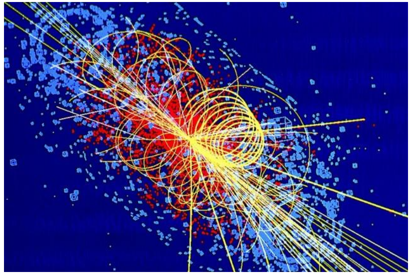
Probabilistic Programming for Inverse Problems in Physical Sciences

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

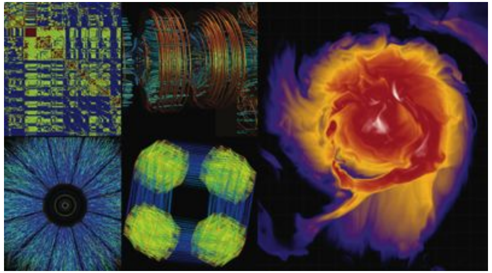
Stanford SLAC AI Seminar
25 Sep 2020

Simulation and physical sciences

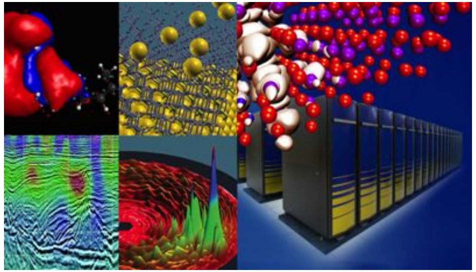
Computational models and simulation are key to scientific advance at all scales



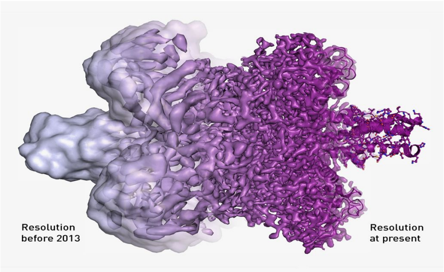
Particle physics



Nuclear physics

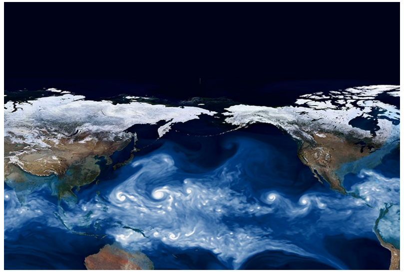


Material design

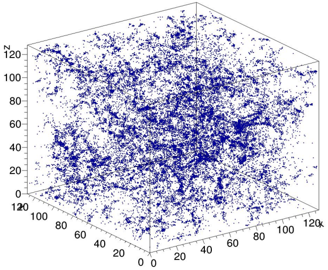


Drug discovery

Weather

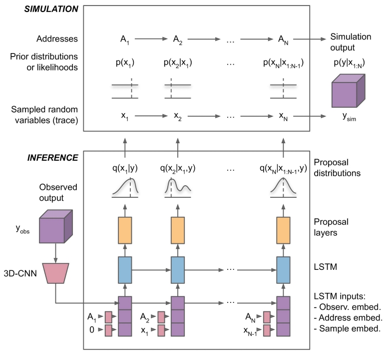


Climate science

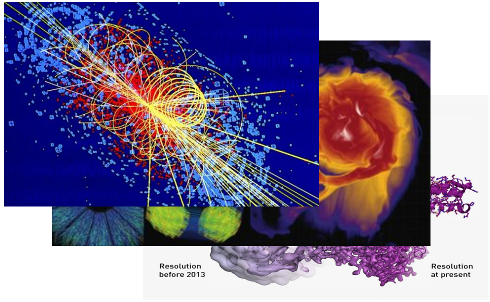


Cosmology

Introducing a new way to use existing simulators

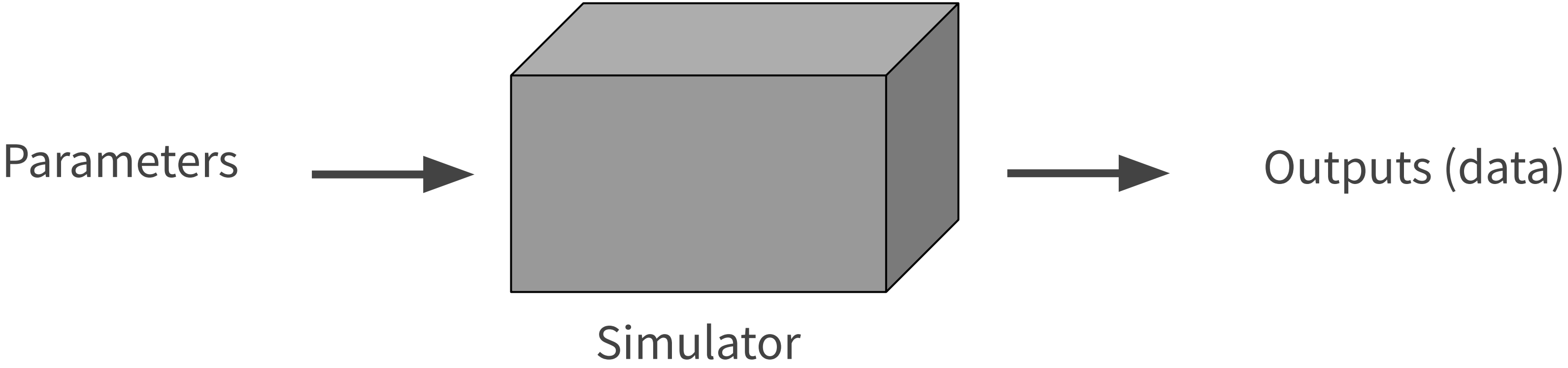


Probabilistic programming

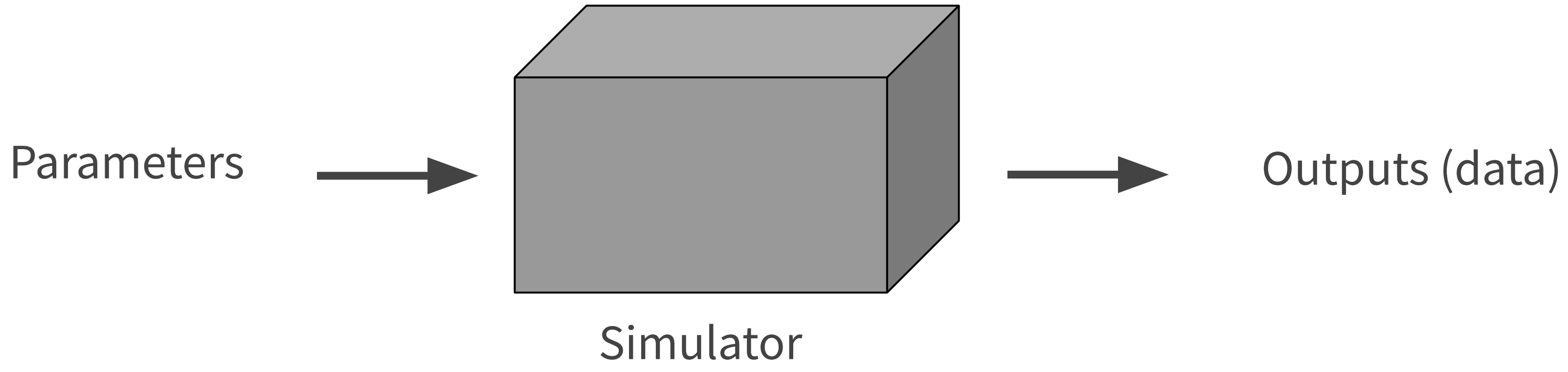


Simulation

Simulators



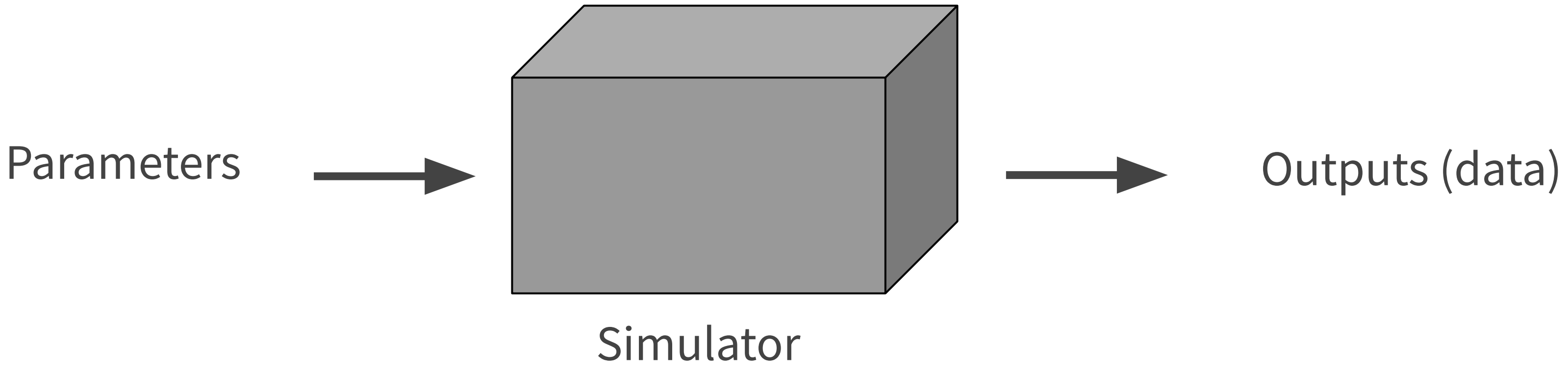
Simulators



Prediction:

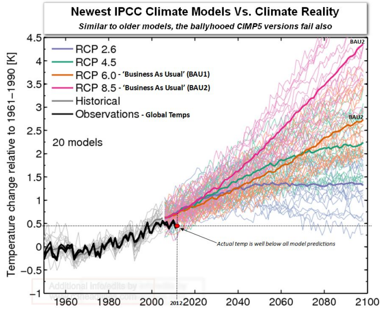
- Simulate forward evolution of the system
- Generate samples of output

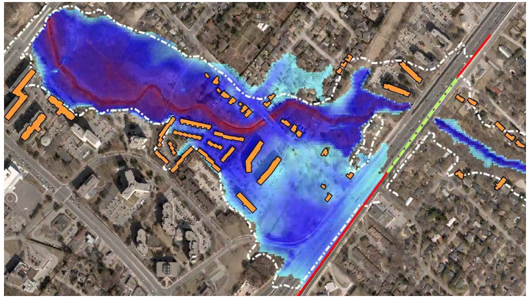
Simulators



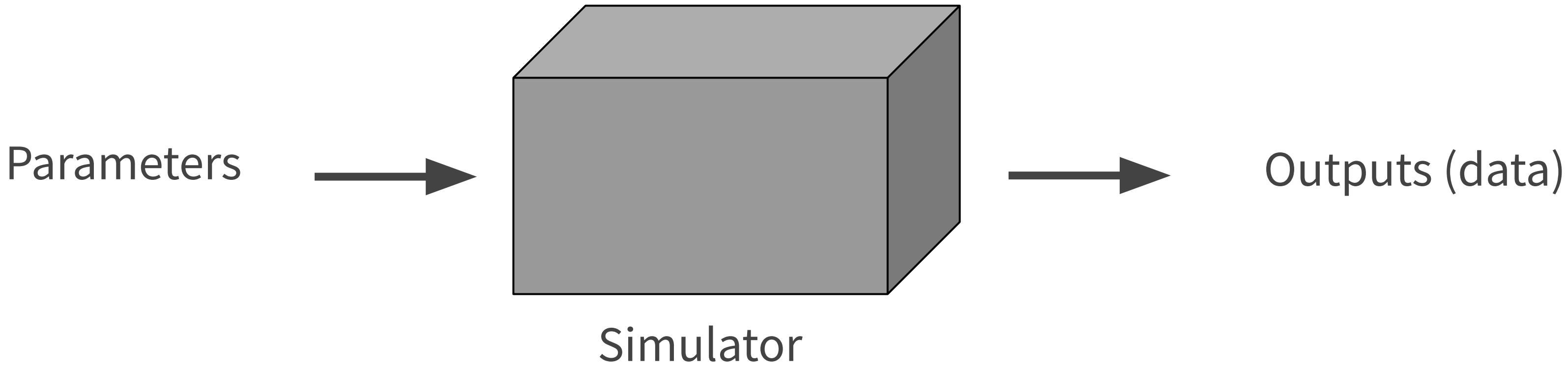
Prediction:

- Simulate forward evolution of the system
- Generate samples of output





Simulators

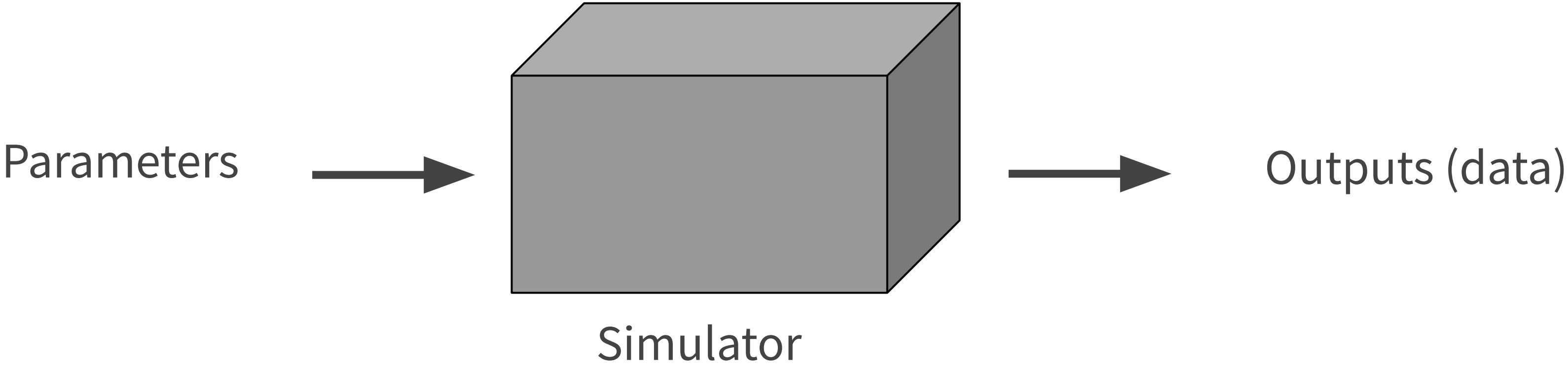


Prediction:

- Simulate forward evolution of the system
- Generate samples of output

WE NEED THE INVERSE!

Simulators



Prediction:

- Simulate forward evolution of the system
- Generate samples of output

Inference:

- Find parameters that can produce (explain) observed data
- Inverse problem
- Often a manual process

Simulators

Parameters

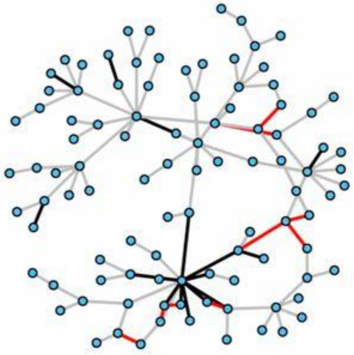
Outputs (data)

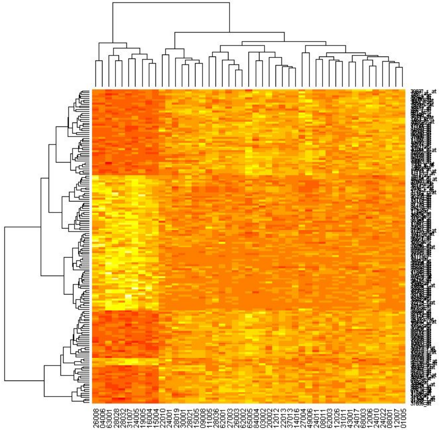
Simulator

Inferred parameters

Observed data

Gene network





Gene expression

Simulators

Parameters

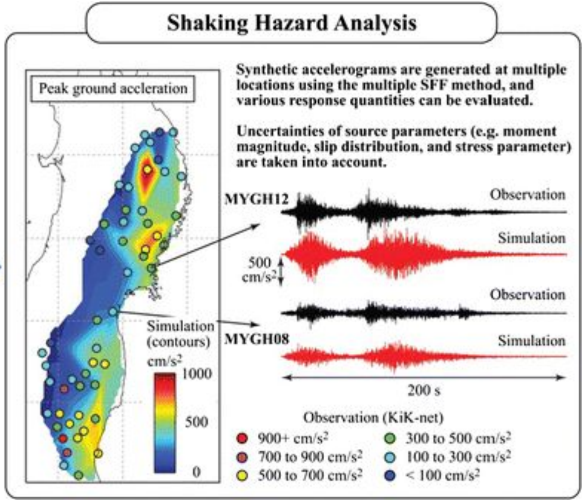
Outputs (data)

Simulator

Inferred parameters

Observed data

Earthquake location & characteristics



Seismometer readings

Simulators

Parameters

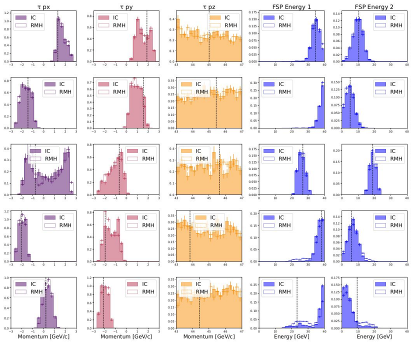
Outputs (data)

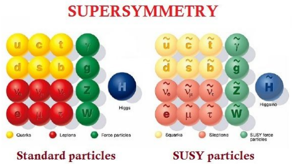
Simulator

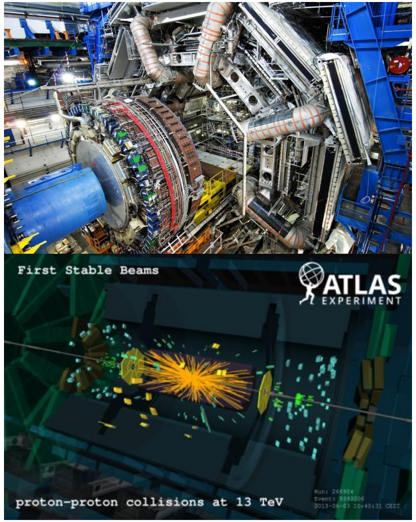
Inferred parameters

Observed data

Event analyses & new particle discoveries

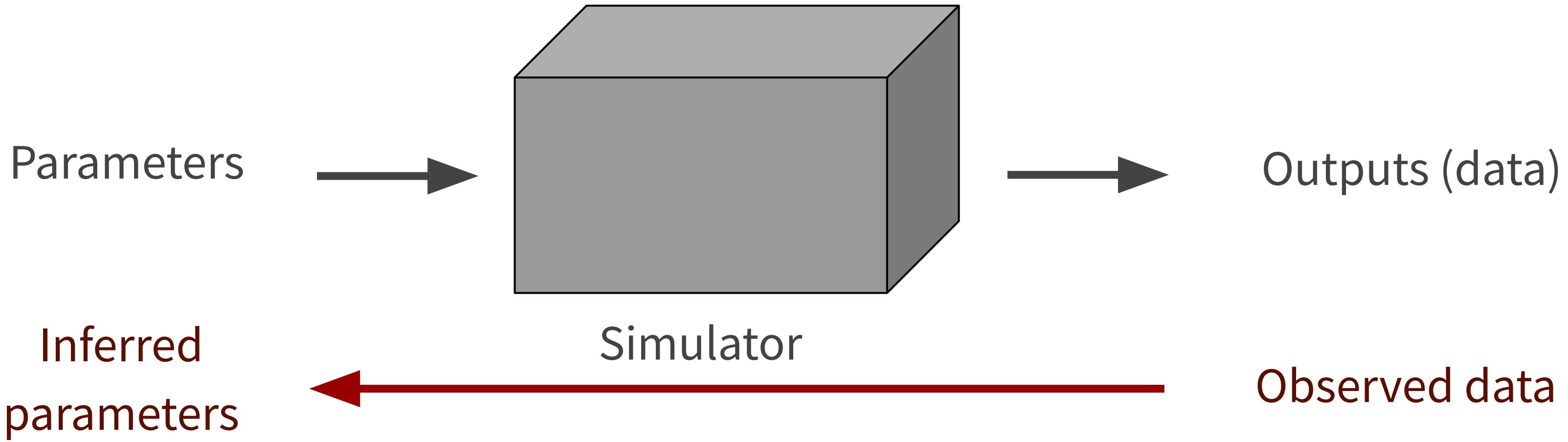






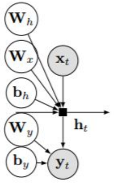
Particle detector readings

Inverting a simulator



Probabilistic programming is a machine learning framework allowing us to

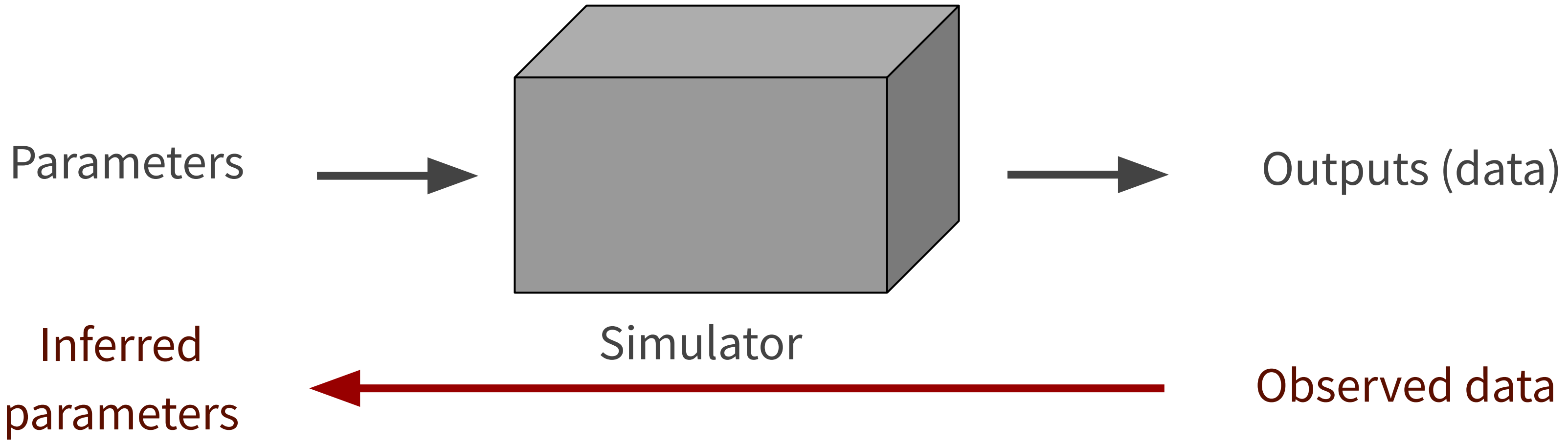
- write ***programs that define probabilistic models***
- run automated Bayesian ***inference of parameters conditioned on observed outputs*** (data)




```
def rnn_cell(hprev, xt):
  return tf.tanh(tf.dot(hprev, Wh) + tf.dot(xt, Wx) + bh)

Wh = Normal(mu=tf.zeros([H, H]), sigma=tf.ones([H, H]))
Wx = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
Wy = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
bh = Normal(mu=tf.zeros(H), sigma=tf.ones(H))
by = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.placeholder(tf.float32, [None, D])
h = tf.scan(rnn_cell, x, initializer=tf.zeros(H))
y = Normal(mu=tf.matmul(h, Wy) + by, sigma=1.0)
```

Pyro Edward Stan

Inverting a simulator



Probabilistic programming is a machine learning framework allowing us to

- writ
- run

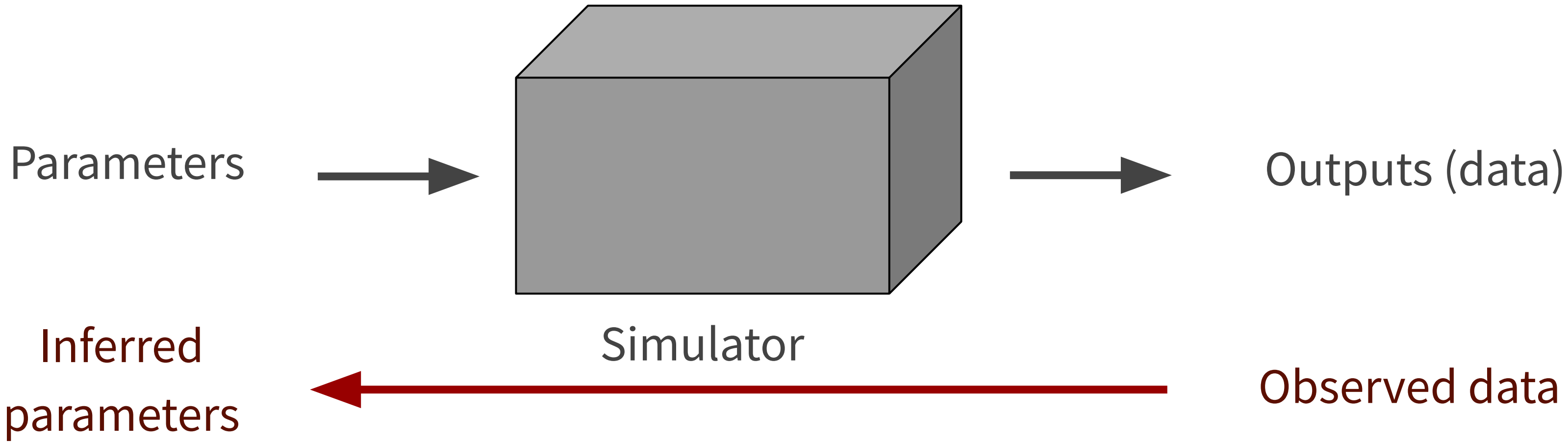
con

- Has been limited to **toy and small-scale problems**
- Normally requires one to **implement a probabilistic model from scratch** in the chosen language/system

```
cell(hprev, xt):
 tf.tanh(tf.dot(hprev, Wh) + tf.dot(xt, Wx) + bh)

mal(mu=tf.zeros([H, H]), sigma=tf.ones([H, H]))
mal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
mal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
mal(mu=tf.zeros(H), sigma=tf.ones(H))
mal(mu=tf.zeros(1), sigma=tf.ones(1))

laceholder(tf.float32, [None, D])
can(rnn_cell, x, initializer=tf.zeros(H))
al(mu=tf.matmul(h, Wy) + by, sigma=1.0)
```

Pyro Edward Stan

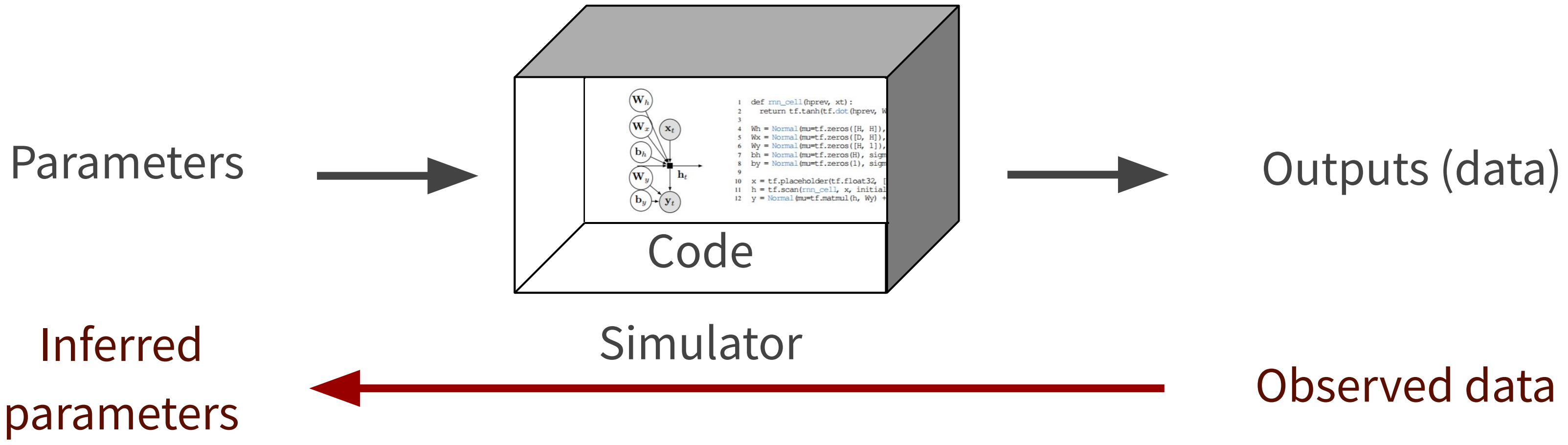
Inverting a simulator



Key idea:

Many simulators are stochastic and they define probabilistic models by sampling random numbers

Inverting a simulator

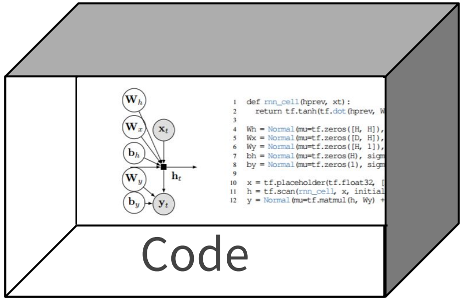


Key idea:
Many simulators are stochastic and they define probabilistic models by sampling random numbers

Simulators are probabilistic programs!

Inverting a simulator

Parameters



Outputs (data)

Simulator

Inferred parameters

Observed data

Key idea:

Many simulators are stochastic and they define probabilistic models by sampling random numbers

Simulators are probabilistic programs!

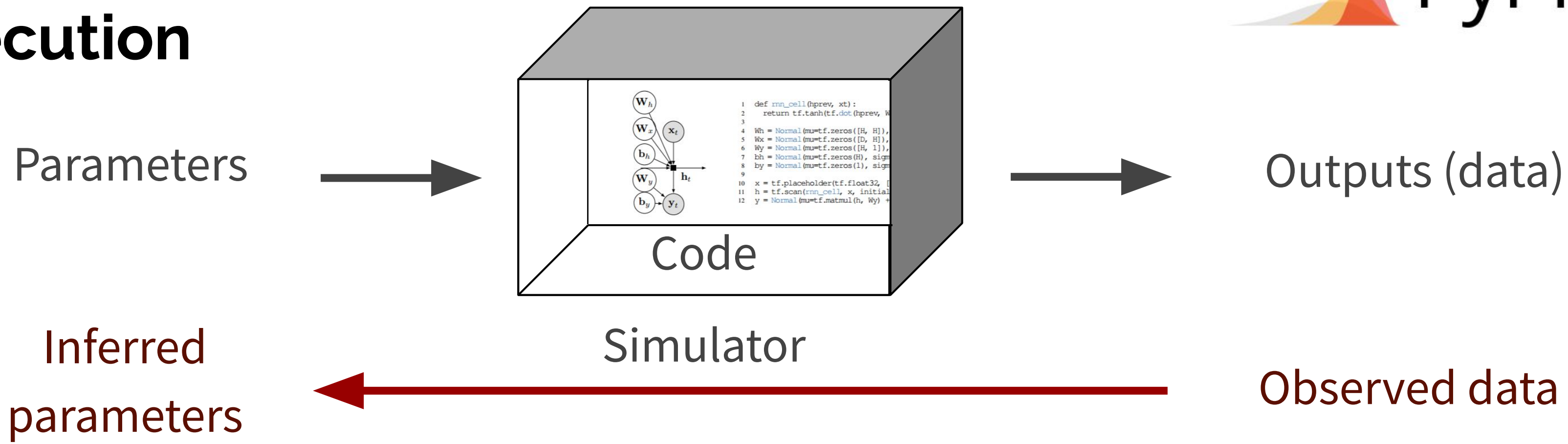
We “just” need an infrastructure to execute them as such

PyProb

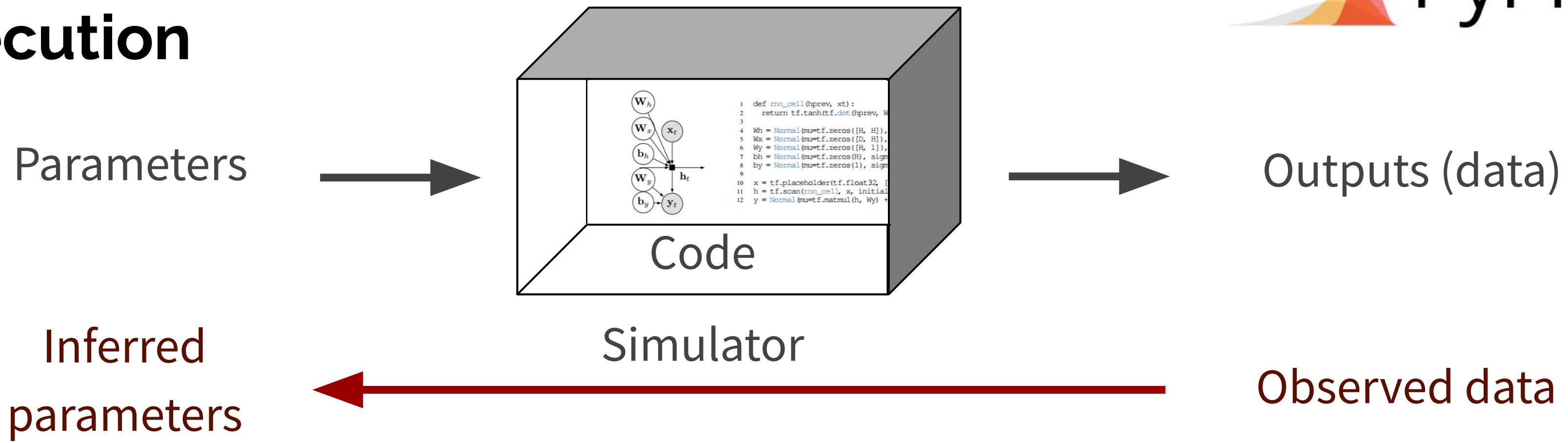
A new probabilistic programming system for existing simulators (in any language) based on PyTorch

Probabilistic execution

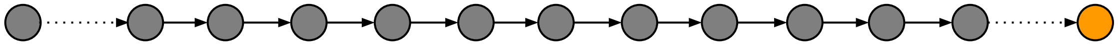
PyProb



Probabilistic execution

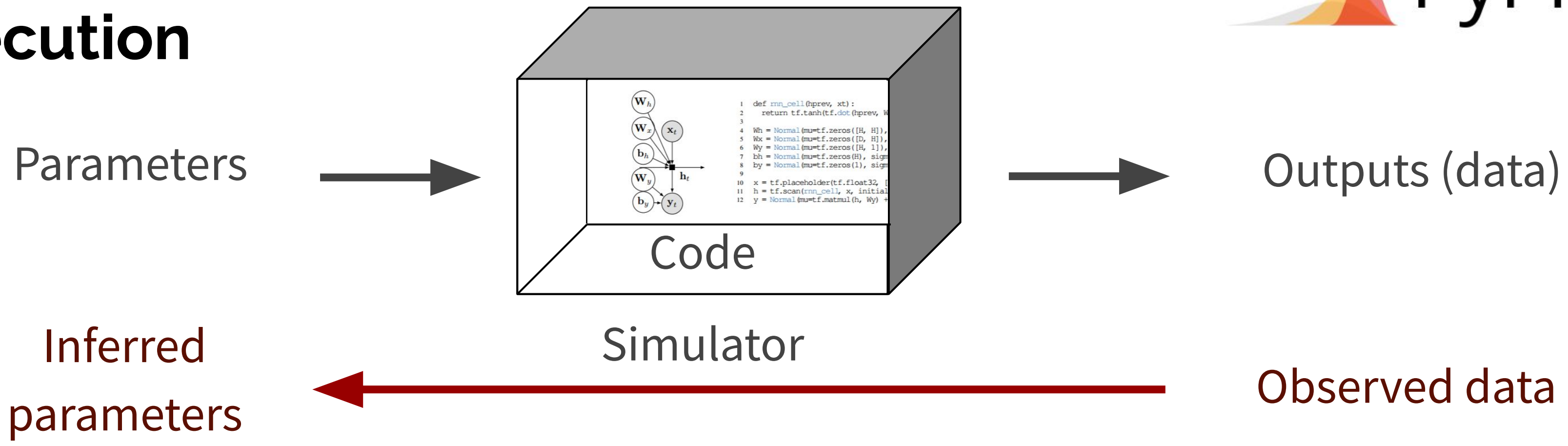


- **Run forward & catch all random choices** (“hijack” all calls to RNG)
- Record an **execution trace**: a record of all parameters, random choices, outputs

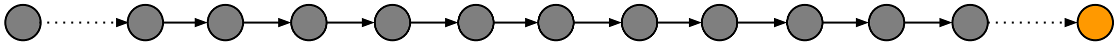


Probabilistic execution

PyProb



- **Run forward & catch all random choices** (“hijack” all calls to RNG)
- Record an **execution trace**: a record of all parameters, random choices, outputs



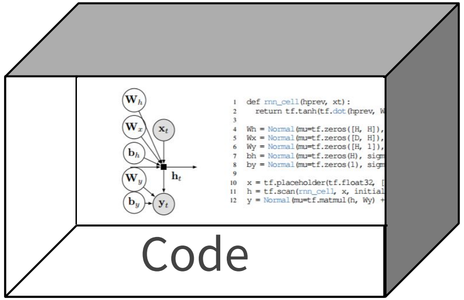
PPX

Probabilistic **P**rogramming e**X**ecution protocol
C++, C#, Dart, Go, Java, JavaScript, Lua, Python, Rust and others

Probabilistic execution

PyProb

Parameters



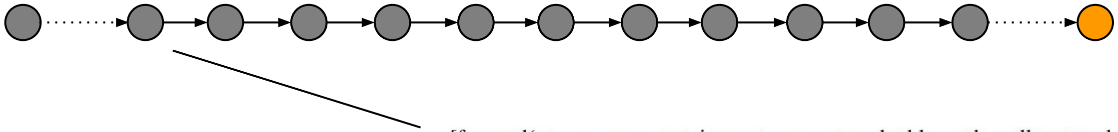
Outputs (data)

Simulator

Inferred parameters

Observed data

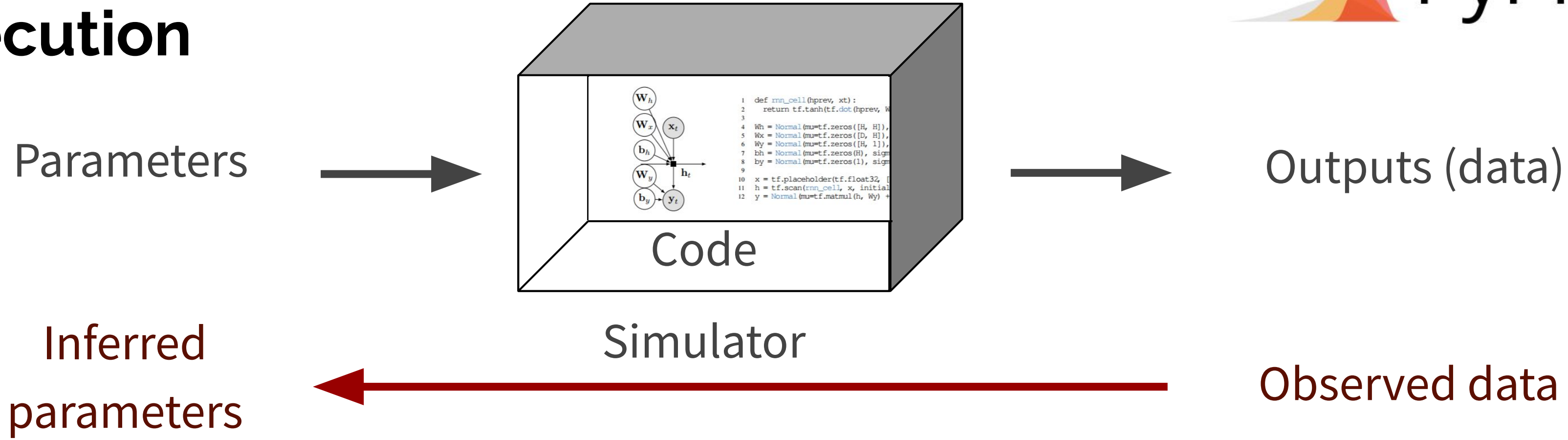
- **Run forward & catch all random choices** (“hijack” all calls to RNG)
- Record an **execution trace**: a record of all parameters, random choices, outputs



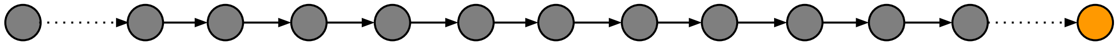
Uniquely label each choice at runtime by “addresses” of stack frames

[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

Probabilistic execution



- **Run forward & catch all random choices** (“hijack” all calls to RNG)
- Record an **execution trace**: a record of all parameters, random choices, outputs

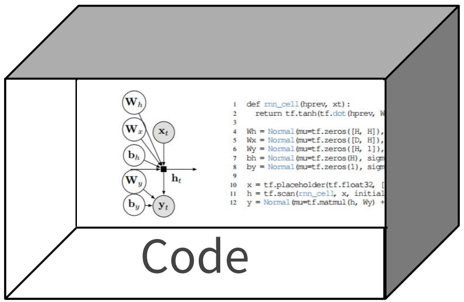


  - Conditioning: compare ***simulated output*** and ***observed data***
- **Approximate the distribution of parameters** that can produce (explain) observed data, using inference engines like Markov-chain Monte Carlo (MCMC)

Probabilistic execution

PyProb

Parameters



Outputs (data)

Simulator

Inferred parameters

Observed data

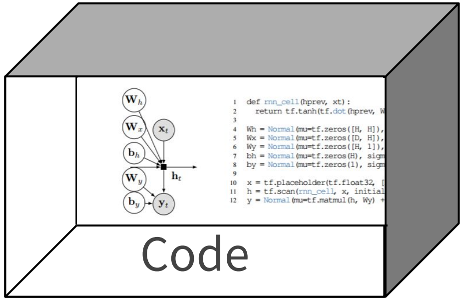
Simulators = giant probability models so inference is hard and computationally costly

- Need to run simulator up to millions of times
- Simulator execution and MCMC inference are sequential
- MCMC has “burn-in period” and autocorrelation

- **Ru**
- Re … outputs
- **Ap** … n)
ob … MCMC)

Probabilistic execution

PyProb

Parameters →  → Outputs (data)

Simulator

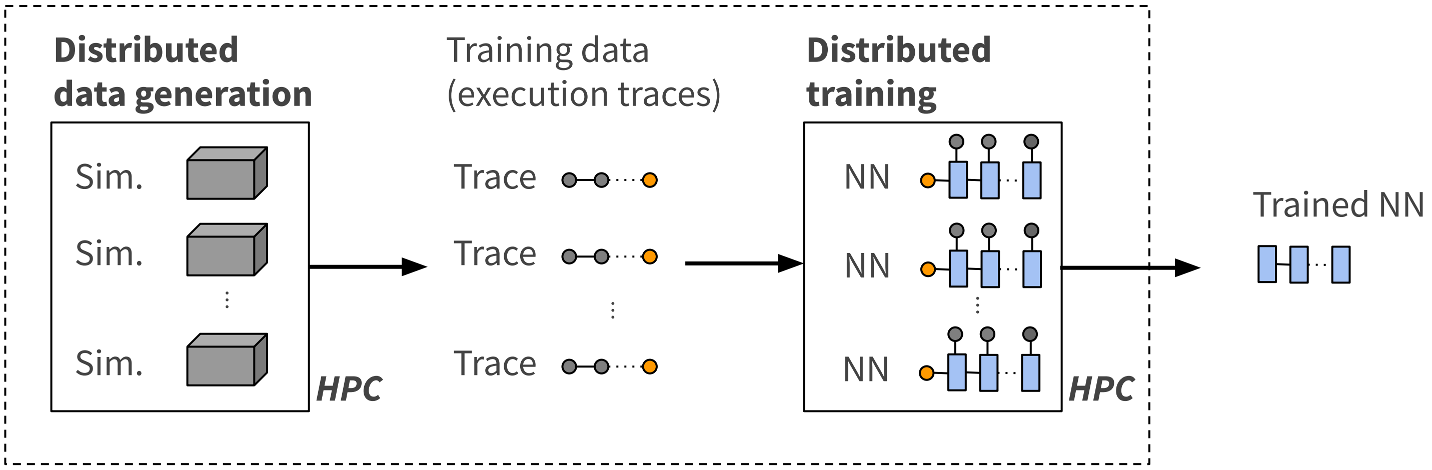
Inferred parameters ← Observed data

- Ru
- Re … outputs
  - ○
- Ap … n)
  ob … MCMC)

Simulators = giant probability models so inference is hard and computationally costly

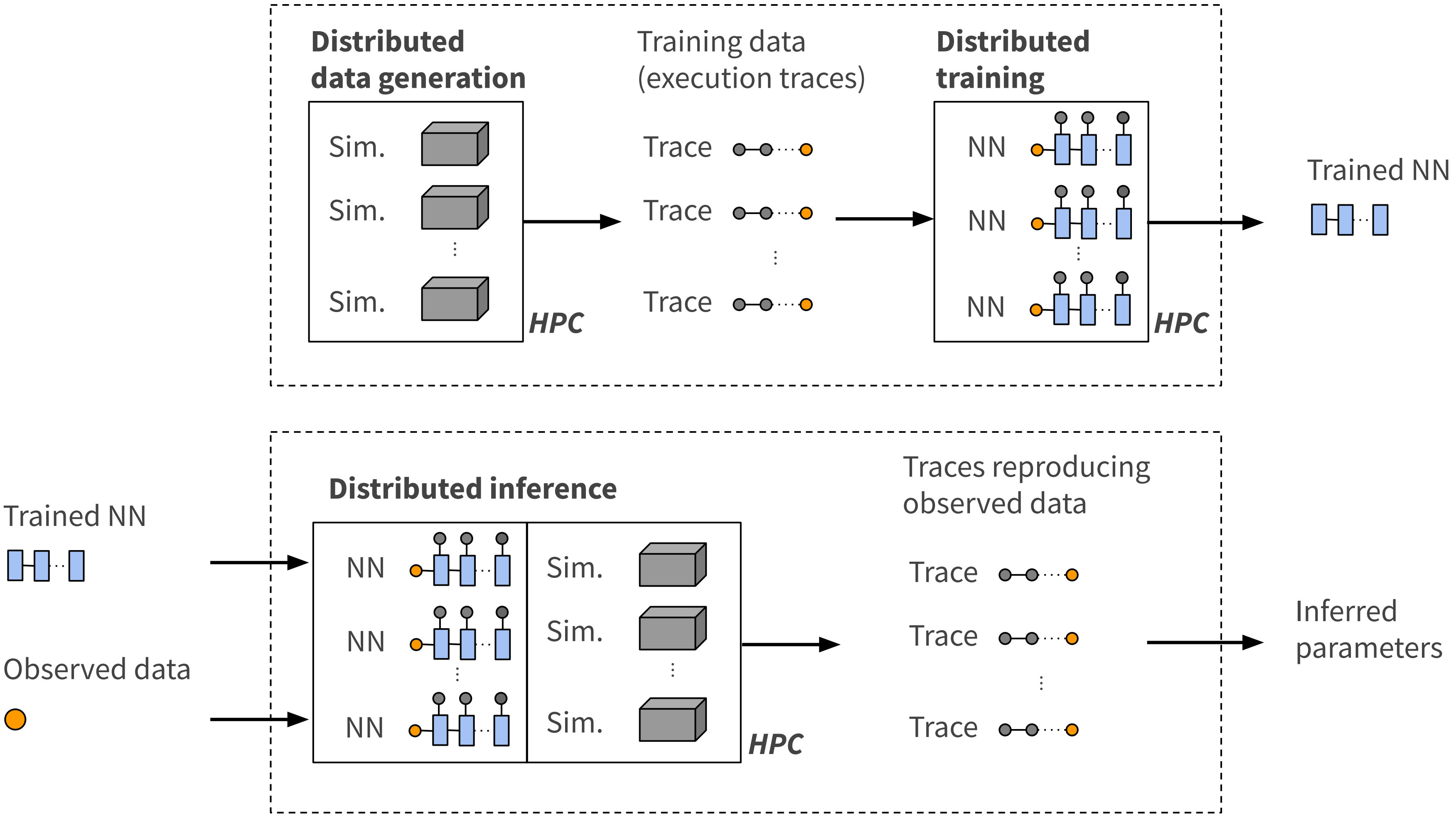
- Need to run simulator up to millions of times
- Simulator execution and MCMC inference are sequential
- MCMC has “burn-in period” and autocorrelation

But we can amortize the cost of inference using deep learning



Training (recording simulator behavior)

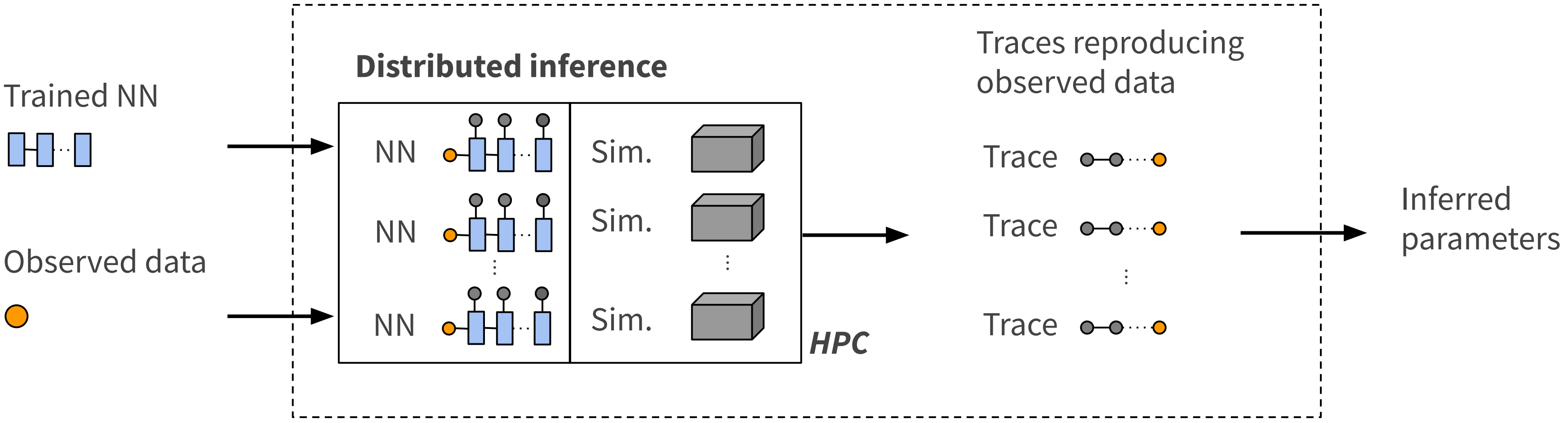
- Deep recurrent neural network learns all random choices in simulator
- Dynamic NN: grows with simulator complexity
  - Layers get created as we learn more of the simulator
  - 100s of millions of parameters in particle physics simulation
- Costly, but amortized: we need to train only once per given model



Distributed data generation
Sim.
Sim.
Sim.
HPC
Training data (execution traces)
Trace
Trace
Trace
Distributed training
NN
NN
NN
HPC
Trained NN
Trained NN
Observed data
Distributed inference
NN
NN
NN
Sim.
Sim.
Sim.
HPC
Traces reproducing observed data
Trace
Trace
Trace
Inferred parameters

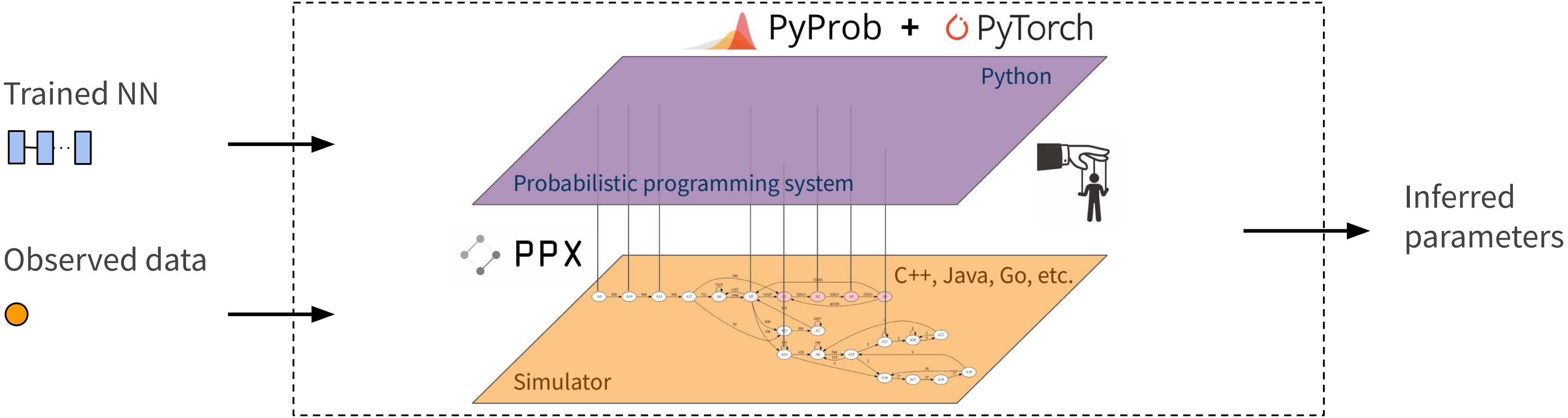
Inference (controlling simulator behavior)

- Trained deep NN makes intelligent choices given data observation
- Embarrassingly parallel distributed inference
- No “burn in period”
- No autocorrelation: every sample is independent



Inference (controlling simulator behavior)

- Trained deep NN makes intelligent choices given data observation
- Embarrassingly parallel distributed inference
- No “burn in period”
- No autocorrelation: every sample is independent



Probabilistic programming with simulators

PyProb https://github.com/pyprob/pyprob

- Probabilistic programming system for simulators and HPC, based on PyTorch
  Distributed training and inference, efficient support for multi-TB distribution files
  Optimized memory usage, parallel trace processing and combination

PPX https://github.com/pyprob/ppx

- *P*robabilistic *P*rogramming e*X*ecution protocol
  Simulator and inference/NN executed in separate processes and machines across network
  Using Google flatbuffers to support C++, C#, Dart, Go, Java, JavaScript, Lua, Python, Rust and others
  Probabilistic programming analogue to Open Neural Network Exchange (ONNX) for deep learning

Pyprob_cpp, RNG front end for C++ simulators https://github.com/pyprob/pyprob_cpp

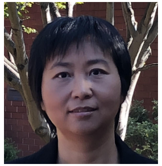
docker SHIFTER Containerized workflow
Develop locally, deploy to HPC on many nodes for experiments

etalumis → | ← simulate

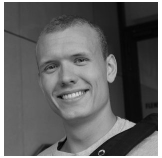

Atılım Güneş Baydin

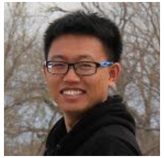

Lukas Heinrich


Wahid Bhimji


Lei Shao

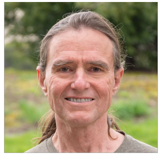

Saeid Naderiparizi


Andreas Munk

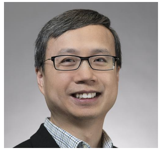

Jialin Liu


Bradley Gram-Hansen


Gilles Louppe


Lawrence Meadows

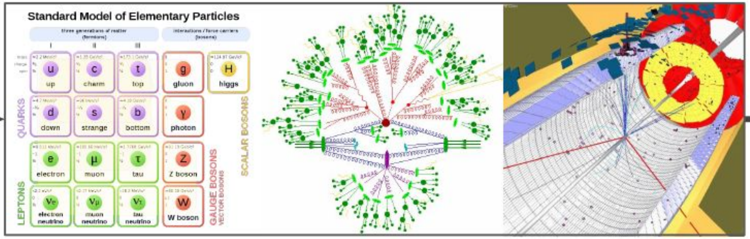

Phil Torr


Victor Lee


Prabhat

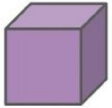

Kyle Cranmer


Frank Wood



Inputs

$\mathbf{y}$



Simulated data (detector response)

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

Likelihood Prior

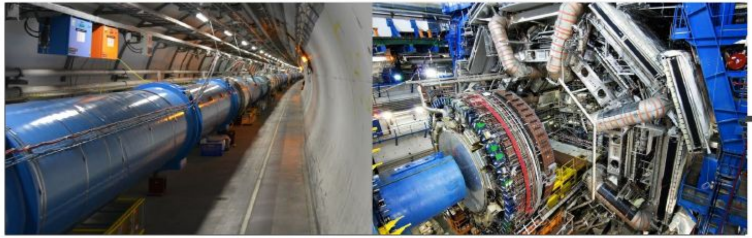
Inputs

Posterior $p(\mathbf{x}|\mathbf{y})$

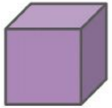
$\mathrm{observe}(p(\mathbf{y}|\mathbf{x}), \mathbf{y}_{\mathrm{obs}})$

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

Real world system (e.g., Large Hadron Collider)



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



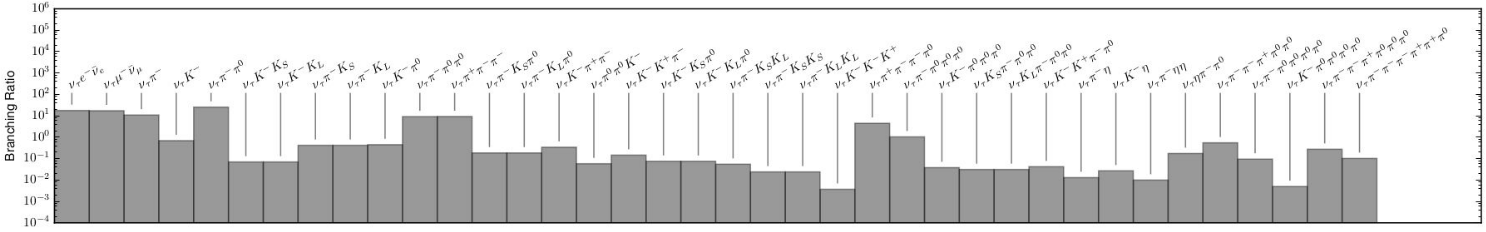
Observed data (detector response)

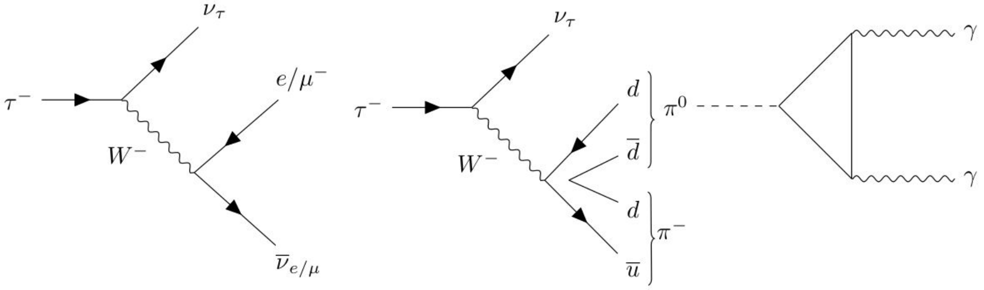
Baydin, Heinrich, Bhimji, Shao, Naderiparizi, Munk, Liu, Gram-Hansen, Louppe, Meadows, Torr, Lee, Prabhat, Cranmer, Wood.
NeurIPS 2019

Tau lepton decay

We study tau lepton decay using the state-of-the-art Sherpa simulator (C++)

Coupled to a fast approximate calorimeter simulation in C++





Baydin, Heinrich, Bhimji, Shao, Naderiparizi, Munk, Liu, Gram-Hansen, Louppe, Meadows, Torr, Lee, Prabhat, Cranmer, Wood.
NeurIPS 2019

Latent variables in Sherpa

We found Sherpa to contain **at least 25k addresses (latent variables)**

Note: the **simulator defines an unlimited number of latents** due to Turing-complete host language (C++) and presence of many sampling loops

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
...	

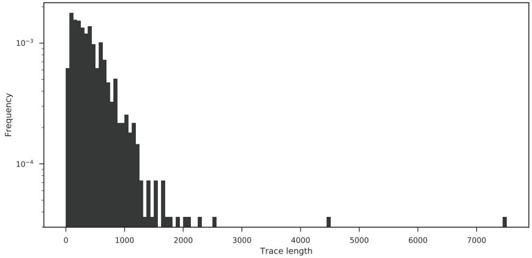
Common trace types in Sherpa

Approximately 450 trace types encountered

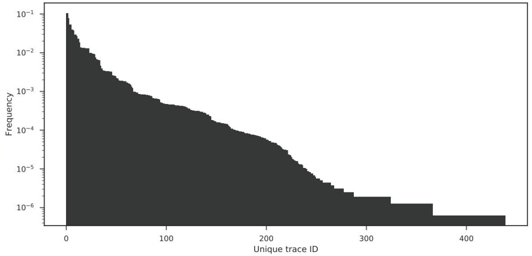
Trace type: unique sequencing of addresses (with different sampled values)

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

...



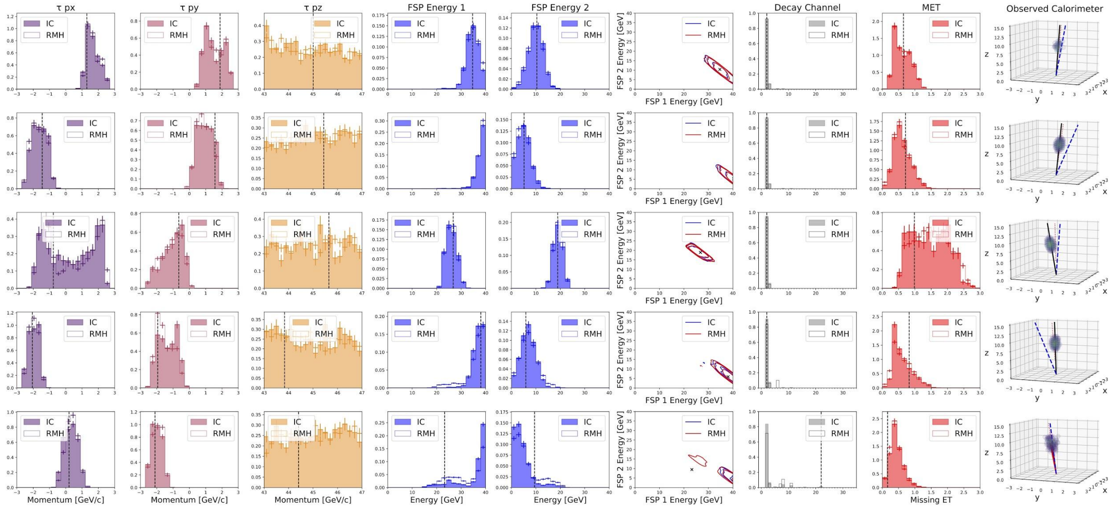
(a) Distribution of trace lengths (all addresses). Min: 13, max: 7,514, mean: 383.58.



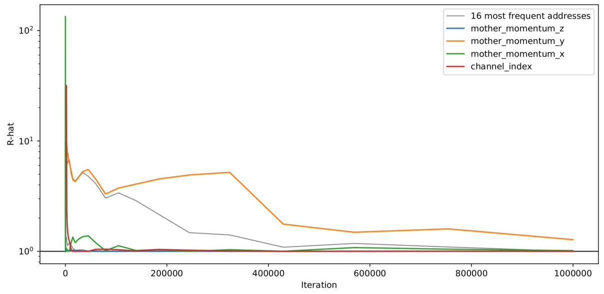
(c) Distribution of trace types, sorted in decreasing frequency.

Inference results

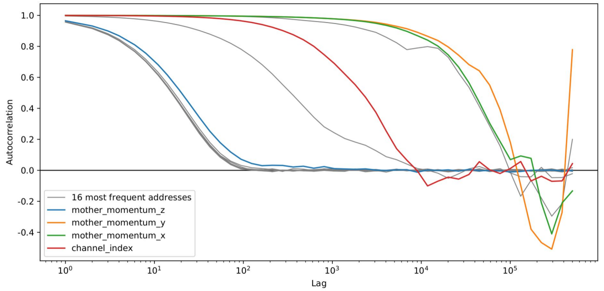
- Achieved MCMC (RMH) “ground truth”
- **First tractable Bayesian inference for LHC physics**
  - Posterior over full latent space (>25k latent variables)
  - Autocorrelation typically around 10^5
- Amortized inference (IC) closely matches MCMC (RMH)
  - No autocorrelation, embarrassingly parallel
  - MCMC: 115 hours, IC: 30 minutes



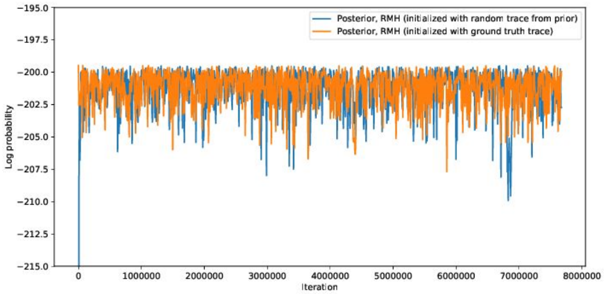
Gelman-Rubin convergence diagnostic



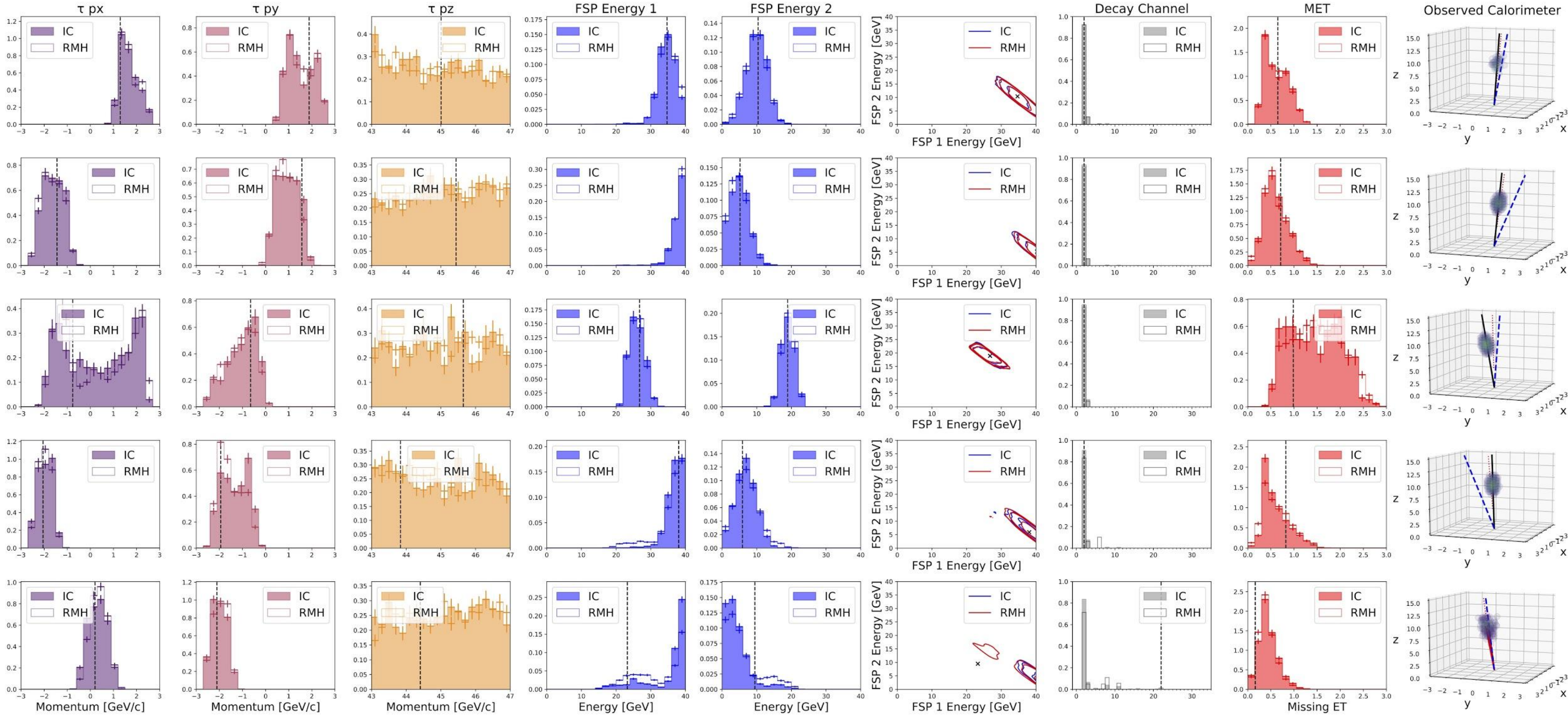
Autocorrelation

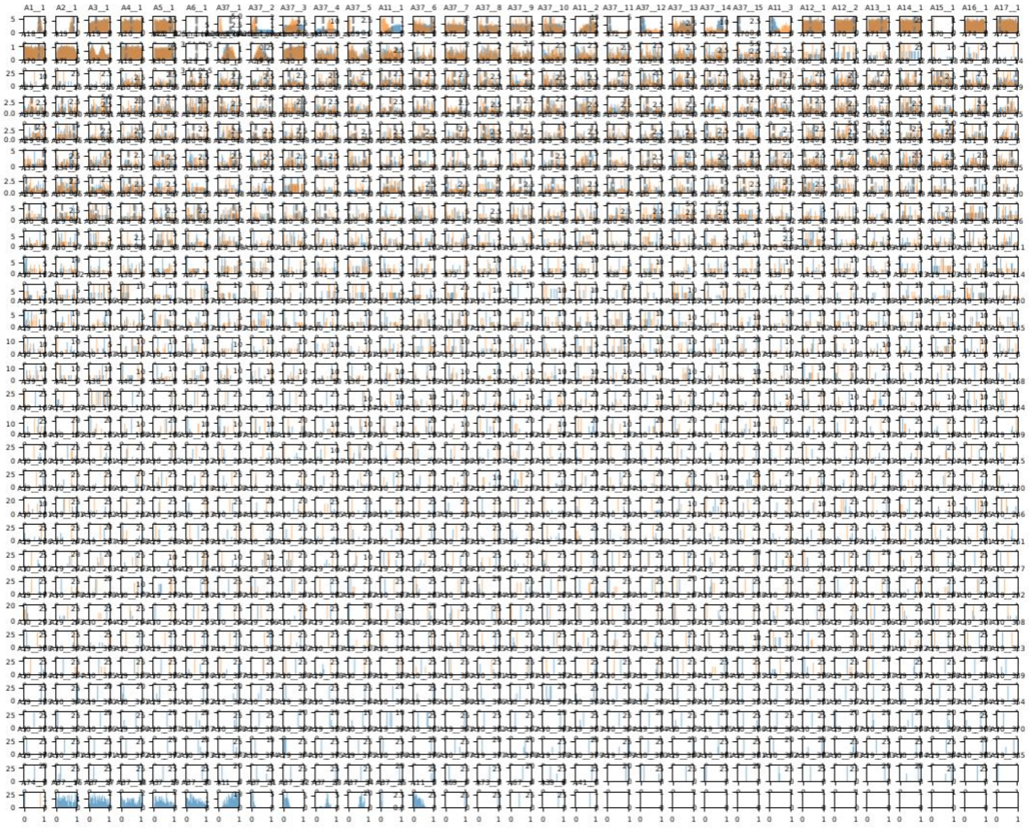


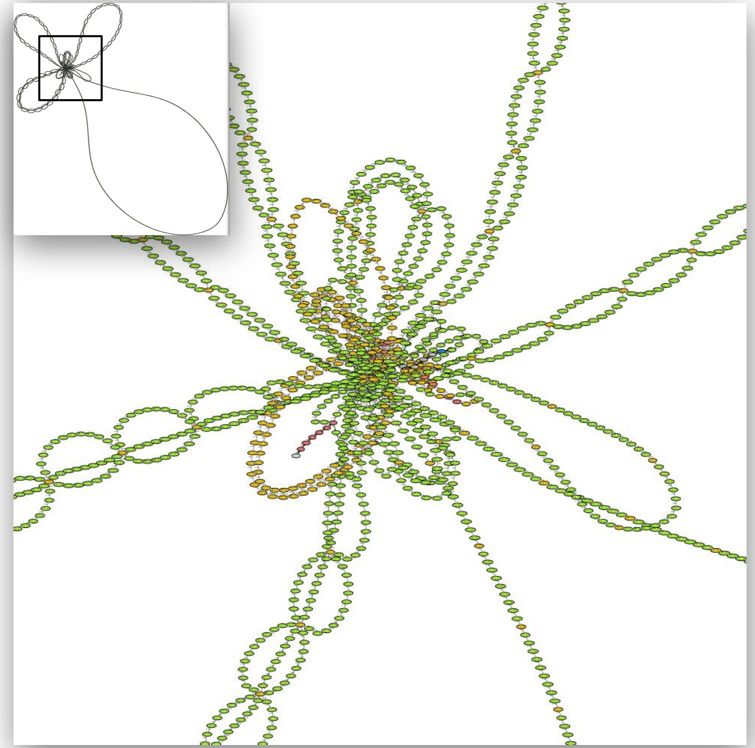
Trace log-probability



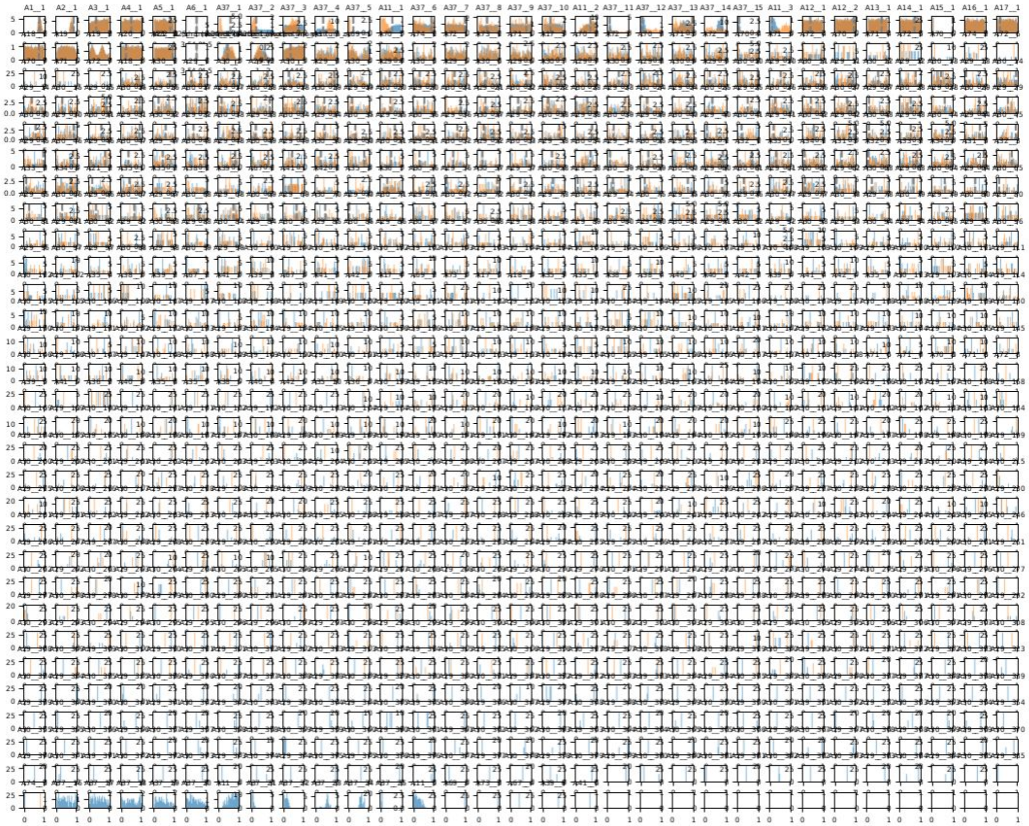
More physics events







Etalumis gives access to all latent variables: allows answering *any* model-based question



Etalumis gives access to all latent variab
any model-based question

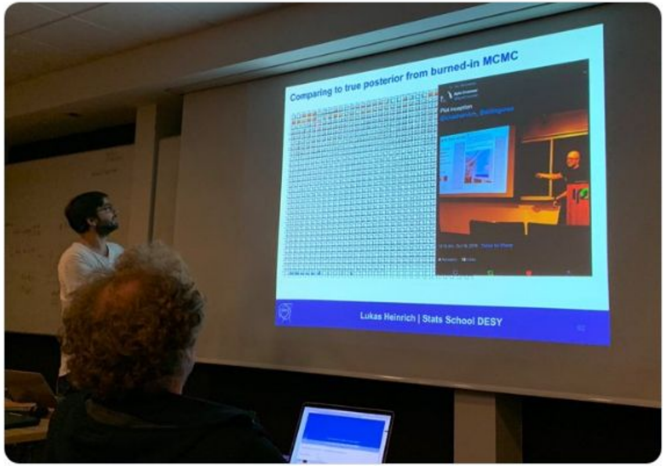
Nathan Simpson @ CERN
@phi_nate

The plot-ception saga continues!!!

Many congratulations to @lukasheinrich_ for reclaiming his title of most plots in a single slide here at the first @INSIGHTS_EU advanced statistics school held at @desy.

How will the competition respond? ;)

cc @atilimgunes @KyleCranmer

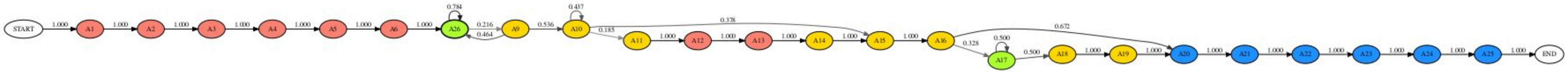


4:34 PM · Oct 29, 2019 · Twitter Web App

3 Retweets **20** Likes

Interpretability

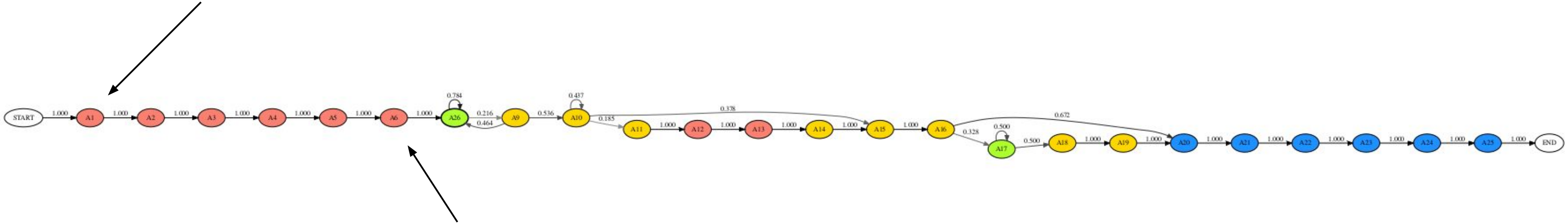
Latent probabilistic structure of **10** most frequent trace types



Interpretability

Latent probabilistic structure of **10** most frequent trace types

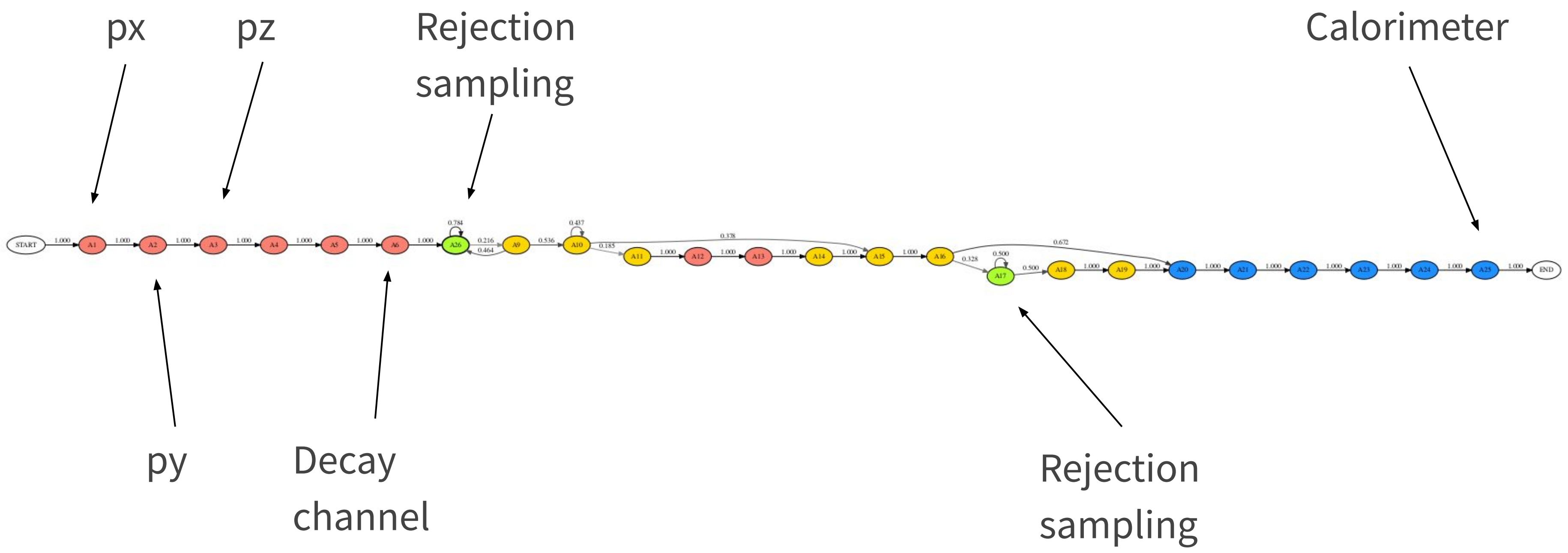
[forward(xt:: xarray_container<xt:: uvector<double, std:: allocator<double> >, (xt:: layout_type)1, xt:: svector<unsigned long, 4ul, std:: allocator<unsigned long>, true>, xt:: xtensor_expression_tag>)+0x5f; SherpaGenerator:: Generate()+0x36; SHERPA:: Sherpa:: GenerateOneEvent(bool)+0x2fa; SHERPA:: Event_Handler::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

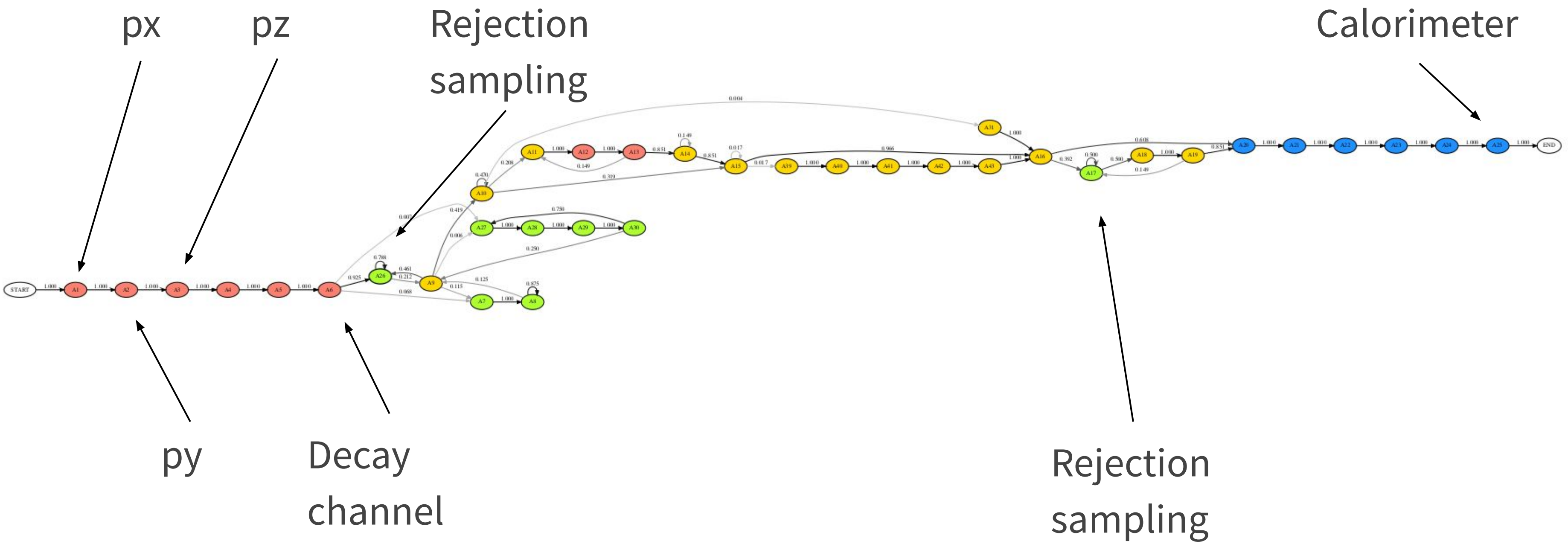
Interpretability

Latent probabilistic structure of **10** most frequent trace types



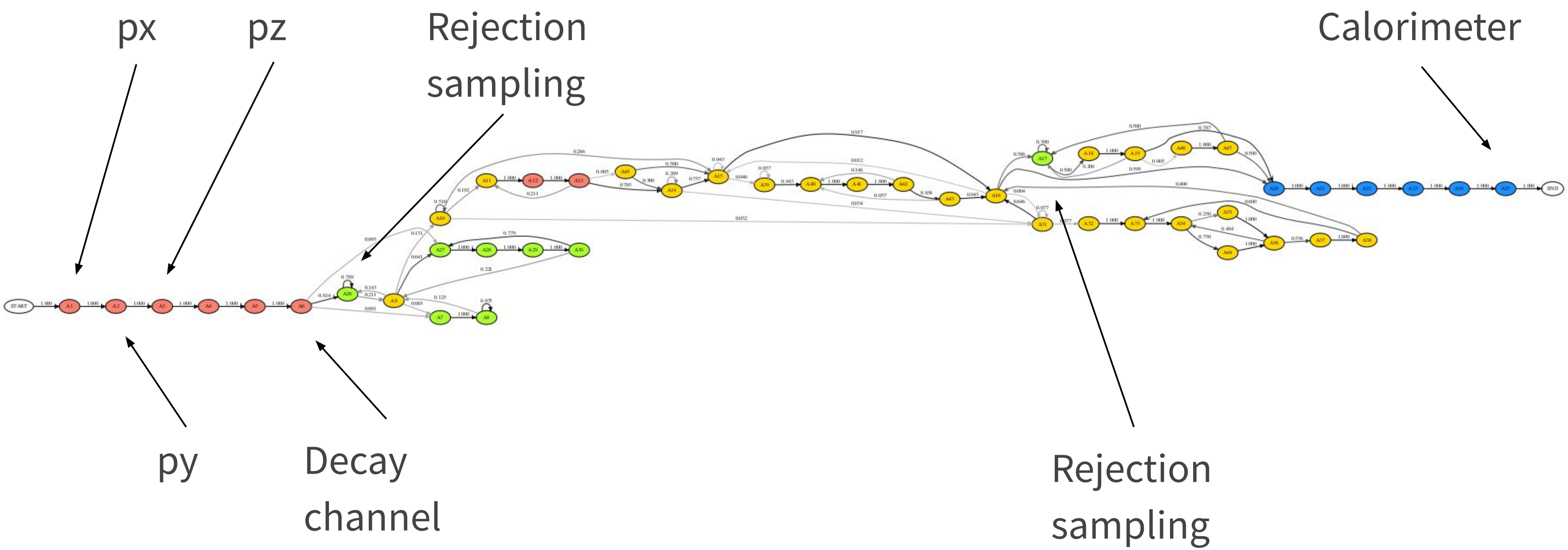
Interpretability

Latent probabilistic structure of **25** most frequent trace types



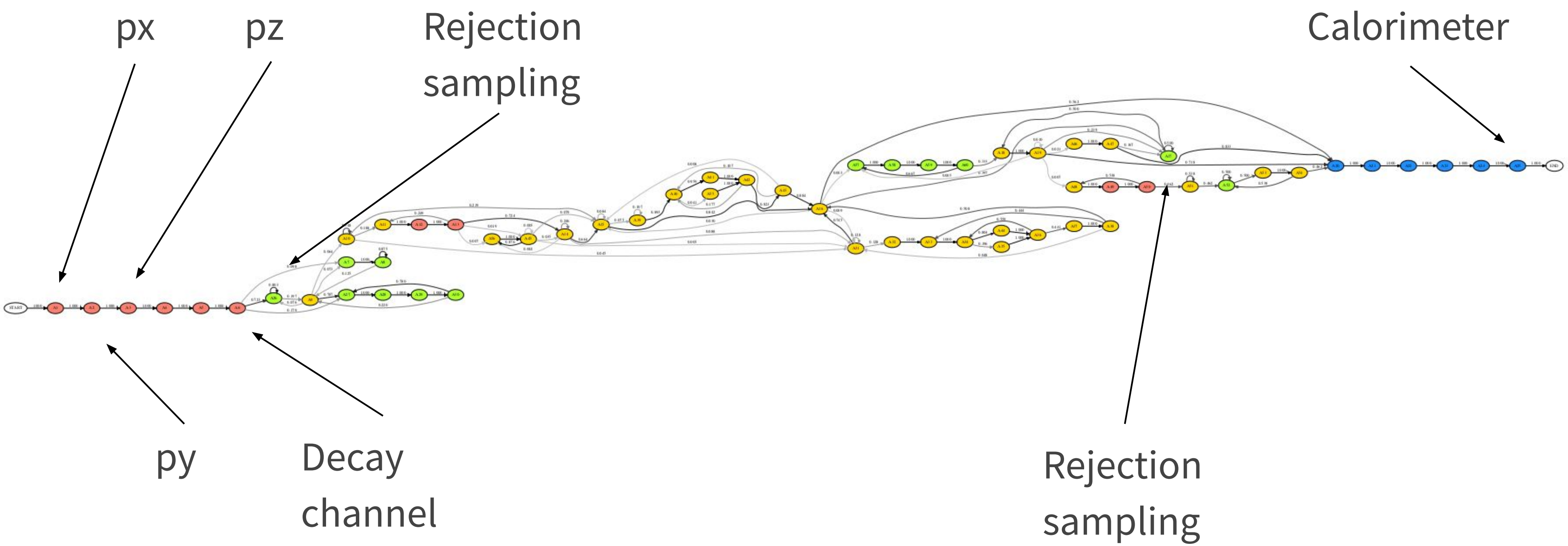
Interpretability

Latent probabilistic structure of **100** most frequent trace types



Interpretability

Latent probabilistic structure of **250** most frequent trace types



What’s next?

Current and upcoming work

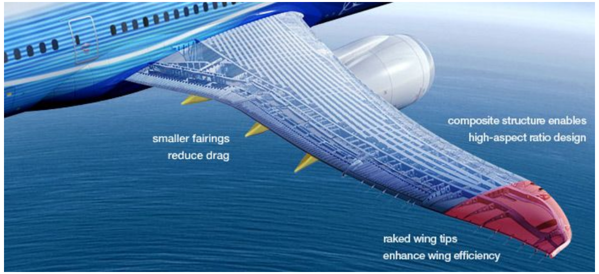
- Autodiff through PPX protocol
- **Learning simulator surrogates** (approximate forward simulators)
- **Rejection sampling loops** (weighting schemes)
- Rare event simulation for compilation ("prior inflation")
- Batching of open-ended traces for NN training
- Distributed training of dynamic networks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

Probabilistic programming is
for the first time practical
for large-scale real-world science models

This is just the beginning ...

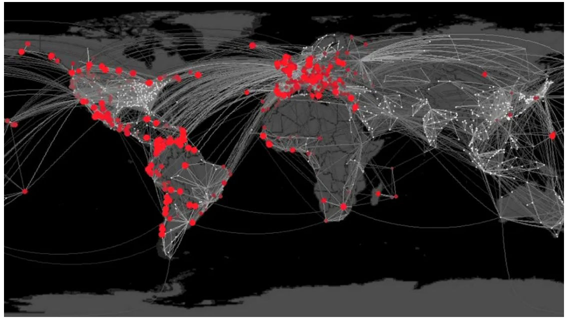
Spacecraft collision prevention

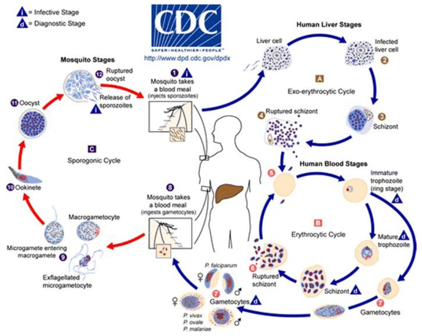
Collaboration with ESA



Simulation of composite materials

Munk et al. 2019. arXiv:1910.11950

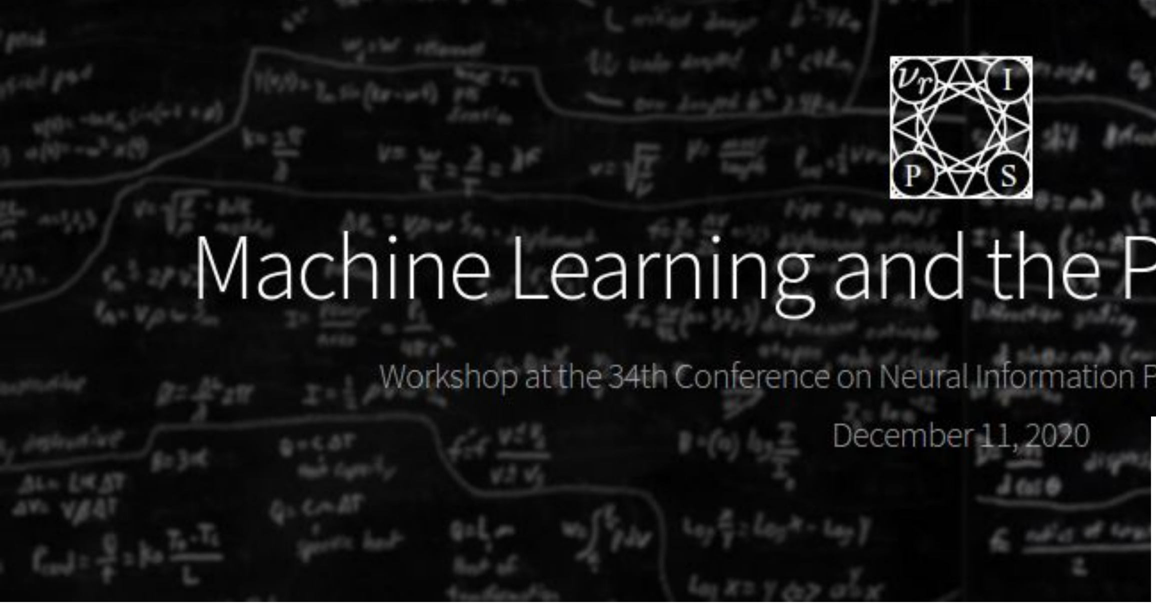




Simulation-based inference in health

Schroeder de Witt et al. 2020. arXiv:2005.07062

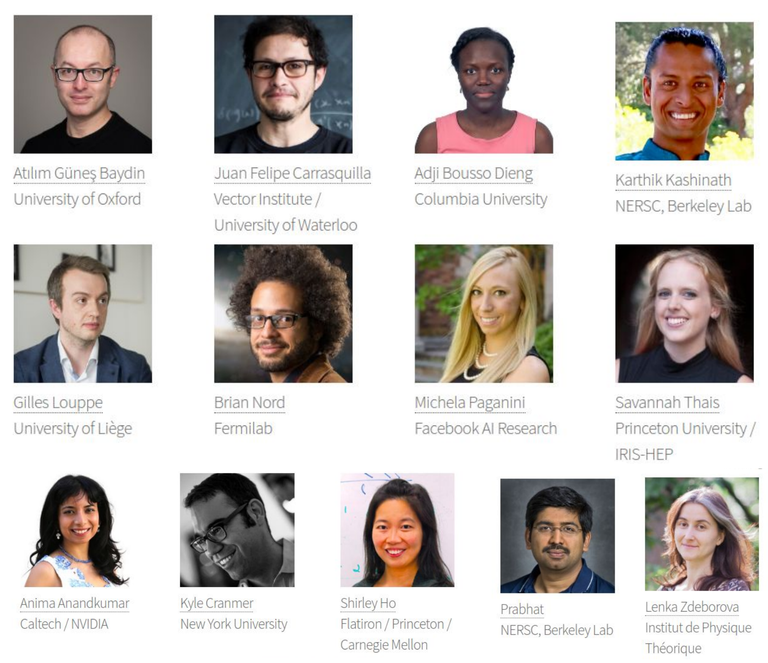
Gram-Hansen et al. 2019. arXiv:1905.12432



Neural Information Processing Systems (NeurIPS) workshop

Expecting your papers at the intersection of machine learning and physical sciences!

Paper deadline: 2 Oct 2020; workshop: 11 Dec 2020
https://ml4physicalsciences.github.io/



Thank you for listening

Stanford SLAC AI Seminar
25 Sep 2020

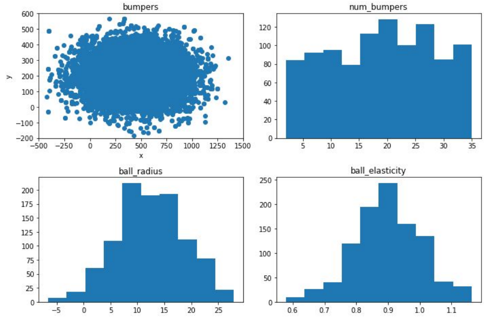
UNIVERSITY OF
OXFORD

Live demo

Jupyter notebook:

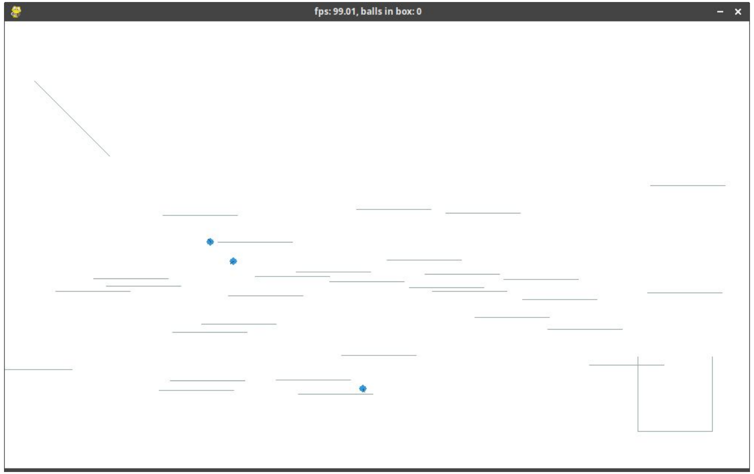
https://github.com/gbaydin/mlhep2020/blob/master/notebooks/probprog-physics-example.ipynb

```
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
```




```python
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')
```

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



References

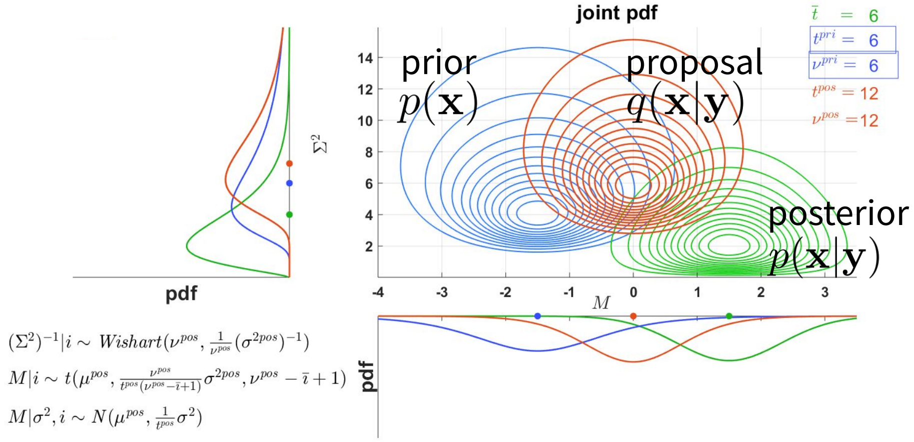
Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, Frank Wood. 2019. *"Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard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

Extra slides

Inference engines

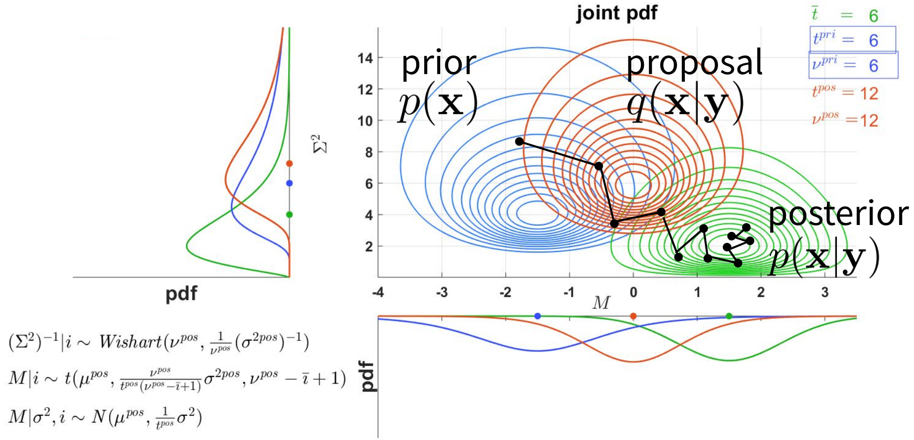
- Markov chain Monte Carlo
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- Importance sampling
  - Propose from prior $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ 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

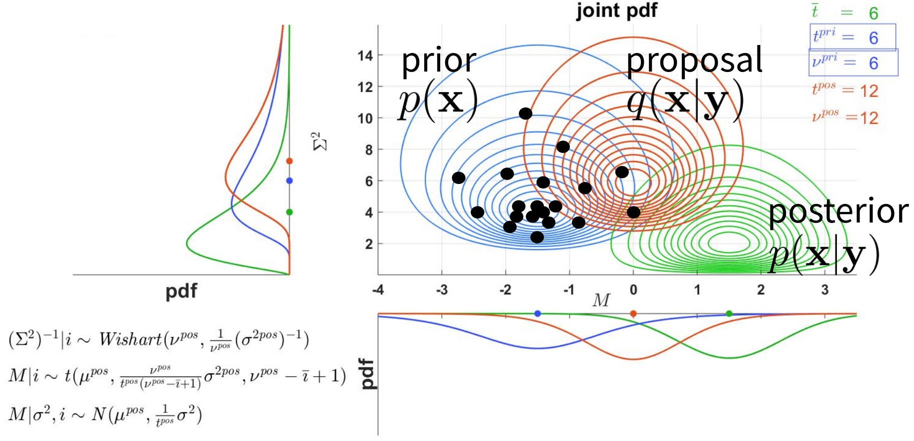
- **Markov chain Monte Carlo**
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- Importance sampling
  - Propose from prior $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ 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

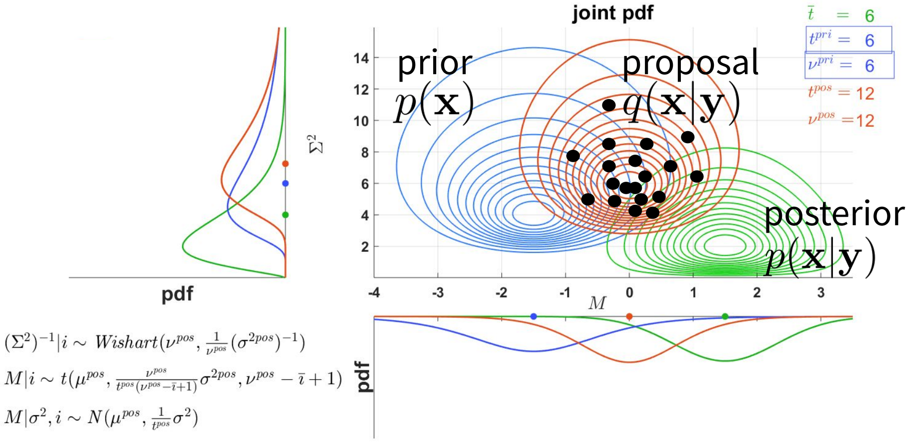
- Markov chain Monte Carlo
  - Probprog-specific:
    - Lightweight Metropolis–Hastings
    - Random-walk Metropolis–Hastings
  - Sequential
  - Autocorrelation in samples
  - “Burn in” period
- **Importance sampling**
  - **Propose from prior** $p(\mathbf{x})$
  - Use learned proposal $q(\mathbf{x}|\mathbf{y})$ 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

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