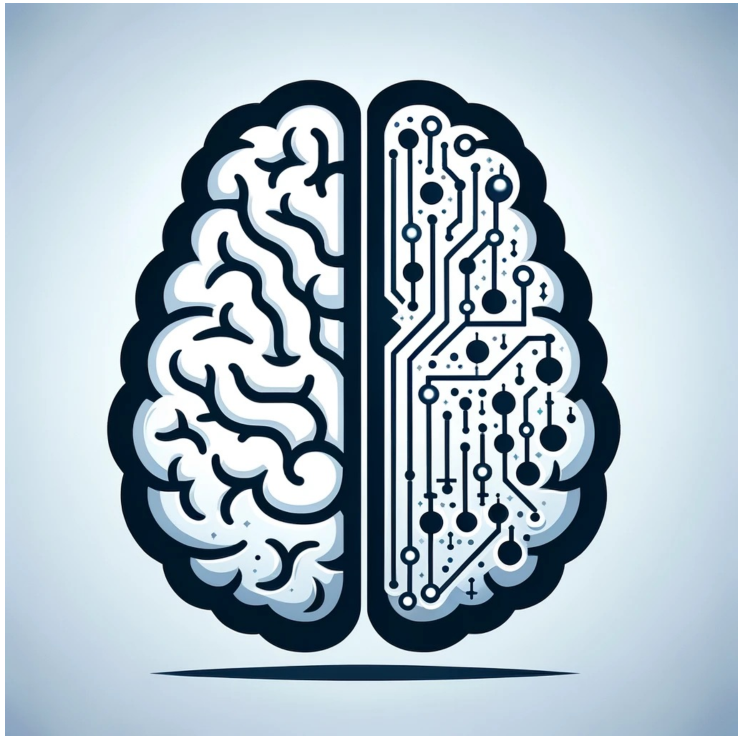
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)



Lecture 8: Intro to Gen, importance sampling

Tianmin Shu

Common questions about Pset 1

- Relevant lecture notes: Lecture 4
- Hypothesis averaging for prediction

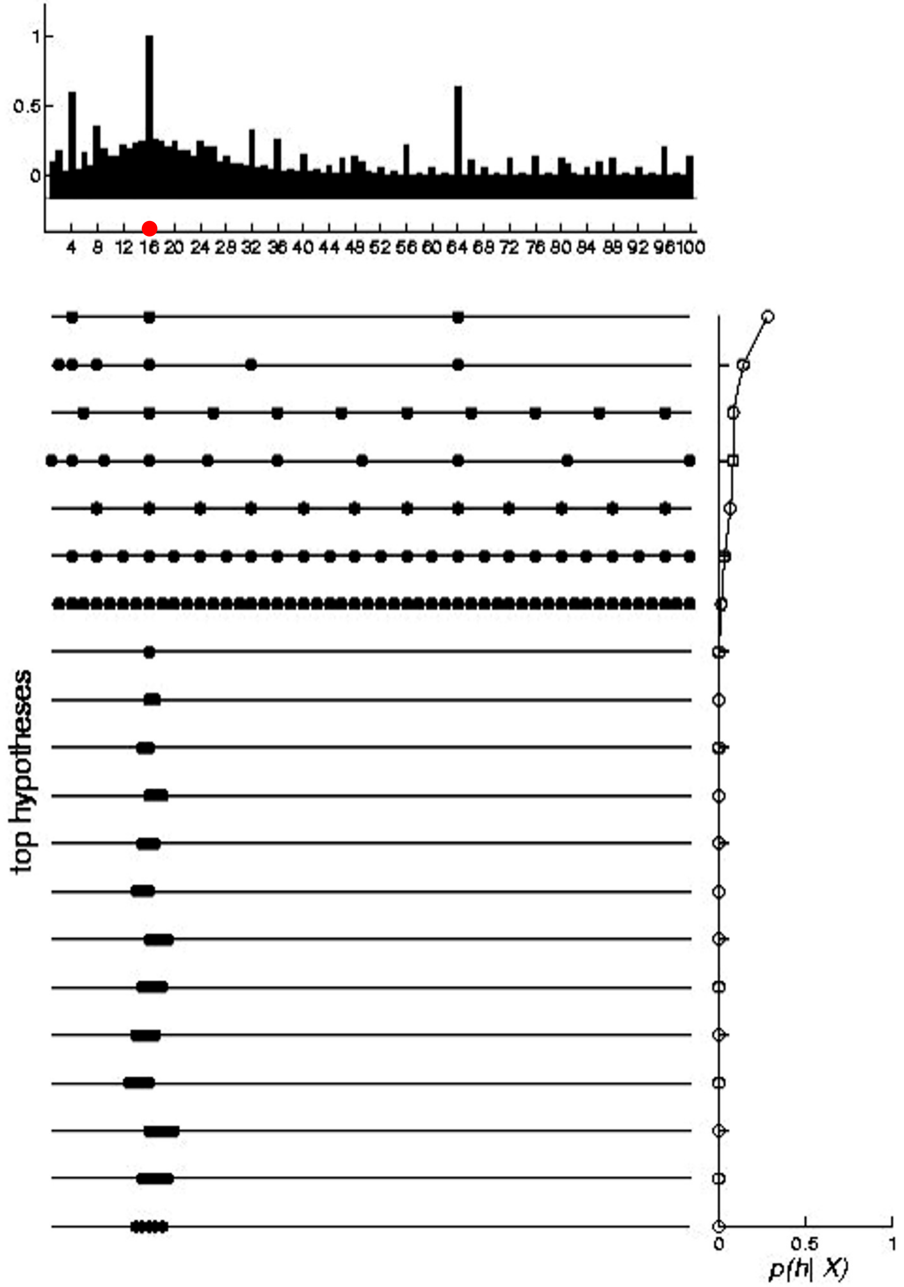
Hypothesis averaging:

Compute the probability that *C* applies to some new object *y* (i.e., *y* is a yes number) by averaging the predictions of all hypotheses *h*, weighted by *p*(*h*|*X*):

$$p(y \in C \mid X) = \sum_{h \in H} \underbrace{p(y \in C \mid h)}_{= \begin{cases} 1 \text{ if } y \in h \\ 0 \text{ if } y \notin h \end{cases}} \; p(h \mid X)$$

Step 1:
Concept inference
$p(h \mid X)$

Examples:
16

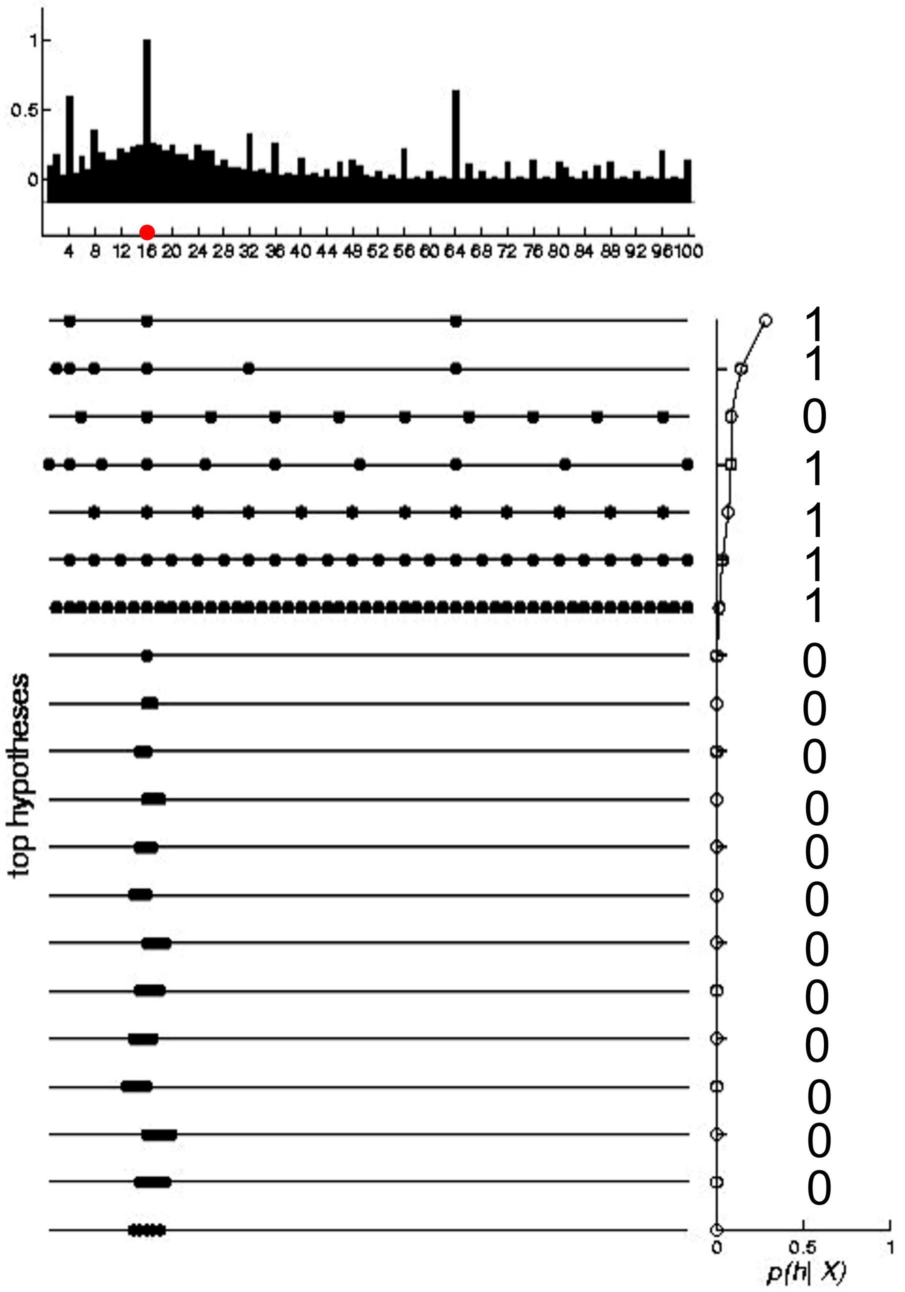


Step 2: Prediction

For a new number *y*:
64

$$p(y \in C \mid X) = \sum_{h \in H} \underbrace{p(y \in C \mid h)}_{= \begin{cases} 1 \text{ if } y \in h \\ 0 \text{ if } y \notin h \end{cases}} \; p(h \mid X)$$

Weighted average of 0s and 1s

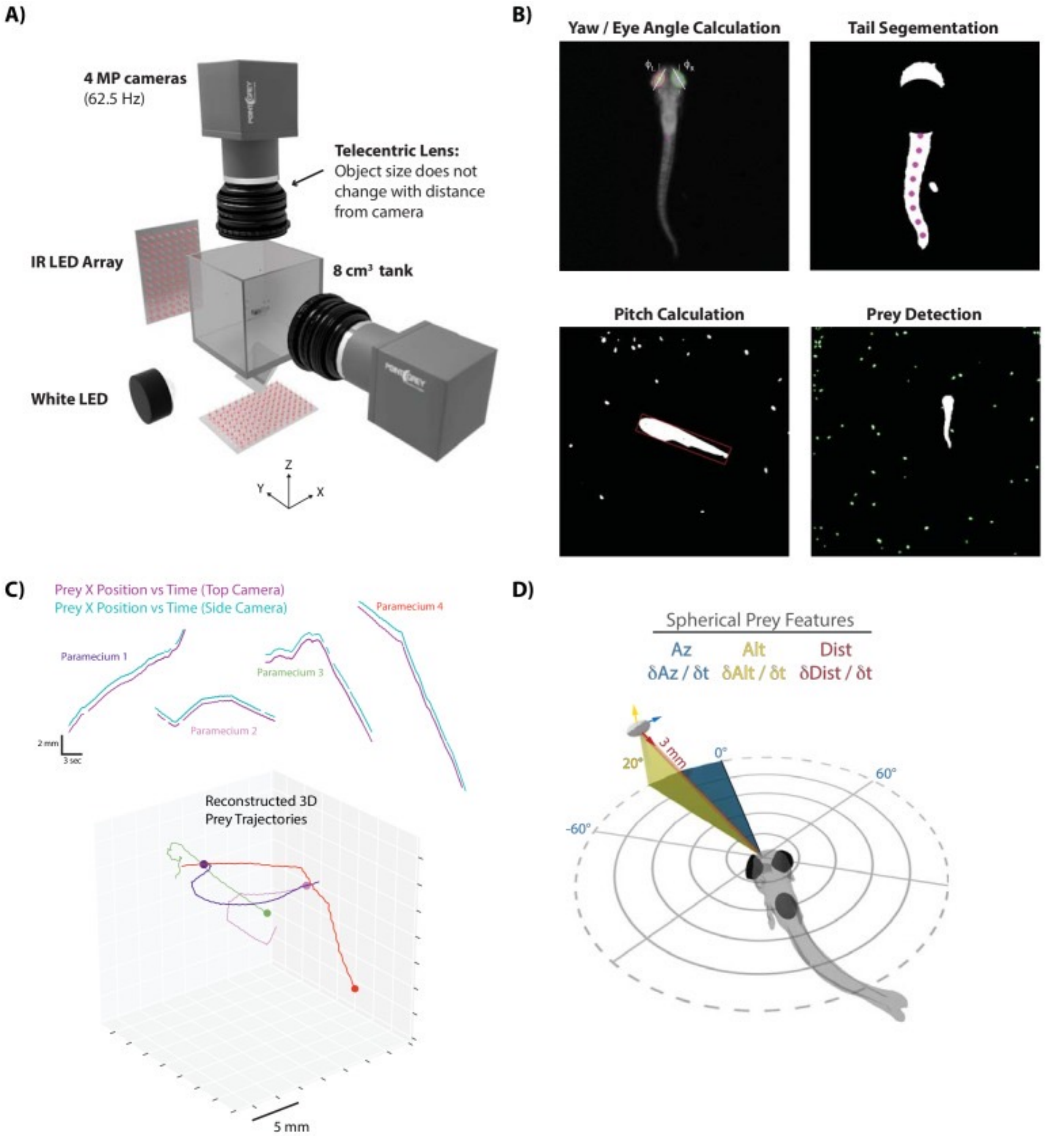


Introduction to Gen

- Gen vs Conventional PPLs
- “Hello world” in Gen: write a simple generative program
- Trace, weights
- Importance sampling & importance resampling

Probabilistic programming in Science

Experimental Neuroscience

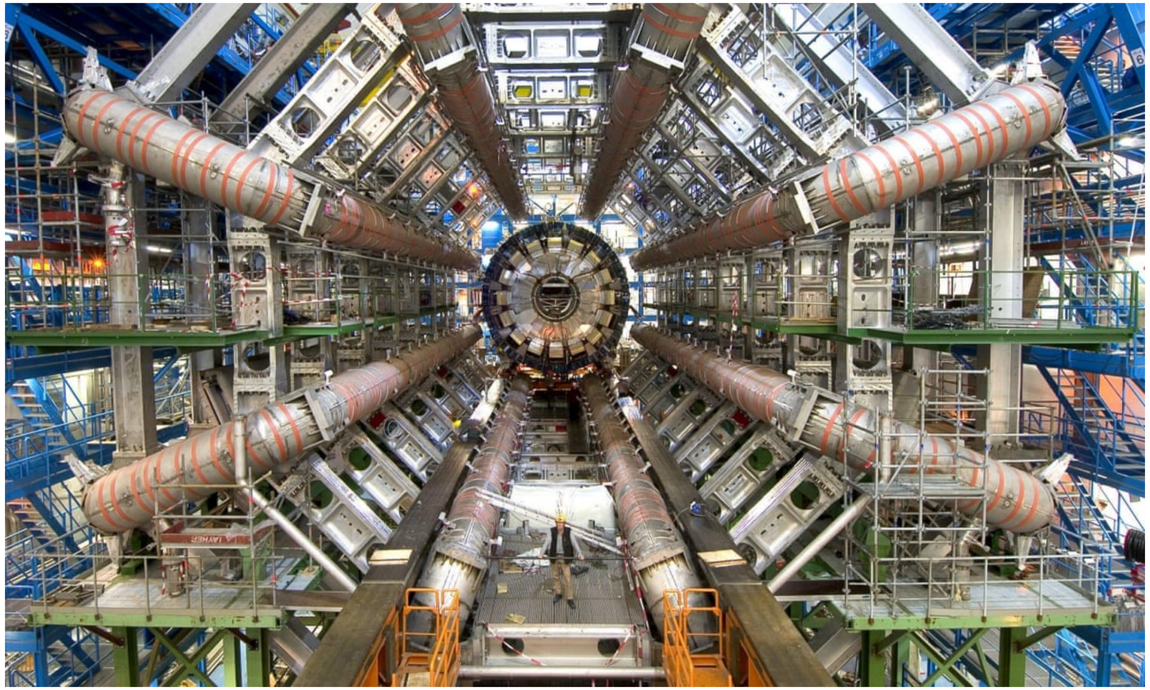


Bolton et al. (2019) using **BayesDB**

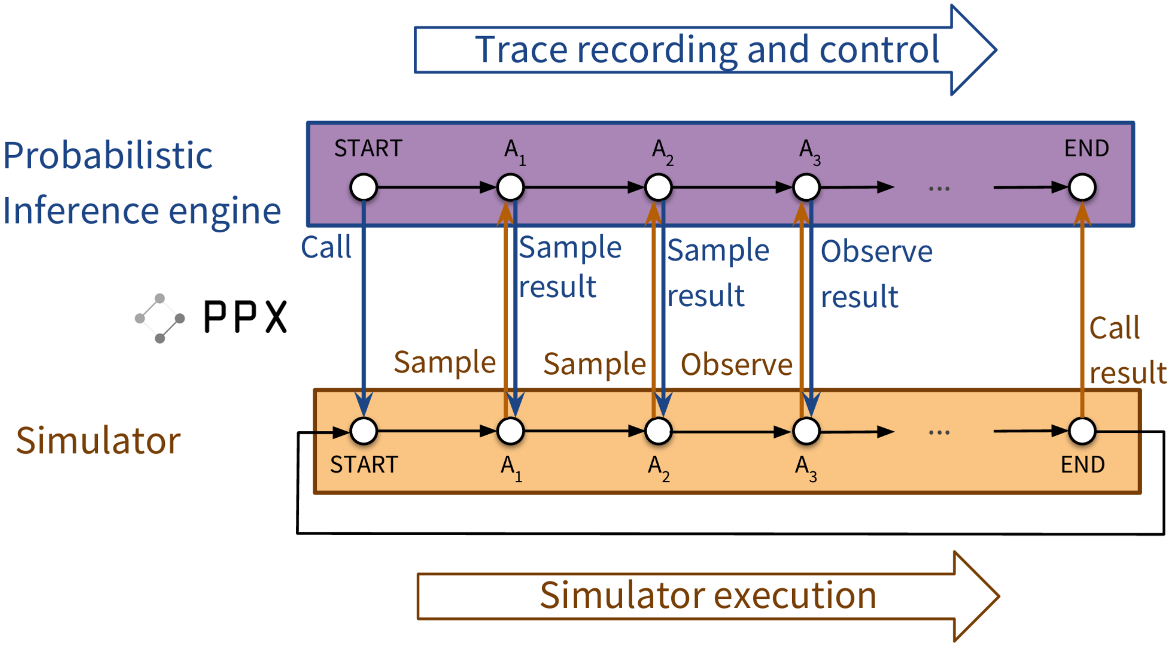
Probabilistic programming in Science

Particle Physics

Determine the properties of particles at the Large Hadron Collider (LHC) at CERN



Bolton et al. (2019) using **pyprob (PyTorch-based PPL) + Sherpa (C++ Simulator)**



Probabilistic programming in Science

Epidemiology

Estimating the number of infections and the impact of non-pharmaceutical interventions on COVID-19 in European countries: technical description update

Seth Flaxman[*], Swapnil Mishra[*], Axel Gandy[*], H Juliette T Unwin, Helen Coupland, Thomas A Mellan, Harrison Zhu, Tresnia Berah, Jeffrey W Eaton, Pablo N P Guzman, Nora Schmit, Lucia Callizo, Imperial College COVID-19 Response Team, Charles Whittaker, Peter Winskill, Xiaoyue Xi, Azra Ghani, Christl A. Donnelly, Steven Riley, Lucy C Okell, Michaela A C Vollmer, Neil M. Ferguson and Samir Bhatt[*,1]

Using Stan PPL

Two types of PPLs

- Write your generative programs and let a black box engine run the inference
- Programmable generative models and programmable inference

Stan
infer.net
BLOG
PSI SOLVER
Church
Anglican
ΣΠPL

Automatic Inference

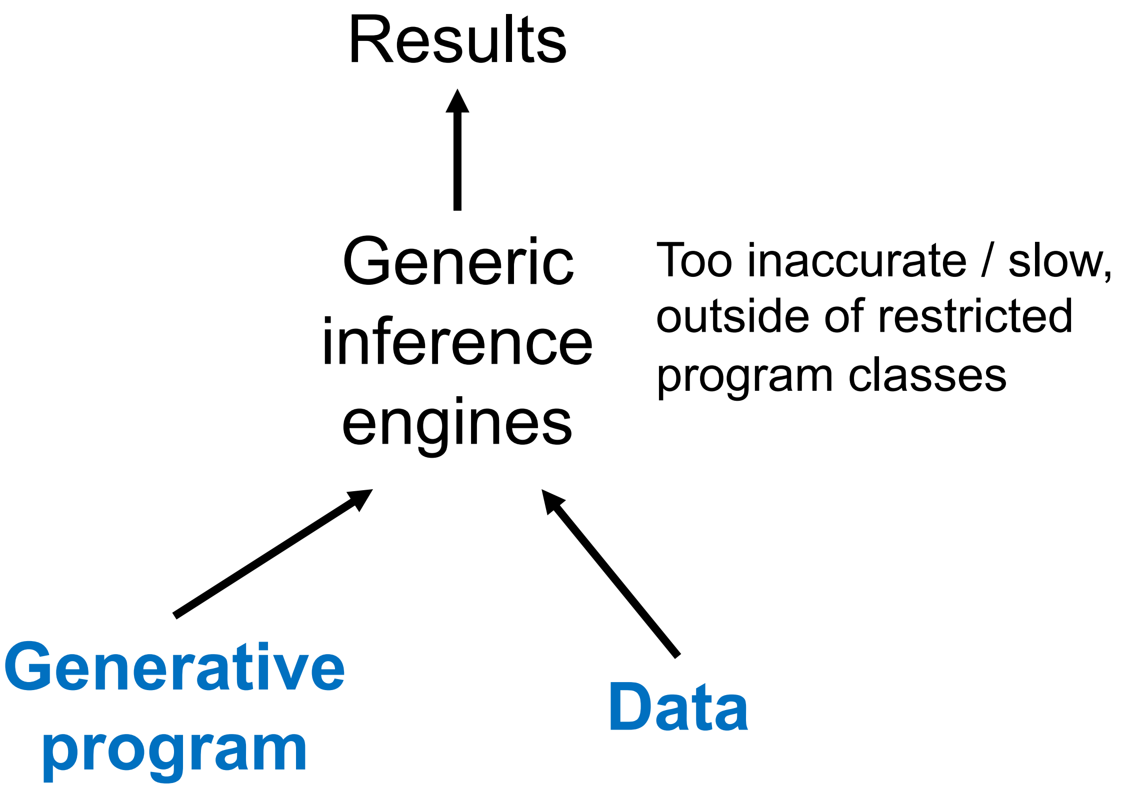
Gen
PYRO
Venturecxx
Turing.jl

Programmable Inference

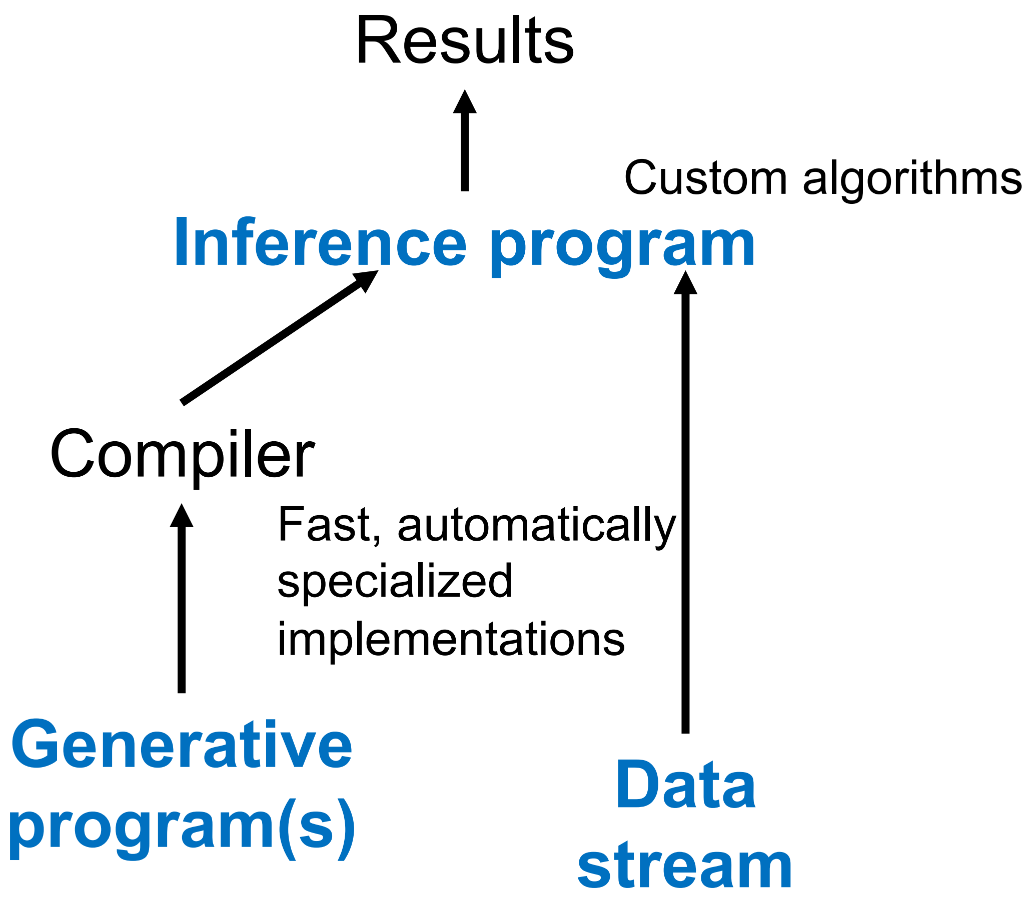
Two types of PPLs

(**BLUE**: specified by users)

Traditional PPLs



Gen



Competitive performance compared against restricted PPLs

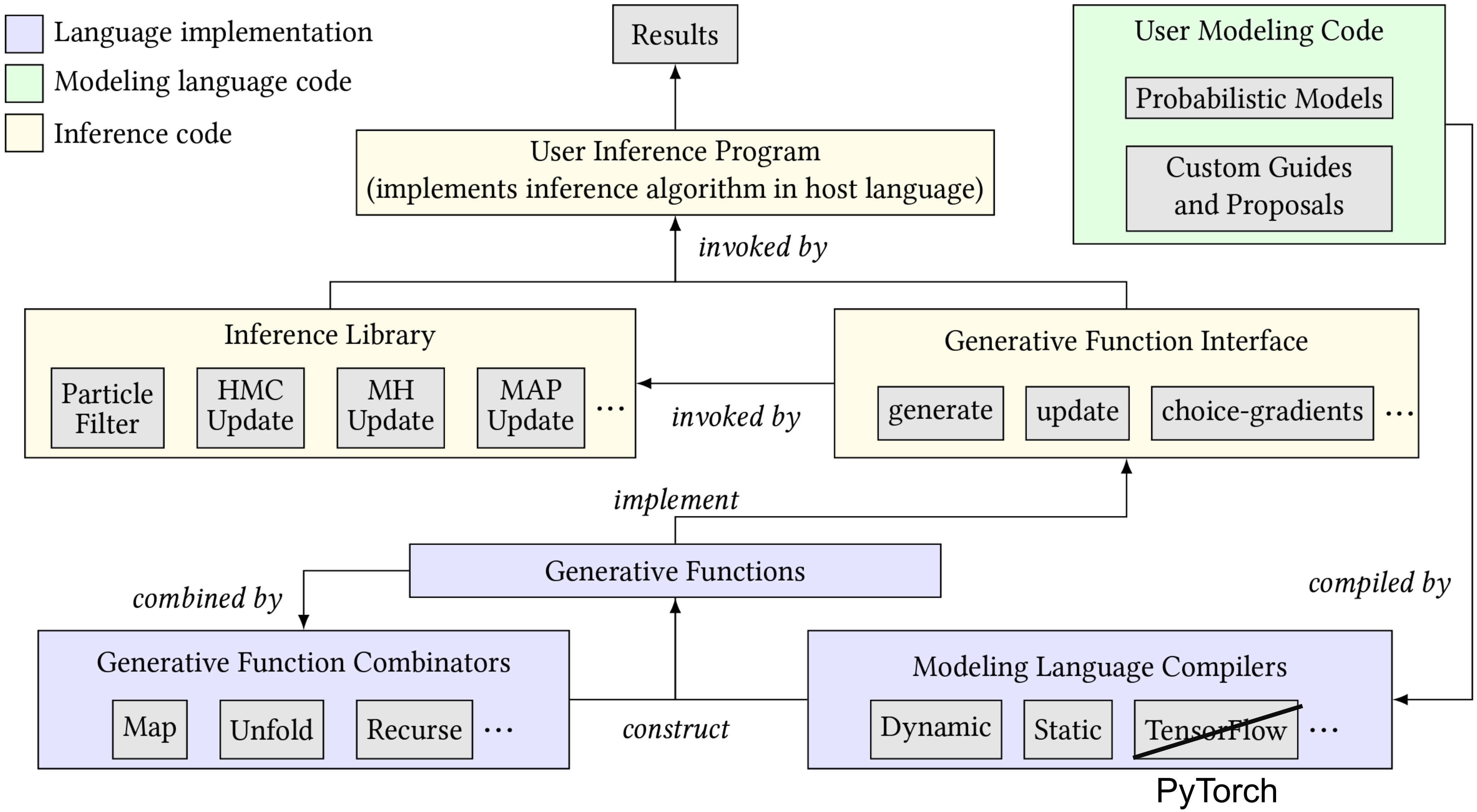
	Inference Algorithm	Runtime (ms)
Stan	Hamiltonian Monte Carlo (NUTS)	53.4ms
Gen (SML + `Map`)	Gaussian Drift Metropolis Hastings	75.3ms
Edward	Hamiltonian Monte Carlo	76.6ms
Anglican	Gaussian Drift Metropolis Hastings	783ms
Venture	Gaussian Drift Metropolis Hastings	1.3×10^6ms

Bayesian linear regression

	Proposal Distribution	Runtime (ms)
Gen (DML + `Unfold`)	Custom	4.9ms (± 0.07)
Gen (DML + `Unfold`)	Generic	82ms (± 3.6)
Anglican	Generic	275ms (± 11)
Turing	Generic	1174ms (± 25)
Venture	Generic	$>10^6$ms

Nonlinear state-space model

Gen's architecture



An example generative model in Gen

Defining a generative model in `Julia`

- Sample n uniformly from 1-10
- With prob of p, multiply n by 2
- With 0.5, sample n, and with 0.5 sample uniformly from the remaining 19 numbers in 1-20

```
using Gen: uniform_discrete, bernoulli, categorical

function f(p)
    n = uniform_discrete(1, 10)
    if bernoulli(p)
        n *= 2
    end
    return categorical([i == n ? 0.5 : 0.5/19 for i=1:20])
end;
```

`@gen` macro for defining a generative model in Gen

```
var = {:address} ~ distribution(args...)
      address ~ distribution(args...)
```

```
using Gen: @gen

@gen function gen_f(p)
    n = {:initial_n} ~ uniform_discrete(1, 10)
    if ({:do_branch} ~ bernoulli(p))
        n *= 2
    end
    return {:result} ~ categorical([i == n ? 0.5 : 0.5/19 for i=1:20])
end;
```

Trace

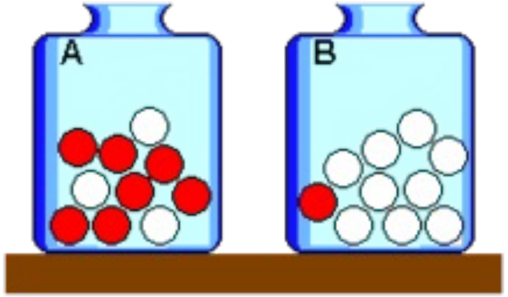
```julia
using Gen: @gen

@gen function gen_f(p)
    n = {:initial_n} ~ uniform_discrete(1, 10)
    if ({:do_branch} ~ bernoulli(p))
        n *= 2
    end
    return {:result} ~ categorical([i == n ? 0.5 : 0.5/19 for i=1:20])
end;
```

```julia
trace = simulate(gen_f, (0.3,));
```

```julia
get_choices(trace)
```

```
│
├── :result : 19
│
├── :do_branch : true
│
└── :initial_n : 7
```

Generating sequence



- Example: Unknown urn problem
- Determine the ratio of the balls based on observations

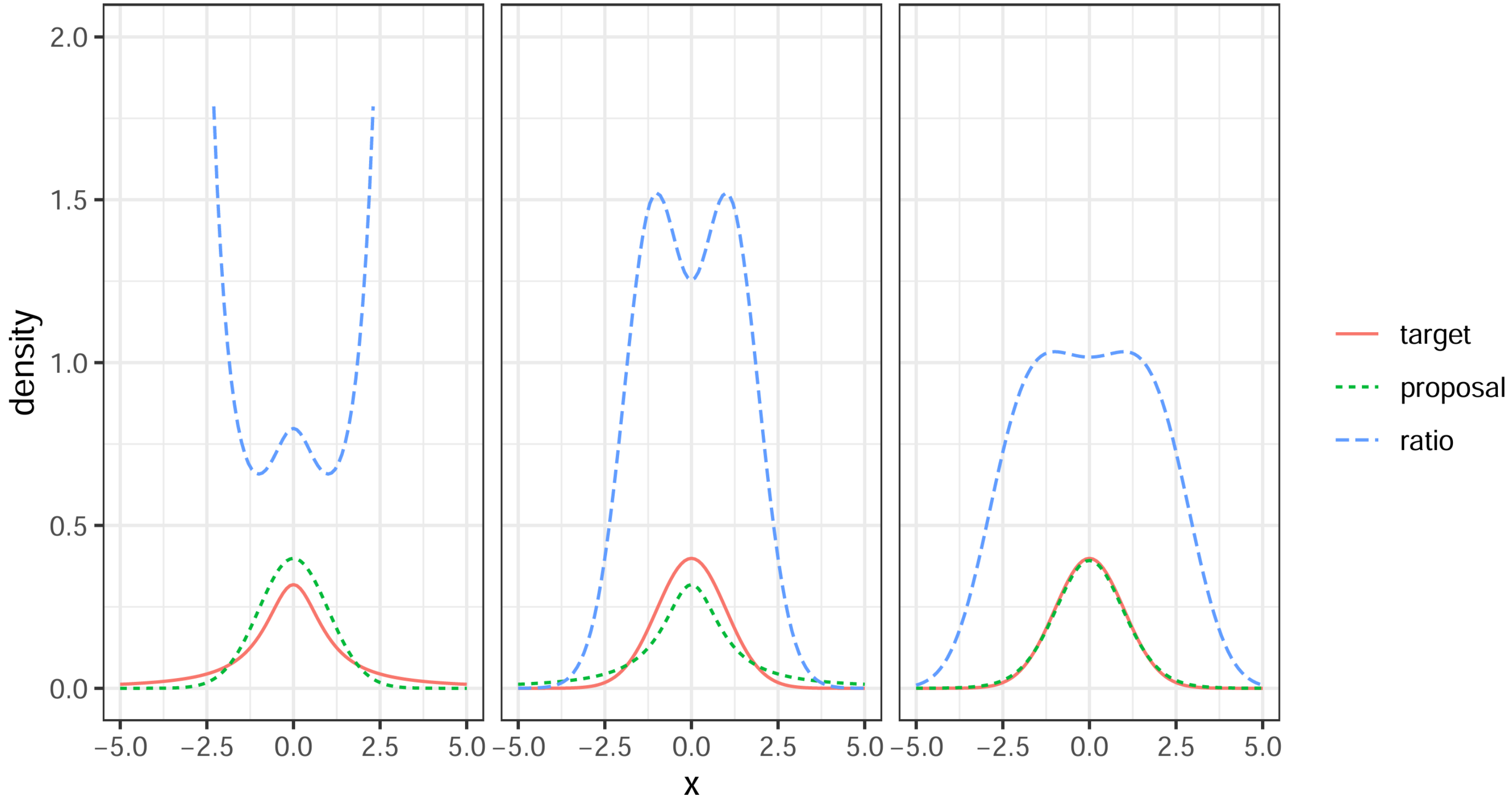
```julia
@gen function unknown_urn()
    # p(θ)~Uniform(0,1)  [prior distribution]
    theta ~ uniform(0, 1)
    for i=1:100
        # p(y=1|θ) ~ Bernoulli(θ) [likelihood function]
        {:data => i => :y} ~ bernoulli(theta)
    end
end
```

```julia
(trace, _) = generate(unknown_urn_static, (3,))
get_choices(trace)
```

```
│
├── :theta : 0.48592872586107905
│
└── :data
     │
     ├── 1
     │   │
     │   └── :y : false
     │
     ├── 2
     │   │
     │   └── :y : true
     │
     └── 3
         │
         └── :y : false
```

Importance sampling

- **Goal:** approximate a *target* distribution $P(x)$, which is hard to sample from
- **Main idea:** Sample a set of particles from a simple *proposal* distribution $q(x)$ (e.g., a uniform distribution) and *weight* them to approximate the distribution
- For each particle *i*
  - Sample: $x_i \sim q(x)$
  - Weight: $w_i = \frac{p(x_i)}{q(x_i)}$
  - (x_i, w_i)



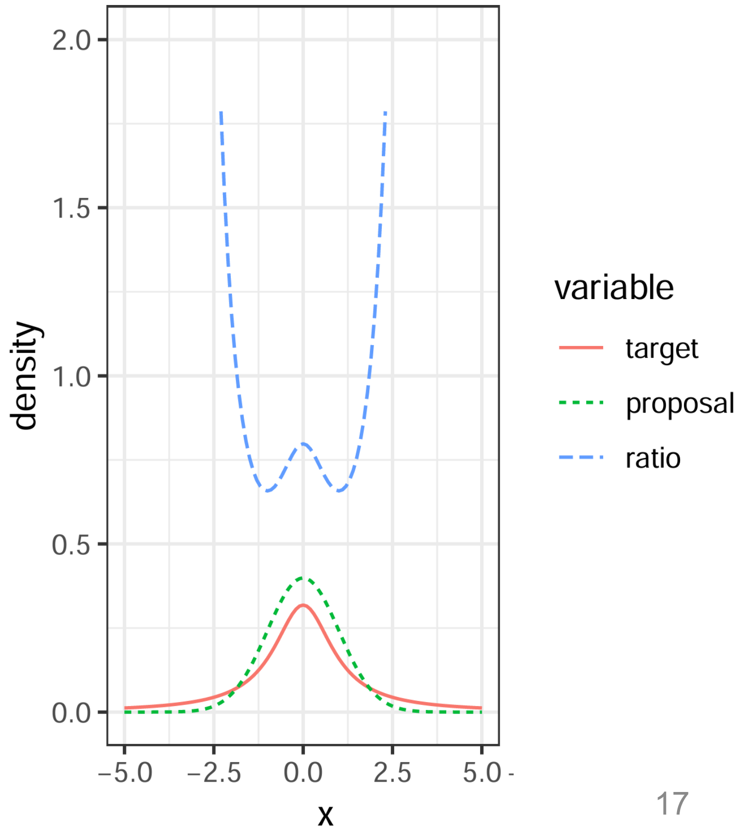
Importance resampling

- **Goal:** sample from the approximated distribution based on the particles and their weights
- Normalize the weights of particles (probability of each particle assuming the whole population is the sampled particles):

$$w_i = \frac{w_i}{\sum_j w_j}$$

- For each new sample x,
  - Sample $j \in \{1,2,\cdots,K\}$, from with probability w
  - The new sample $x = x_j$
  - Set weight $w_i = 1/K$

Limit: proposal distribution needs to be reasonably close to the target distribution



Importance resampling in Bayesian inference

- Target: $h_i \sim P(h|D)$
- Sample: $h_i \sim Q(h)$
- Weight: $w_i \propto \frac{P(D|h_i)P(h_i)}{Q(h_i)}$ $\quad w_i = \frac{w_i}{\sum_j w_j}$
- (h_i, w_i)
- For each new sample h,
  - Sample $j \in \{1,2,\cdots,K\}$, from with probability w_j
  - The new sample $\mathrm{h} = h_j$

Sample, retrieve, reject, weight, update a trace

See the jupyter notebook