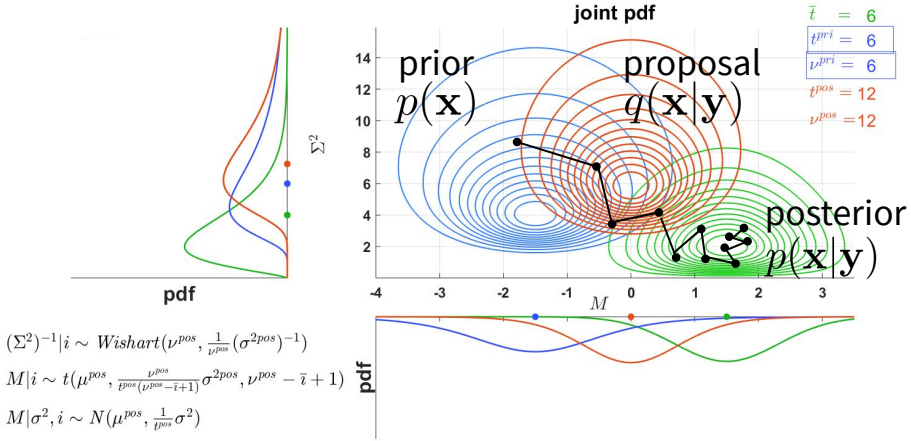
- Markov chain Monte Carlo
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- Importance sampling
  - Propose from prior $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ parameterized by observations
  - No autocorrelation or burn in
  - Each sample is independent (parallelizable)
- Others: variational inference, Hamiltonian Monte Carlo, etc.



We sample in trace space:
each sample (trace) is one full execution of the model/simulator!

Inference engines

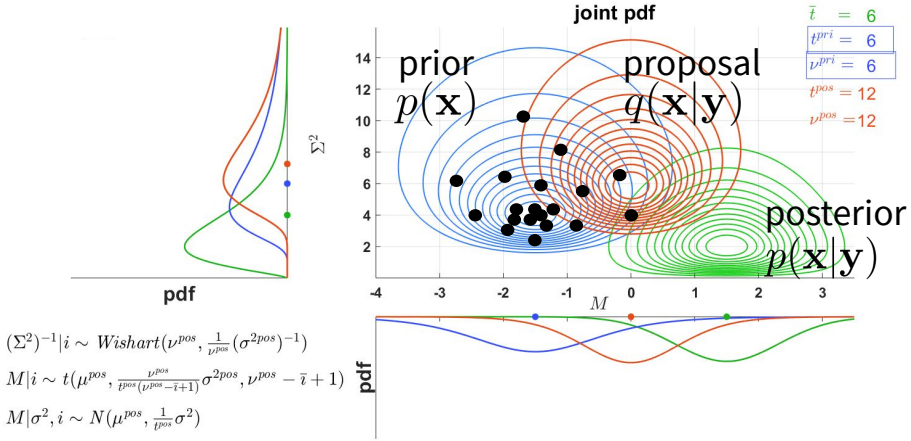
- **Markov chain Monte Carlo**
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- Importance sampling
  - Propose from prior $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ parameterized by observations
  - No autocorrelation or burn in
  - Each sample is independent (parallelizable)
- Others: variational inference, Hamiltonian Monte Carlo, etc.



We sample in trace space:
each sample (trace) is one full execution of the model/simulator!

Inference engines

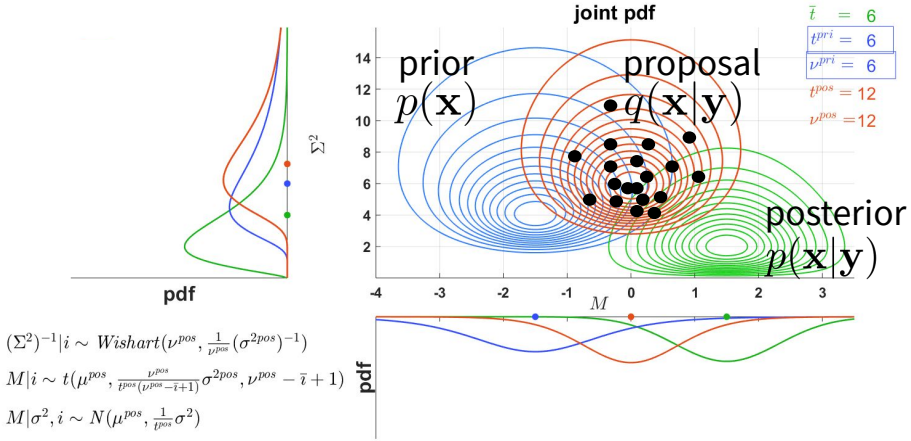
- Markov chain Monte Carlo
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- **Importance sampling**
  - **Propose from prior** $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ parameterized by observations
  - No autocorrelation or burn in
  - Each sample is independent (parallelizable)
- Others: variational inference, Hamiltonian Monte Carlo, etc.



We sample in trace space:
each sample (trace) is one full execution of the model/simulator!

Inference engines

- Markov chain Monte Carlo
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- **Importance sampling**
  - Propose from prior $p(\mathbf{x})$
  - **Use learned proposal** $q(\mathbf{x}|\mathbf{y})$ parameterized by observations
  - No autocorrelation or burn in
  - Each sample is independent (parallelizable)
- Others: variational inference, Hamiltonian Monte Carlo, etc.



We sample in trace space:
each sample (trace) is one full execution of the model/simulator!

Probabilistic programming languages (PPLs)

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

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

Existing simulators as probabilistic programs

Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, Frank Wood. 2019. “Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model.” **NeurIPS 2019**

Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence F. Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Kyle Cranmer, Victor Lee, Prabhat, Frank Wood. 2019. “Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale.” International Conference for High Performance Computing, Networking, Storage, and Analysis - **SC19**

Execute existing simulators as probprog

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

Key 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

Execute existing 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)
- Hamiltonian Monte Carlo (in progress)

pyprob

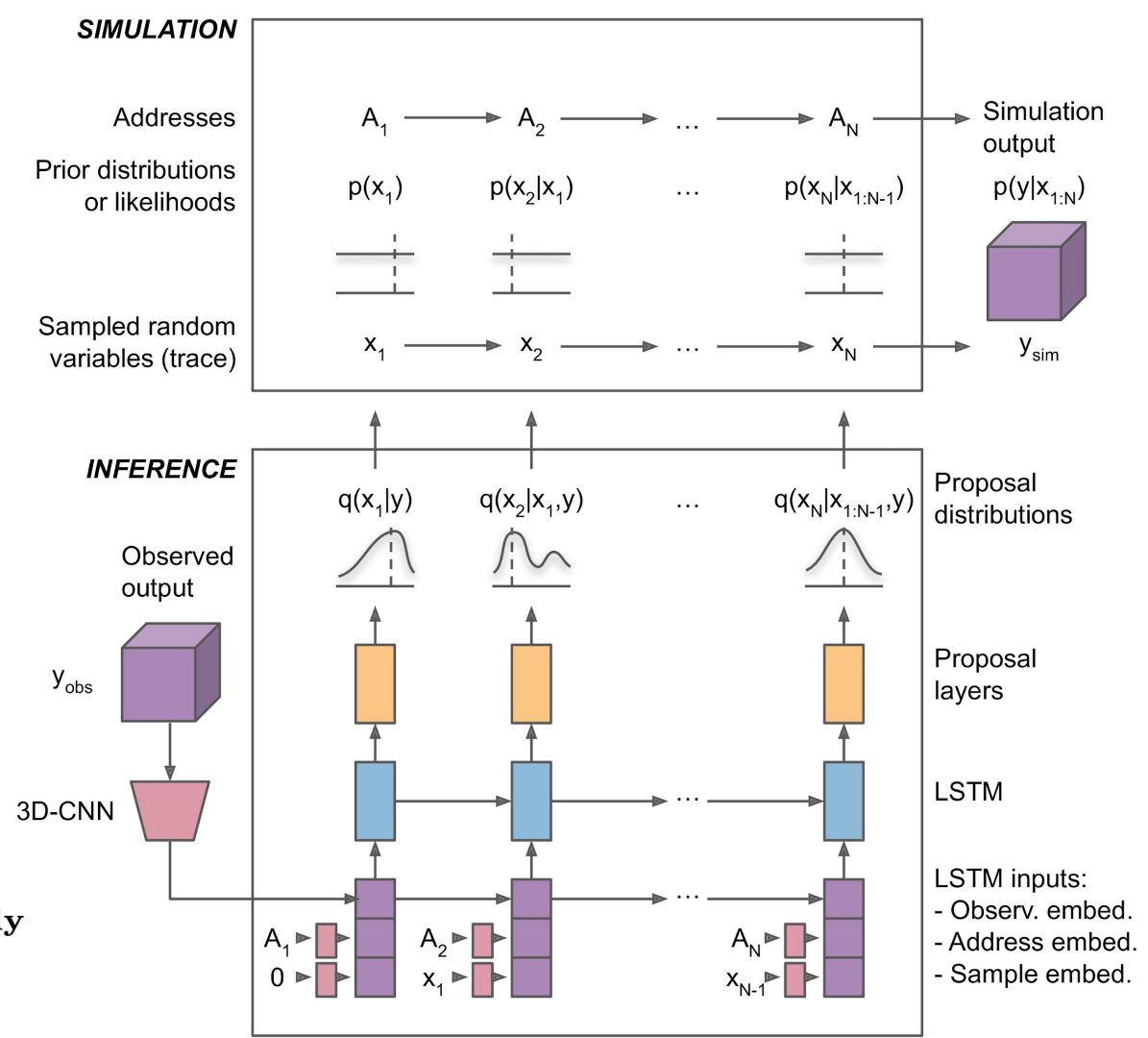
https://github.com/probprog/pyprob

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)**

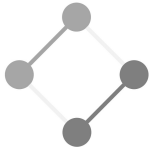
Le, Baydin and Wood. Inference Compilation and Universal Probabilistic Programming. AISTATS 2017



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

PPX

https://github.com/probprog/ppx



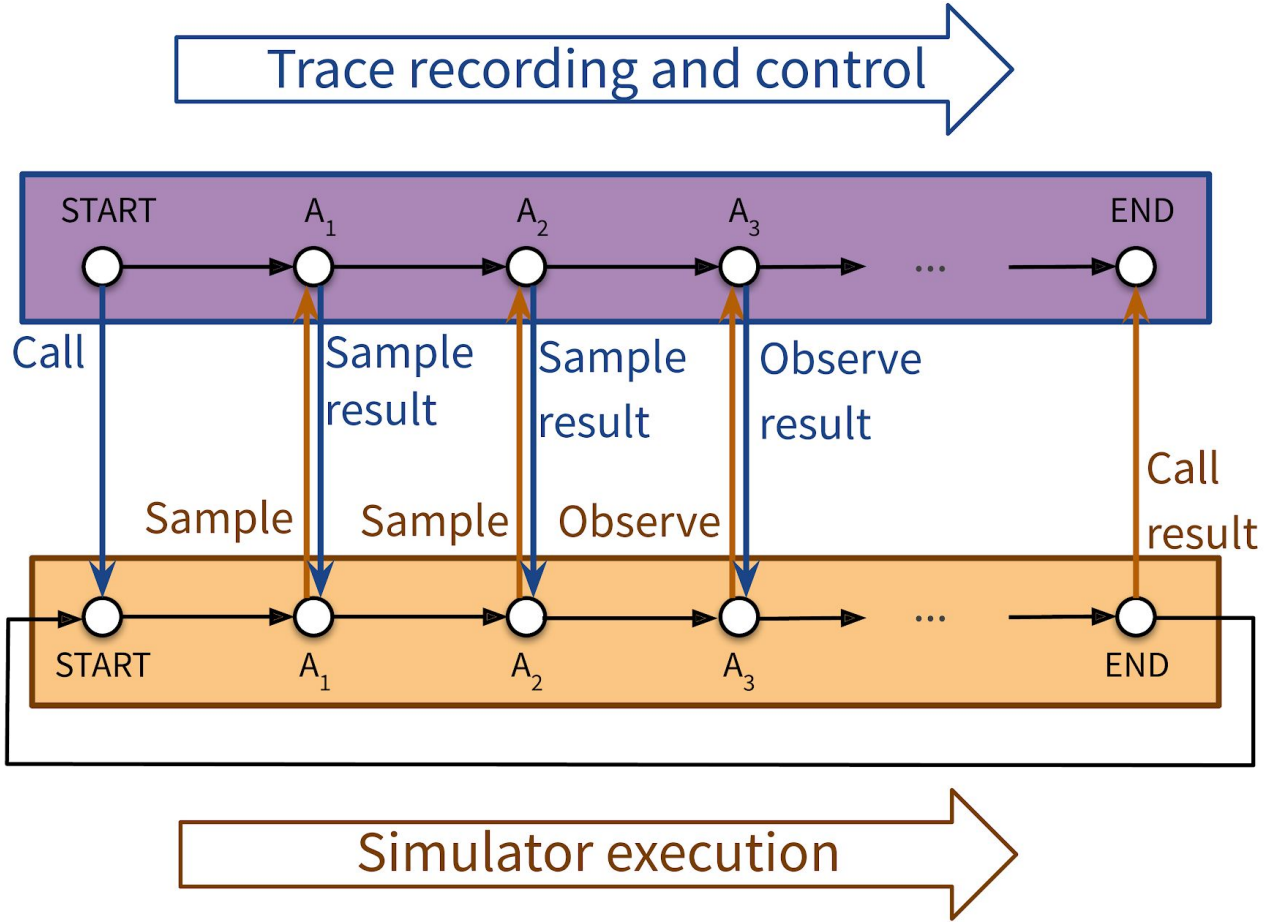
PPX

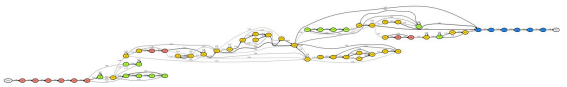
Probabilistic **P**rogramming e**X**ecution protocol

- 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



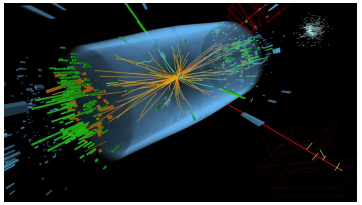


Probabilistic programming system

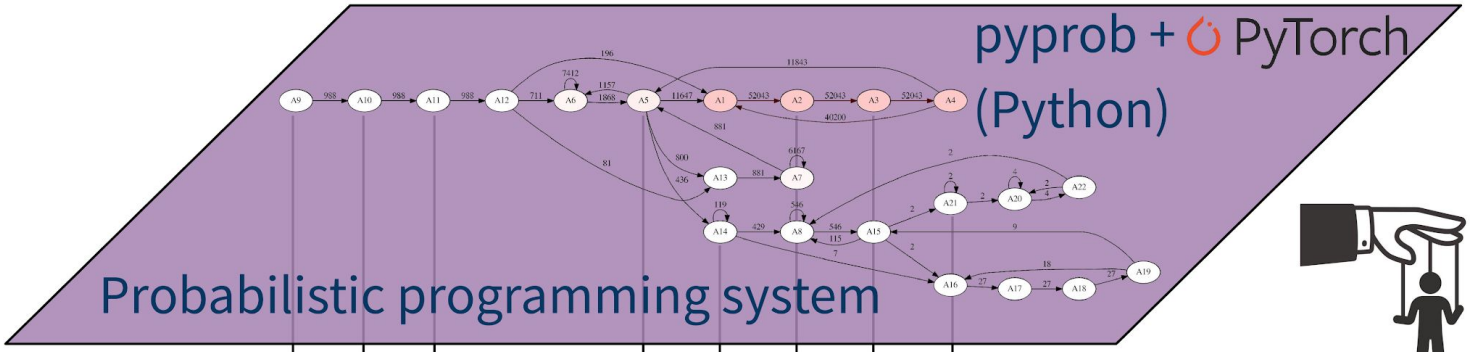
PPX

Simulator

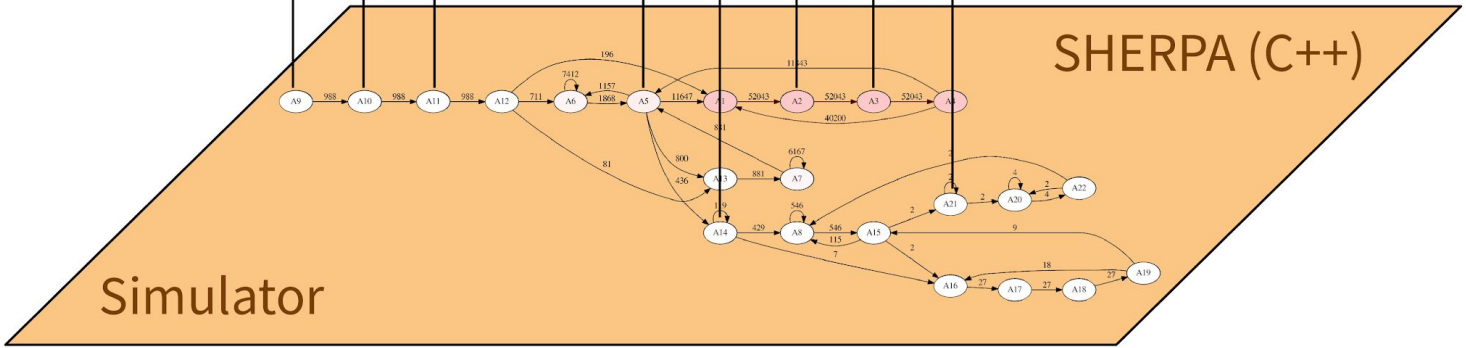
E.g., SHERPA, GEANT



PPX


pyprob + PyTorch
(Python)
Probabilistic programming system

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;
}
```

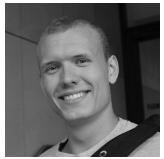
Probprog and high-energy physics
“etalumis”

simulate

etalumis | simulate


Atılım Güneş Baydin


Lukas Heinrich


Andreas Munk

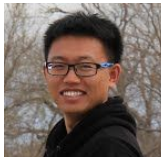

Wahid Bhimji


Lei Shao


Bradley Gram-Hansen


Gilles Louppe


Saeid Naderiparizi


Jialin Liu


Larry Meadows

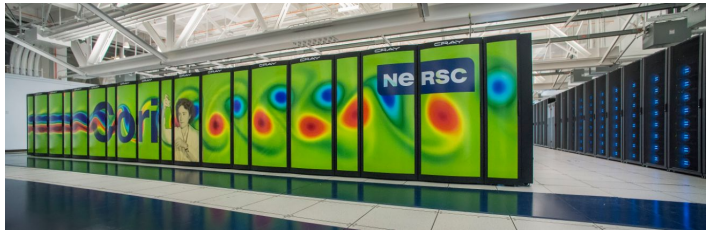

Phil Torr


Kyle Cranmer


Frank Wood


Prabhat


Victor Lee



Cori supercomputer, Lawrence Berkeley Lab
2,388 Haswell nodes (32 cores per node)
9,688 KNL nodes (68 cores per node)

High-energy physics simulators

Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, Frank Wood. 2019. “Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model.” **NeurIPS 2019**

https://arxiv.org/abs/1807.07706

Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence F. Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Kyle Cranmer, Victor Lee, Prabhat, Frank Wood. 2019. “Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale.” International Conference for High Performance Computing, Networking, Storage, and Analysis - **SC19**

https://arxiv.org/abs/1907.03382

- **PPL with HPC** features: **multi-TB empirical distributions**, **distributed inference and training**
- The largest scale posterior inference in a Turing-complete PPL; approx. 25,000 latents expressed by Sherpa code base of 1M lines of C++ code
- Synchronous data parallel training of a dynamic 3DCNN–LSTM with PyTorch MPI, at the scale of 1,024 nodes (**32,768 CPU cores**) with **128k global minibatch size**. Largest scale use of PyTorch MPI, largest minibatch size for this form of NN

Best Paper Finalist

International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19), Denver, CO, November 17–22, 2019

pyprob_cpp and Sherpa

```cpp
#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;
}
```

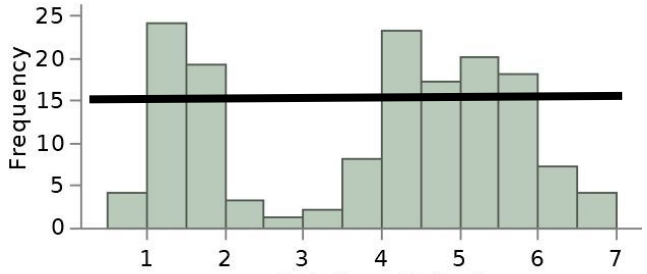
Main challenges

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

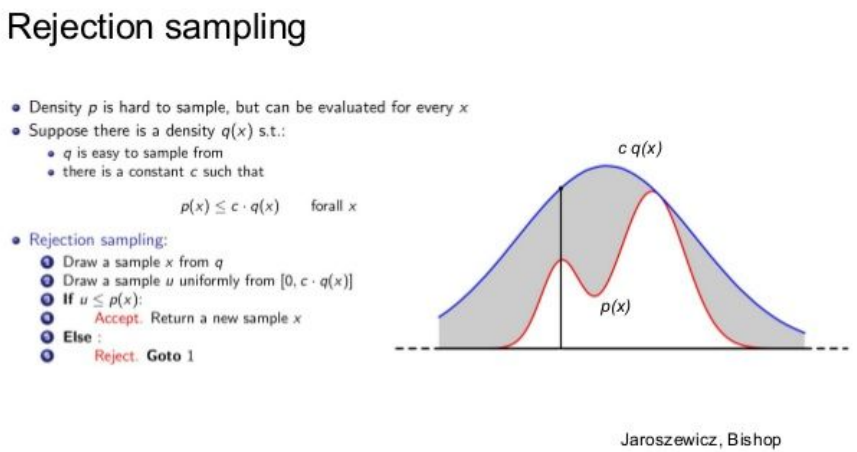
    HEP: differential cross-section = phase space * matrix element



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



Experiments

Tau lepton decay

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

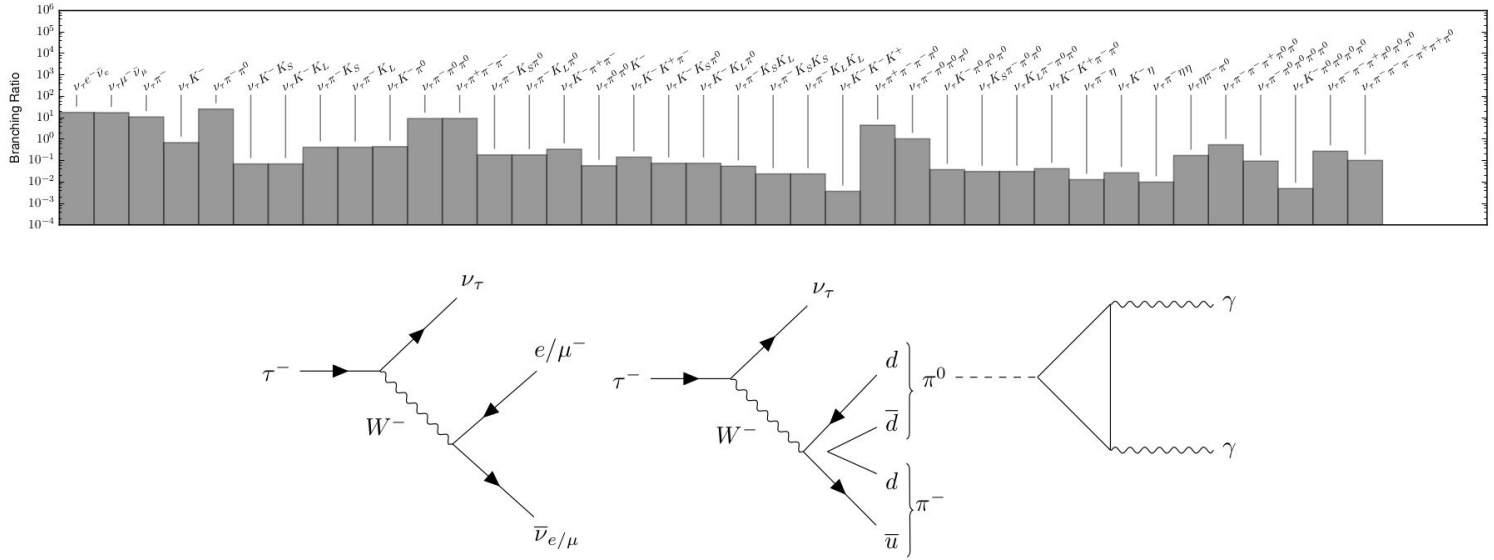


Figure 2: *Top:* branching ratios of the τ lepton, effectively the prior distribution of the decay channels in SHERPA. Note that the scale is logarithmic. *Bottom:* Feynman diagrams for τ decays illustrating that these can produce multiple detected particles.

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

...

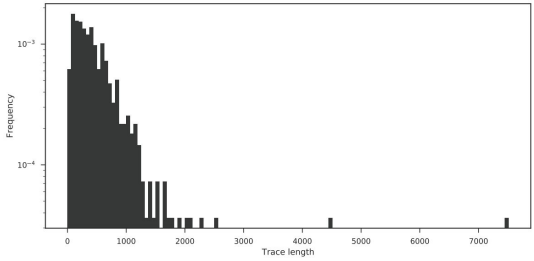
Common trace types in Sherpa

Approximately 450 trace types encountered

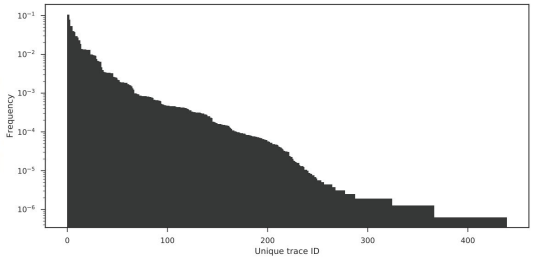
Trace type: unique sequencing of addresses (with different sampled values)

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

...



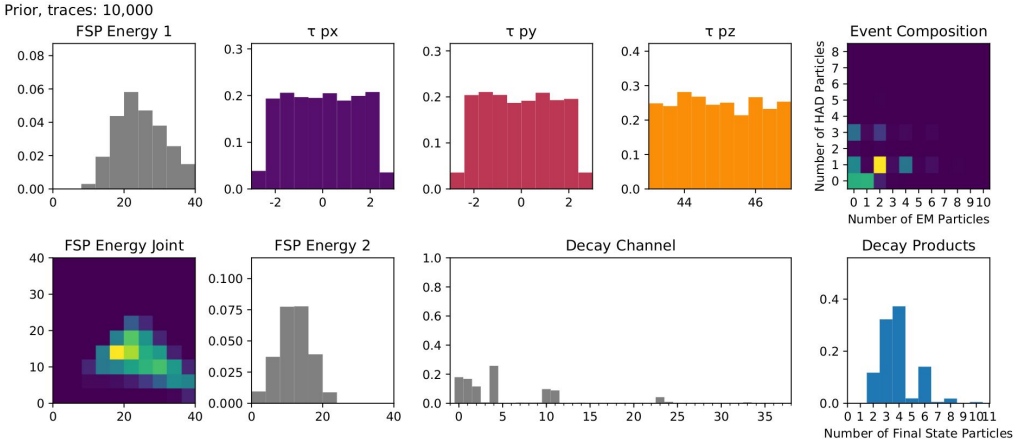
(a) Distribution of trace lengths (all addresses). Min: 13, max: 7,514, mean: 383.58.



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

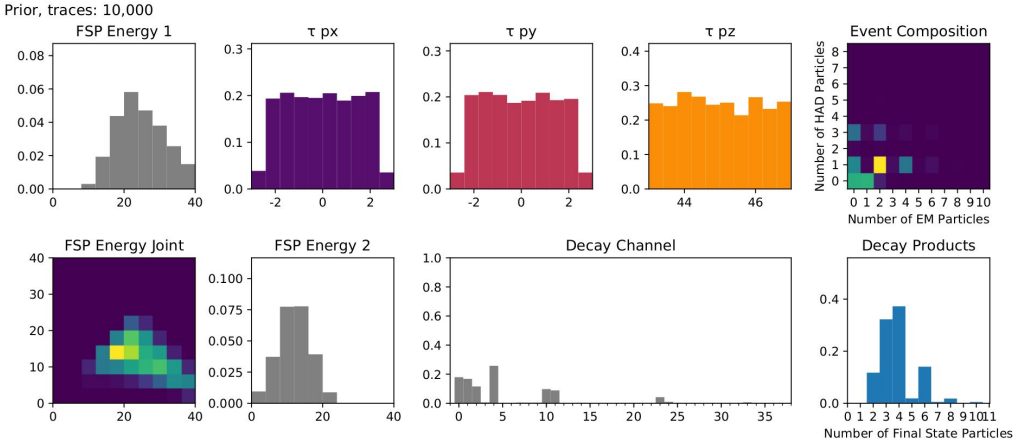
Inference results with MCMC engine

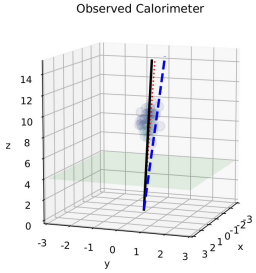
Prior



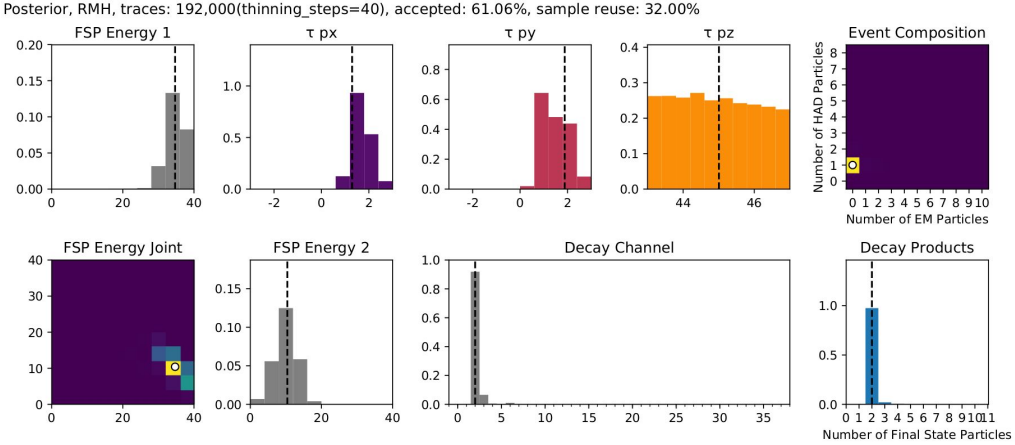
Inference results with MCMC engine

Prior





MCMC Posterior
conditioned on
calorimeter



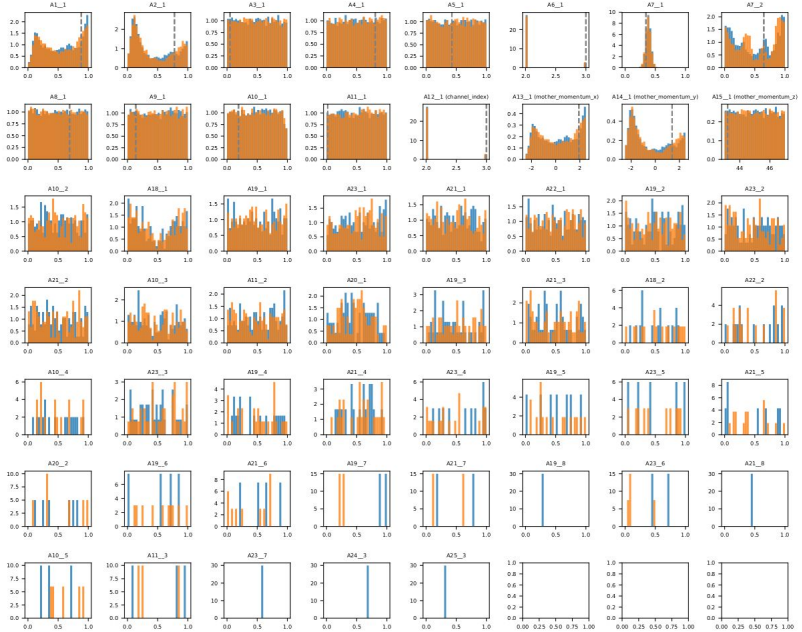
7,700,000 samples

Slow and has to run single node

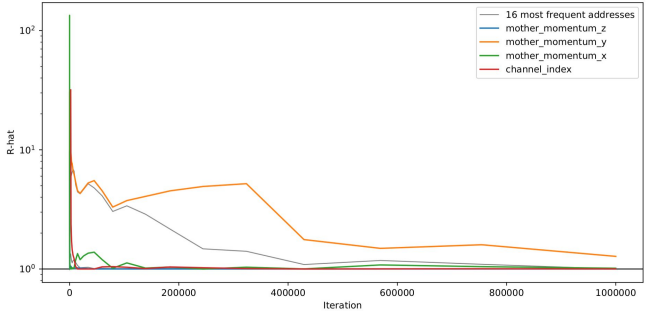
Convergence to true posterior

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

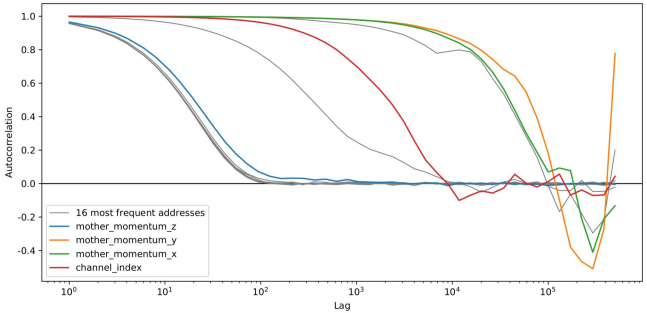
- Chain initialized with random trace from prior
- Chain initialized with known ground-truth trace



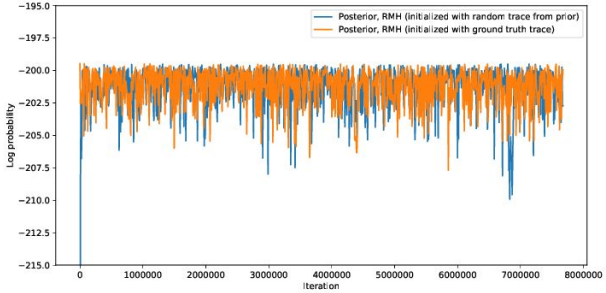
Gelman-Rubin convergence diagnostic



Autocorrelation



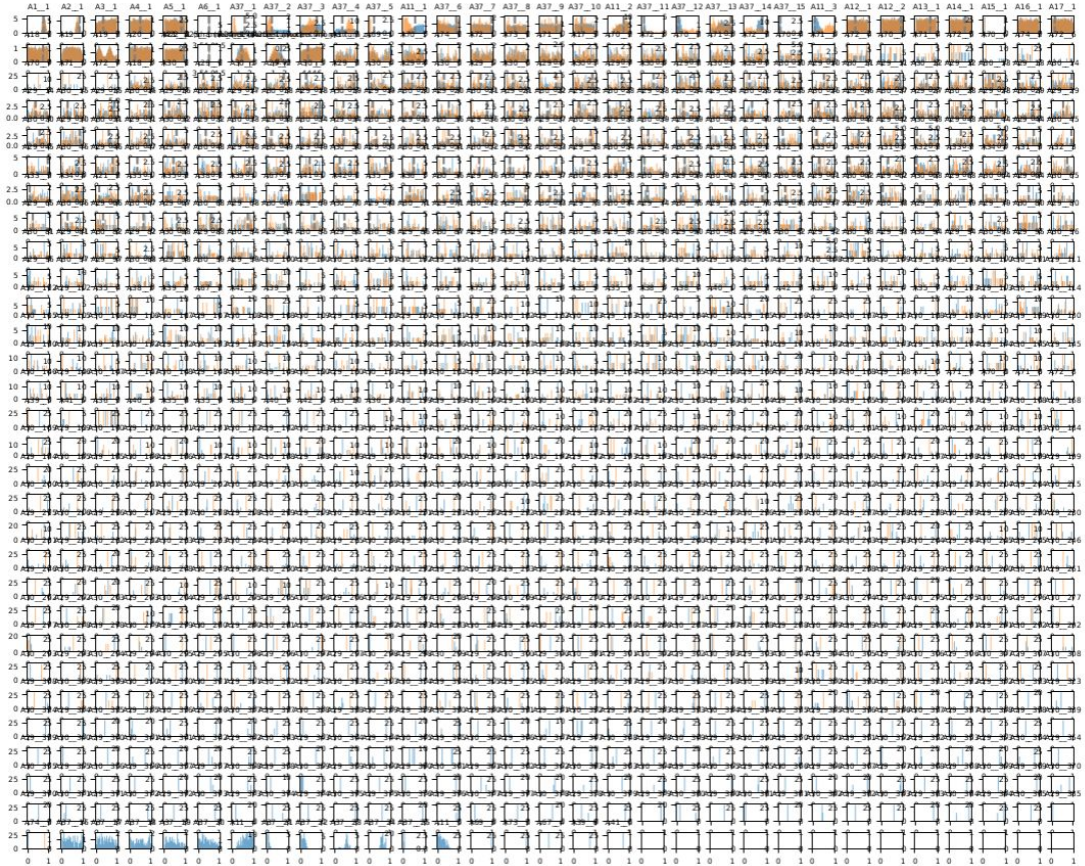
Trace log-probability



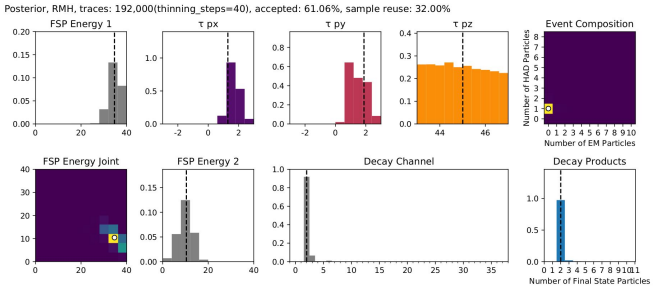
Convergence to true posterior

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:

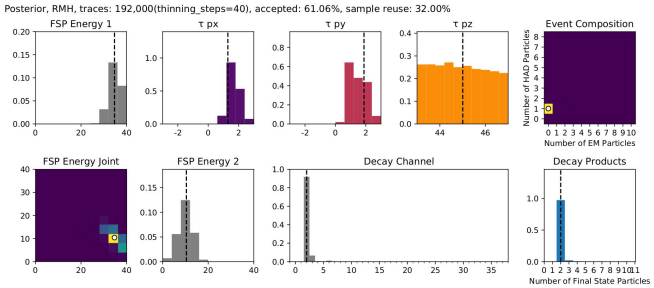


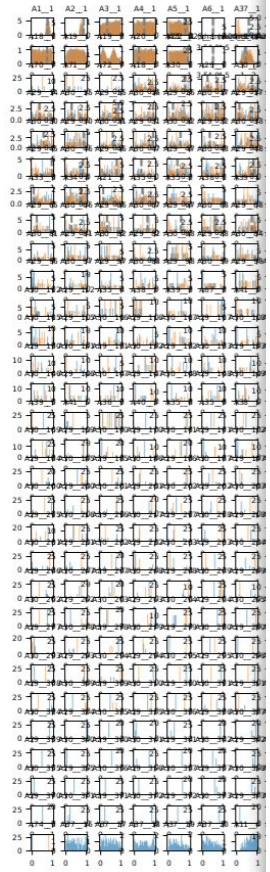
Convergence to true poste

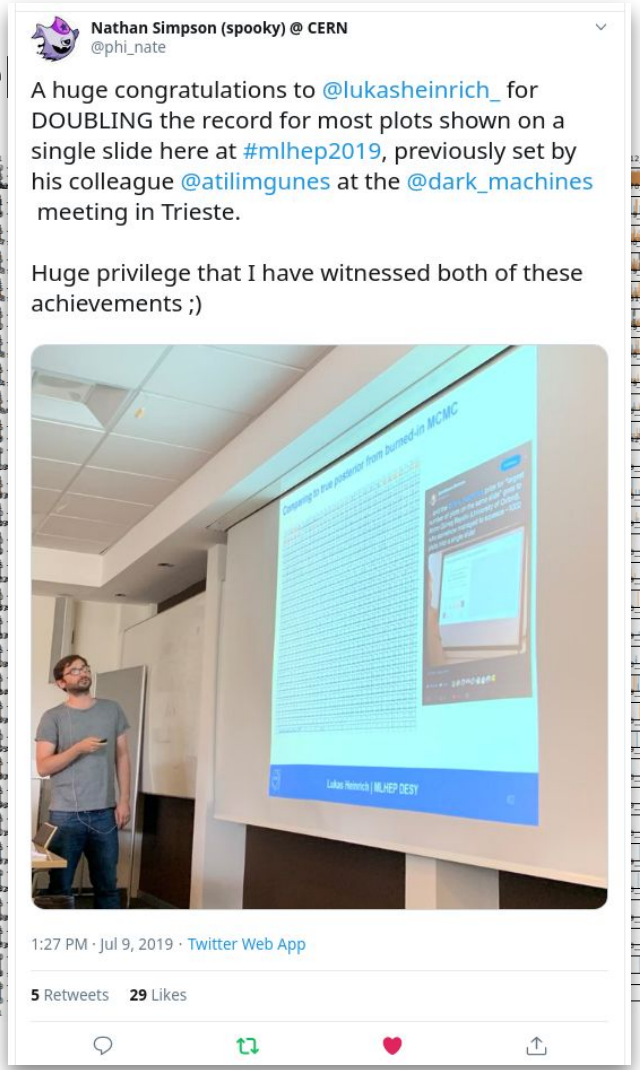
Important:

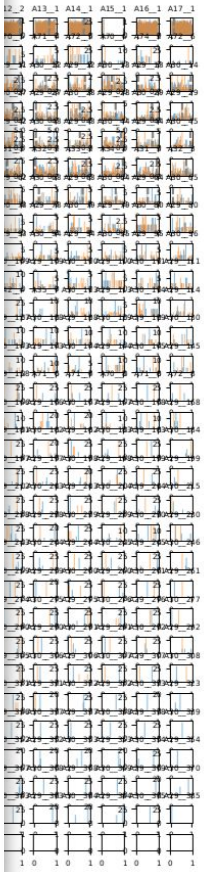
- We get **posteriors over the whole Sherpa address space, 1000s of addresses**
- Trace complexity varies depending on observed event

This is just a selected subset:



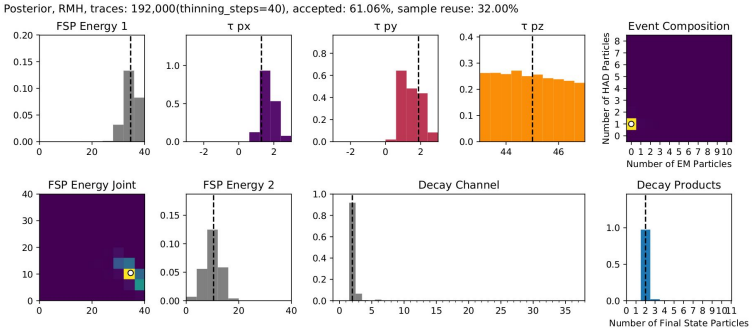






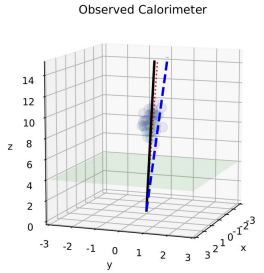
Inference results with IC engine

MCMC true posterior (7.7M single node)



Inference results with IC engine

MCMC true posterior (7.7M single node)

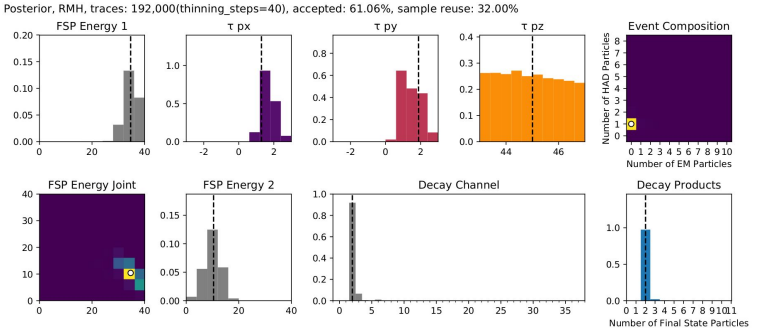


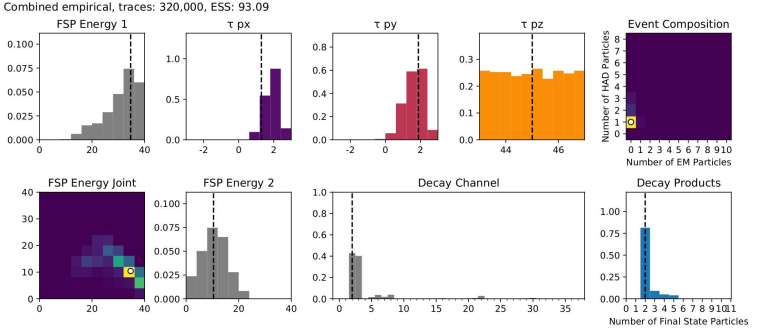
IC proposal
from trained NN

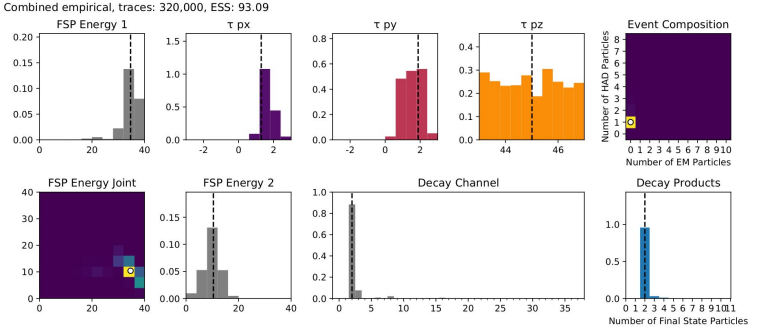
IC posterior
after importance weighting

320,000 samples

Fast “embarrassingly” parallel multi-node

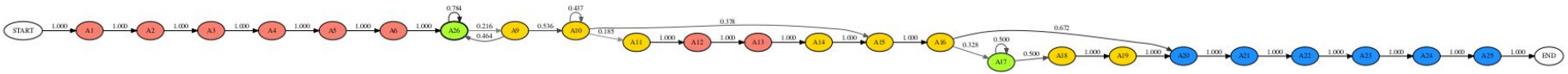






Interpretability

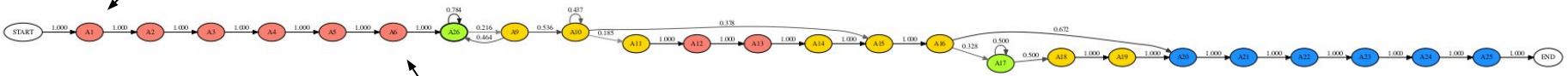
Latent probabilistic structure of 10 most frequent trace types



Interpretability

Latent probabilistic structure of 10 most frequent trace types

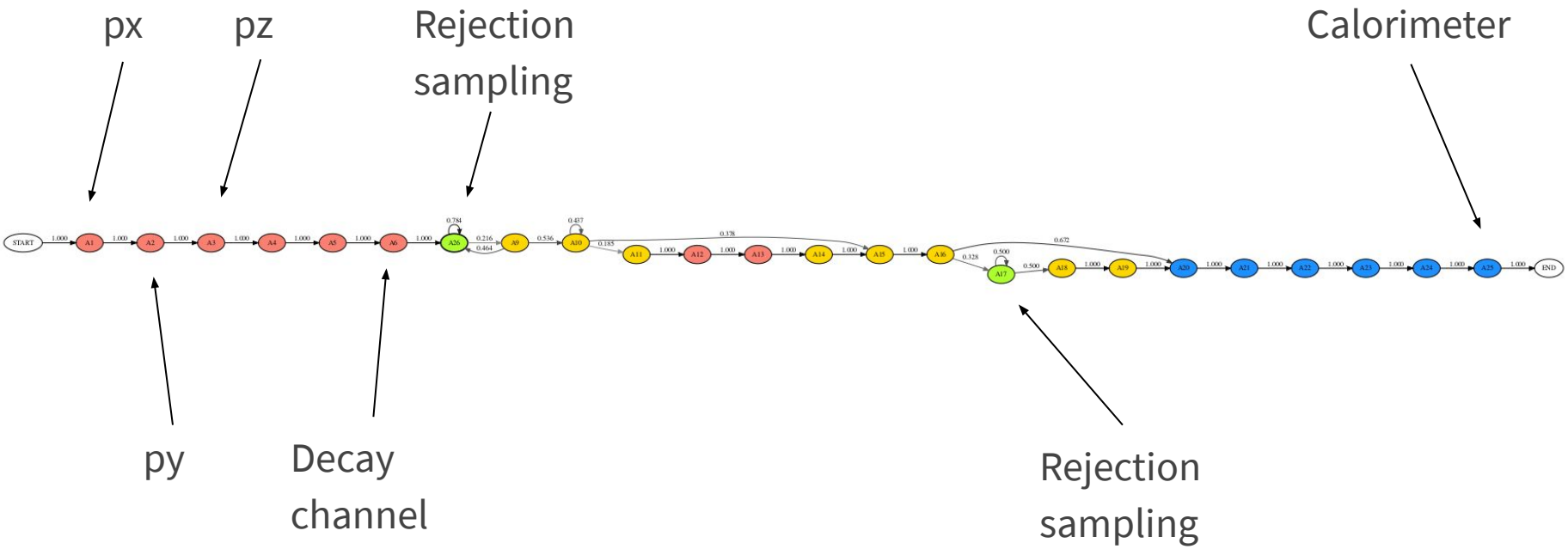
[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler::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

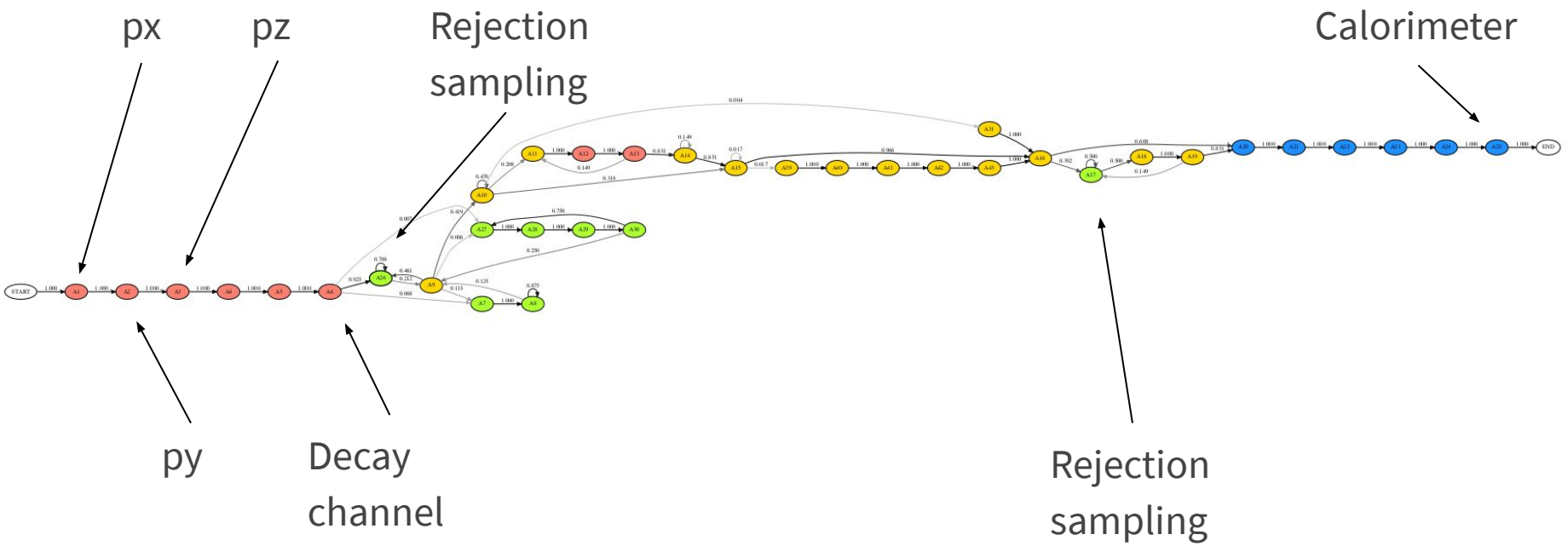
Interpretability

Latent probabilistic structure of 10 most frequent trace types



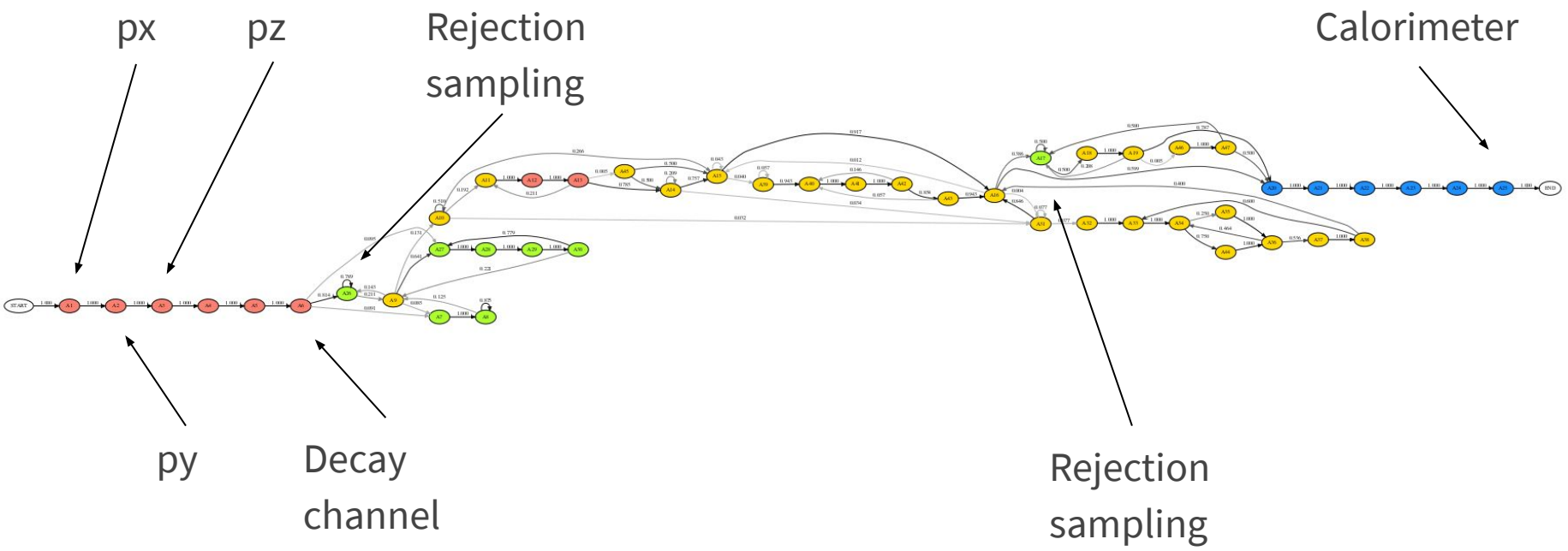
Interpretability

Latent probabilistic structure of 25 most frequent trace types



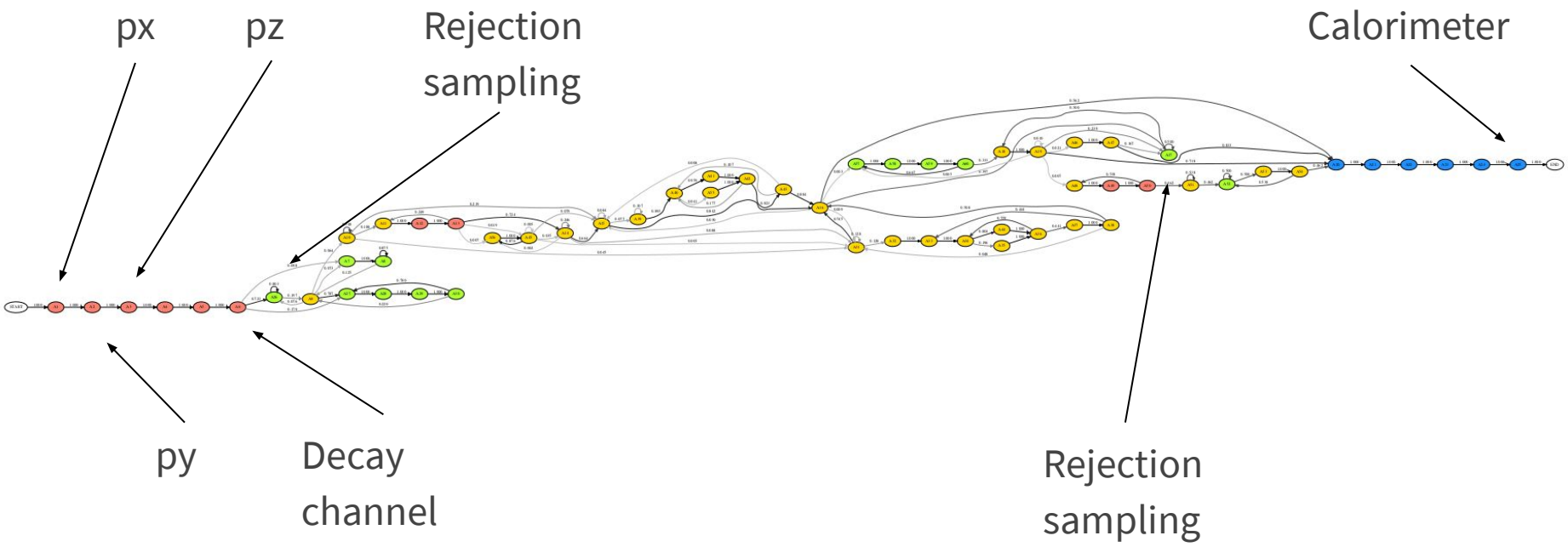
Interpretability

Latent probabilistic structure of 100 most frequent trace types

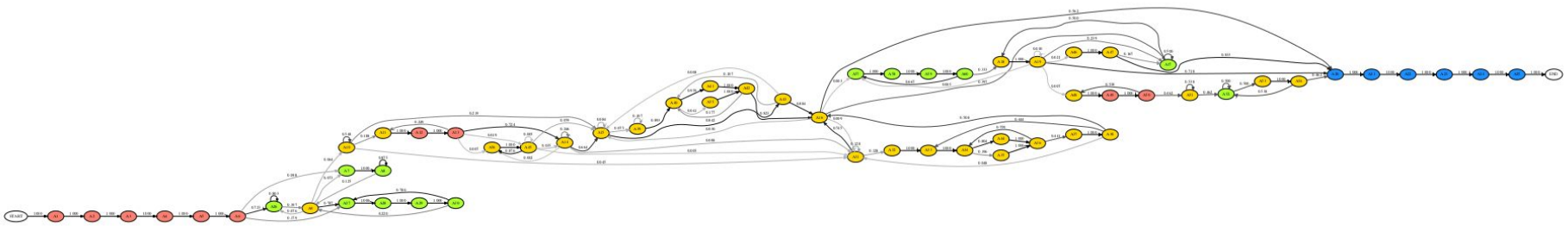


Interpretability

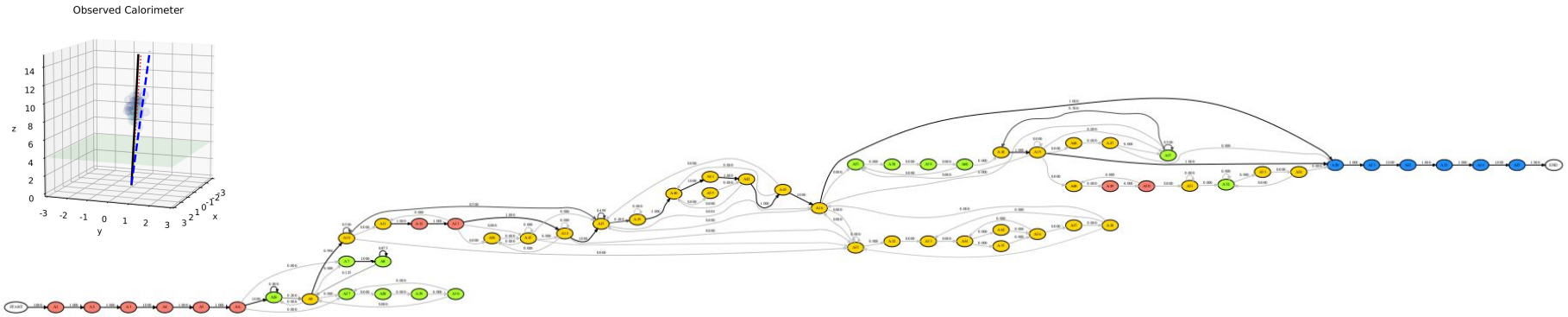
Latent probabilistic structure of 250 most frequent trace types



Interpretability



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



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

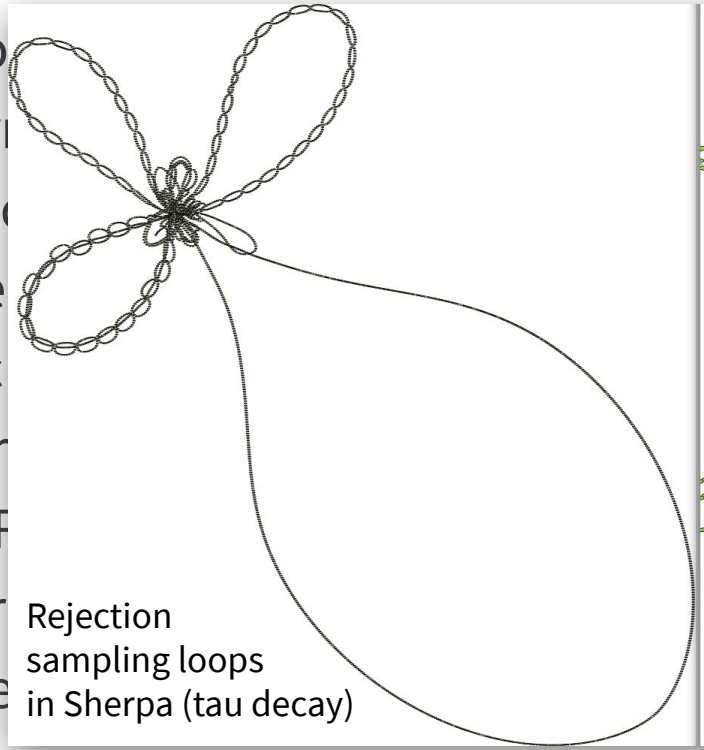
What’s next?

Current and upcoming work

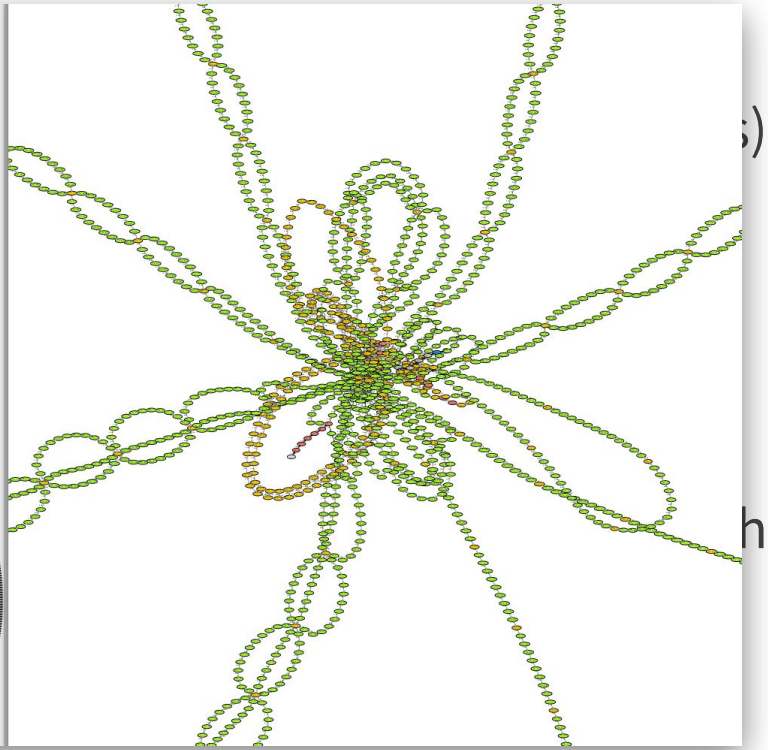
- Autodiff through PPX protocol
- **Learning simulator surrogates** (approximate forward simulators)
- **Rejection sampling loops** (weighting schemes)
- Rare event simulation for compilation (“prior inflation”)
- Batching of open-ended traces for NN training
- Distributed training of dynamic networks
  - Recently ran on 32k CPU cores on Cori (largest-scale PyTorch MPI)
- User features: posterior code highlighting, etc.
- Other simulators: astrophysics, epidemiology, computer vision

Current and upcoming work

- Auto
- Lear
- Reje
- Rare
- Batc
- Distr
  - F
- User
- Othe



Rejection
sampling loops
in Sherpa (tau decay)



Machine Learning and the Physical Sciences

Workshop at *Neural Information Processing Systems (NeurIPS)* conference

December 14, 2019, Vancouver, Canada

- Machine learning for physical sciences
- Physics for machine learning

Invited talks: **Alan Aspuru-Guzik, Yasaman Bahri, Katie Bouman, Bernhard Schölkopf, Maria Schuld, Lenka Zdeborova**

Contributed talks: **MilesCranmer, Eric Metodiev, Danilo Jimenez Rezende, Alvaro Sanchez-Gonzalez, Samuel Schoenholz, Rose Yu**

https://ml4physicalsciences.github.io/

Thank you for listening

Institute for Pure & Applied Mathematics, UCLA
15 October 2019

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

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

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

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

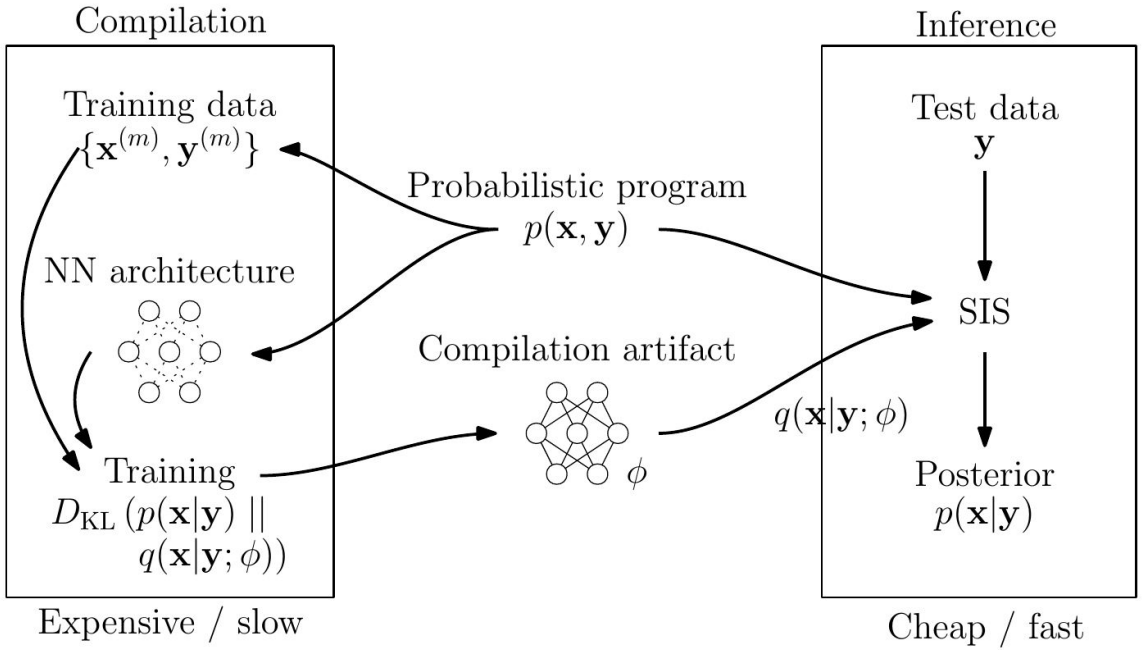
Inference engines

Model writing is decoupled from running inference

- 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)
    - Importance sampling (IS)
      - Inference compilation (IC)

Inference compilation

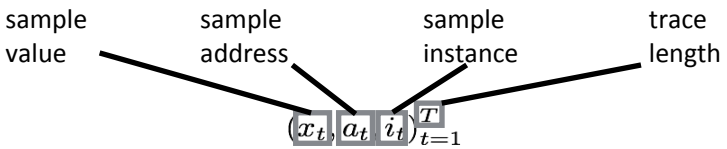
Transform a generative model implemented as a probabilistic program into a trained neural network artifact for performing inference



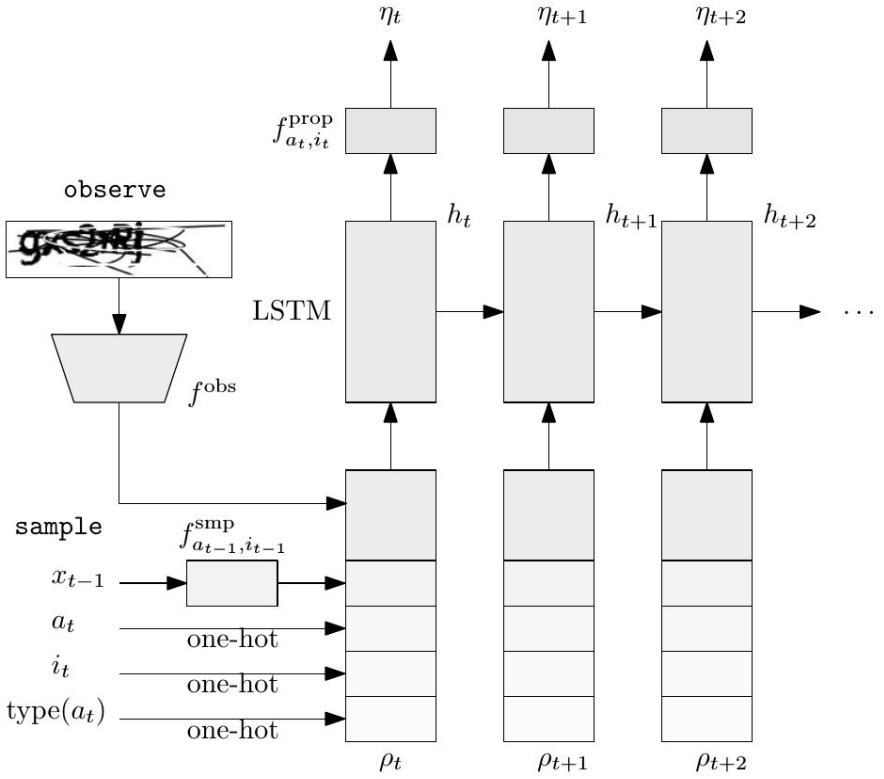
Inference compilation

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

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



Proposal distribution parameters

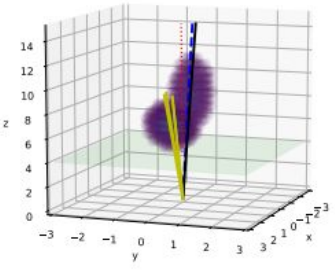


Tau lepton decay

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