Filtrer sans s'appauvrir

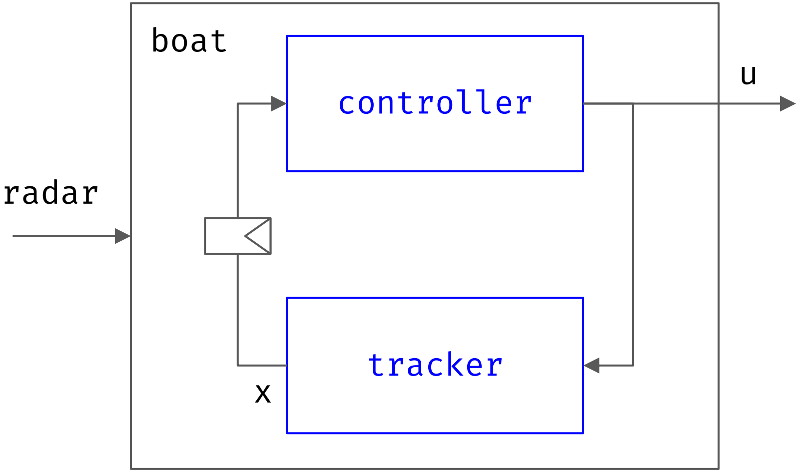
Inférer les paramètres constants des modèles réactifs probabilistes

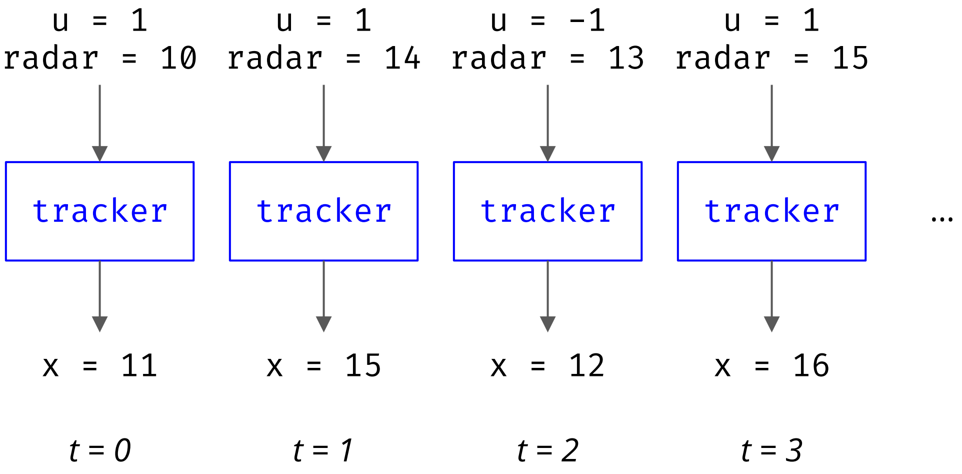
Guillaume Baudart, **Grégoire Bussone**, Louis Mandel, Christine Tasson

Synchronous Programming

Synchronous data-flow languages

- Inputs/Outputs: streams of values
- Functions: stream processors





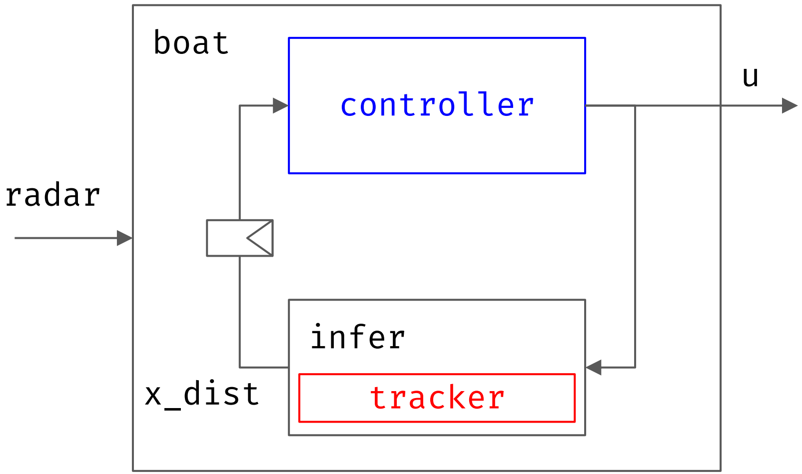
Synchronous Probabilistic Programming

Synchronous data-flow languages

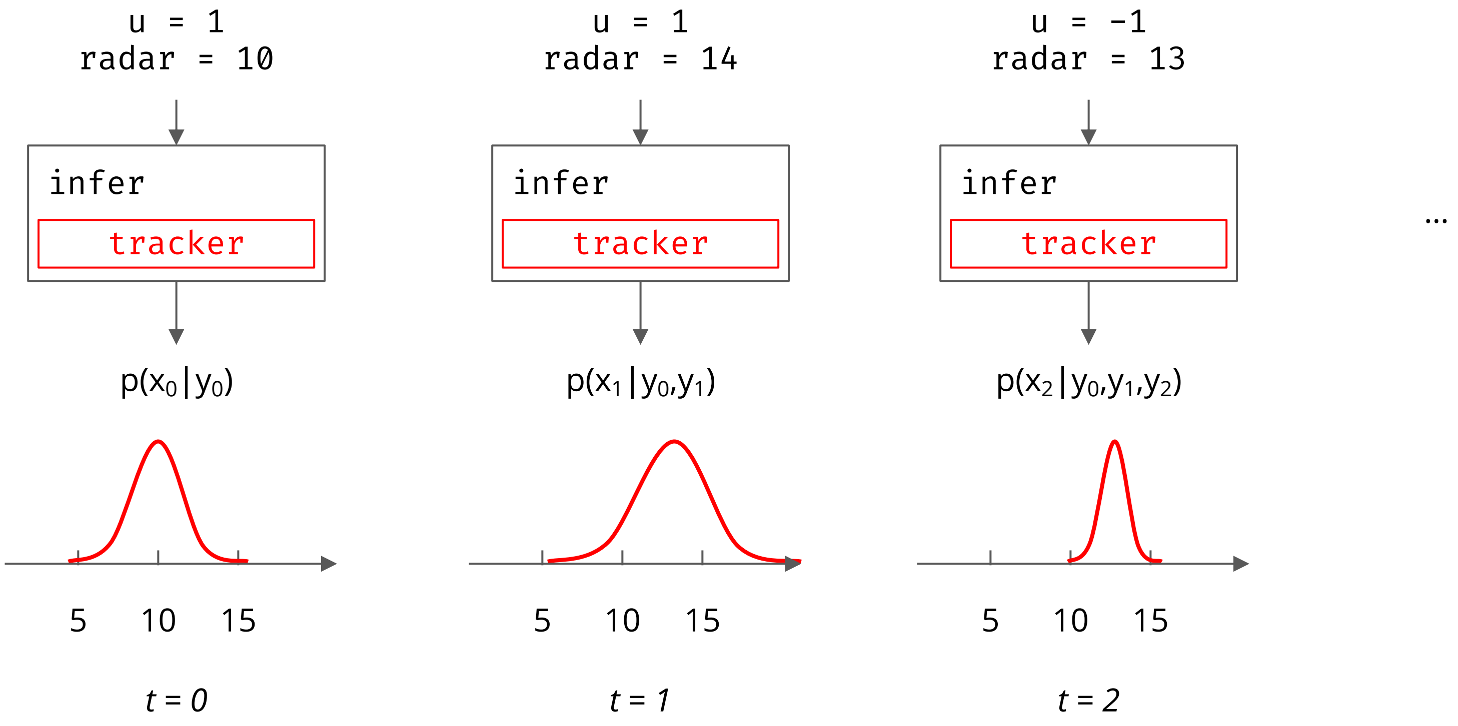
- Inputs/Outputs: streams of values
- Functions: stream processors

ProbZelus: add support to deal with uncertainty

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



Synchronous Probabilistic Programming



Bayesian Inference

From prior to posterior distribution

$$\underbrace{\mathbb{P}(params|data)}_{\text{Posterior}} \propto \underbrace{\mathbb{P}(params)}_{\text{Prior}}\underbrace{\mathbb{P}(data|params)}_{\text{Likelihood}}$$

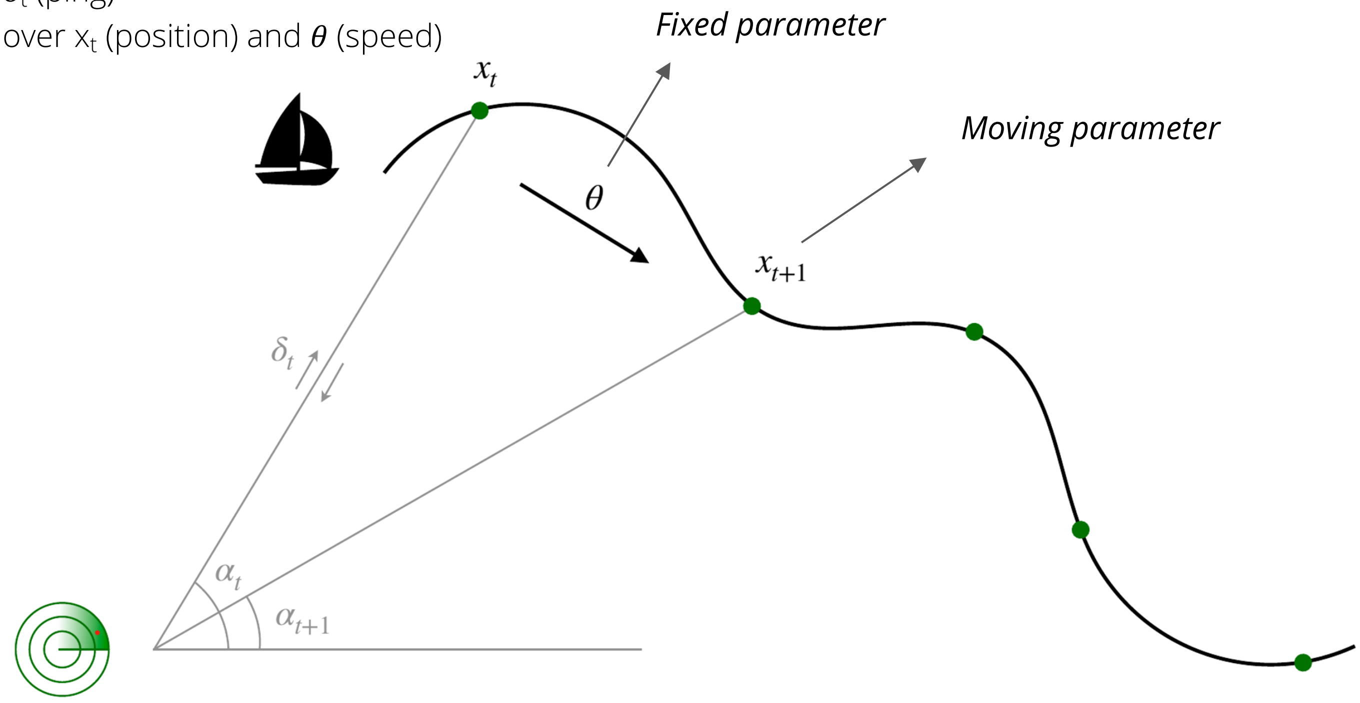
Probabilistic constructs

```
● sample d          (* Prior: sample from distribution d                 *)
● observe (d, y)    (* Condition: assume y was sampled from distribution d *)
● infer model y     (* Infer: compute the posterior of model given y      *)
```

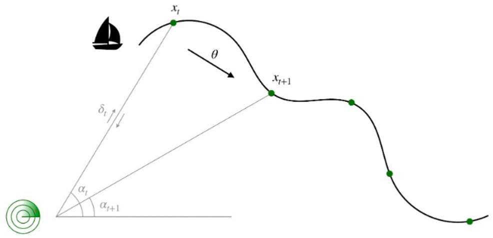
Example: Boat

Use a radar to infer the position of a boat

- Inputs: α_t (angle) and δ_t (ping)
- Outputs: distribution over x_t (position) and θ (speed)



Example: Boat

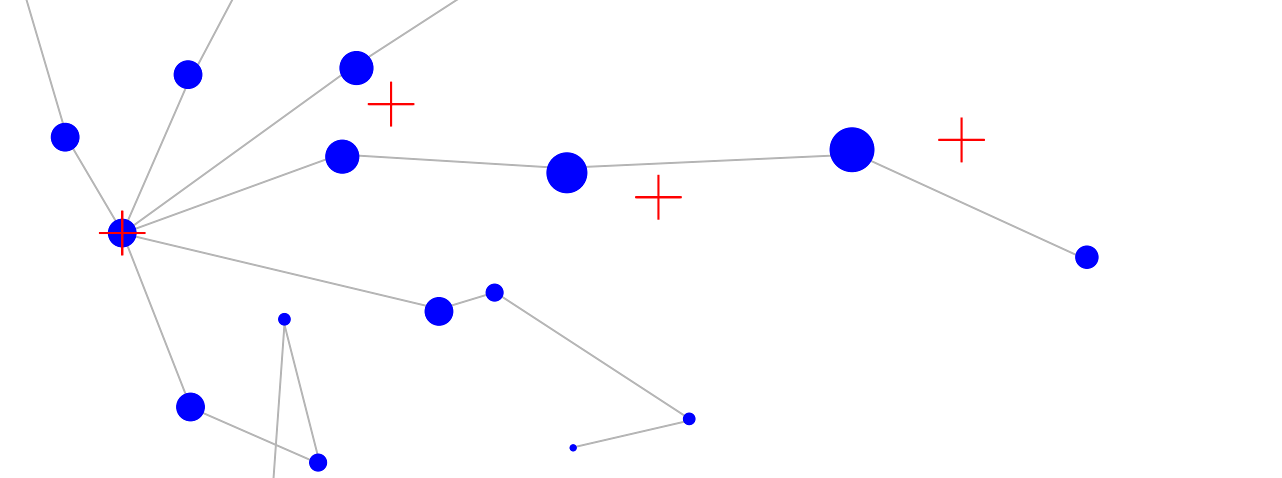



```
proba radar (delta, alpha) = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 -> sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)

node main (delta, alpha) = d_theta, d_x where
  rec d = infer radar (delta, alpha)
  and d_theta, d_x = Dist.split d
```

Approximate Inference: Basic Monte Carlo

Importance Sampling

- Run a set of n independent executions, called particles
- Sample: draw a sample from a distribution
- Observe: associate a score to the current execution
- Gather output values and scores to approximate the posterior distribution



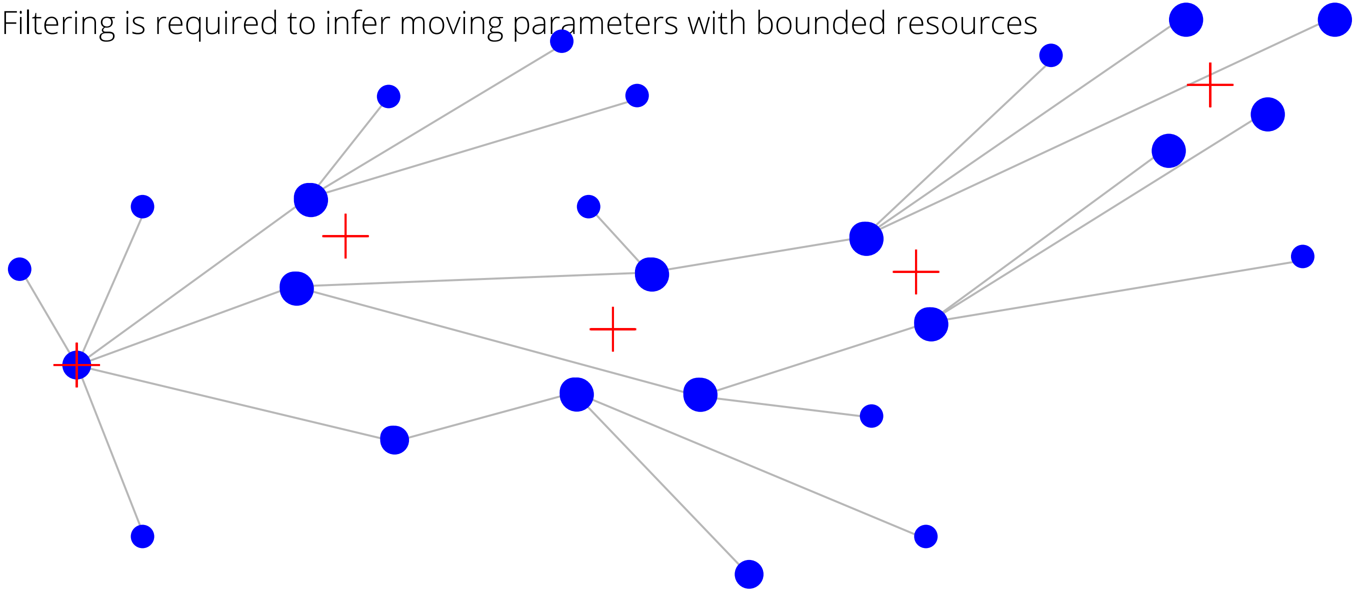
Bad estimation

E.g., estimate the boat position from noisy measurements

Approximate Inference: Sequential Monte Carlo

Filtering: Duplicate most significant particles, kill least significant particles

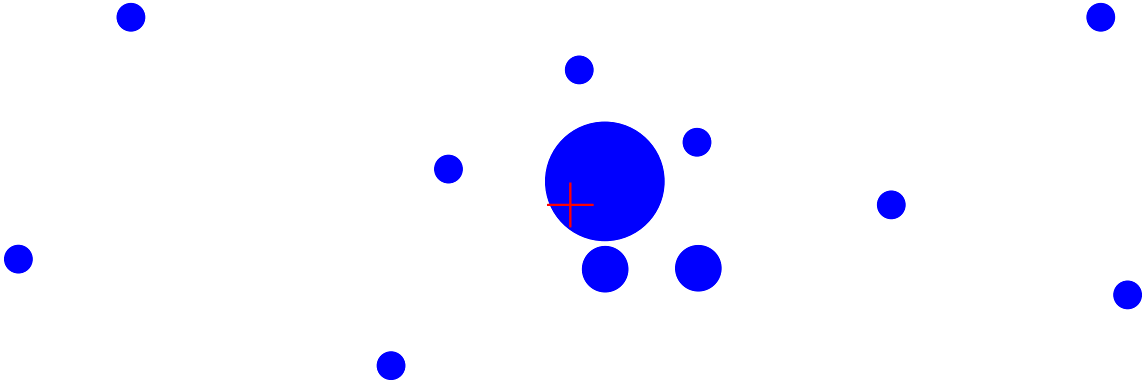
- Recenter inference on the most significant estimations
- Filtering is required to infer moving parameters with bounded resources



Better estimation

E.g., estimate the boat position from noisy measurements

Particle Impoverishment



Filtering is problematic for fixed parameters.
Information loss at each filtering step

E.g., estimate the boat drift (fixed parameter)

Assumed Parameter Filter

Each particle maintains a distribution of fixed parameters. At each step:

- `filter` the set of particles
- `sample` to compute moving parameters
- `update` the distribution of fixed parameters

Input: data y_t, previous particles distribution μ_{t-1}
Output: current particles distribution μ_t

For each particle $i = 1$ to N do

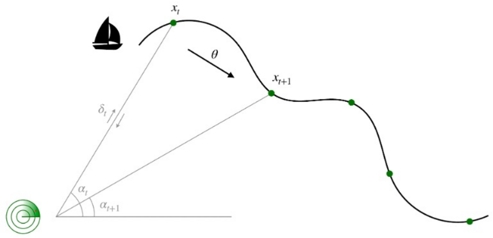
$x^i_{t-1}, \Theta^i_{t-1} = \texttt{sample}(\mu_{t-1})$

$\theta^i = \texttt{sample}(\Theta^i_{t-1})$

$x^i_t, w^i_t = \texttt{model}(y_t \mid \theta^i, x^i_{t-1})$

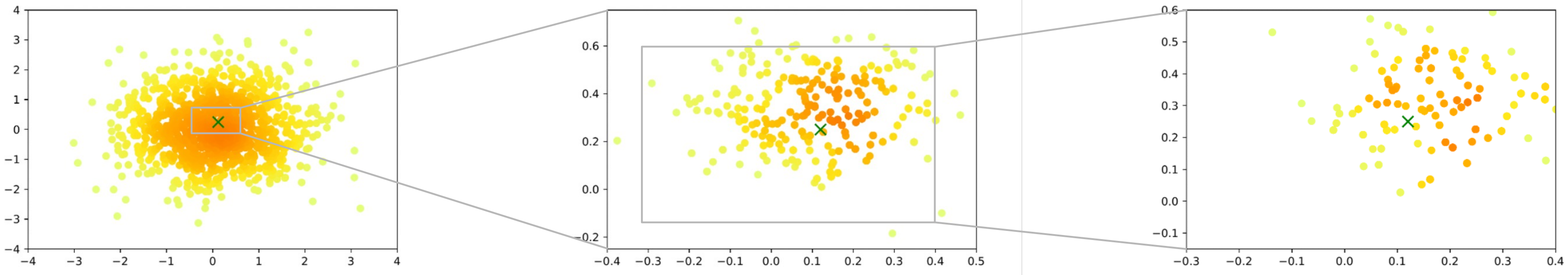
$\Theta^i_t = \texttt{update}(\Theta^i_{t-1}, \lambda\theta.\, \texttt{model}(y_t \mid \theta, x^i_t, x^i_{t-1}))$

$\mu_t = \mathcal{M}(\{w^i_t, (x^i_t, \Theta^i_t)\}_{1 \le i \le N})$

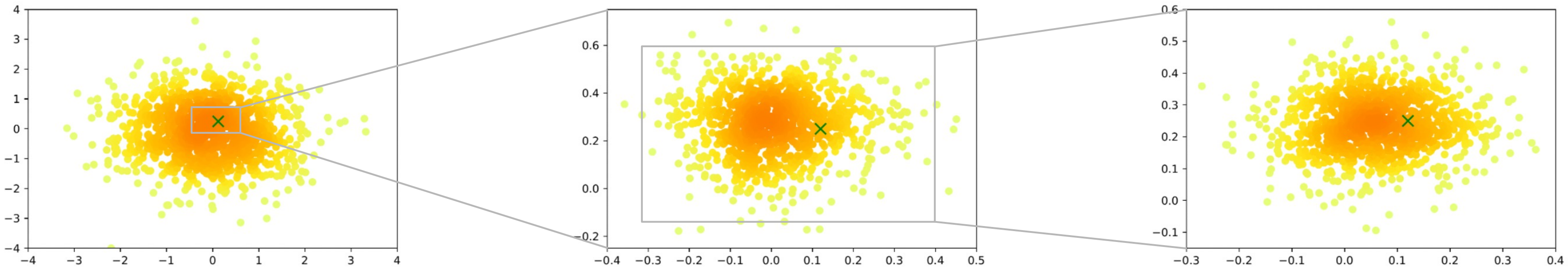


APF Results

Particle filter



APF



θ, t = 0 θ, t = 30 θ, t = 50

No more impoverishment for θ

Problem: APF requires the ability to fix the value of some parameters at runtime

```
Input: data yt, previous particles distribution μt-1
Output: current particles distribution μt

For each particle i = 1 to N do
  | xit-1, Θit-1 = sample(μt-1)
  | θi = sample(Θit-1)
  | xit, wit = model(yt | θi, xit-1)
  | Θit = update(Θit-1, λθ. model(yt | θ, xit, xit-1))
μt = M({wit, (xit, Θit)}1≤i≤N)
```

$x^i_{t-1}, \Theta^i_{t-1}$ = sample(μ_{t-1})

θ^i = sample(Θ^i_{t-1})

x^i_t, w^i_t = model($y_t \mid \theta^i, x^i_{t-1}$)

Θ^i_t = update($\Theta^i_{t-1}, \lambda\theta.$ model($y_t \mid \theta, x^i_t, x^i_{t-1}$))

$\mu_t = \mathcal{M}(\{w^i_t, (x^i_t, \Theta^i_t)\}_{1 \le i \le N})$

Sample θ outside the model

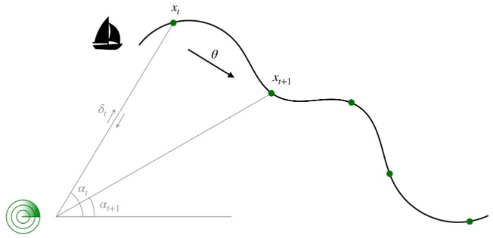
Replay a transition multiple times

Contributions: APF for ProbZelus

- Static analysis to identify fixed parameters
- Compilation pass required by APF
- New APF runtime for ProbZelus

Static Analysis and Compilation

Static Analysis



```
proba radar (delta, alpha) = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 -> sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)

node main (delta, alpha) = d_theta, d_x where
  rec d = infer radar (delta, alpha)
  and d_theta, d_x = Dist.split d
```

$\theta \sim \mathcal{N}(0, I_2)$ fixed
$x_{t+1} \sim \mathcal{N}(x_t\theta, 0.5 I_2)$ moving

Can we define a type system to identify fixed parameters and their priors?

Static Analysis

Extract prior distribution

$$\vdash^{p} \texttt{sample}(e_d) : e_d$$

Only depends on globals

$$C \vdash^{l} e_d$$

Global constants

Extract fixed parameters

$$C \vdash e : \xi$$

Parameters map

Global constants

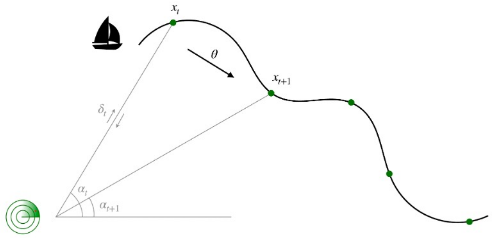
New global constants

$$H, C \vdash d : H', C'$$

Parameters map

Declaration

New parameters map



Boat: Static Analysis

```
proba radar (delta, alpha) = th
  rec init theta = sample(mv_ga
  and x = x0 -> sample(mv_gauss
  and d = sqrt((vec_get x 0) **
  and a = atan(vec_get x 1 /. v
  and () = observe(gaussian(2.
  and () = observe(gaussian(a,

node main (delta, alpha) = d_th
  rec d = infer radar (delta, alpha)
  and d_theta, d_x = Dist.split d
```

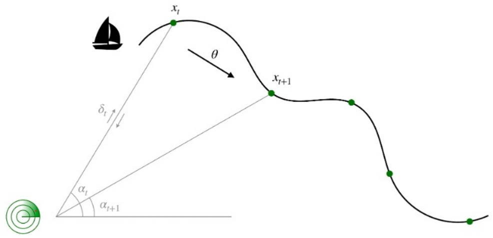
$$\vdash^p \texttt{sample}(\mathcal{N}(0, I_2)) : \mathcal{N}(0, I_2)$$
$$\vdash^p \texttt{sample}(\mathcal{N}(\texttt{pre x} + \theta, 0.5 I_2)) : \mathcal{N}(\texttt{pre x} + \theta, 0.5 I_2)$$

$$\{0, I_2\} \vdash^l \mathcal{N}(0, I_2)$$
$$\{0, I_2\} \not\vdash^l \mathcal{N}(\texttt{pre x} + \theta, 0.5 I_2)$$

$$\{0, I_2\} \vdash \texttt{theta, x where ...} : \{\texttt{theta} \leftarrow \mathcal{N}(0, I_2)\}$$

$$\emptyset, \emptyset \vdash \texttt{prog} : \{\texttt{radar} \leftarrow \{\texttt{theta} \leftarrow \mathcal{N}(0, I_2)\}\}, \{0, I_2\}$$

Boat: Compilation

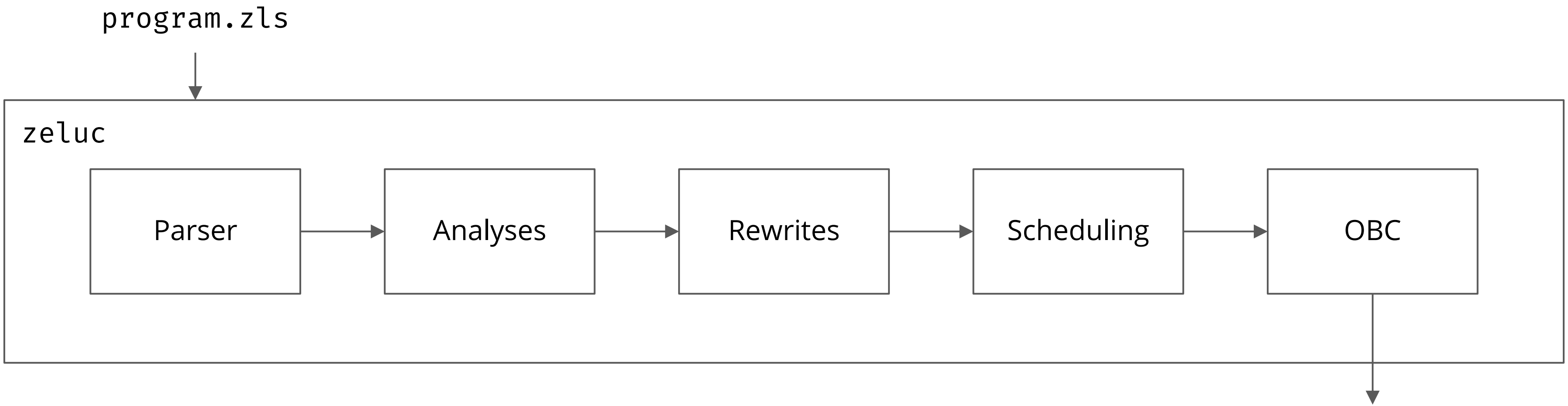


```
let radar_prior = mv_gaussian(zeros, i2)                         (* Define priors *)

proba radar (delta, alpha) = theta, x where
proba radar_model (theta, (delta, alpha)) = theta, x where       (* Add inputs *)
  rec init theta = sample(mv_gaussian(zeros, i2))                (* Remove definitions *)
  rec x = x0 -> sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)

node main (delta, alpha) = d_theta, d_x where
  rec d = infer radar (delta, alpha)
  rec d = APF.infer radar_prior radar_model (delta, alpha)      (* Change inference *)
  and d_theta, d_x = Dist.split d
```

$$\boxed{\emptyset, \emptyset \vdash \texttt{prog} : \{\texttt{radar} \leftarrow \{\texttt{theta} \leftarrow \mathcal{N}(0, I_2)\}\}, \{0, I_2\}}$$

Runtime

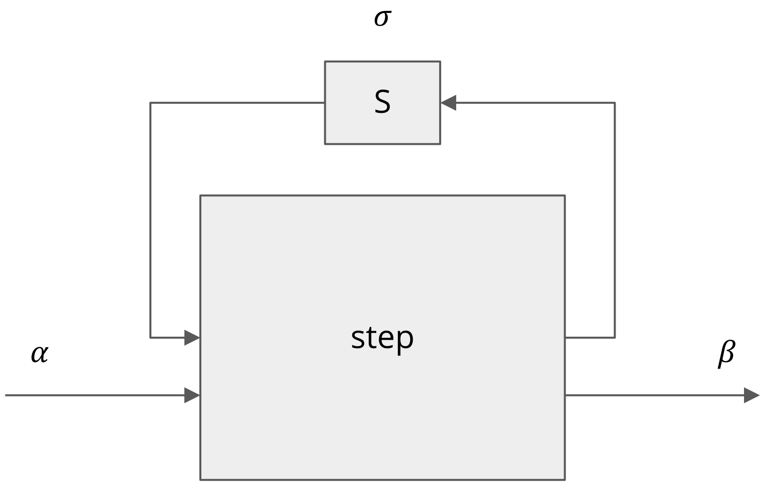
Zelus Compilation Pipeline



Embedded code (OCaml)
Imperative updates to the state
`program.byte`

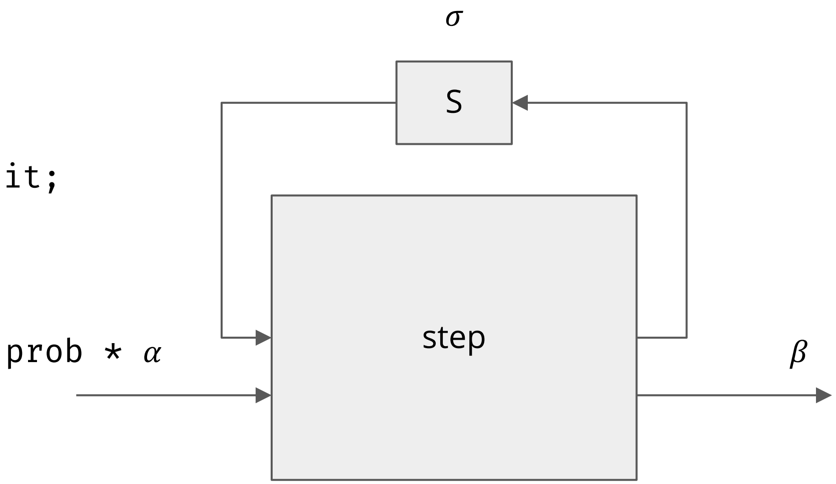
Synchronous State Machines

```
type (α, β) cnode = Cnode: { alloc : unit -> σ;
                             reset : σ -> unit;
                             step  : σ -> α -> β;
                           } -> (α, β) cnode
```



Probabilistic Synchronous State Machines

```
type (α, β) cnode = Cnode: { alloc : unit -> σ;
                             reset : σ -> unit;
                             step  : σ -> α -> β;
                             copy  : σ -> σ -> unit;
                           } -> (α, β) cnode

type (α, β) pnode = (prob * α, β) cnode
```



Infer: Sequential Monte Carlo

```
val infer_smc : int -> (α, β) pnode -> (α, β dist) cnode

type σ infer_state = { mutable sigma : σ dist }

let infer_smc n (Cnode { alloc; reset; step; copy }) =
  let i_step state obs =
    let particles = Array.init n
      (fun _ -> let p = alloc () in
                let s = Dist.draw state.sigma in             (* Filtering *)
                copy s p; p)
    in
    let scores = Array.make n 0. in
    let values = Array.mapi                                  (* Particle execution *)
      (fun i s -> step s ({ id = i; scores }, obs)) particles
    in
    state.sigma <- multinomial particles scores;             (* Next state distribution *)
    multinomial values scores                                (* Output distribution *)
  in ...
```

Particle Filter

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


let sample (prob, d) = draw d

let observe (prob, d, x) =
  prob.scores.(prob.id) <- prob.scores.(prob.id) +. logpdf d x



let infer_pf n model = infer_smc n model
```

Assumed Parameter Filter

Each particle maintains a distribution of fixed parameters. At each step:

- `filter` the set of particles
- `sample` to compute moving parameters
- `update` the distribution of fixed parameters

Input: data y_t, previous particles distribution μ_{t-1}
Output: current particles distribution μ_t

For each particle $i = 1$ to N do

$x^i_{t-1}, \Theta^i_{t-1} = \mathtt{sample}(\mu_{t-1})$

$\theta^i = \mathtt{sample}(\Theta^i_{t-1})$

$x^i_t, w^i_t = \mathtt{model}(y_t \mid \theta^i, x^i_{t-1})$

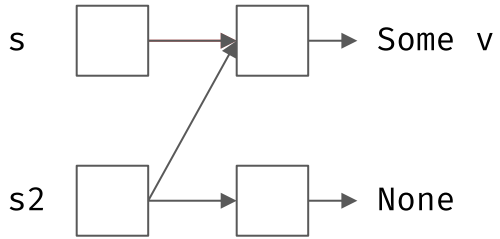
$\Theta^i_t = \mathtt{update}(\Theta^i_{t-1}, \lambda\theta.\ \mathtt{model}(y_t \mid \theta, x^i_t, x^i_{t-1}))$

$\mu_t = \mathcal{M}(\{w^i_t, (x^i_t, \Theta^i_t)\}_{1 \leq i \leq N})$

We need to replay the same transition multiple times.
Problem: Each call to `sample` returns a different value

Replay with `sample` as a Node

```
type prob = { id : int; scores : float array; replay : bool }

let sample =
  let alloc () = ref (ref None) in
  let reset state = !state := None in
  let step state (prob, dist) =
    match prob.replay with
    | false -> let v = Dist.draw dist in !state := Some v; v
    | true  -> let v = Option.get (! (!state)) in
               observe (prob, dist, v); v
  in
  let copy src dst = dst := !src in
  Cnode { alloc; reset; copy; step }
```



Assumed Parameter Filter

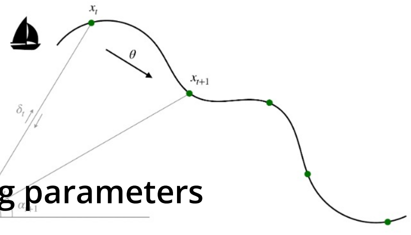
```
type (α, β) apf_state = { p_state : α; mutable d_theta : β dist }

let apf_particle m_prior (Cnode { alloc; reset; step; copy } as m_model) =
  let p_step ({ p_state; d_theta } as state) (prob, obs) =
    let start_state = alloc () in copy p_state start_state;
    let theta = Dist.draw d_theta in                                    (* Sampling *)
    let o = step p_state ({ prob with replay = false }, (theta, obs)) in  (* Sampling *)
    let tmp_state = alloc () in
    state.d_theta <- update                                             (* Update *)
      (fun theta ->
        copy start_state tmp_state;
        let prob = {id = 0; replay = true; scores = [|0.|]} in
        let _ = step tmp_state (prob, (theta, obs)) in
        prob.scores.(prob.id))
      d_theta;
    o
  in ...
```

Assumed Parameter Filter

```
let APF.infer n m_prior m_model =
  let particle = apf_particle m_prior m_model in   (* Sampling and update *)
  infer_smc n particle                             (* Filtering *)
```

Evaluation

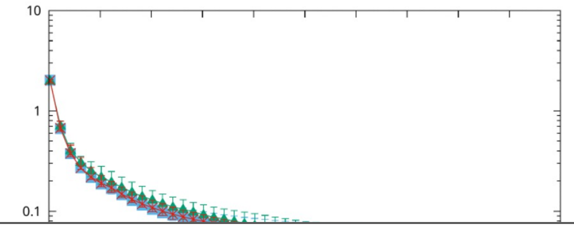
Evaluation

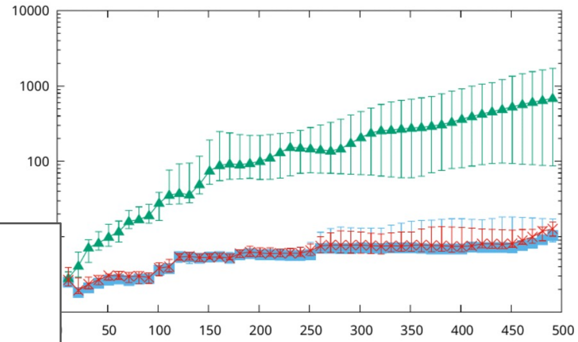


Fixed parameters

Moving parameters

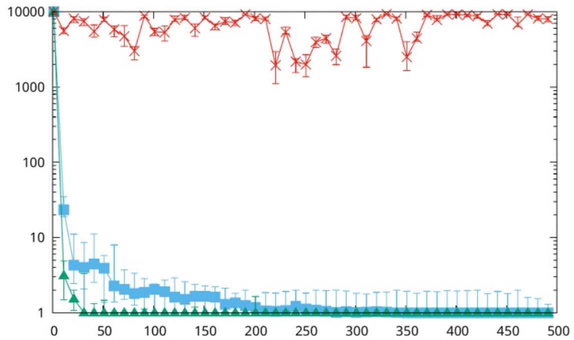
Precision
Metrics: Mean Square Error





Time cost: between 1.3x and 3x slower

Impoverishment
Metrics: Effective Sample Size



Particle Filter ■

Importance Sampling ▲

Assumed Parameter Filter ╳

Takeaway

Assumed Parameter Filter (APF)

- Problem: particle impoverishment for fixed parameters
- APF: split moving and fixed parameters
- Requires a program transformation

Static Analysis and Compilation

- Identify fixed parameters in ProbZelus models
- Move fixed parameters as inputs of the model

Runtime

- Sequential Monte Carlo for probabilistic state machines
- Sample with replay using stratified memory states

Also in the paper

- Evaluation on existing models
- Simplified correctness proof of the compilation

Filtrer sans s'appauvrir : inférer les paramètres constants des modèles réactifs probabilistes

Guillaume Baudart,[1,2] Grégoire Bussone,[2,3]
Louis Mandel,[4] et Christine Tasson[3]

[1] Inria Paris
[2] École normale supérieure – PSL university
[3] Sorbonne Université
[4] IBM Research

Résumé

ProbZelus étend le langage synchrone Zelus pour permettre de décrire des modèles probabilistes synchrones. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions. On peut par exemple estimer la position d'un objet en mouvement à partir d'observations bruitées ou estimer l'incertitude d'un capteur à partir d'une suite d'observations. Ces problèmes mêlent des flots de variables aléatoires – les *paramètres d'état* qui changent au cours du temps – et des variables aléatoires constantes – les *paramètres constants*.

Pour estimer les paramètres d'état, les algorithmes d'inférence bayésienne de Monte Carlo séquentiels reposent sur des techniques de filtrage. Le filtrage est une méthode approchée qui consiste à perdre volontairement de l'information sur l'approximation actuelle pour recentrer les estimations futures autour de l'information la plus significative. Malheureusement, cette perte d'information est dommageable pour l'estimation des paramètres constants qui n'évoluent pas au cours du temps. Ce phénomène s'appelle l'appauvrissement.

Inspirés de la méthode d'inférence *Assumed Parameter Filter* (APF), nous proposons (1) une analyse statique, (2) une passe de compilation et (3) une nouvelle méthode d'inférence modulaire pour ProbZelus qui évite l'appauvrissement pour les paramètres constants tout en gardant les performances des algorithmes de Monte Carlo séquentiels pour les paramètres d'états.

1 Introduction

La programmation synchrone flot de données [5] est un paradigme dans lequel les fonctions, appelées nœuds, manipulent des suites infinies, appelées flots. Ces flots progressent au même rythme, de manière synchrone. L'expressivité des langages synchrones est volontairement restreinte, ce qui permet à des compilateurs spécialisés de générer du code efficace et correct par construction avec de fortes garanties sur le temps d'exécution et sur l'utilisation de la mémoire [19]. Le langage industriel Scade est notamment utilisé pour la conception de systèmes embarqués critiques [13].

Les langages probabilistes [6, 14, 17, 18] étendent des langages de programmation généralistes avec des constructions probabilistes. Suivant la méthode bayésienne, un *modèle* probabiliste décrit une distribution de probabilité (la distribution *a posteriori*) à partir d'une croyance initiale (la distribution *a priori*) et d'observations concrètes (les entrées).

Dans la même lignée de langages, ProbZelus [2] est une extension probabiliste du langage synchrone flot de données Zelus [8]. Ce langage permet de décrire des modèles réactifs probabilistes. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions.

https://github.com/rpl-lab/jfla23-apf