# Reactive Probabilistic Programming

Guillaume Baudart,
Louis Mandel

*IBM Research*

Marc Pouzet

*ENS Paris*

Eric Atkinson,
Benjamin Sherman,
Michael Carbin

*MIT*

# Probabilistic Programming

# Probabilistic Programming

Programming and reasoning with uncertainty

- Program functions with uncertainty: sample from distributions
- Condition on observed data: inputs of the model

Probabilistic Programming Languages

- Bugs, Stan, Church, Blog, Anglican, Venture, Figaro, WebPL, Pyro, Edward, ...

Probabilistic constructs:

- x = **sample**(d): introduce a random variable x of distribution d
- **observe**(d, y): measure the likelihood of an observation y w.r.t d
- **infer** m obs: compute output distribution of a model m given obs

Inference: compute probability distribution defined by a model given observations or data (similar to learning in machine learning)

# ProbZelus: Design Choices

Zelus extended with probabilistic constructs

Inference in the loop

- Interaction between deterministic processes and probabilistic models
- Models receives input from the environment
- Deterministic processes can access intermediate results
- Feedback between inferred distribution and deterministic processes

Streaming inference

- Inference runs in parallel with deterministic processes (non-terminating)
- Should run with bounded ressources
- Multiples inference algorithms with different trade-off cost/accuracy
- Sequential Monte-Carlo (SMC) vs. streaming delayed sampling

# Bayesian Inference (in-the-loop)

Learn parameters from data

- Latent parameters at instant t $\theta_t$
- Observed data $x_1, \dots x_t$

Compute the distribution $p(\theta_t \mid x_1, \dots x_t)$ at each time step

$$p(\theta_t \mid x_1, \dots x_t) = \frac{p(\theta_t) p(x_1, \dots, x_t \mid \theta_t)}{p(x_1, \dots, x_t)} \qquad \text{(Bayes' theorem)}$$

# Bayesian Inference (in-the-loop)

Learn parameters from data

- Latent parameters at instant t $\theta_t$
- Observed data $x_1, \ldots x_t$

Compute the distribution $p(\theta_t \mid x_1, \ldots x_t)$ at each time step

$$p(\theta_t \mid x_1, \ldots x_t) = \frac{p(\theta_t) p(x_1, \ldots, x_t \mid \theta_t)}{p(x_1, \ldots, x_t)} \qquad \text{(Bayes' theorem)}$$

$$\propto p(\theta_t) p(x_1, \ldots, x_t \mid \theta_t) \qquad \text{(Data are constants)}$$

# Bayesian Inference (in-the-loop)

Learn parameters from data

- Latent parameters at instant t $\theta_t$
- Observed data $x_1, \ldots x_t$

Compute the distribution $p(\theta_t \mid x_1, \ldots x_t)$ at each time step

$$p(\theta_t \mid x_1, \ldots x_t) = \frac{p(\theta_t) p(x_1, \ldots, x_t \mid \theta_t)}{p(x_1, \ldots, x_t)} \qquad \text{(Bayes' theorem)}$$

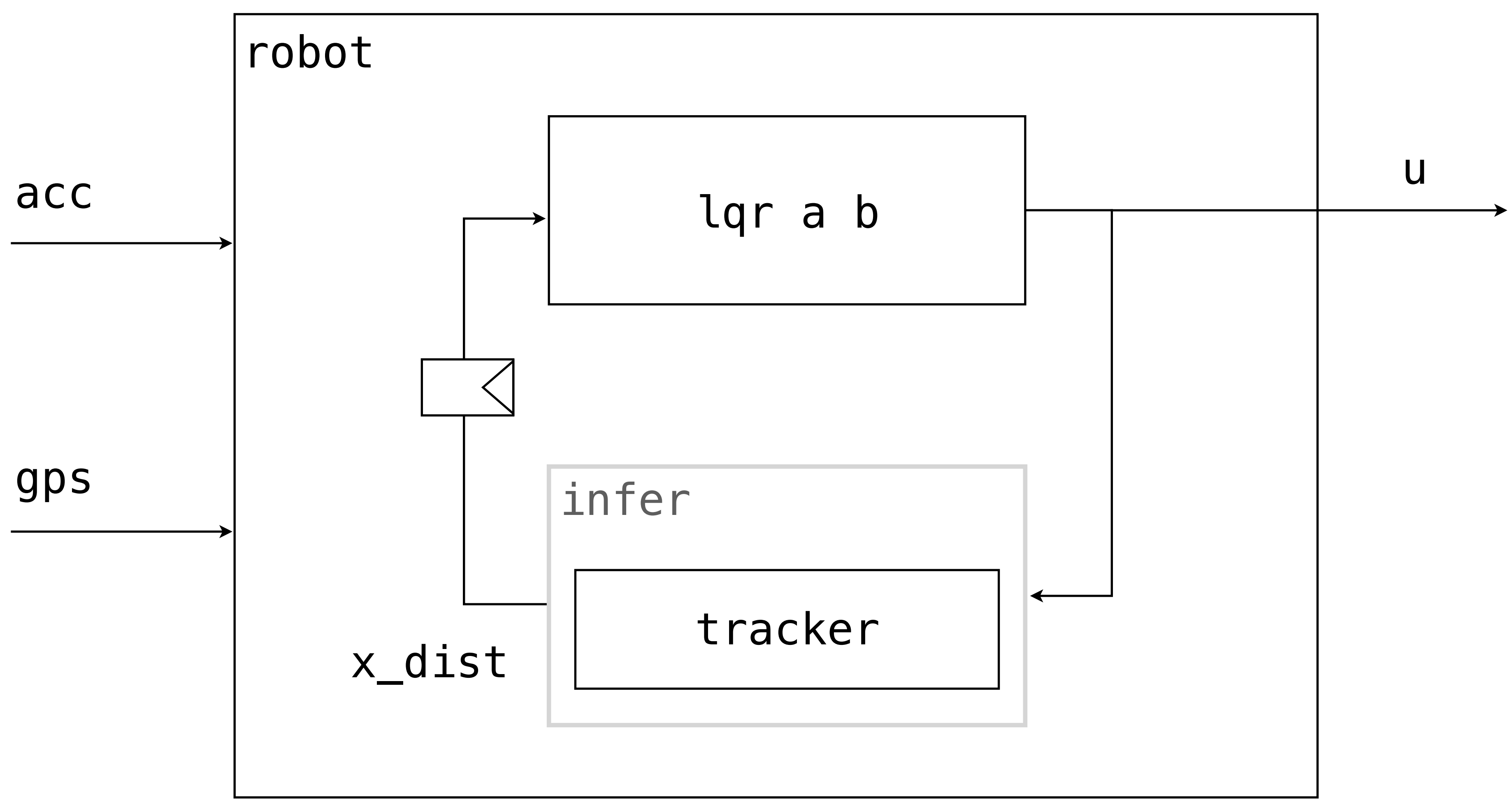
$$\propto p(\theta_t) p(x_1, \ldots, x_t \mid \theta_t) \qquad \text{(Data are constants)}$$

*prior:* **sample** *likelihood:* **observe**

# Example

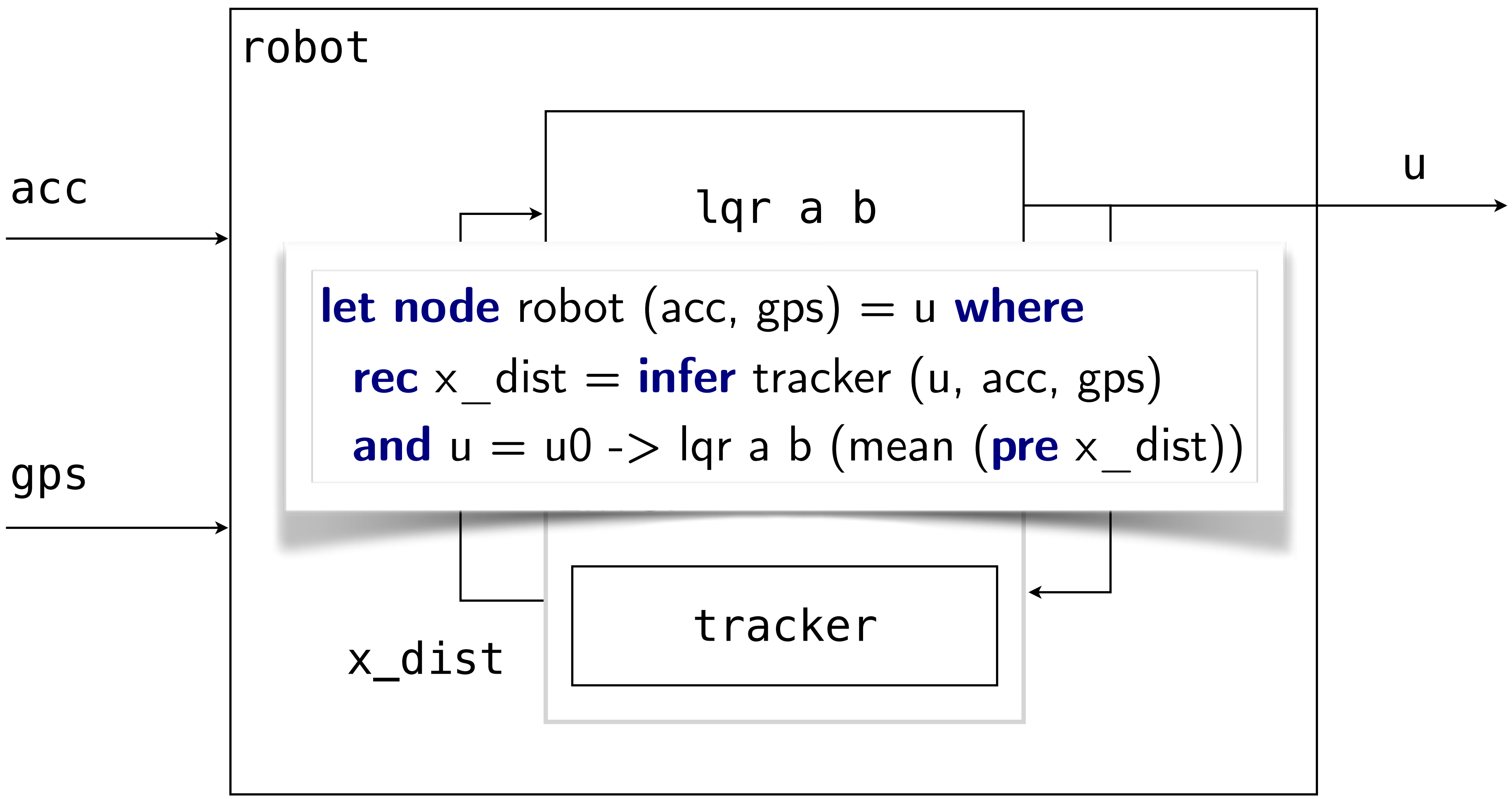
# Robot Controller

- Input: noisy acceleration `acc` (at each step), noisy position `gps` (sporadic)
- Output: command `u` to drive the robot to a given target
- State: $x_t$ = (position, velocity, acceleration)
- Motion model: $x_{t+1} = A.x_t + B.u_t$ (A, B are constant matrices)



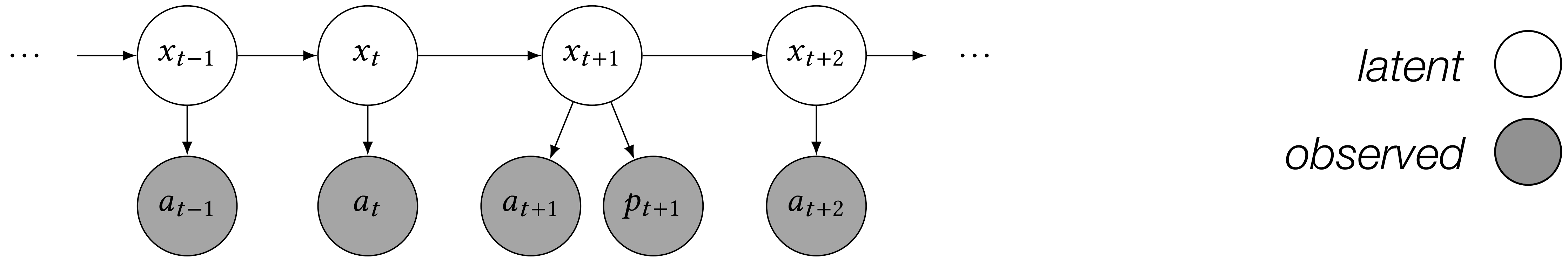
# Robot Controller

- Input: noisy acceleration **`acc`** (at each step), noisy position **`gps`** (sporadic)
- Output: command **`u`** to drive the robot to a given target
- State: $x_t$ = (position, velocity, acceleration)
- Motion model: $x_{t+1} = A.x_t + B.u_t$ (A, B are constant matrices)




```
let node robot (acc, gps) = u where
  rec x_dist = infer tracker (u, acc, gps)
  and u = u0 -> lqr a b (mean (pre x_dist))
```

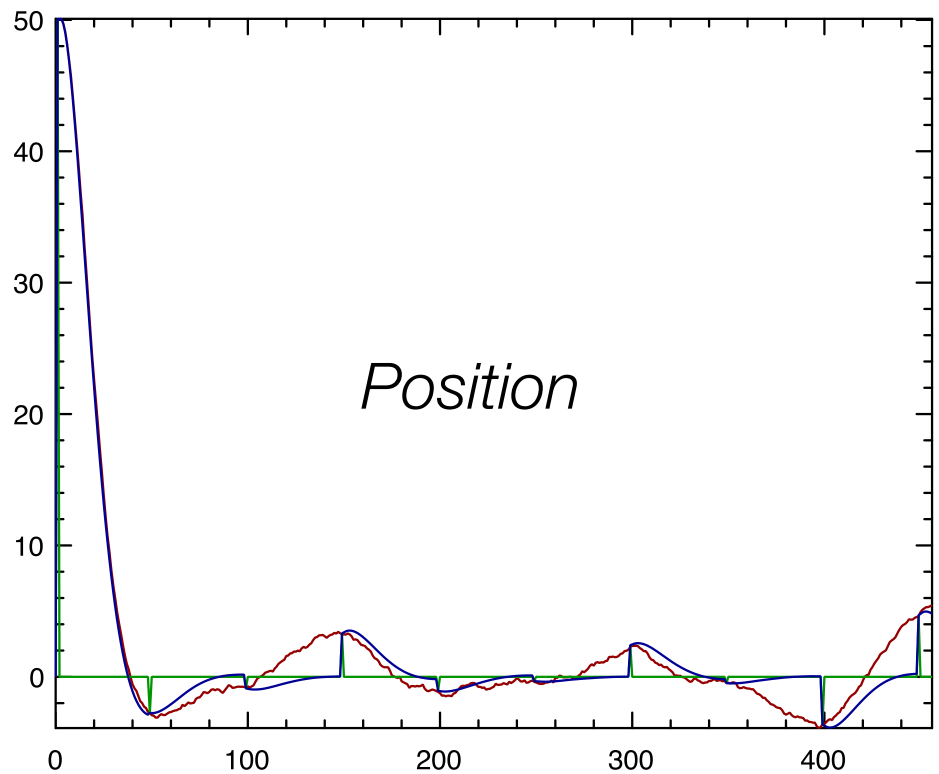
# Robot Controller

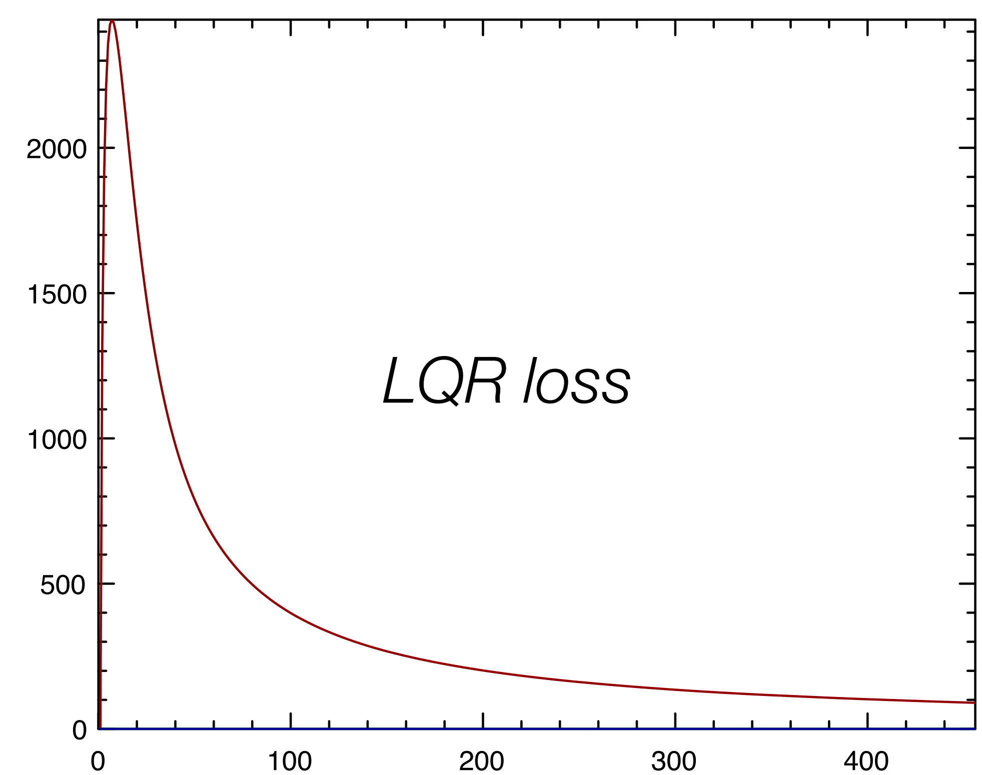


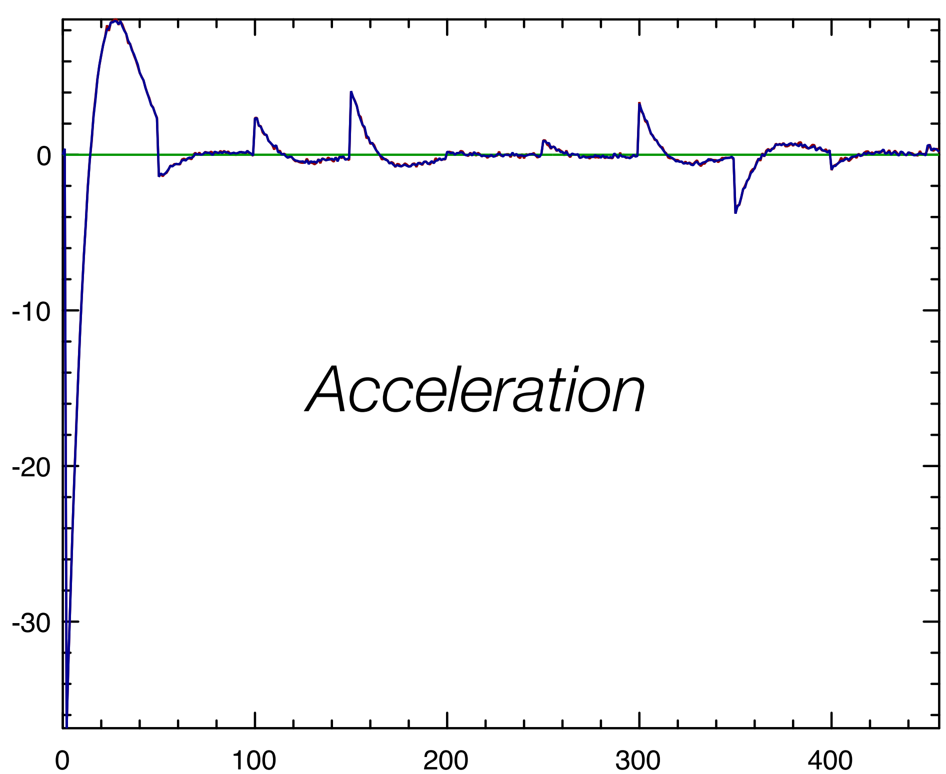

```
let proba kalman (u, acc, gps) = x where
  rec mu = x0 -> (a *@ pre x) +@ (b *@ u)          (* xt+1 = A.xt + B.ut *)
  and x = sample (mv_gaussian (mu, noise))
  and () = observe (gaussian (vec_get x 2, 1.0), acc)
  and present gps (pos) ->
        do () = observe (gaussian (vec_get x 0, 0.01), pos) done

let node robot (acc, gps) = u where
  rec x_dist = infer 100 tracker (u, acc, gps)
  and u = u0 -> lqr a b (mean (pre x_dist))
```

# Robot Controller




- *exact value*
- *estimated value*
- *gps readings*

# Language

# Syntax

$d ::=$ **let node** $f\ x = e \mid$ **let proba** $f\ x = e \mid d\ d$

$e ::= c \mid x \mid (e,e) \mid op(e) \mid f(e) \mid$ **last** $x \mid e$ **where rec** $E$

$\mid$ **present** $e$ −> $e$ **else** $e \mid$ **reset** $e$ **every** $e$

$\mid$ **sample**$(e) \mid$ **observe**$(e, e) \mid$ **infer**$(e)$

$E ::= x = e \mid$ **init** $x = c \mid E$ **and** $E$

- Other constructs can be expressed in this kernel.
- Probabilistic models are nodes (proba)
- Local equations in e **where rec** E are scheduled

```
x where
  rec init x1 = c1
  and init x2 = c2
  and x1 = e1
  and x2 = e2
```

# Syntax

$d$ ::= let node $f\ x = e$ | let proba $f\ x = e$ | $d\ d$
$e$ ::= $c$ | $x$ | $(e,e)$ | $op(e)$ | $f(e)$ | last $x$ | $e$ where rec $E$
| present $e$ −> $e$ else $e$ | reset $e$ every $e$
| sample($e$) | observe($e$, $e$) | infer($e$)
$E$ ::= $x = e$ | init $x = c$ | $E$ and $E$

- Other constructs can be expressed in this kernel.
- Probabilistic models are nodes (proba)
- Local equations in e **where rec** E are scheduled

```
x where
  rec init x1 = c1
  and init x2 = c2
  and x1 = e1
  and x2 = e2
```

```
x = 0 -> pre x + 1
```

---

```
x where
  rec init fst = true
  and init x = 0
  and fst = false
  and x = if last fst then 0 else last x + 1
```

# Typing: D vs. P

$$\frac{G \vdash^{\mathsf{D}} e : T \text{ dist}}{G \vdash^{\mathsf{P}} \text{sample}(e) : T} \qquad \frac{G \vdash^{\mathsf{D}} e_1 : T \text{ dist} \quad G \vdash^{\mathsf{D}} e_2 : T}{G \vdash^{\mathsf{P}} \text{observe}(e_1, e_2) : \text{unit}}$$

$$\frac{G \vdash^{\mathsf{D}} e : T}{G \vdash^{\mathsf{P}} e : T} \qquad \frac{G \vdash^{\mathsf{P}} e : T}{G \vdash^{\mathsf{D}} \text{infer}(e) : T \text{ dist}}$$

- Add a kind D (deterministic) or P (probabilistic)
- **sample** and **observe** can only be used in a probabilistic context
- Deterministic expression can be lifted to probabilistic ones
- Transition realized by **infer**
- Add a datatype for distributions $T$ dist

[Benveniste et al. 11]

# Co-iteration Semantics

Deterministic Stream: Initial state, transition function

$$CoStream(T, S) = S \times (S \longrightarrow T \times S)$$
$$CoNode(T, T', S) = S \times (S \longrightarrow T \longrightarrow T' \times S).$$

$$[\![e]\!]_\gamma \; : CoStream(T, S) = [\![e]\!]^i_\gamma, [\![e]\!]^s_\gamma$$

[Caspi, Pouzet 94, Staton 17]

# Co-iteration Semantics

Deterministic Stream: Initial state, transition function

$$CoStream(T, S) = S \times (S \rightarrow T \times S)$$
$$CoNode(T, T', S) = S \times (S \rightarrow T \rightarrow T' \times S).$$

$$[\![e]\!]_\gamma \, : \, CoStream(T, S) = [\![e]\!]^i_\gamma, [\![e]\!]^s_\gamma$$

Probabilistic Stream: transition function returns a *measure* over pairs (result, state)

$$CoPStream(T, S) = S \times (S \rightarrow (\Sigma_{T \times S} \rightarrow [0, \infty]))$$
$$CoPNode(T, T', S) = S \times (S \rightarrow T \rightarrow (\Sigma_{T' \times S} \rightarrow [0, \infty]))$$

$$\{\![e]\!\}_\gamma \, : \, CoPStream(T, S) = \{\![e]\!\}^i_\gamma, \{\![e]\!\}^s_\gamma$$

[Caspi, Pouzet 94, Staton 17]

# Co-iteration Semantics

Deterministic Stream: Initial state, transition function

$$CoStream(T, S) = S \times (S \to T \times S)$$
$$CoNode(T, T', S) = S \times (S \to T \to T' \times S).$$

$$[\![e]\!]_\gamma : CoStream(T, S) = [\![e]\!]^i_\gamma, [\![e]\!]^s_\gamma$$

Probabilistic Stream: transition function returns a *measure* over pairs (result, state)

$$CoPStream(T, S) = S \times (S \to (\Sigma_{T \times S} \to [0, \infty]))$$
$$CoPNode(T, T', S) = S \times (S \to T \to (\Sigma_{T' \times S} \to [0, \infty]))$$

$$\{\!\![e]\!\!\}_\gamma : CoPStream(T, S) = \{\!\![e]\!\!\}^i_\gamma, \{\!\![e]\!\!\}^s_\gamma$$

Normalize the measure to obtain a distribution

$$\mu : \Sigma_D \to [0, \infty]$$
$$d : T \text{ dist} = \frac{\mu}{\int_D \mu(dx)}$$

[Caspi, Pouzet 94, Staton 17]

# Co-iteration Semantics: D

$$
\begin{array}{lcl}
[\![x]\!]^i_\gamma & = & () \\
[\![x]\!]^s_\gamma & = & \lambda s.\ (\gamma(x), s)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^i_\gamma & = & ([\![e]\!]^i_\gamma, [\![e_1]\!]^i_\gamma, [\![e_2]\!]^i_\gamma) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^s_\gamma & = & \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = [\![e]\!]^s_\gamma(s)\ \mathit{in} \\
& & \quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s'_1 = [\![e_1]\!]^s_\gamma(s_1)\ \mathit{in}\ (v_1, (s', s'_1, s_2)) \\
& & \quad \quad \mathit{else}\ \ \mathit{let}\ v_2, s'_2 = [\![e_2]\!]^s_\gamma(s_2)\ \mathit{in}\ (v_2, (s', s_1, s'_2))
\end{array}
$$

$$
\left[\!\!\left[
\begin{array}{l}
e\ \texttt{where} \\
\quad \texttt{rec init}\ x_1 = c_1\ \texttt{and init}\ x_2 = c_2 \\
\quad \texttt{and}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2
\end{array}
\right]\!\!\right]^i_\gamma
=
\left(
\begin{array}{c}
(c_1, c_2), \\
([\![e_1]\!]^i_\gamma, [\![e_2]\!]^i_\gamma), \\
[\![e]\!]^i_\gamma
\end{array}
\right)
$$

$$
\left[\!\!\left[
\begin{array}{l}
e\ \texttt{where} \\
\quad \texttt{rec init}\ x_1 = c_1\ \texttt{and init}\ x_2 = c_2 \\
\quad \texttt{and}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2
\end{array}
\right]\!\!\right]^s_\gamma
=
\begin{array}{l}
\lambda((m_1, m_2), (s_1, s_2), s). \\
\quad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\_\mathrm{last}]\ \mathit{in}\ \mathit{let}\ \gamma_2 = \gamma_2[m_2/x_2\_\mathrm{last}]\ \mathit{in} \\
\quad \mathit{let}\ v_1, s'_1 = [\![e_1]\!]^s_{\gamma_2}(s_1)\ \mathit{in}\ \mathit{let}\ \gamma'_1 = \gamma_2[v_1/x_1]\ \mathit{in} \\
\quad \mathit{let}\ v_2, s'_2 = [\![e_2]\!]^s_{\gamma'_1}(s_2)\ \mathit{in}\ \mathit{let}\ \gamma'_2 = \gamma'_1[v_2/x_2]\ \mathit{in} \\
\quad \mathit{let}\ v, s' = [\![e]\!]^s_{\gamma'_2}(s)\ \mathit{in} \\
\quad v, ((\gamma'_2[x_1], \gamma'_2[x_2]), (s'_1, s'_2), s')
\end{array}
$$

# Co-iteration Semantics: P

$$\{\!\![ e ]\!\!\}^i_\gamma = [\![ e ]\!]^i_\gamma \qquad \textit{if kindOf}(\mathrm{e}) = \mathsf{D}$$
$$\{\!\![ e ]\!\!\}^s_\gamma = \lambda s.\ \lambda U.\ \delta_{[\![ e ]\!]^s_\gamma(s)}(U) \qquad \textit{if kindOf}(e) = \mathsf{D}$$

$$\{\!\![ \mathsf{sample}(e) ]\!\!\}^i_\gamma = [\![ e ]\!]^i_\gamma$$
$$\{\!\![ \mathsf{sample}(e) ]\!\!\}^s_\gamma = \lambda s.\ \lambda U.\ \textit{let}\ \mu, s' = [\![ e ]\!]^s_\gamma(s)\ \textit{in}\ \int_T \mu(dv)\ \delta_{v,s'}(U)$$

$$\{\!\![ \mathsf{observe}(e_1, e_2) ]\!\!\}^i_\gamma = ([\![ e_1 ]\!]^i_\gamma, [\![ e_2 ]\!]^i_\gamma)$$
$$\{\!\![ \mathsf{observe}(e_1, e_2) ]\!\!\}^s_\gamma = \lambda(s_1, s_2).\ \lambda U.$$
$$\textit{let}\ \mu, s_1' = [\![ e_1 ]\!]^s_\gamma(s_1)\ \textit{in}$$
$$\textit{let}\ v, s_2' = [\![ e_2 ]\!]^s_\gamma(s_2)\ \textit{in}\ \mu_{\mathrm{pdf}}(v) * \delta_{(),(s_1', s_2)}(U)$$

# Co-iteration Semantics: P

$$\{\!\![e]\!\!\}^i_\gamma = [\![e]\!]^i_\gamma \qquad \text{if } kindOf(\mathrm{e}) = \mathsf{D}$$
$$\{\!\![e]\!\!\}^s_\gamma = \lambda s.\ \lambda U.\ \delta_{[\![e]\!]^s_\gamma(s)}(U) \qquad \text{if } kindOf(e) = \mathsf{D}$$

$$\{\!\![\mathsf{sample}(e)]\!\!\}^i_\gamma = [\![e]\!]^i_\gamma$$
$$\{\!\![\mathsf{sample}(e)]\!\!\}^s_\gamma = \lambda s.\ \lambda U.\ let\ \mu, s' = [\![e]\!]^s_\gamma(s)\ in\ \int_T \mu(dv)\ \delta_{v,s'}(U)$$

$$\{\!\![\mathsf{observe}(e_1, e_2)]\!\!\}^i_\gamma = ([\![e_1]\!]^i_\gamma, [\![e_2]\!]^i_\gamma)$$
$$\begin{aligned}\{\!\![\mathsf{observe}(e_1, e_2)]\!\!\}^s_\gamma = {} & \lambda(s_1, s_2).\ \lambda U.\\ & \quad let\ \mu, s_1' = [\![e_1]\!]^s_\gamma(s_1)\ in\\ & \quad let\ v, s_2' = [\![e_2]\!]^s_\gamma(s_2)\ in\ \mu_{\mathrm{pdf}}(v) * \delta_{(),(s_1', s_2)}(U)\end{aligned}$$

$$\left\{\!\!\left[\begin{array}{l} e\ \mathsf{where} \\ \quad \mathsf{rec\ init}\ x_1 = c_1\ \mathsf{and\ init}\ x_2 = c_2 \\ \quad \mathsf{and}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2 \end{array}\right]\!\!\right\}^s_\gamma = \begin{array}{l} \lambda((m_1, m_2), (s_1, s_2), s).\ \lambda U. \\ \quad let\ \gamma_1 = \gamma[m_1/x_1\_\mathsf{last}]\ in\ let\ \gamma_2 = \gamma_1[m_2/x_2\_\mathsf{last}]\ in \\ \quad let\ \mu_1 = \{\!\![e_1]\!\!\}^s_{\gamma_2}(s_1)\ in \\ \quad \int \mu_1(dv_1, ds_1')let\ \gamma_1' = \gamma_2[v_1/x_1]\ in \\ \quad\quad let\ \mu_2 = [\![e_2]\!]^s_{\gamma_1'}(s_2)\ in \\ \quad\quad \int \mu_2(dv_2, ds_2')let\ \gamma_2' = \gamma_1'[v_2/x_2]\ in \\ \quad\quad\quad let\ \mu = \{\!\![e]\!\!\}^s_{\gamma_2'}(s)\ in \\ \quad\quad\quad \int \mu(dv, ds')\ \delta_{v,((\gamma_2'[x_1], \gamma_2'[x_2]),(s_1', s_2'), s')}(U) \end{array}$$

# Co-iteration Semantics: infer

$$\llbracket \mathsf{infer}(e) \rrbracket^i_\gamma = \lambda U.\ \delta_{\llbracket e \rrbracket^i_\gamma}(U)$$

$$\llbracket \mathsf{infer}(e) \rrbracket^s_\gamma = \lambda \sigma.\ let\ \mu = \lambda U.\ \frac{\int_S \sigma(ds) \{\!\!\{ e \}\!\!\}^s_\gamma(s)(U)}{\int_S \sigma(ds) \{\!\!\{ e \}\!\!\}^s_\gamma(s)(\top)}\ in\ (\pi_{1*}(\mu), \pi_{2*}(\mu))$$

- The state of **infer** is a distribution
- At each step **infer** compute a distribution of results, and a distribution of states
- Free variables in $e$ capture input from deterministic processes
- The distribution of results can be used by other deterministic processes
- The distribution of state is used for the next step

*Inference-in-the-loop*

# Inference

# Particle Filtering

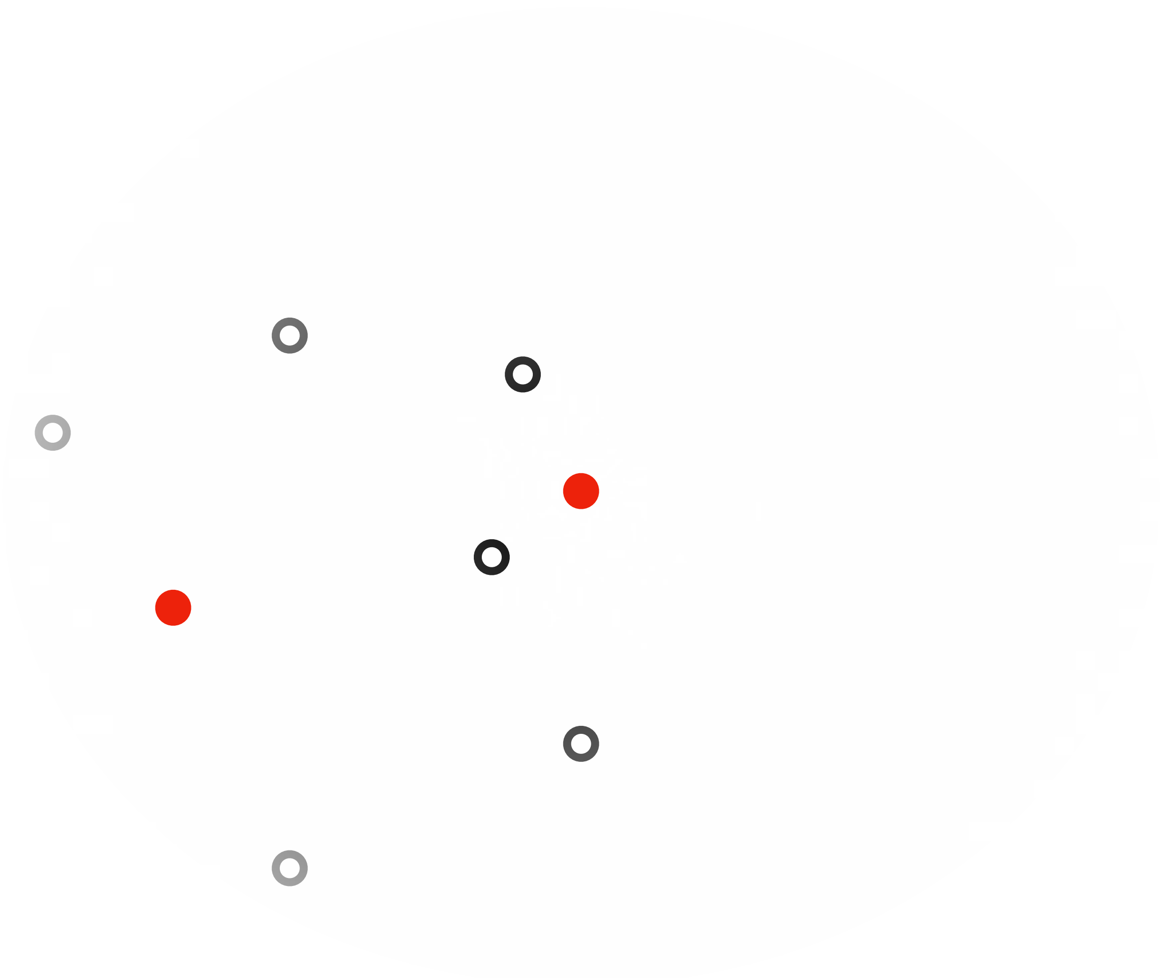
Launch N particles. At each step:

- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution

# Particle Filtering

Launch N particles. At each step:

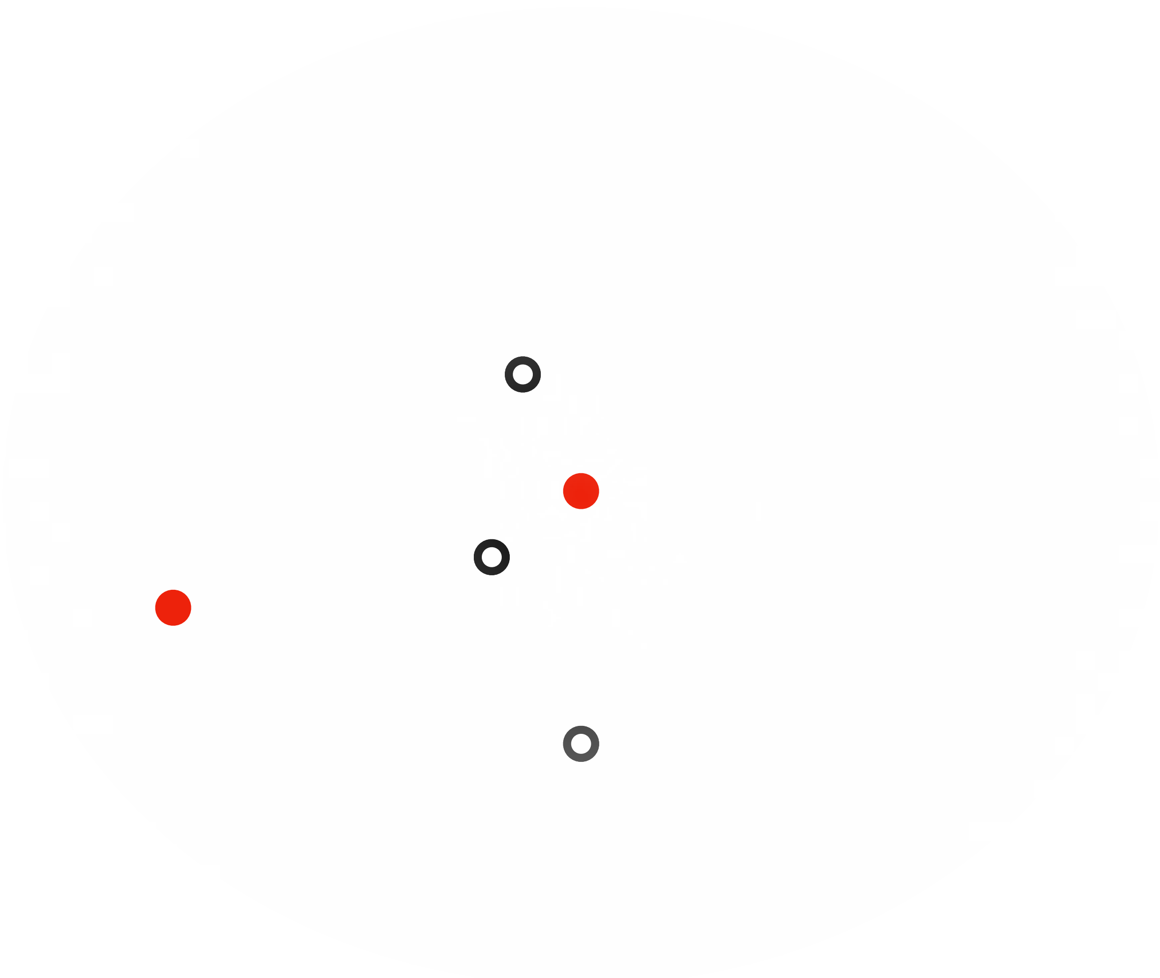
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

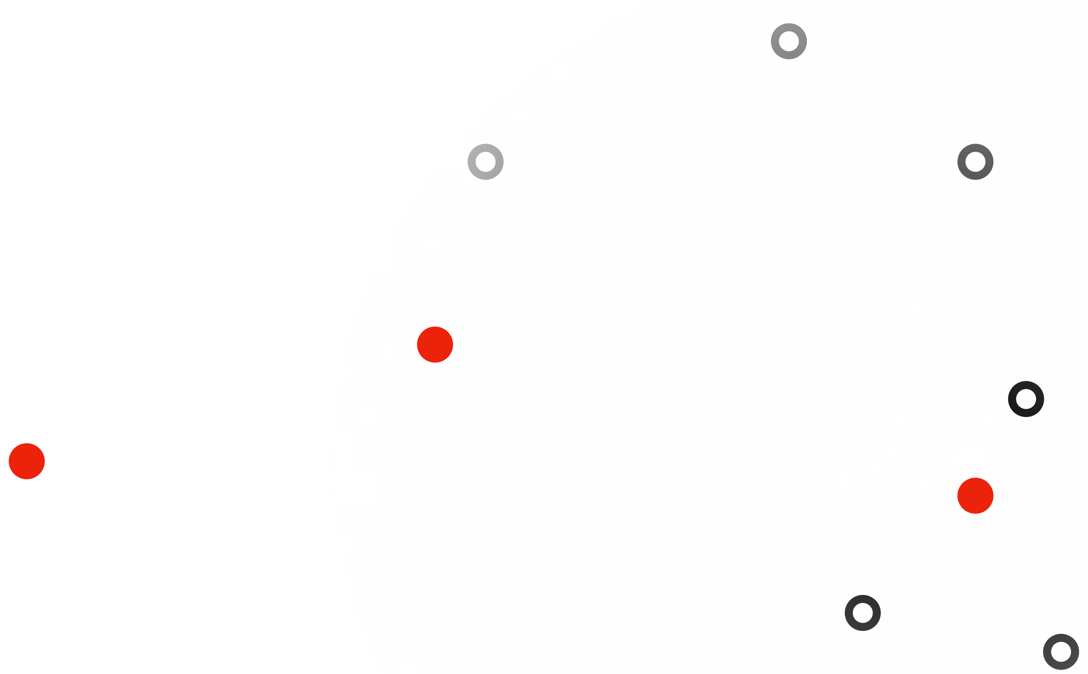
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

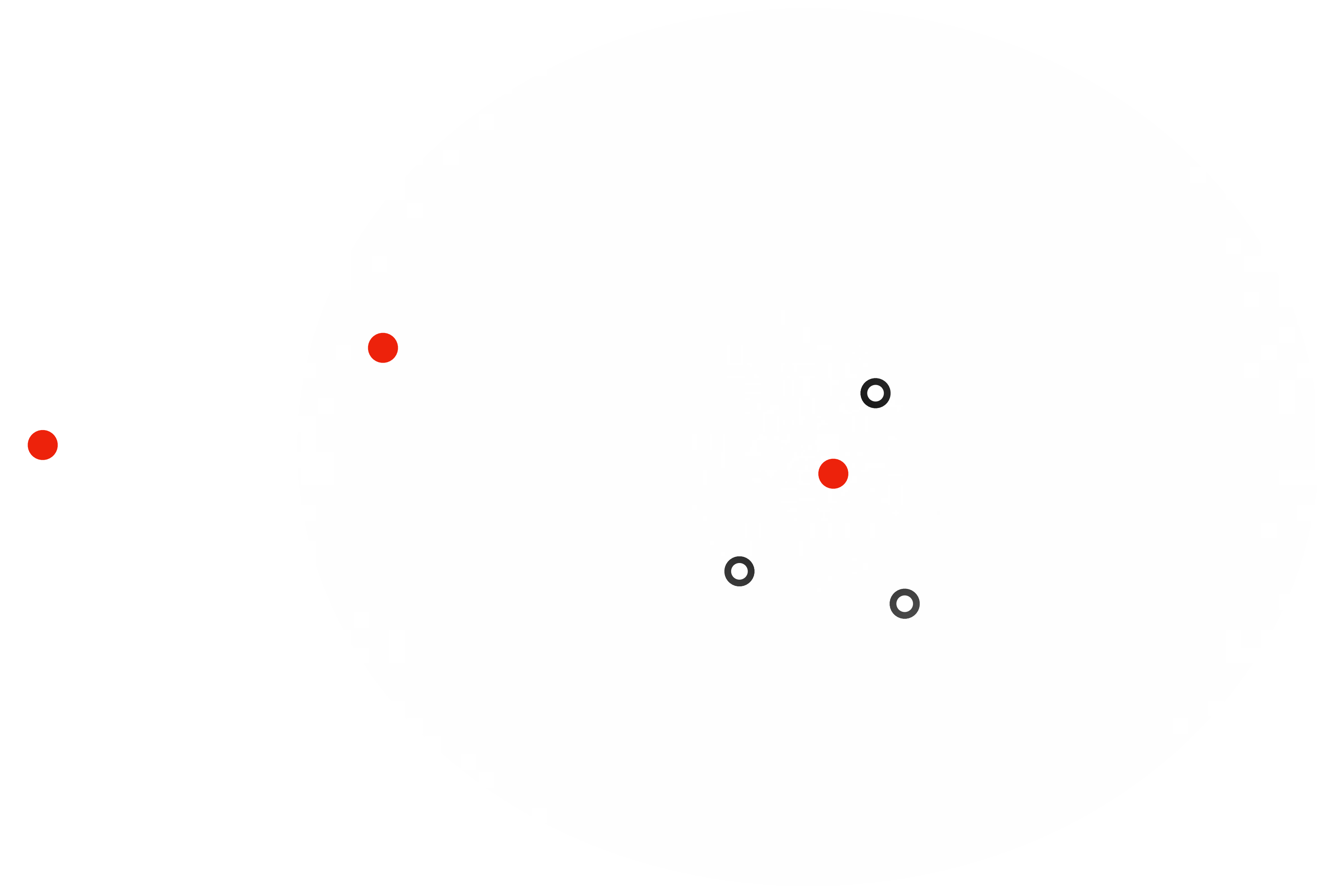
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

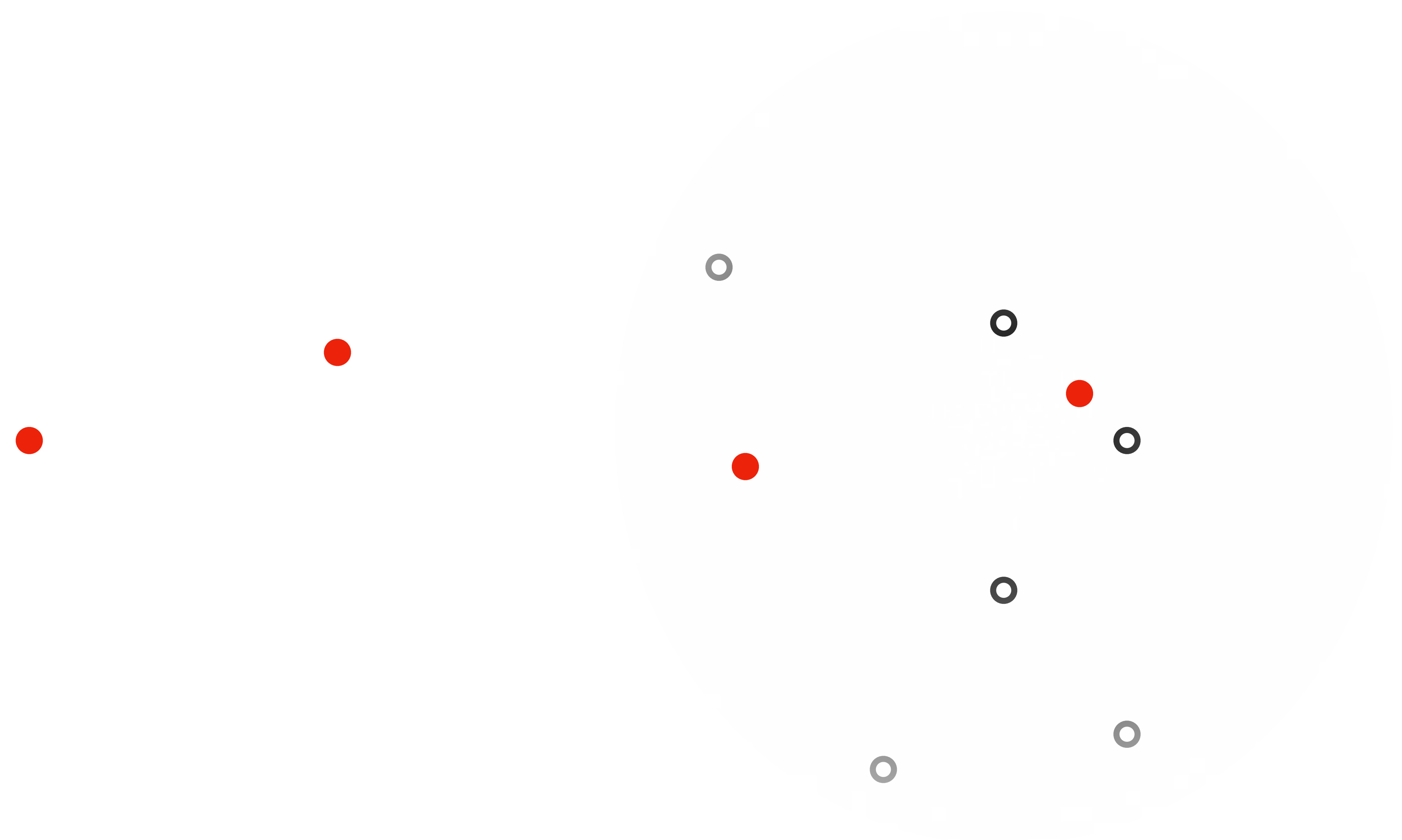
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

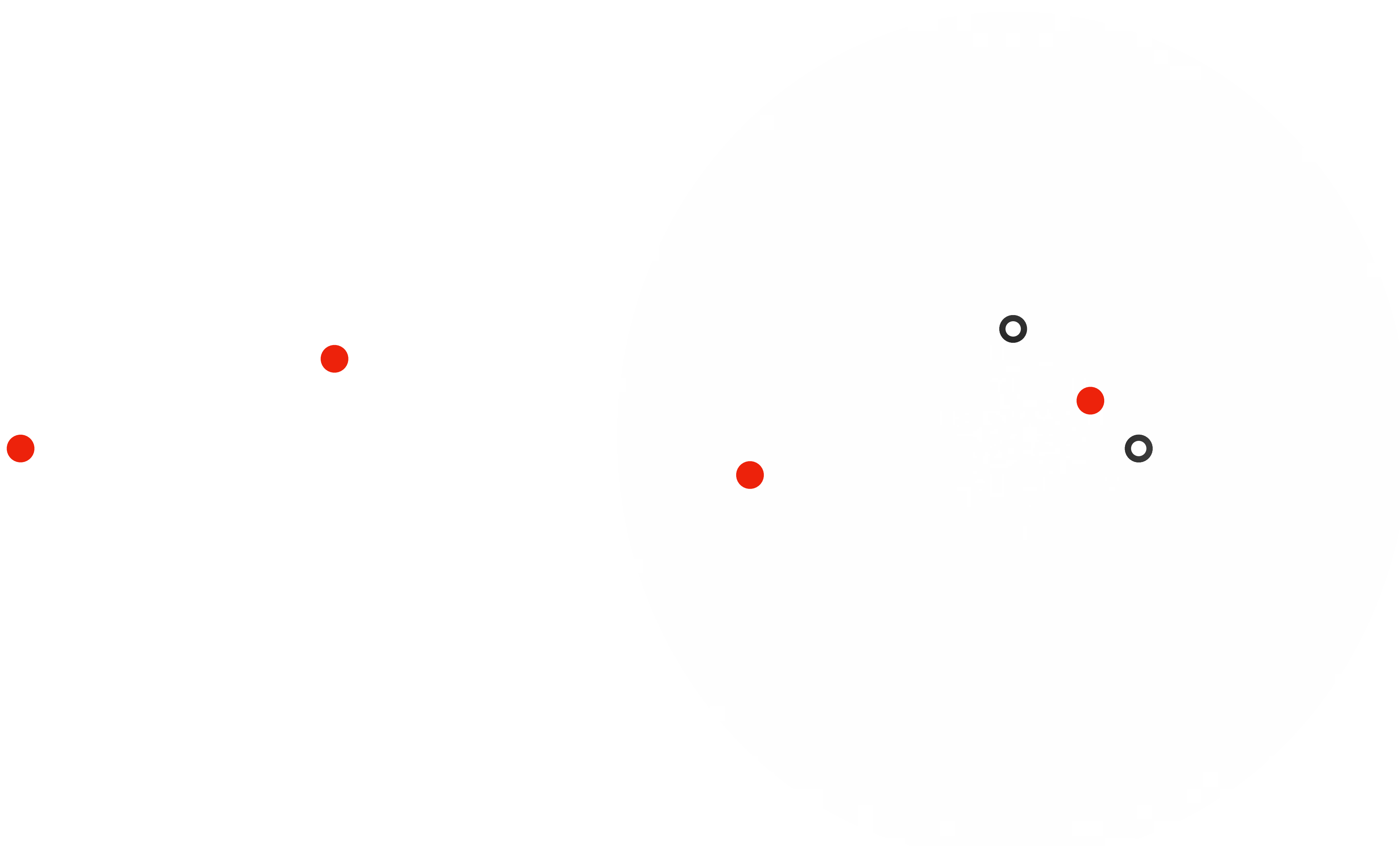
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

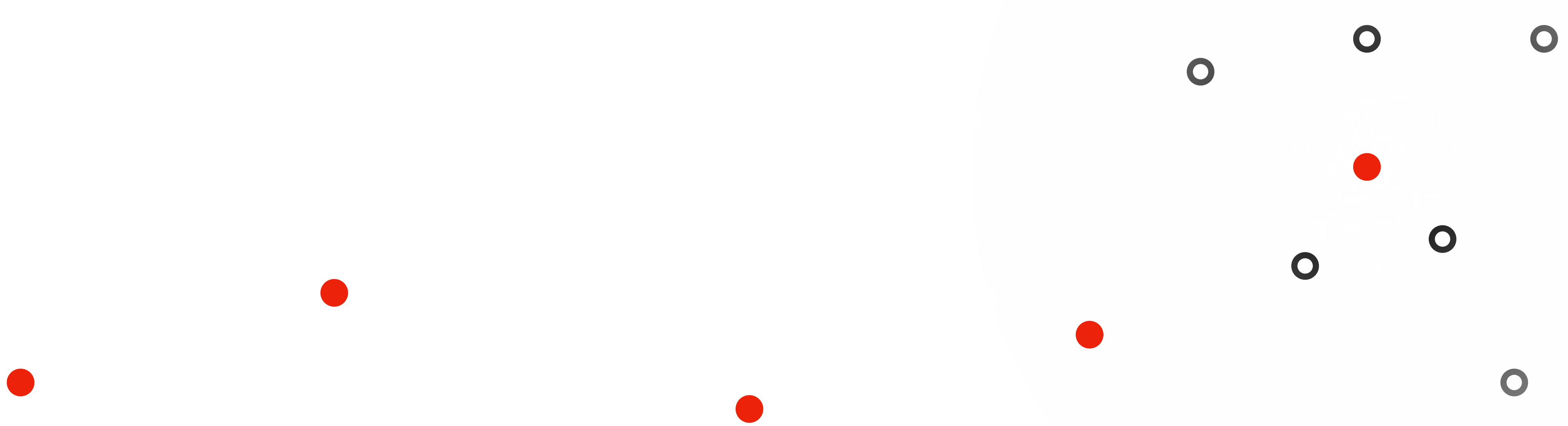
- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Particle Filtering

Launch N particles. At each step:

- Each particle generate pairs (result, score) with an importance score
- Normalize the pairs based on the score
- Re-sample a new set of particles from this distribution



# Delayed Sampling

Particle filter + symbolic computations

- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise

[Murray et al. 2018]

# Delayed Sampling

Particle filter + symbolic computations

- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise

# Delayed Sampling

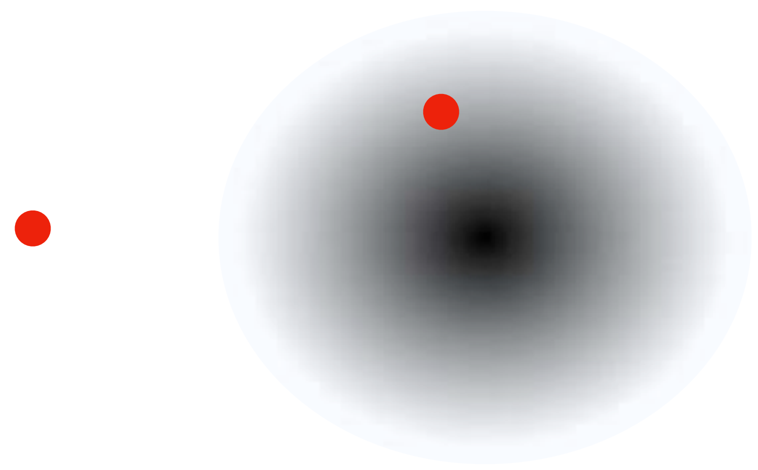
Particle filter + symbolic computations

- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise

# Delayed Sampling

Particle filter + symbolic computations

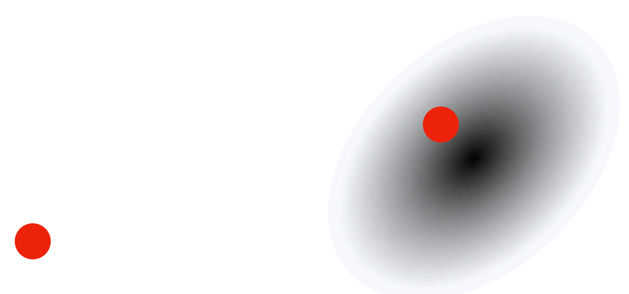
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

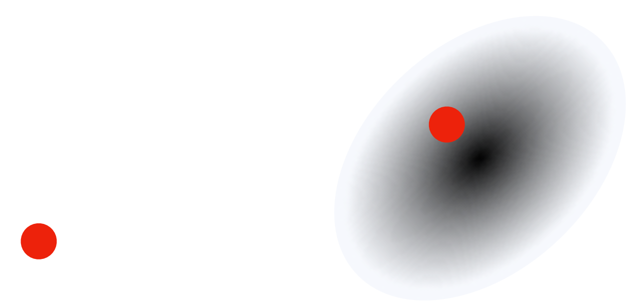
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

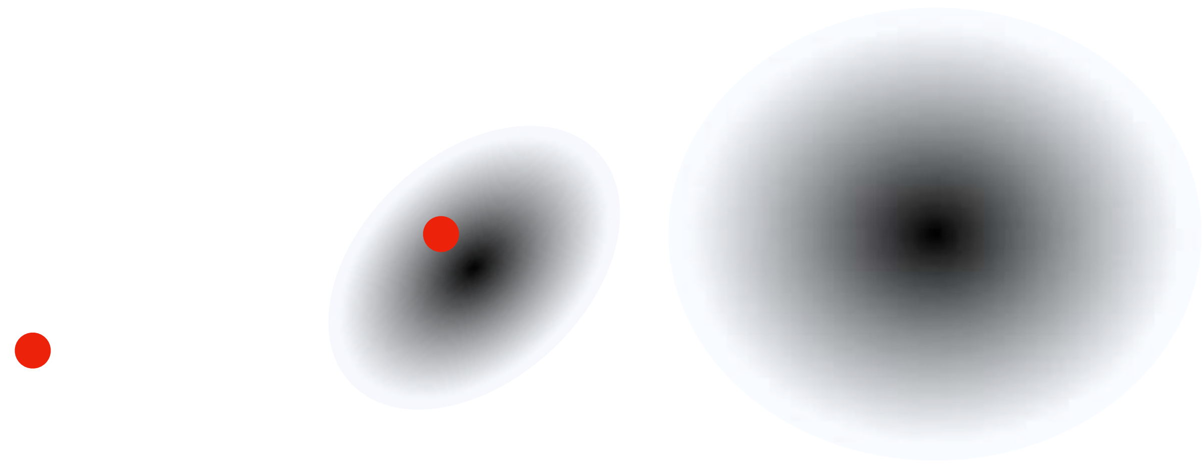
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

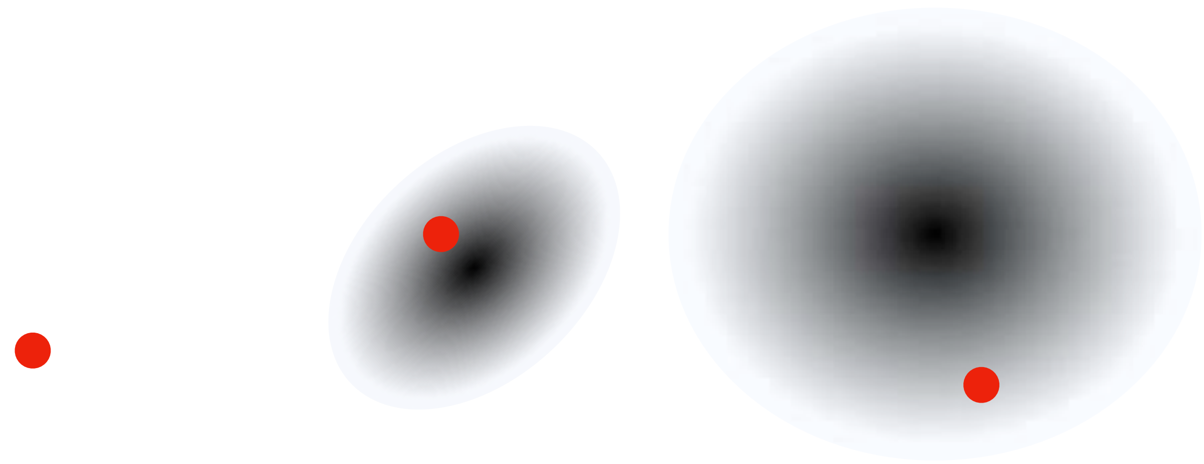
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

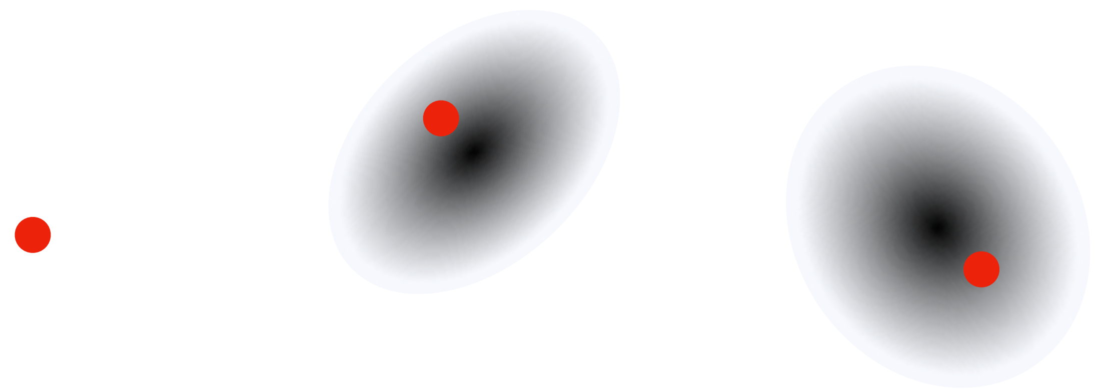
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

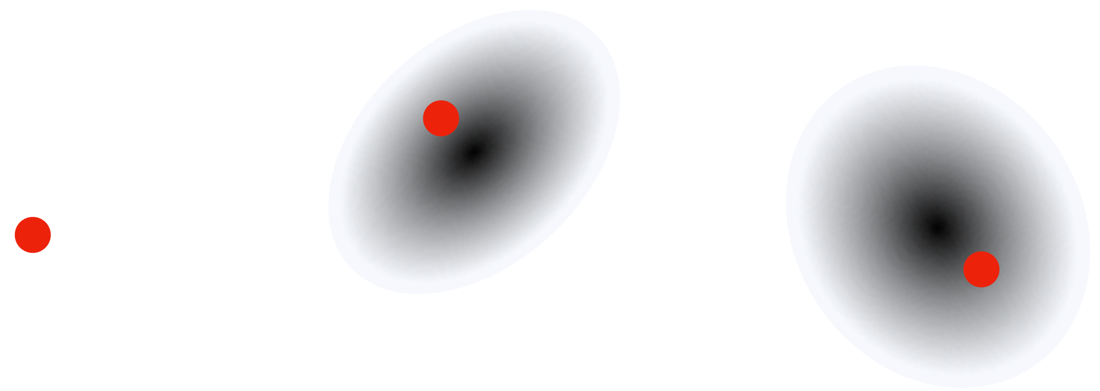
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

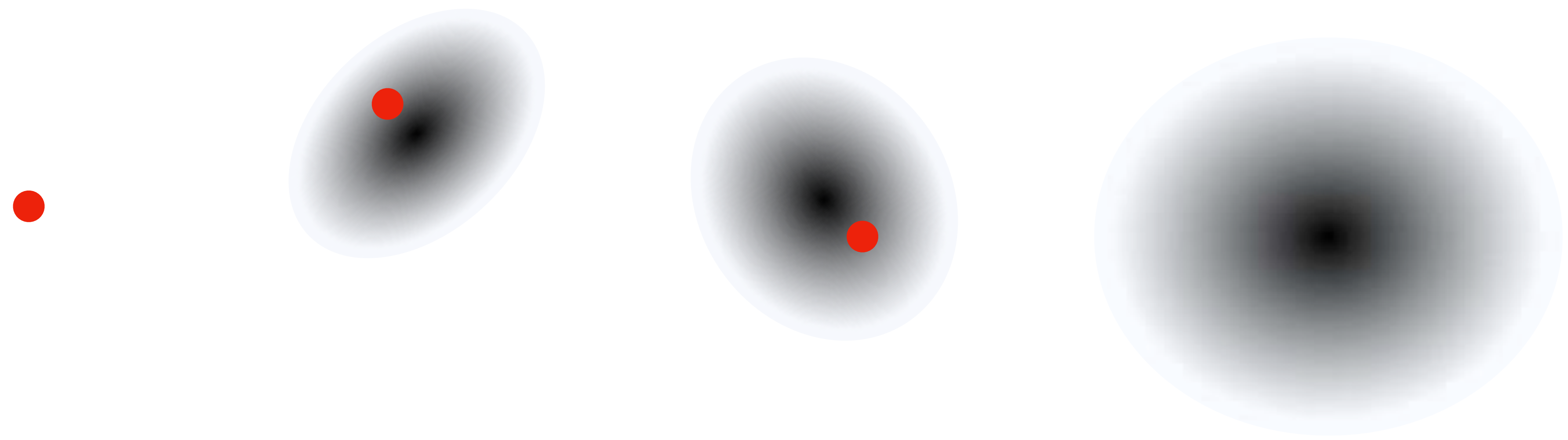
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

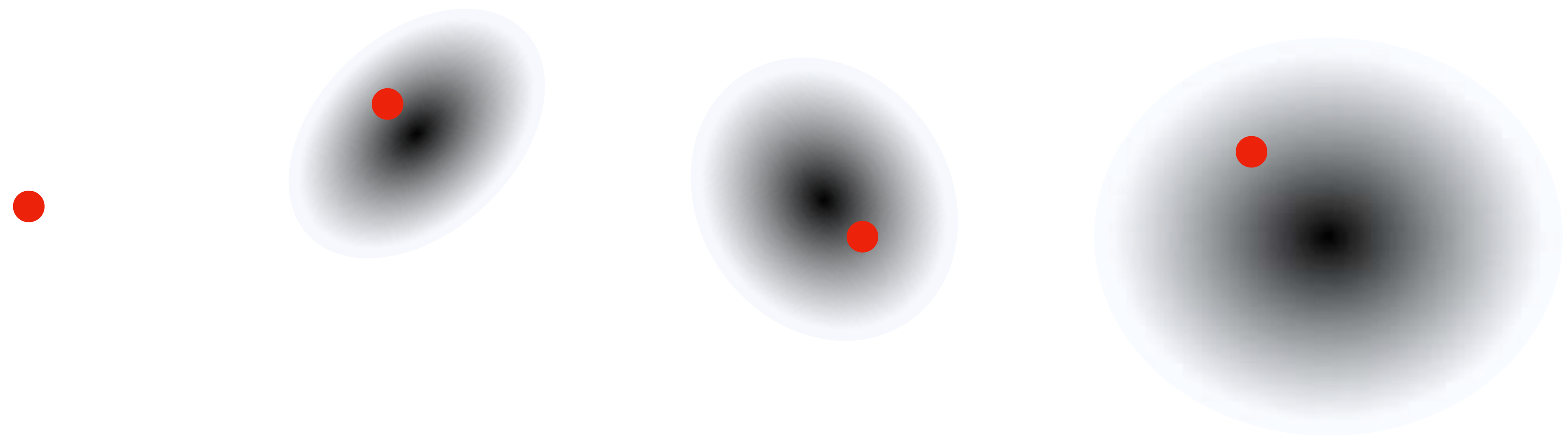
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

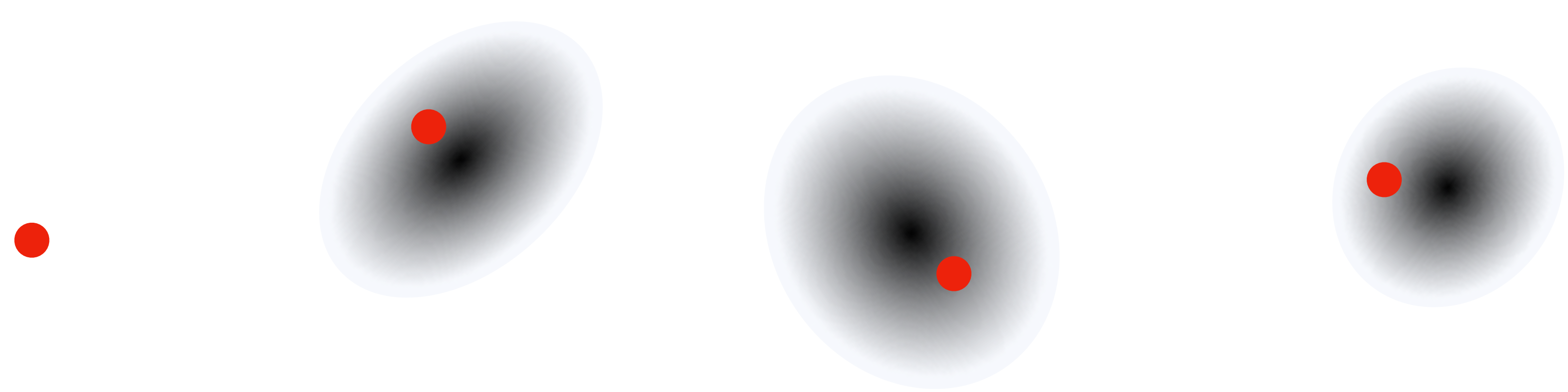
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

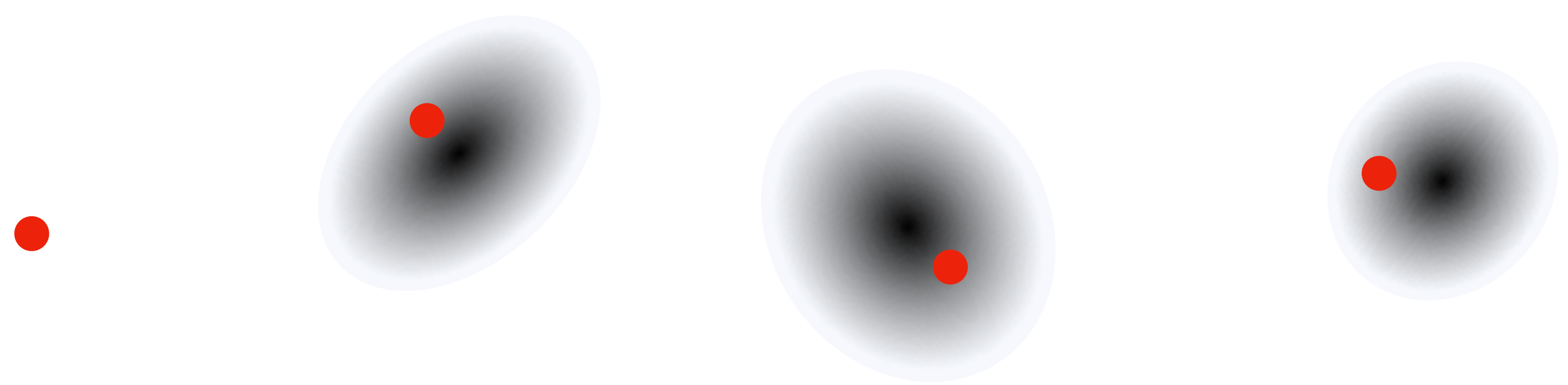
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

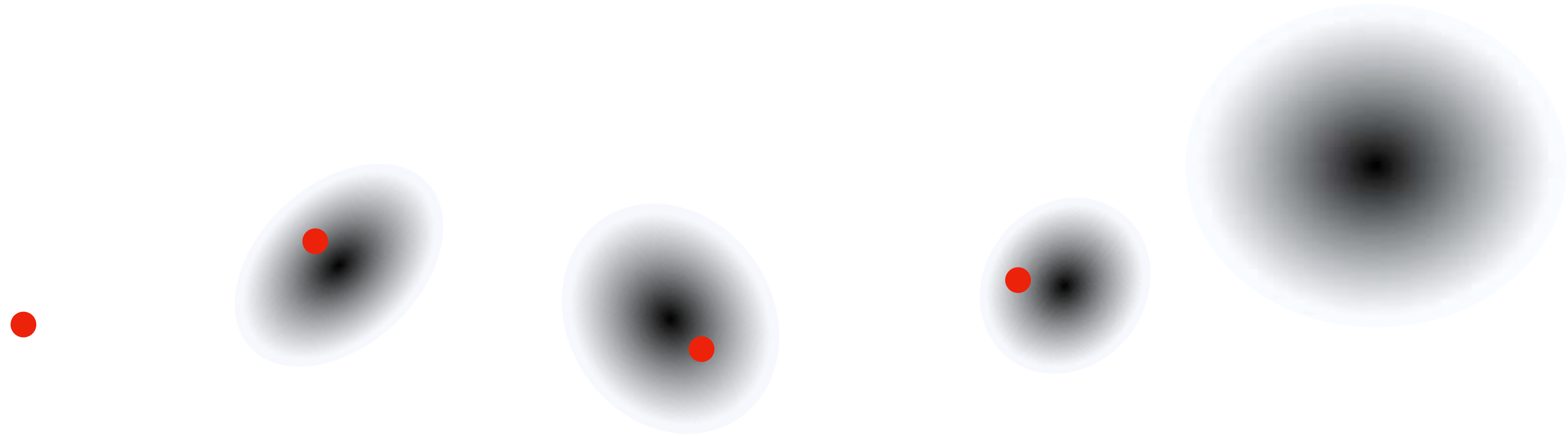
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

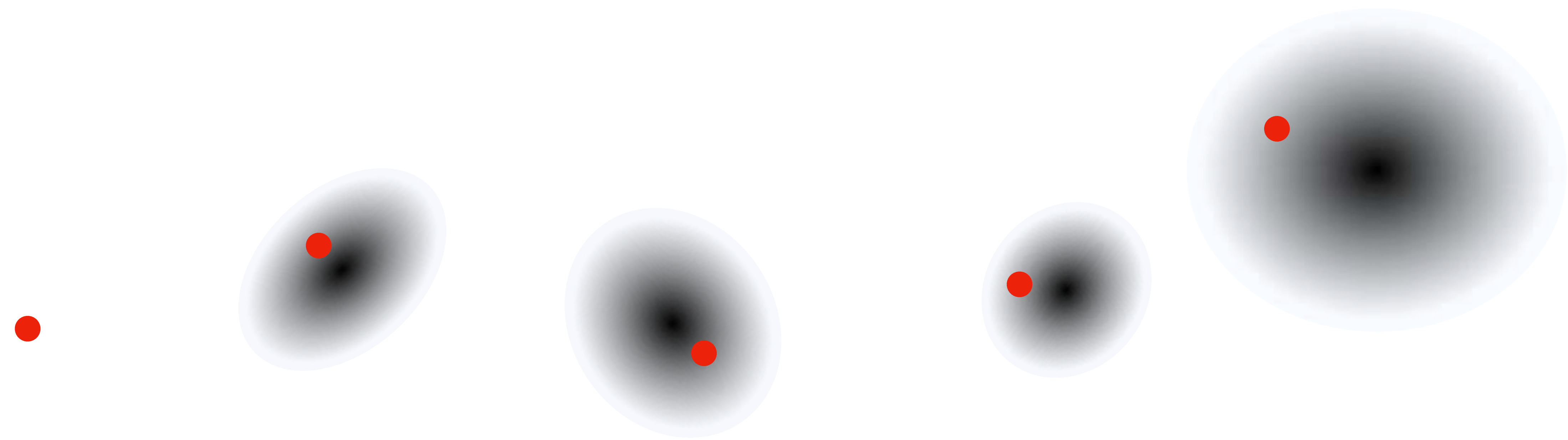
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

Particle filter + symbolic computations

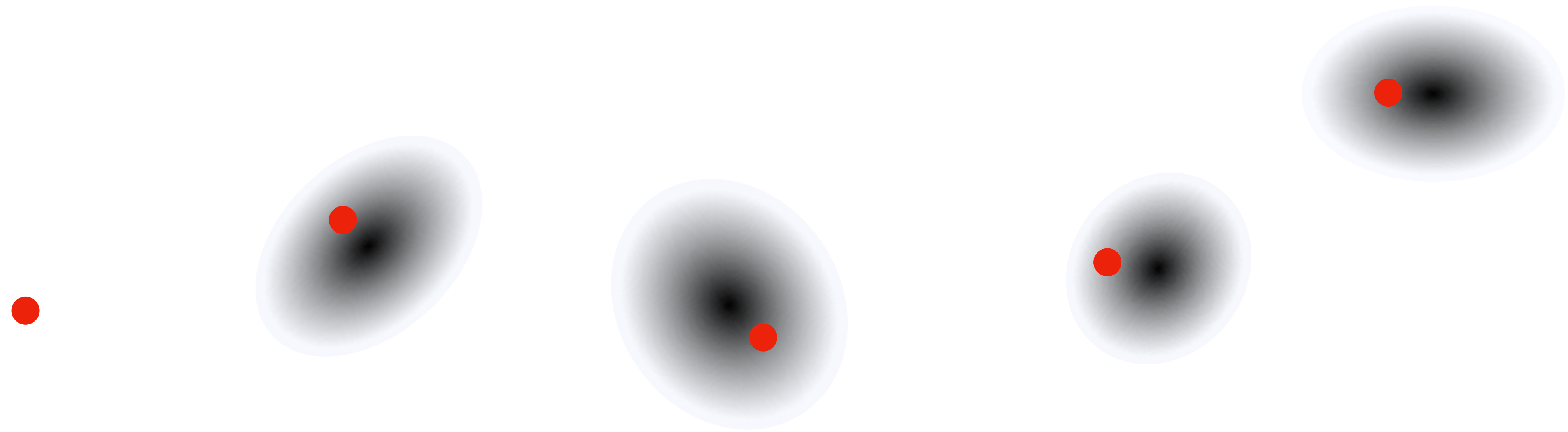
- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise



# Delayed Sampling

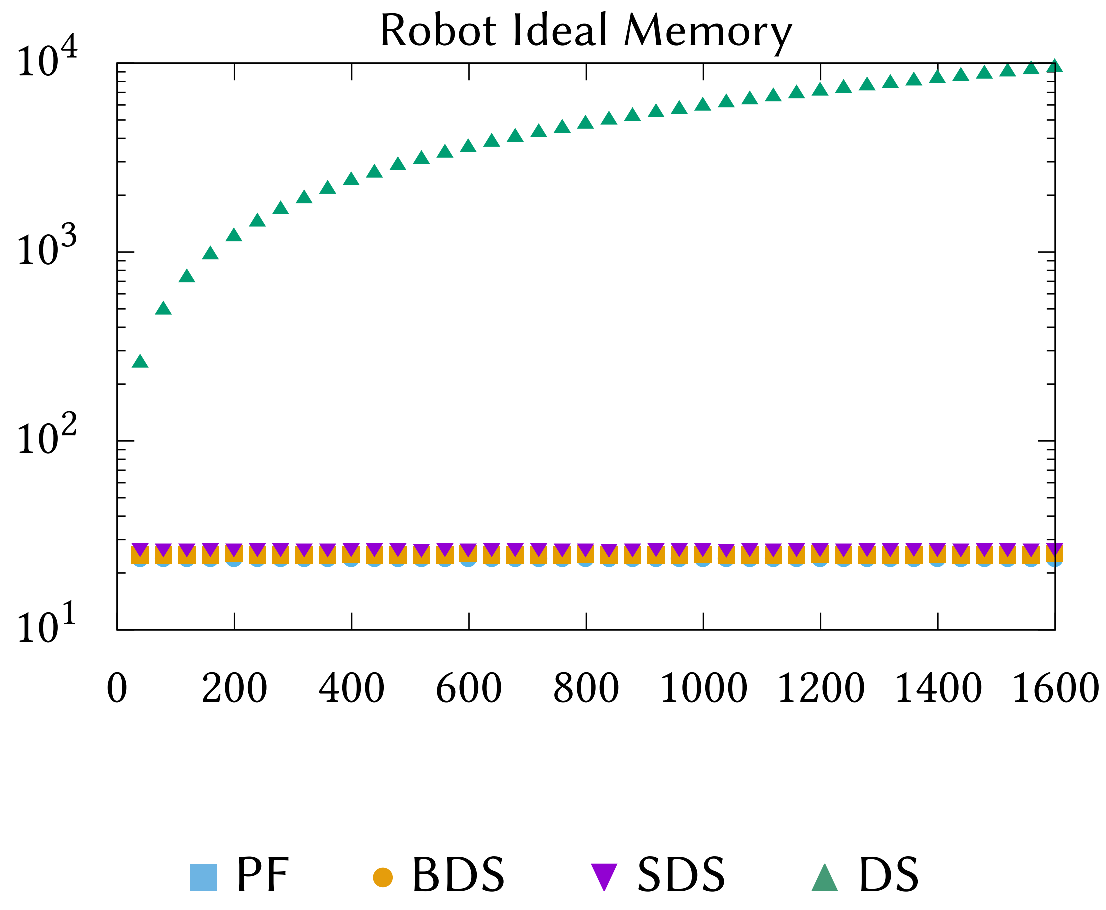
Particle filter + symbolic computations

- Exploit relations between random variables to maintain a *Bayesian network*
- Observation can be incorporated by analytically *conditioning* the network
- Exact solution if possible, default to particle filtering otherwise

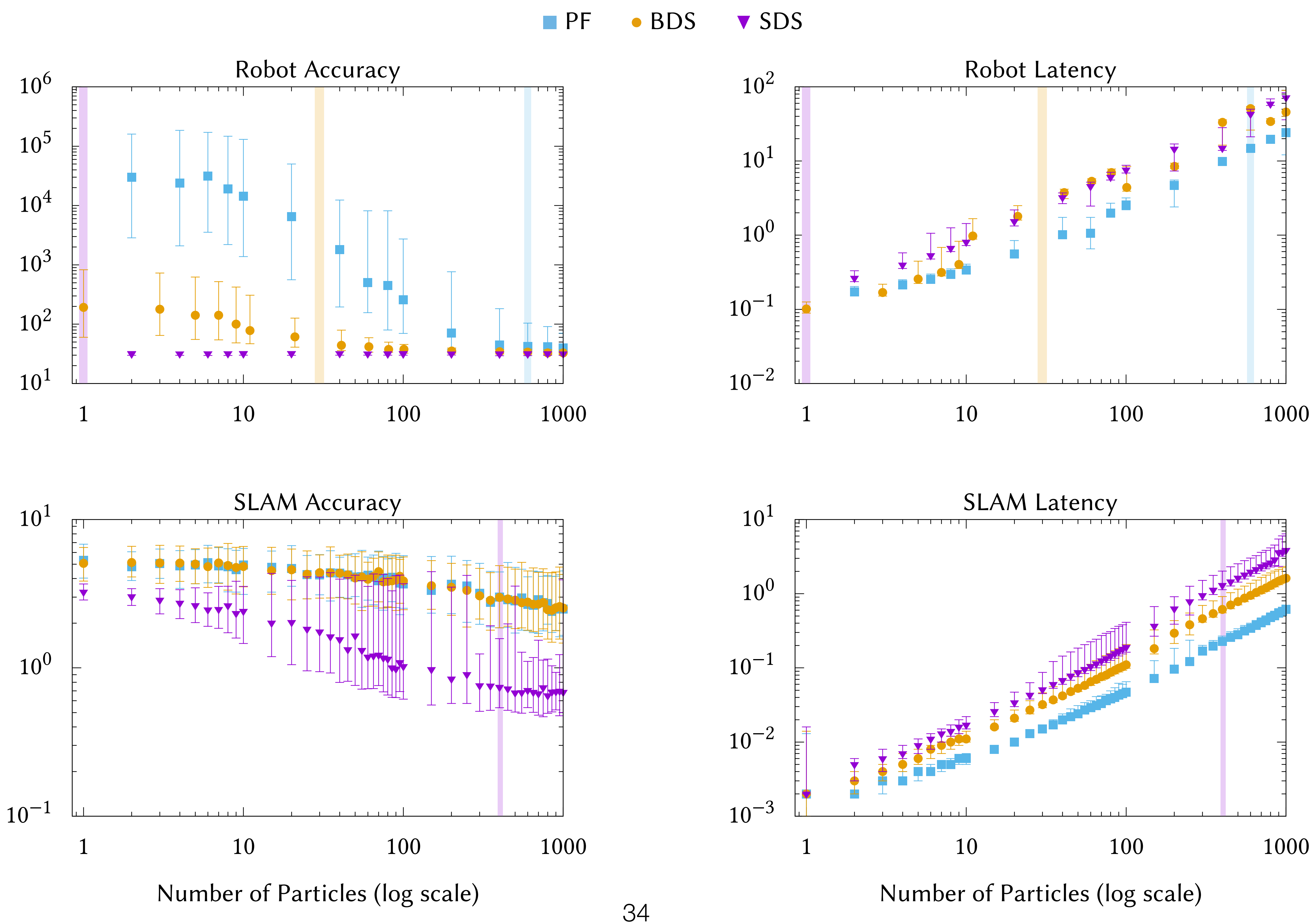


# Streaming Delayed Sampling

- Problem: the size of the network is linear in the number of samples
- Novel implementations (SDS, and BDS) run in bounded memory

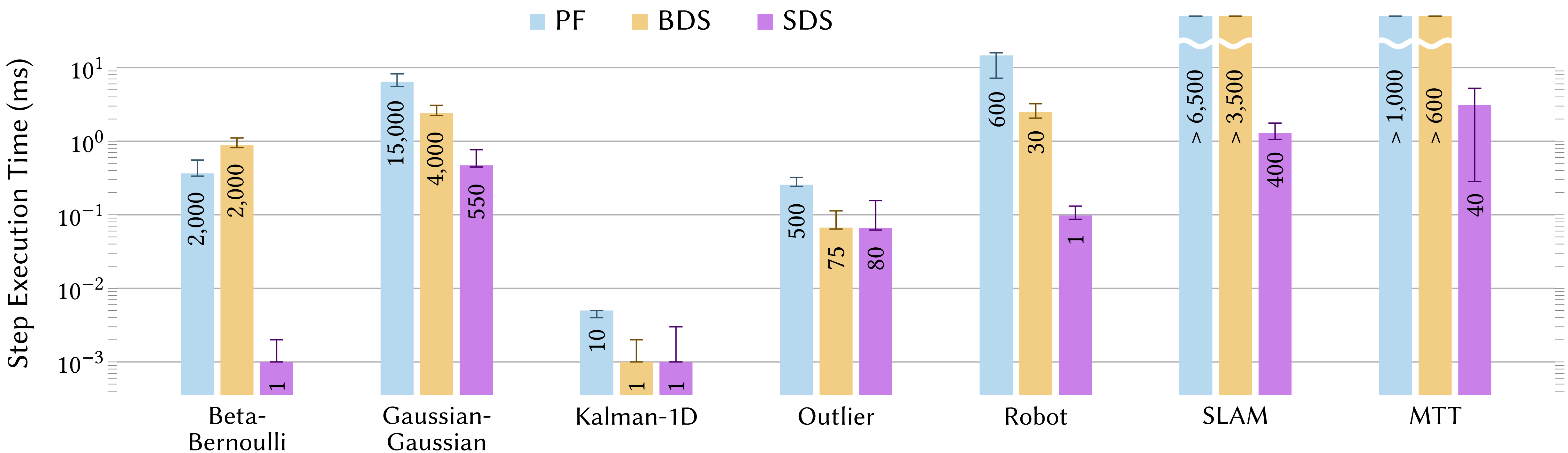


# Streaming Delayed Sampling



# Streaming Delayed Sampling

- Benchmarks illustrate: fixed parameters, trajectory, inference-in-the-loop
- Baseline: accuracy of SDS with 500 particles
- Latency of the inference algorithms to reach comparable accuracy



# Conclusion

ProbZelus

- Synchronous language extended with probabilistic constructs
- Inference-in-the-loop
- Efficient streaming inference algorithms

Design, Semantics, Compilation

- Type system to discriminate deterministic and probabilistic processes
- Measure-based co-iterative semantics
- Semantics preserving compilation scheme

Streaming inference

- Adapt particle filtering and delayed sampling to run on stream processors
- Streaming delayed sampling implementation that run in bounded memory