Probabilistic Programming Languages

Guillaume Baudart
Christine Tasson

MPRI 2021-2022

Overview

Probabilistic Programming Languages

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$\underbrace{p(\theta \mid x_1, \ldots x_n)}_{\text{posterior}} = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto \underbrace{p(\theta)}_{\text{prior}}\ \underbrace{p(x_1, \ldots, x_n \mid \theta)}_{\text{likelihood}} \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Probabilistic Programming Languages

General purpose programming languages extended with probabilistic constructs

- `sample`: draw a sample from a distribution
- `assume`, `factor`, `observe`: condition the model on inputs (e.g., observed data)
- `infer`: compute the posterior distribution of a model given the inputs

Multiple examples:

- Church, Anglican (lisp, clojure), 2008
- WebPPL (javascript), 2014
- Pyro/NumPyro (python), 2017/2019
- Gen (julia), 2018
- ProbZelus (Zelus), 2019
- ...

More and more, incorporating new ideas:

- New inference techniques, e.g., stochastic variational inference (SVI)
- Interaction with neural nets (deep probabilistic programming)

Inference in Practice

Overview

Rejection Sampling

```
module Rejection_sampling = struct
  exception Reject

  let sample d = Distribution.draw d
  let assume p = if not p then raise Reject

  let infer ?(n = 1000) model obs =
    let rec exec i = try model Prob obs with Reject → exec i in
    let values = Array.init n exec in
    Distribution.uniform_support ~values
end
```

Executing the model generates one sample

- `sample`: draw from a distribution
- `assume`/`observe`: hard conditioning, reject invalid samples
- Terminates with n valid samples

Importance Sampling

```
let coin prob data =
  type prob = { id : int; scores : float array }

  let sample _prob d = Distribution.draw d
  let factor prob s = prob.scores.(prob.id) ← prob.scores.(prob.id) +. s

  let infer ?(n = 1000) model obs =
    let scores = Array.make n 0. in
    let values = Array.mapi (fun i _ → model { id = i; scores } obs) scores in
    Distribution.support ~values ~logits:scores
end
```

Executing the model generates a pair (sample, weight)

- `sample`: draw from a distribution
- `factor`/`observe`: soft conditioning, assign a score
- Terminates with n pairs (sample, weight)

Particle Filter

basic.ml

```
module Particle_filter = struct
  include Importance_sampling

  let resample particles = ... (* resample an array of (particle, score) *)

  let factor s k prob = ...
    (* execute all the particle until the next factor statement *)
    (* resample given the score so far *)
    (* resume execution until the next factor statement *)
end
```

Inference algorithm : importance sampling, but...

- Add a resampling step at each `factor`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Using CPS models, we can easily clone, stop, and resume particles the middle of an execution.

Coin

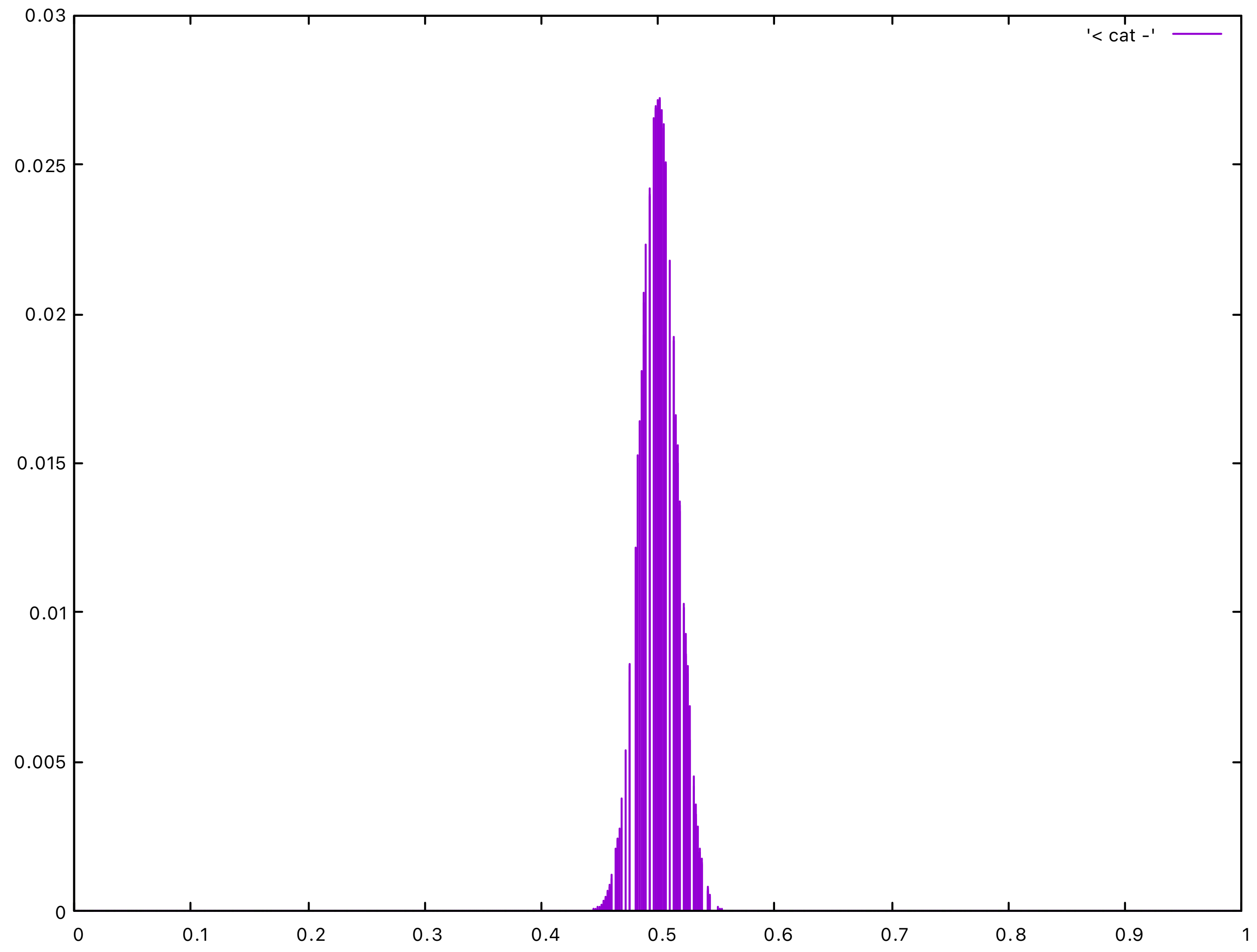
coin.ml

```
let _ =
  let data = List.init 1000 (fun i → i mod 2) in
  let dist = infer coin data in
  plot dist
```

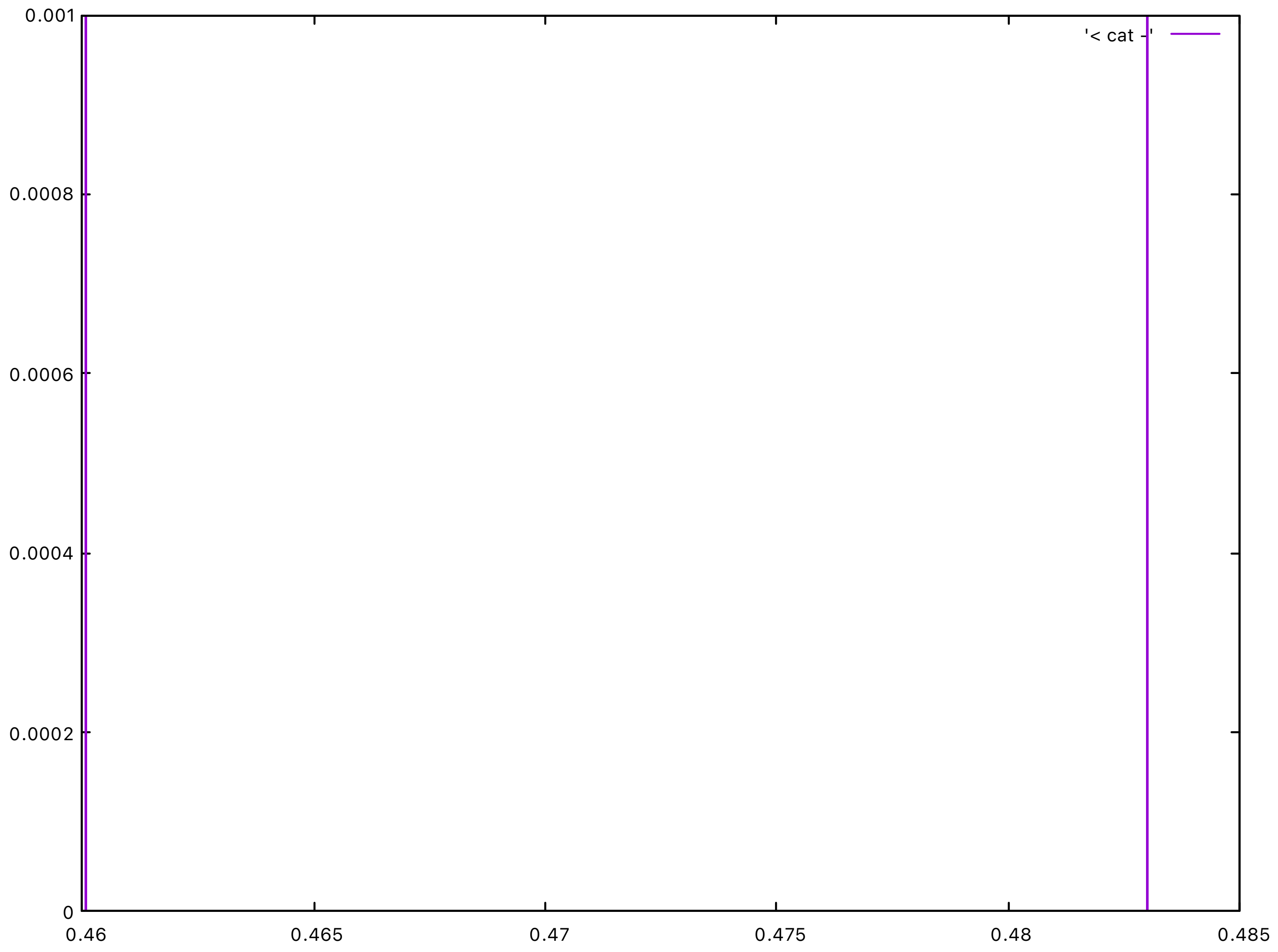
Rejection Sampling

Very (very) slow!

Importance Sampling



Particle Filter



Particle impoverishment

HMM

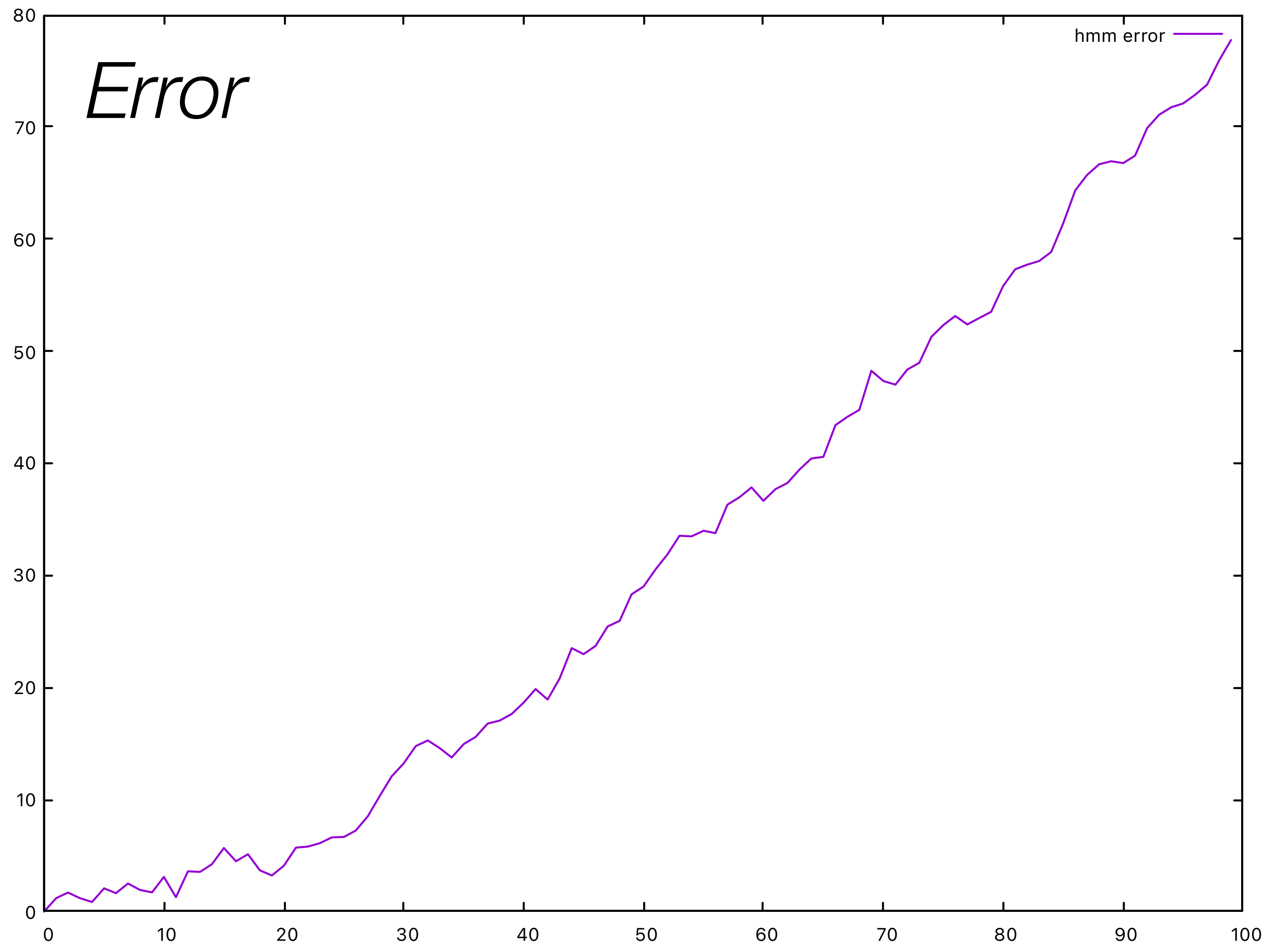
hmm.ml

```
let _ =
  let data = Owl.Arr.linspace 0. 100. 100 ▷ Owl.Arr.to_array ▷ Array.to_list in
  let dist = Distribution.split_list (infer hmm data) in
  plot (error (List.rev dist) data)
```

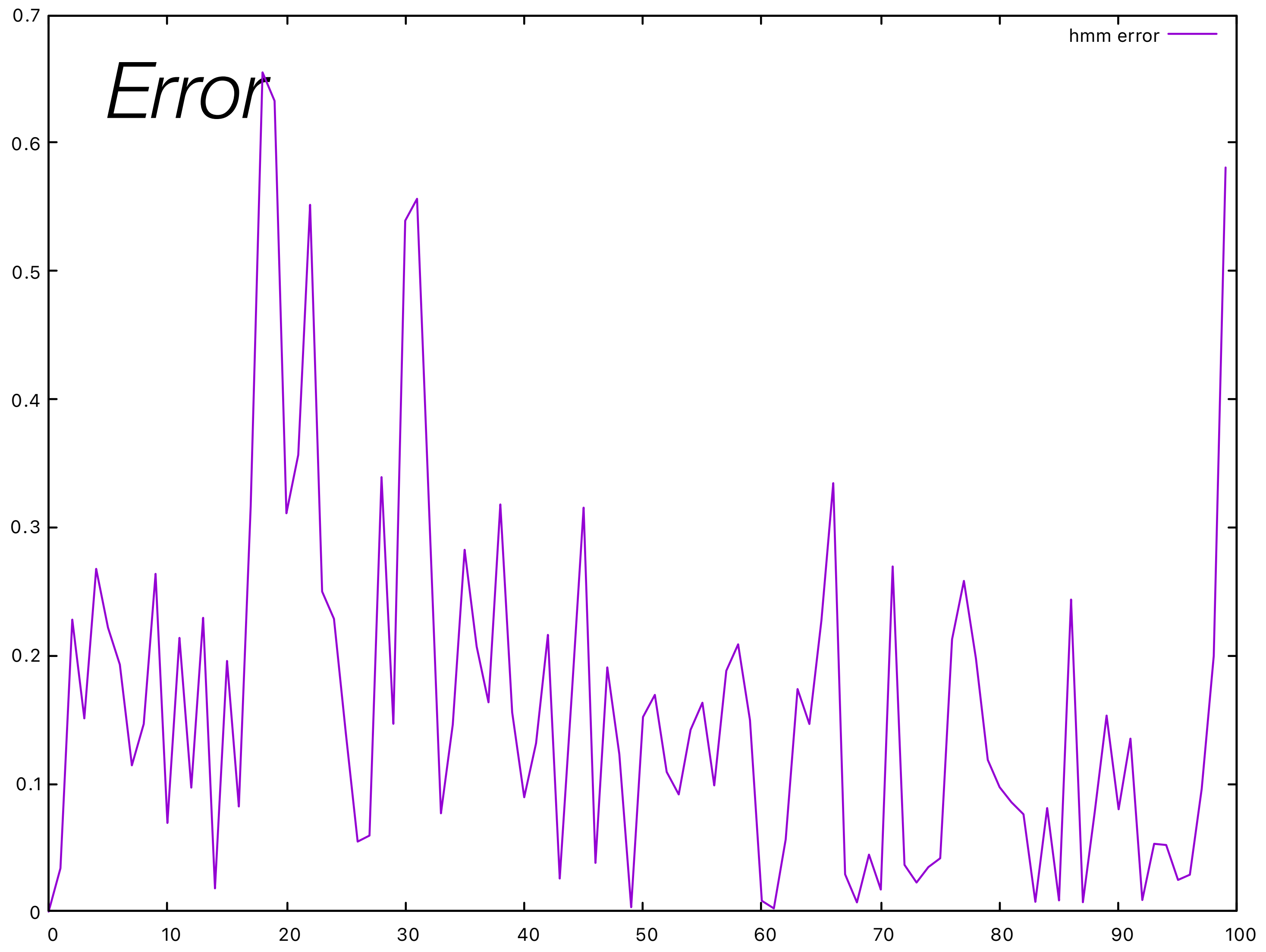
Rejection Sampling

Never terminate!

Importance Sampling



Particle Filter



Denotational Semantics

Overview

Deterministic vs. Probabilistic

Deterministic semantics $G \vdash^{\mathtt{D}} e : t$

- Classic denotational semantics
- Given an environment $\gamma : \Gamma$, $[\![e]\!]_\gamma$ returns a value of type t
- $[\![e]\!] : \Gamma \to t$

Probabilistic semantics $G \vdash^{\mathtt{P}} e : t$

- Expressions are interpreted as kernels
- Given an environment $\gamma : \Gamma$, $\{\![e]\!\}_\gamma$ is a measure on values of type t
- $\{\![e]\!\} : \Gamma \times \Sigma_{[\![t]\!]} \to [0,\infty)$

Probabilistic Programming

set of possible outcomes

model

$$\{\![e]\!\}_\gamma : \Sigma_{[\![t]\!]} \to [0,\infty)$$

score

environment

Unnormalized measure

$$[\![\text{infer}(e)]\!]_\gamma = \frac{\{\![e]\!\}_\gamma}{\{\![e]\!\}_\gamma([\![typeOf(e)]\!])}$$

Distribution

normalize over all possible values

Probabilistic Semantics

$$\{\![e]\!\}_\gamma \quad = \quad \lambda U.\ \delta_{[\![e]\!]_\gamma}(U)\ \mathit{if}\ \mathit{kindOf}(e) = \mathtt{D}$$

$$\{\![\mathtt{let}\ p\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!\}_\gamma \quad = \quad \lambda U.\ \int_{[\![\mathit{typeOf}(e_1)]\!]} \{\![e_1]\!\}_\gamma(du)\{\![e_2]\!\}_{\gamma+[u/p]}(U)$$

$$\{\![\mathtt{sample}(e)]\!\}_\gamma \quad = \quad \lambda U.\ [\![e]\!]_\gamma(U)$$

$$\{\![\mathtt{factor}(e)]\!\}_\gamma \quad = \quad \lambda U.\ \exp([\![e]\!]_\gamma) * \delta_{()}(U)$$

Example : Beta


```
let my_beta a b =
  let x = sample (uniform ~a:0 ~b:1) in
  let () = observe (beta ~a ~b) x in
  x
```

$$
\begin{aligned}
[\![\texttt{my\_beta a b}]\!]\_(U) &= \int_0^1 [\![\texttt{sample (uniform 0 1)}]\!]_{\{a/\mathrm{a},\ b/\mathrm{b}\}}(dx) \\
&\qquad \int_{()} [\![\texttt{observe (beta a b) x}]\!]_{\{a/\mathrm{a},\ b/\mathrm{b},\ x/\mathrm{x}\}}(du)[\![\texttt{x}]\!]_{\{a/\mathrm{a},\ b/\mathrm{b},\ x/\mathrm{x},\ u/()\}}(U) \\
&= \int_0^1 Uniform(dx)\ Beta_{\mathrm{pdf}}(x)\ \delta_x(U) \\
&= \int_U Beta(a,b)_{\mathrm{pdf}}(x)dx \\
&= Beta(a,b)(U)
\end{aligned}
$$

Operational Semantics

Overview

Sampler

Probabilistic semantics $G \vdash^{\mathtt{P}} e : t$

- Expressions are interpreted as samplers
- Given an environment $\gamma : \Gamma$, and a weight $w \in [0, \infty)$, $\{\!\![e]\!\!\}_{\gamma,w} = v, w'$ returns a pair (value, new score)
- $\{\!\![e]\!\!\} : \Gamma \times [0,\infty) \rightarrow [0,\infty)$

$$
\begin{array}{lcl}
\{\!\![c]\!\!\}_{\gamma,w} & = & c, w \\
\{\!\![x]\!\!\}_{\gamma,w} & = & \gamma(x), w \\
\{\!\![\mathtt{sample}(e)]\!\!\}_{\gamma,w} & = & let\ d, w' = [\![e]\!]_{\gamma,w}\ in\ \mathrm{draw}(d), w' \\
\{\!\![\mathtt{factor}(e)]\!\!\}_{\gamma,w} & = & let\ s, w' = [\![e]\!]_{\gamma,w}\ in\ (), s * w'
\end{array}
$$

Importance Sampling

Inference algorithm

- Run a set of n independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

$$[\![\texttt{infer}(e)]\!]_\gamma \quad = \quad let \ \underbrace{[(v_i, w'_i) = \{\![e]\!\}_{\gamma, w_i}]_{1 \leq i \leq N}}_{\text{particles}} \ in \ \underbrace{\lambda U. \ \textstyle\sum_{1 \leq i \leq N} \overline{w'_i} * \delta_{v_i}(U)}_{\text{distribution}}$$

Normalized weights: $\overline{w_i} = \dfrac{w_i}{\sum_{i=1}^{n} w_i}$

Reactive Probabilistic Programming

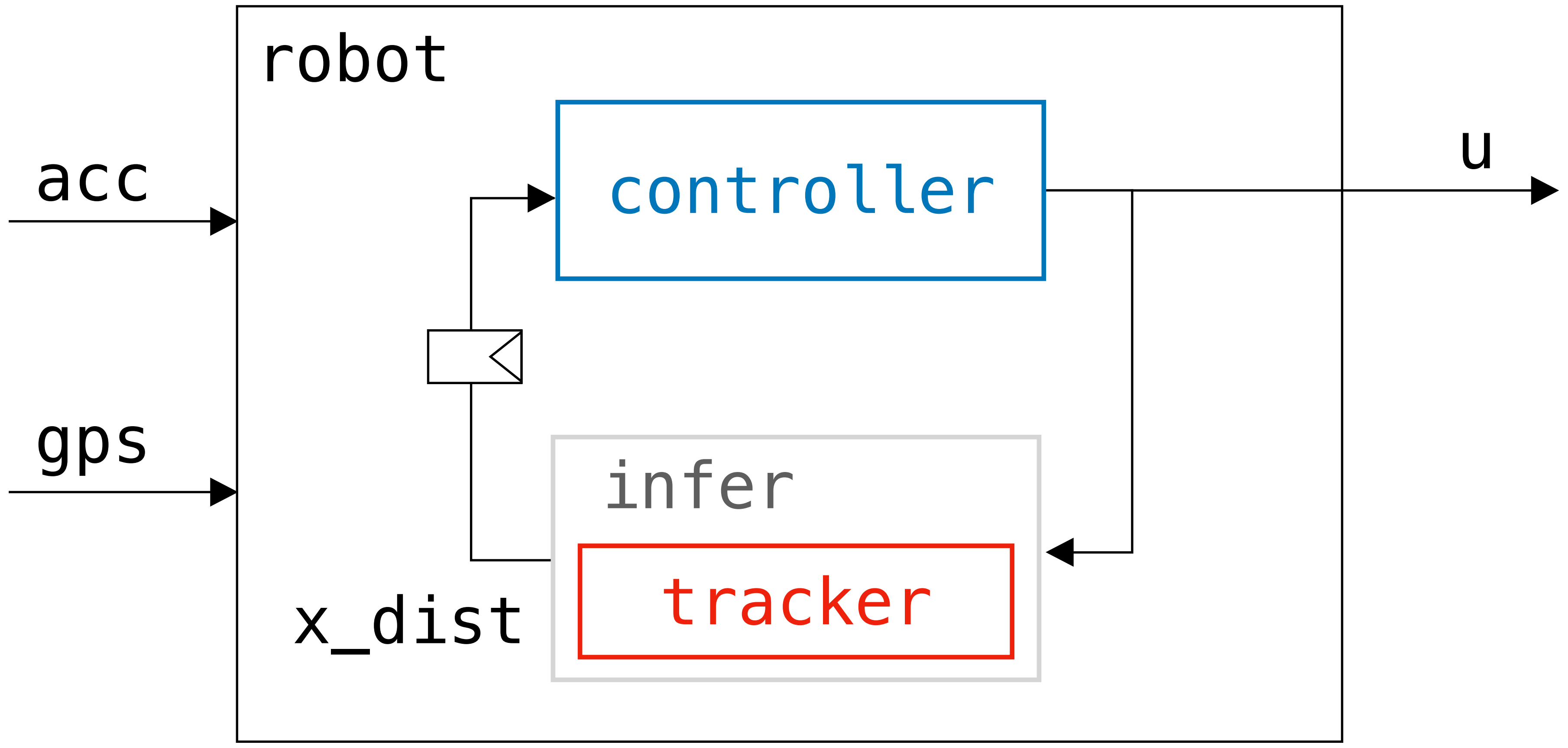
Overview

Reactive Probabilistic Systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

State: `x_dist`: (*position* × *velocity* × *acceleration*) `dist`



ProbZelus: add support to deal with uncertainty

- Extend a synchronous language
- Parallel composition: deterministic/probabilistic
- Inference-in-the-loop
- Streaming inference

Probabilistic programming with streams

```
let node robot (acc, gps) = u where
  rec u = controller (x0_dist → pre x_dist)
  and x_dist = infer tracker (u, acc, gps)
```

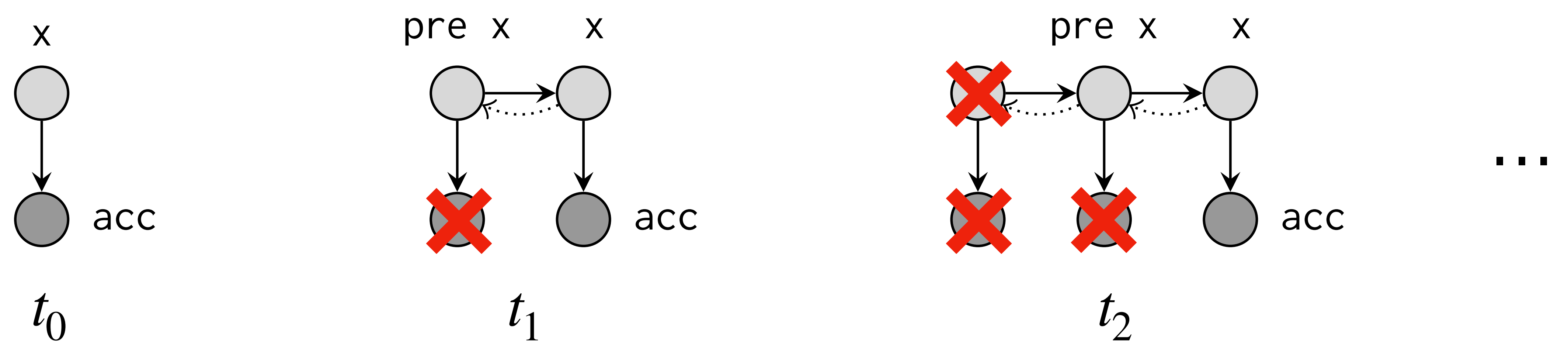
Reactive Probabilistic Programming

Language design and implementation

- Type system deterministic / probabilistic
- Measure-based formal semantics
- Compilation to transition functions

Streaming inference

- Sequential Monte-Carlo method for reactive models
- Semi-symbolic delayed sampling with bounded memory



Evaluation

- Multiple benchmarks from simple to advanced
- Inference schemes comparison: speed vs. accuracy

https://github.com/IBM/probzelus