Filtrer sans s'appauvrir : inférer les paramètres constants de 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 la valeur d'un paramètre comme l'incertitude d'un capteur à partir d'une suite d'observations. Ces problèmes mêlent paramètres d'état qui changent au cours du temps et 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 pour ProbZelus qui permet de séparer l'inférence des paramètres constants de celle des paramètres d'état. Nous pouvons ainsi rendre modulaire l'inférence pour les paramètres constants et éviter l'appauvrissement tout en gardant les performances des algorithmes de Monte Carlo séquentiels pour les paramètres d'état.

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].

Dans la lignée des langages probabilistes récents [6, 14, 17, 18] qui étendent des langages de programmation généralistes avec des constructions probabilistes, ProbZelus [2] est une extension probabiliste du langage synchrone flot de données Zelus [8]. Ce langage permet de décrire des modèles réactifs probabilistes. Suivant la méthode bayésienne, un modèle exprime une croyance *a priori* sur la distribution de paramètres qui est affinée *a posteriori* par les observations concrètes. 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 sont appelés des *State-Space Models* (SSM) [10]. Ils mêlent paramètres constants et paramètres d'état qui changent au cours du temps. Tout programme probabiliste synchrone peut s'exprimer comme un SSM. Un exemple classique est le problème de localisation et cartographie simultanées (SLAM pour *Simultaneous Localization and Mapping*) [20]. Un robot cherche à inférer à la fois sa position et une carte de son environnement. La position est un paramètre d'état. À chaque instant, une nouvelle position doit être estimée à partir de la position précédente et des nouvelles observations. La carte est un paramètre constant dont la valeur est progressivement affinée à chaque nouvelle observation à partir d'une croyance initiale grossière.

Pour estimer les paramètres d'état, les algorithmes d'inférence de Monte Carlo séquentiels (SMC pour *Sequential Monte Carlo*) reposent sur des techniques de filtrage [10,15]. 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. Ces méthodes sont donc particulièrement bien adaptées au contexte réactif où un système en interaction avec son environnement ne s'arrête jamais et doit donc s'exécuter avec des ressources bornées. Toutes les méthodes d'inférence de ProbZelus appartiennent à cette famille [1–3]. 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.

Pour éviter ce problème, l'algorithme d'inférence *Assumed Parameter Filter* (APF) [16] propose, à chaque instant, de décomposer l'inférence en deux temps : (1) estimation des paramètres d'état, et (2) mise à jour des paramètres constants. Cette méthode suppose que les paramètres constants sont bien identifiés et que le modèle est écrit sous une forme qui permet de mettre à jour les distributions des paramètres constants sachant la valeur des paramètres d'état.

Dans cette article, nous proposons une nouvelle méthode d'inférence fondée sur APF pour ProbZelus. Nous présentons en particulier les contributions suivantes :

— Section 3 une analyse statique pour identifier les paramètres constants,
— Section 4 une passe de compilation pour générer un modèle exploitable par APF,
— Section 5 une implémentation de l'algorithme APF pour les modèles ProbZelus.

Nous évaluons cette nouvelle méthode d'inférence sur un ensemble de modèles ProbZelus. Le code est disponible à l'adresse suivante : `https://github.com/rpl-lab/jfla23-apf`.

2 Motivation

Pour motiver notre approche, considérons l'exemple d'un radar adapté de [10, Section 2.4.1] illustré Figure 1. On cherche à estimer la position x d'un bateau à partir d'une mesure de distance (par exemple à l'aide d'un sonar), et d'une mesure angulaire. On suppose que le bateau dérive à vitesse constante θ et on cherche également à estimer cette vitesse. Ce modèle est donc un *State-Space Model* (SSM) avec un paramètre d'état x et un paramètre constant θ.

2.1 Un modèle de radar en ProbZelus

Un modèle de mouvement pour le bateau peut être programmé de la manière suivante :

```
proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
```

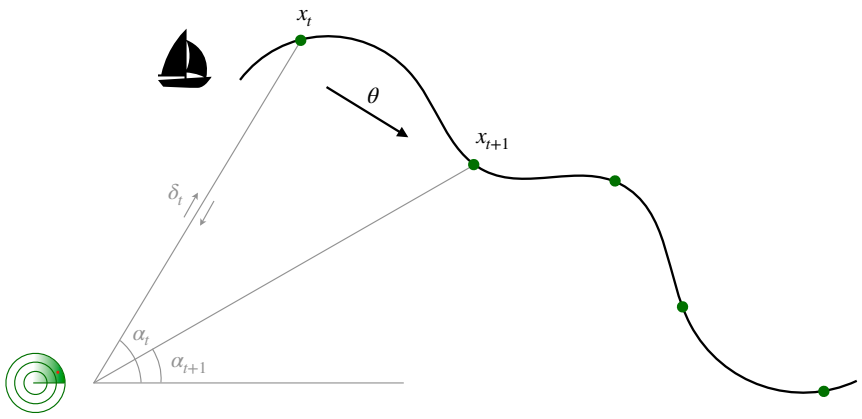


FIGURE 1 – Le radar cherche à estimer la position courante du bateau x_t et sa vitesse de dérive constante θ à partir du délai de rebond d'un sonar δ_t et d'une observation angulaire α_t.

Le mot-clé `proba` introduit un modèle probabiliste appelé `move` qui prend en argument la position initiale `x0` et renvoie, à chaque instant, la vitesse `theta` et la position courante `x`. Les variables `x0`, `theta`, et `x` sont des vecteurs à deux dimensions dans $\mathbb{R}^2$. La vitesse `theta` est un paramètre constant de notre modèle introduit par le mot-clé `init`. On suppose, *a priori*, que `theta` est échantillonné dans une Gaussienne multivariée centrée sur $(0,0)^T$ et de covariance I_2, la matrice identité de dimension 2 (on suppose que les deux composantes de la vitesse ne sont pas corrélées). La définition de la position `x` utilise les opérateurs d'initialisation $\rightarrow$ et de délai unitaire `pre`. Au premier instant, $\mathtt{x}_0 = \mathtt{x0}$. Puis, aux instants $t > 0$ suivants, $\mathtt{x}_t$ est échantillonné dans une Gaussienne centrée sur la valeur précédente $\mathtt{x}_{t-1}$ translatée de `theta` et de covariance $0.5 * I_2$.

Le modèle du radar suivant utilise le modèle de mouvement pour estimer la position et la vitesse du bateau à partir de deux observations bruitées : `delta` le délai de rebond du sonar et `alpha` la mesure angulaire.

```
proba radar (delta, alpha) = theta, x where
  rec theta, x = move x0
  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)
```

À partir de la vitesse `theta` et de la position courante $\mathtt{x} = (x_1, x_2)^T$ estimée, on calcule la distance $\mathtt{d} = \sqrt{x_1^2 + x_2^2}$, et l'angle $\mathtt{a} = \mathrm{atan}(x_2/x_1)$. La construction `observe` permet ensuite de contraindre le modèle en supposant (1) que le temps de rebond observé `delta` a été échantillonné dans une Gaussienne centrée sur `2d/c` (où `c` est la vitesse du signal envoyé par le sonar), et (2) que la mesure angulaire `angle` a été échantillonnée dans une Gaussienne centrée sur `a`.

On peut maintenant lancer l'inférence pour estimer à chaque instant la distribution *a posteriori* des sorties du modèle `radar` en fonction des distributions *a priori* et des flots d'observations.

```
node main (delta, alpha) =
  let d = infer radar (delta, alpha) in
  let theta, x = split d in plot theta; plot x
```

L'opérateur `split` permet de séparer une distribution sur les paires en une paire de distributions *marginales* décorrélées, plus facile à analyser.

2.2 Le filtrage et ses limites

Les méthodes d'inférence proposées par ProbZelus [1–3] appartiennent à la famille des algorithmes de Monte Carlo séquentiels [10, 15]. Ces méthodes reposent sur un ensemble de simulations indépendantes, appelées *particules* pour approximer la distribution recherchée. Chaque particule est associée à un *score*, et un modèle probabiliste devient un échantillonneur qui, à chaque instant, renvoie une valeur de sortie associée à un score. La construction `sample(d)` tire aléatoirement une valeur dans la distribution `d`, et la construction `observe(d, x)` modifie le score courant de la particule en lui ajoutant la *vraisemblance* de l'observation `x` par rapport à la distribution `d`. On ne considère pour cette construction que des distributions dont la fonction de densité est bien définie. La vraisemblance d'une observation est alors la valeur de la densité au point d'observation $\mathtt{d}_{\text{pdf}}(x)$. À chaque instant, l'opérateur `infer` accumule le résultat de chaque particule pondéré par son score pour approximer la distribution recherchée.

Filtre particulaire. Si le modèle fait appel à l'opérateur `sample` à chaque instant, par exemple pour estimer la position du bateau, la méthode précédente implémente une marche aléatoire pour chaque particule. À mesure que le temps avance, il devient de plus en plus improbable qu'une des marches aléatoires coïncide avec le flot d'observations. Le score associé à chaque particule dégringole rapidement vers 0.

Pour régler ce problème, les méthodes de Monte Carlo séquentielles ajoutent une étape de filtrage. À chaque instant, l'opérateur `infer` échantillonne un nouvel ensemble de particules à partir des résultats de l'instant précédent. Les particules les plus probables sont dupliquées, les moins probables sont éliminées et les scores sont réinitialisés : au début de l'instant suivant, toutes les particules sont équiprobables. On recentre ainsi l'inférence autour de l'information la plus significative tout en conservant le même nombre de particules tout au long de l'exécution. Ces méthodes peuvent donc s'exécuter en mémoire bornée, et sont donc particulièrement bien adaptées au contexte réactif où un système ne s'arrête jamais.

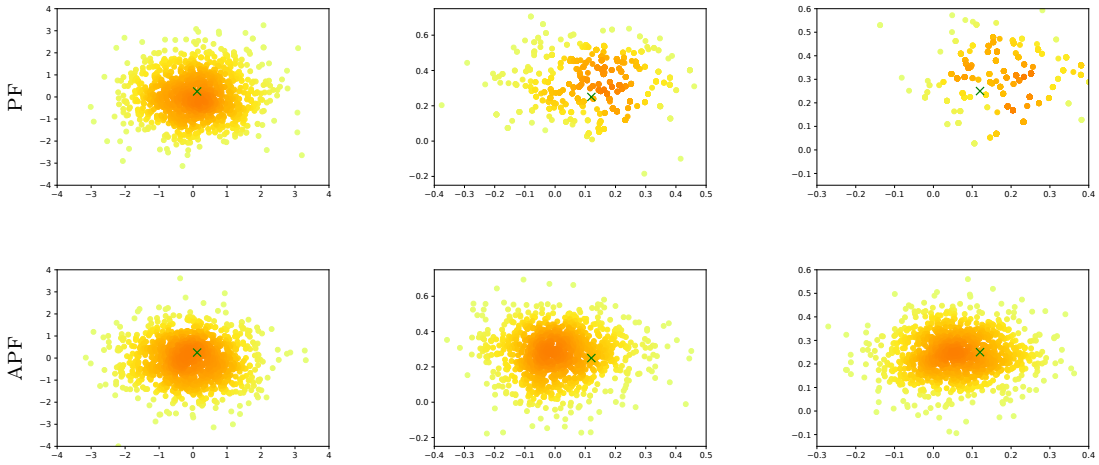
Appauvrissement. Malheureusement, cette perte d'information est dommageable pour l'estimation de paramètres constants. Sur l'exemple du radar, au premier instant, chaque particule tire une valeur aléatoire pour le paramètre `theta`. À chaque instant, les particules dupliquées partagent la même valeur pour `theta`. La quantité d'information utile pour estimer `theta` décroît donc à chaque nouveau filtrage et, au bout d'un certain temps, il ne reste plus qu'une seule valeur possible. La partie haute de la Figure 2 illustre graphiquement ce phénomène.

Plus formellement, on peut montrer que le filtre particulaire souffre d'appauvrissement sur les paramètres constants quel que soit le modèle.

Proposition 1 (Appauvrissement)**.** *Considérons un filtre particulaire avec $N \in \mathbb{N}^*$ particules pour estimer un paramètre constant θ à valeur dans V. À chaque instant, on note $\Theta^t \in V^N$ l'estimation obtenue à l'instant $t \in \mathbb{N}$, par définition un ensemble de valeurs possibles pour θ. Il existe presque sûrement un instant $T \in \mathbb{N}$ tel que $\forall t > T, |\Theta^t| = 1$.*

Démonstration. On note à chaque instant $t \in \mathbb{N}$, $\Theta^t = \{\theta_i^t\}_{1 \leq i \leq N}$ l'ensemble des valeurs possibles et $s(\theta_i^t) \in]0, \infty[$ le score associé à la valeur θ_i^t. À chaque filtrage, θ_i^t est dupliquée avec la probabilité $\overline{s(\theta_i^t)}$ (le score normalisé de θ_i^t). Pour $t \in \mathbb{N}$, s'il reste au moins deux particules distinctes $|\Theta^t| \geq 2$, on note A la valeur de Θ^t associée au score minimal qui a donc le plus de chances d'être éliminée : $A = \underset{a \in \Theta^t}{\operatorname{argmin}} \sum_{\theta_i^t = a} s(\theta_i^t)$. On a donc $\forall 1 \leq i \leq N$, $\mathbb{P}(\theta_i^{t+1} = A) \leq \frac{1}{2}$ et :

$$\mathbb{P}(\exists a.a \in \Theta^t \text{ et } a \notin \Theta^{t+1}) \geq \mathbb{P}(A \notin \Theta^{t+1}) = \prod_{i=1}^{N} \mathbb{P}(\theta_i^{t+1} \neq A) \geq \frac{1}{2^N} = \epsilon > 0.$$



(a) $t = 0$ (b) $t = 30$ (c) $t = 50$

FIGURE 2 – Estimations du paramètre `theta` de l'exemple du radar au cours du temps avec un filtre particulaire (PF, en haut) et *Assumed Parameter Filter* (APF, en bas). La vraie vitesse de dérive est indiquée par une croix verte. Le gradient de couleur représente la densité de points. L'échelle change au cours du temps. Les résultats peuvent différer d'une exécution à l'autre.

Comme ϵ ne dépend que de N, $\forall n \in \mathbb{N}$, on a : $\mathbb{P}(\Theta^{t+n} = \Theta^t) \leq (1-\epsilon)^n$. Il existe donc presque sûrement $t' > t$ tel que $\Theta^{t'} \subsetneq \Theta^t$. Par récurrence sur la taille de Θ^0, il existe presque sûrement $T \in \mathbb{N}$ tel que $|\Theta^T| = 1$ □

2.3 *Assumed Parameter Filter*

Plutôt que d'échantillonner au début de l'exécution une valeur pour chaque paramètre constant qui s'appauvrira inexorablement, l'idée de l'algorithme *Assumed Parameter Filter* (APF) [16] est de maintenir une distribution. Lors de l'exécution, l'inférence alterne ensuite entre une passe d'échantillonnage pour estimer les paramètres d'état, et une passe d'optimisation qui permet de mettre à jour les paramètres constants. On évite ainsi l'appauvrissement pour l'estimation des paramètres constants. La partie basse de la Figure 2 illustre les résultats de APF sur l'exemple du radar.

Plus formellement, l'Algorithme 1 décrit l'exécution de APF. Chaque particule $1 \leq i \leq N$ maintient une distribution de paramètres constants Θ_i^t initialisée avec Θ^0, la distribution *a priori*. À chaque instant $t > 0$, chaque particule échantillonne une valeur θ_i dans la distribution obtenue à l'instant précédent Θ_i^{t-1}. Comme pour un filtre particulaire classique, on peut alors exécuter un pas du modèle pour obtenir un échantillon des paramètres d'état x_t associé à un score w_t sachant la valeur des paramètres constants θ_i et de l'état précédent x_i^{t-1}. Il reste alors à calculer Θ_i^t en explorant les autres valeurs possibles pour θ sachant que la particule a choisi la transition $x_i^{t-1} \to x_i^t$. À la fin de l'instant, on construit une distribution P où chaque particule est associée à son score. $\mathcal{M}(\{w_i, v_i\}_{1 \leq i \leq N})$ est une distribution multinomiale, où la valeur v_i est associée à la probabilité w_i. On peut enfin ré-échantillonner N nouvelles particules dans P pour démarrer l'instant suivant.

Données : le modèle `model`, le flot d'entrées y, la distribution *a priori* Θ^0.
Résultat : à chaque instant t, P une approximation de la distribution de x_t et θ.

Initialiser $\{\Theta_i^0 = \Theta^0\}_{1 \leq i \leq N}$.
pour *toujours, à chaque instant* $t > 0$ **faire**
  pour *chaque particule* $i = 1$ **à** N **faire**
    $\theta_i = \texttt{sample}(\Theta_i^{t-1})$
    $x_i^t, w_i^t = \texttt{model}(y_t \mid \theta_i, x_i^{t-1})$
    $\Theta_i^t = \mathit{Udpate}(\Theta_i^{t-1}, \lambda \mathit{theta}.\ \texttt{model}(y_t \mid \theta, x_i^{t-1}, x_i^t))$
  $P = \mathcal{M}(\{w_i^t, (x_i^t, \Theta_i^t)\}_{1 \leq i \leq N})$
  $\{x_i^t, \Theta_i^t = \texttt{sample}(P)\}_{1 \leq i \leq N}$

Algorithme 1 : *Assumed Parameter Filter* [16]

$d ::= \texttt{let}\ x\ \texttt{=}\ e \mid \texttt{node}\ f\ x\ \texttt{=}\ e \mid \texttt{proba}\ f\ x\ \texttt{=}\ e \mid d\ d$
$e ::= c \mid x \mid \texttt{(}e\texttt{,}e\texttt{)} \mid op\texttt{(}e\texttt{)} \mid f^{\alpha}\texttt{(}e\texttt{)} \mid \texttt{last}\ x$
$\quad \mid e\ \texttt{where rec init}\ x\ \texttt{=}\ e\ \texttt{and}\ \ldots\ \texttt{and init}\ x\ \texttt{=}\ e\ \texttt{and}\ p\ \texttt{=}\ e\ \texttt{and}\ \ldots\ \texttt{and}\ p\ \texttt{=}\ e$
$\quad \mid \texttt{present}\ e\ \texttt{->}\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e \mid \texttt{sample(}e\texttt{)} \mid \texttt{observe(}e\texttt{,}e\texttt{)} \mid \texttt{infer(}f\texttt{(}e\texttt{))}$

FIGURE 3 – Syntaxe du noyau de ProbZelus

La difficulté principale est qu'à chaque instant, l'Algorithme 1 doit ré-exécuter plusieurs fois le modèle en fixant la valeur de certaines variables aléatoires. D'abord pour calculer les paramètres d'état sachant la valeur échantillonnée θ_i, puis pour mettre à jour la distribution Θ_i^t en explorant plusieurs valeurs possibles pour θ, sachant la transition $x_i^{t-1} \to x_i^t$.

Pour implémenter APF pour des modèles ProbZelus arbitraires, nous proposons donc une analyse statique pour identifier les paramètres constants *a priori* (Section 3) et une passe de compilation qui transforme ces paramètres en entrée supplémentaire du modèle (Section 4).

3 Analyse Statique

Dans cette section, nous présentons un noyau de ProbZelus. L'ensemble du langage, y compris les structures de contrôle de haut niveau comme les automates hiérarchiques, peut être compilé vers ce noyau [12]. Nous présentons ensuite l'analyse statique qui permet d'identifier les paramètres constants et leur distribution *a priori* sur ce noyau.

3.1 Un noyau de ProbZelus

La syntaxe du noyau de ProbZelus est présentée dans la Figure 3. Un programme est une série de déclarations d. Une déclaration peut donner un nom au résultat d'une expression (`let` x `=` e) ou définir un nœud déterministe (`node` f x `=` e) ou probabiliste (`proba` f x `=` e). Le nom de chaque déclaration est unique. Une expression peut être une constante (c), une variable (x), une paire (`(`e`,`e`)`), l'application d'une primitive (op`(`e`)`), un appel de nœud (f^{α}`(`e`)`), un délai (`last` x), un ensemble d'équations locales mutuellement récursives (e `where rec` E), une structure de contrôle (`present` e `->` e `else` e), une structure de réinitialisation (`reset` e `every` e) ou une construction probabiliste (`sample(`e`)`, `observe(`e`,`e`)` ou `infer(`f`(`e`))`).

$$\frac{}{C \vdash c : \emptyset} \qquad \frac{}{C \vdash x : \emptyset} \qquad \frac{C \vdash e_1 : \xi_1 \qquad C \vdash e_2 : \xi_2}{C \vdash (e_1, e_2) : \xi_1 \cup \xi_2} \qquad \frac{C \vdash e : \xi}{C \vdash op(e) : \xi}$$

$$\frac{C \vdash e : \xi}{C \vdash f^{\alpha}(e) : \xi \cup \{\alpha \leftarrow f\_\texttt{prior}\}} \qquad \frac{}{C \vdash \texttt{last}\ x : \emptyset} \qquad \frac{C \vdash e_1 : \xi}{C \vdash \texttt{present}\ e_1\ \texttt{->}\ e_2\ \texttt{else}\ e_3 : \xi}$$

$$\frac{C \vdash e_2 : \xi}{C \vdash \texttt{reset}\ e_1\ \texttt{every}\ e_2 : \xi} \qquad \frac{C \vdash e : \xi}{C \vdash \texttt{sample}(e) : \xi} \qquad \frac{C \vdash e_1 : \xi_1 \qquad C \vdash e_2 : \xi_2}{C \vdash \texttt{observe}(e_1, e_2) : \xi_1 \cup \xi_2}$$

$$\frac{\begin{array}{c} C \vdash e : \xi \qquad \forall 1 \le i \le k, C \vdash e_i : \xi_i \qquad \forall 1 \le j \le n, C \vdash e'_j : \xi'_j \\ \xi' = \{x_i \leftarrow d_i \mid x_i = y_j \wedge e'_j = \texttt{last}\ y_j \wedge \vdash^p e_i : d_i \ \wedge\ C \vdash^l d_i\} \end{array}}{C \vdash e\ \texttt{where rec}\ (\texttt{init}\ x_i = e_i)_{1 \le i \le k}\ \texttt{and}\ (y_j = e'_j)_{1 \le j \le n} : \xi \cup (\cup_{i=1}^{k} \xi_i) \cup (\cup_{j=1}^{n} \xi'_j) \cup \xi'}$$

$$\frac{C \vdash e : \xi}{H, C \vdash \texttt{proba}\ f\ x = e : H \cup \{f \leftarrow \xi\}, C} \qquad \frac{}{H, C \vdash \texttt{node}\ f\ x = e : H \cup \{f \leftarrow \emptyset\}, C}$$

$$\frac{C \vdash^l e}{H, C \vdash \texttt{let}\ x = e : H, C \cup \{x\}} \qquad \frac{H, C \vdash d_1 : H_1, C_1 \qquad H_1, C_1 \vdash d_2 : H', C'}{H, C \vdash d_1\ d_2 : H', C'}$$

FIGURE 4 – Identification des variables aléatoires constantes avec leurs distributions *a priori*.

On impose que toutes les variables initialisées soient définies par une équation, quitte à ajouter `x = last x`, et que les équations soient ordonnancées. Dans une expression de la forme e `where rec` $(\texttt{init}\ x_i = e_i)_{1 \le i \le k}$ `and` $(y_j = e'_j)_{1 \le j \le n}$, on a donc $\{x_i\}_{1 \le i \le k} \subset \{y_j\}_{1 \le j \le n}$ et e'_j ne peut dépendre de $y_{j'}$ avec $j \le j'$ qu'à travers l'opérateur `last`.

Ce noyau est proche de celui utilisé pour définir la sémantique formelle de ProbZelus [2]. La différence principale est qu'il est possible d'initialiser une variable avec une expression arbitraire là où le noyau original limitait la construction `init` à des constantes : `init` x `=` c. On autorise ainsi `x = sample(d) and x = last x` pour définir des paramètres constants. Par ailleurs, on autorise la définition des variables globales dans les déclarations. Enfin, l'appel de nœud $f^{\alpha}(e)$ est ici annoté avec un nom d'instance α ajouté automatiquement par le compilateur qui permet de différencier deux instances différentes. Par exemple, l'expression `f(0) + f(1)` devient dans ce noyau $\texttt{f}^{\alpha_1}\texttt{(0) + f}^{\alpha_2}\texttt{(1)}$.

Sémantique. La sémantique formelle de ProbZelus est définie dans [2]. Dans un environnement γ qui associe les noms de variables à leur valeur, les expressions déterministes sont des machines à états caractérisées par un état initial $[\![e]\!]^{\text{init}}_{\gamma}$ de type S et une fonction de transition $[\![e]\!]^{\text{step}}_{\gamma}$ de type $S \to O \times S$. La fonction de transition prend en entrée un état et renvoie une sortie et l'état suivant. On obtient un flot de données en itérant la fonction de transition depuis l'état initial.

Les expressions probabilistes sont elles aussi caractérisées par un état initial de type S, mais la fonction de transition $\{\![e]\!\}^{\text{step}}_{\gamma}$ de type $S \to \Sigma_{O \times S} \to [0, \infty)$ renvoie une mesure qui associe un score positif à chaque ensemble mesurable de paires (sortie, état suivant) $\in \Sigma_{O \times S}$.

$$\frac{}{\vdash^p \texttt{sample}(d) : d} \qquad \frac{\vdash^p e : d}{\vdash^p e \texttt{ where rec } E : d \texttt{ where rec } E}$$

FIGURE 5 – Distribution échantillonnée.

$$\frac{}{C \vdash^l c} \qquad \frac{x \in C}{C \vdash^l x} \qquad \frac{C \vdash^l e_1 \qquad C \vdash^l e_2}{C \vdash^l (e_1, e_2)} \qquad \frac{C \vdash^l e}{C \vdash^l op(e)} \qquad \frac{C \vdash^l e_1 \qquad C \vdash^l e_2 \qquad C \vdash^l e_3}{C \vdash^l \texttt{present } e_1 \texttt{ -> } e_2 \texttt{ else } e_3}$$

$$\frac{C \vdash^l e_1 \qquad C \vdash^l e_2}{C \vdash^l \texttt{reset } e_1 \texttt{ every } e_2} \qquad \frac{C \cup \{x_i\}_{1 \le i \le k} \vdash^l e \qquad C \cup \{x_j\}_{1 \le j < i} \vdash^l e_i}{C \vdash^l e \texttt{ where rec } (\texttt{init } x_i \texttt{ = } e_i)_{1 \le i \le k} \texttt{ and } (x_j \texttt{ = last } x_j)_{1 \le j \le k}}$$

FIGURE 6 – Expressions constantes.

3.2 Identifier les paramètres constants

L'analyse statique est définie par un jugement $\emptyset, \emptyset \vdash prog : H, C$ qui, pour un programme *prog*, construit un environnement H associant à chaque nœud probabiliste un type ξ qui identifie les variables aléatoires constantes du nœud et leur distribution *a priori* (l'environnement C contient les variables globales constantes). Pour un nœud `proba` f `x =` e, le type ξ est une fonction telle que chaque variable x dans $dom(\xi)$ est une variable aléatoire constante dans e de distribution *a priori* $\xi(x)$. Le jugement de typage des expressions est de la forme $C \vdash e : \xi$ et repose sur deux jugements auxiliaires : le jugement $\vdash^p e : d$ qui extrait la distribution d échantillonnée par e, et le jugement $C \vdash^l e$ qui garantit que l'expression e est constante et ne dépend que des variables constantes définies dans l'ensemble C.

Paramètres constants. Le jugement principal $C \vdash e : \xi$ est défini Figure 4. Pour simplifier l'analyse, on suppose que toutes les variables et les instances ont des noms différents. Une passe de renommage des variables permet facilement de satisfaire cette précondition. Toutes les règles parcourent les sous-expressions pour collecter les variables aléatoires constantes. La règle pour `reset` e_1 `every` e_2 parcourt uniquement e_2 car l'expression e_1 peut être réinitialisée et donc n'est pas constante. Pour une raison similaire, la règle pour `present` e_1 `->` e_2 `else` e_3 ne parcourt que e_1. La règle pour $f^\alpha(e)$ associe dans ξ le nom d'instance α avec la distribution *a priori* de f que l'on suppose définie dans la variable f`_prior` (qui sera ajoutée par la compilation). La règle pour l'expression e `where rec` $(\texttt{init } x_i \texttt{ = } e_i)_{1 \le i \le k}$ `and` $(y_j \texttt{ = } e'_j)_{1 \le j \le n}$ ajoute dans ξ les variables x_i qui respectent les conditions suivantes.

1. Elle n'est pas modifiée par la fonction de transition ($x_i = y_j \wedge e'_j = \texttt{last } y_j$).
2. Elle échantillonne une distribution d_i ($\vdash^p e_i : d_i$).
3. La distribution d_i est constante et ne dépend que des constantes dans C ($C \vdash^l d_i$).

L'analyse d'un nœud probabiliste f type le corps e du nœud et ajoute ce type dans l'environnement H associé au nom f. Les nœuds déterministes n'ont pas de variables aléatoires par définition. Les déclarations globales doivent être constantes, leurs noms sont ajoutés dans l'ensemble C des variables constantes.

Distributions. Le jugement $\vdash^p e : d$, défini Figure 5, garantit que l'expression e échantillonne une distribution d. La règle principale est donc celle de `sample`(d) qui renvoie la distribution d.

La règle pour `where rec` autorise l'utilisation de variables locales pour définir la distribution échantillonnée.

Constantes. Le jugement $C \vdash^l e$, défini Figure 6, garantit que l'expression e est constante en vérifiant que toutes les sous-expressions sont constantes et ne dépendent que de variables constantes.

Exemple. Illustrons l'analyse statique sur l'exemple du nœud `move` *de la Section 2. L'équation* `theta = last theta` *est ajoutée automatiquement par le compilateur.*

```
proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and theta = last theta
```

La règle `proba` *type le corps du nœud en utilisant la règle* `where rec`*. Le typage de chaque sous-expression renvoie un ensemble vide (aucun paramètre constant local n'est défini). Le système détecte que* `theta` *est un flot constant (*`init theta = ... and theta = last theta`*), donc l'expression* `sample(mv_gaussian(zeros, i2))` *est analysée avec le jugement* $\vdash^p$ *qui renvoie la distribution* `mv_gaussian(zeros, i2)`*. Cette distribution est typée avec succès par le jugement* $\vdash^l$ *dans l'environnement* C *qui contient les constantes* `zeros` *et* `i2`*. Ainsi le corps du nœud* `move` *a le type* { `theta` ← `mv_gaussian(zeros, i2)` } *et le nœud a le type* { `move` ← { `theta` ← `mv_gaussian(zeros, i2)` } }*.*

4 Compilation

Pour exécuter l'algorithme APF présenté en Section 2.3, il faut maintenant compiler les modèles probabilistes pour que les paramètres constants deviennent une entrée supplémentaire. On pourra ainsi ré-exécuter le modèle en explorant différentes valeurs pour mettre à jour la distribution des paramètres constants. La compilation est définie Figure 7 par induction sur la syntaxe du noyau. On commence par typer le programme complet, c'est-à-dire une liste de déclarations $p = d_1 \ \ldots d_n$ dans un environnement vide : $\emptyset, \emptyset \vdash p : H, C$. La fonction de compilation $\mathcal{C}$ est ensuite paramétrée par l'environnement de typage H pour les déclarations, et le type ξ pour les expressions (voir Section 3).

Modèles. Pour un modèle `proba` f `x =` e, si après l'analyse statique $H(f) = \xi$ (*i.e.*, $C \vdash e : \xi$), alors f est compilé en deux déclarations :

- `let` f`_prior =` $img(\xi)$: la distribution *a priori* des paramètres constants dans e, et
- `proba` f`_model` $(dom(\xi), x)$ `=` $\mathcal{C}_\xi(e)$: un nouveau modèle qui prend en argument supplémentaire $dom(\xi)$ pour les paramètres constants.

Expressions. La fonction de compilation d'une expression est paramétrée par le type ξ du nœud qui la contient. Cette fonction supprime les définitions de paramètres constants, $x \in dom(\xi)$, qui sont passés en entrée du nouveau modèle. Les premiers cas dans la Figure 7, de c à `observe`, implémentent une simple descente récursive sur les sous-expressions. Le cas `where`/`rec` supprime effectivement les paramètres constants en ne gardant que les équations définissant les variables x telles que $x \notin dom(\xi)$.

Exemple. Après compilation, le nœud `move` *de l'exemple de la Section 2 devient :*

```
let move_prior = mv_gaussian(zeros, i2)
proba move_model (theta, x0) = theta, x where
  rec x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
```

$$
\begin{array}{lcl}
\mathcal{C}_\xi(c) & = & c \\
\mathcal{C}_\xi(x) & = & x \\
\mathcal{C}_\xi((e_1, e_2)) & = & (\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi(op(e)) & = & op(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\texttt{last}\ x) & = & \texttt{last}\ x \\
\mathcal{C}_\xi(\texttt{present}\ e_1\ \texttt{->}\ e_2\ \texttt{else}\ e_3) & = & \texttt{present}\ \mathcal{C}_\xi(e_1)\ \texttt{->}\ \mathcal{C}_\xi(e_2)\ \texttt{else}\ \mathcal{C}_\xi(e_3) \\
\mathcal{C}_\xi(\texttt{reset}\ e_1\ \texttt{every}\ e_2) & = & \texttt{reset}\ \mathcal{C}_\xi(e_1)\ \texttt{every}\ \mathcal{C}_\xi(e_2) \\
\mathcal{C}_\xi(\texttt{sample}(e)) & = & \texttt{sample}(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\texttt{observe}(e_1, e_2)) & = & \texttt{observe}(\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi\begin{pmatrix} e\ \texttt{where} \\ \quad \texttt{rec}\ (\texttt{init}\ x_i = e_i)_{1\leq i\leq k} \\ \quad \texttt{and}\ (y_j = e'_j)_{1\leq j\leq n} \end{pmatrix} & = & \mathcal{C}_\xi(e)\ \texttt{where} \\
 & & \qquad \texttt{rec}\ (\texttt{init}\ x_i = \mathcal{C}_\xi(e_i))_{1\leq i\leq k, x_i\notin dom(\xi)} \\
 & & \qquad \texttt{and}\ (y_j = \mathcal{C}_\xi(e'_j))_{1\leq j\leq n, y_j\notin dom(\xi)} \\
\mathcal{C}_\xi(f^\alpha(e)) & = & \begin{cases} f(\mathcal{C}_\xi(e)) & \text{si } f \text{ est déterministe} \\ f\_\texttt{model}(\alpha, \mathcal{C}_\xi(e)) & \text{si } \alpha \in dom(\xi) \\ f\_\texttt{model}(\alpha, \mathcal{C}_\xi(e))\ \texttt{where} & \text{sinon} \\ \quad \texttt{rec init}\ \alpha = \texttt{sample}(f\_\texttt{prior}) & \\ \quad \texttt{and}\ \alpha = \texttt{last}\ \alpha & \end{cases} \\
\mathcal{C}_\xi(\texttt{infer}(f(e))) & = & \texttt{APF.infer}(f\_\texttt{prior}, f\_\texttt{model}, e)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{C}_H(\texttt{let}\ x = e) & = & \texttt{let}\ x = e & \\
\mathcal{C}_H(\texttt{node}\ f\ x = e) & = & \texttt{node}\ f\ x = \mathcal{C}_\emptyset(e) & \\
\mathcal{C}_H(\texttt{proba}\ f\ x = e) & = & \texttt{let}\ f\_\texttt{prior} = img(\xi) & \text{avec } H(f) = \xi \\
 & & \texttt{proba}\ f\_\texttt{model}\ (dom(\xi), x) = \mathcal{C}_\xi(e) &
\end{array}
$$

FIGURE 7 – Fonction de compilation d'un programme ProbZelus pour APF. La fonction de compilation est paramétrée par le type ξ pour les expressions et l'environnement de typage H pour les déclarations (voir Section 3).

Appel de nœud. La difficulté principale est de faire remonter les paramètres constants introduits par les appels de nœud. On distingue trois cas.

— Si le nœud est déterministe, il n'y a pas de paramètre constant possible.
— Si les paramètres constants de l'appel de nœud sont également des paramètres constants dans le contexte d'appel, on a $\alpha \in dom(\xi)$ et on se contente de remplacer l'appel à f^α par un appel à f`_model` en utilisant le nom frais d'instance pour les paramètres constants.
— Sinon, il existe des paramètres constants dans f qui ne sont pas constants dans le contexte d'appel car l'instance f^α est utilisée à l'intérieur d'un `reset`/`every` ou d'un `present`/`else`. Dans ce cas, on redéfinit localement ces paramètres en échantillonnant leur distribution *a priori* f`_prior`.

Par ailleurs, dans les nœuds déterministes, on substitue l'appel à `infer`$(f(e))$ par un appel à `APF.infer`$(f\_\texttt{prior}, f\_\texttt{model}, e)$.

Exemple. Le nœud `radar` *de la Section 2 devient :*

```
let radar_prior = move_prior
proba radar_model (i_move_0, (delta, alpha)) = theta, x where
  rec theta, x = move_model (i_move_0, x0)
  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)
```

La variable `i_move_0` *est le nom frais associé à l'instance de* `move` *appelée dans* `radar`*. Il faut enfin substituer l'appel à* `infer` *par un appel à* `APF.infer` *dans le nœud* `main`*.*

```
node main (delta, alpha) =
  let d = APF.infer(radar_prior, radar_model, (delta, alpha)) in
  let theta, x = split d in plot theta; plot x
```

Correction. La propriété de correction établit que l'inference appliquée sur un modèle et son compilé définissent les mêmes flots de distributions.

Pour tout programme prog tel que $\emptyset, \emptyset \vdash prog : H, C$*, pour tout modèle probabiliste* m *de prog,*

$$[\![\texttt{infer}(m(obs))]\!] \equiv [\![\texttt{APF.infer}(m\_\texttt{prior}, m\_\texttt{model}, obs)]\!]$$

La sémantique de `APF.infer` est définie en appliquant `infer` après avoir échantillonné le paramètre constant α au début de l'exécution.

$$[\![\texttt{APF.infer}(m\_\texttt{prior}, m\_\texttt{model}, obs)]\!] = \\ [\![\texttt{infer}(m\_\texttt{model}(\alpha, obs)\ \texttt{where rec init}\ \alpha = \texttt{sample}(m\_\texttt{prior})\ \texttt{and}\ \alpha = \texttt{last}\ \alpha)]\!]$$

Deux expressions probabilistes sont équivalentes si elles définissent les mêmes flots de distributions. La sémantique étant exprimée en terme d'état initial et de fonction de transition, l'équivalence s'exprime avec une relation de bisimilarité [22] initialisée sur les états initiaux et propagée à travers les fonctions de transition [9]. La bisimilarité sur les environnements γ_1 et γ_2 met en rapport les nœuds probabilistes avec leur version compilée et garantit que toutes les autres valeurs sont identiques.

Définition 1 (Bisimilarité d'environnements)**.** Pour tout programme *prog*, pour tous γ_1, γ_2, $\gamma_1 \equiv \gamma_2$ si pour toute variable $x \in dom(\gamma_1) \cap dom(\gamma_2)$, $\gamma_1(x) = \gamma_2(x)$ et pour tout nœud probabiliste `proba` $m\ x$ `=` e tel que $C \vdash e : \xi$ dans *prog*.

$$\begin{cases} \gamma_1(m\_\texttt{init}) = \{\!|e|\!\}^{\text{init}}_{\gamma_1} \\ \gamma_1(m\_\texttt{step}) = \lambda v.\ \{\!|e|\!\}^{\text{step}}_{\gamma_1[x \leftarrow v]} \end{cases} \quad \text{et} \quad \begin{cases} \gamma_2(m\_\texttt{model\_init}) = \{\!|\mathcal{C}_\xi(e)|\!\}^{\text{init}}_{\gamma_2} \\ \gamma_2(m\_\texttt{model\_step}) = \lambda(\theta, v).\ \{\!|\mathcal{C}_\xi(e)|\!\}^{\text{step}}_{\gamma_2[dom(\xi) \leftarrow \theta][x \leftarrow v]} \\ \gamma_2(m\_\texttt{prior}) = img(\xi) \end{cases}$$

La bisimilarité sur les expressions probabilistes met en rapport les flots de mesures renvoyées à chaque instant par les fonctions de transition.

Définition 2 (Bisimilarité probabiliste)**.** Soient e_1 et e_2 des expressions probabilistes, et une bisimulation $\mathcal{P} \subseteq S$ `dist` $\times\ S$ `dist` entre les *distributions* d'états. $\{\!|e_1|\!\} \equiv_{\mathcal{P}} \{\!|e_2|\!\}$ lorsque :

— pour tout $\gamma_1 \equiv \gamma_2$, $(\delta([\![e_1]\!]^{\text{init}}_{\gamma_1}), \delta([\![e_2]\!]^{\text{init}}_{\gamma_2})) \in \mathcal{P}$ et

— pour tout $\gamma_1 \equiv \gamma_2$, si $(\sigma_1, \sigma_2) \in \mathcal{P}$

$$\begin{cases} \mu_1 = \lambda U.\ \int \sigma_1(ds_1) \{\!|e_1|\!\}^{\text{step}}_{\gamma_1}(s_1)(U), & o_1 = \frac{1}{\mu_1(\top)}\ \pi_{1*}\mu_1,\ \sigma_1' = \frac{1}{\mu_1(\top)}\ \pi_{2*}\mu_1 \\ \mu_2 = \lambda U.\ \int \sigma_2(ds_2) \{\!|e_2|\!\}^{\text{step}}_{\gamma_2}(s_2)(U), & o_2 = \frac{1}{\mu_2(\top)}\ \pi_{1*}\mu_2,\ \sigma_2' = \frac{1}{\mu_2(\top)}\ \pi_{2*}\mu_2 \end{cases} \text{alors} \begin{cases} (\sigma_1', \sigma_2') \in \mathcal{P} \\ o_1 = o_2 \end{cases}.$$

Nous prouvons la correction de la compilation dans un cadre simplifié.

Théorème 1. *Pour tout programme prog tel que* $\emptyset, \emptyset \vdash prog : H, C$*, pour tout modèle probabiliste* m *de prog sous la forme* `proba` m x `=` e `where rec init` θ `=` `sample(`d`)` `and` θ `=` `last` θ*, avec* $H(m) = \{\theta \leftarrow d\}$*, et* e *ne contient pas d'appel de nœud, alors* $[\![$`infer(`m`(`obs`))`$]\!] \equiv [\![$`APF.infer(`m`_prior,`m`_model,`obs`)`$]\!]$.

Les contraintes sur m ne sont pas excessives : e peut contenir des définitions locales, θ peut être un tuple de paramètre constants, et on peut inliner les appels de nœuds. De plus, on peut imaginer une analyse statique qui permet de vérifier que m est sous la bonne forme.

Démonstration. Vue la sémantique de `infer` et `APF.infer`, il suffit de montrer que : $\{[\![m(obs)]\!]\} \equiv_{\mathcal{P}} \{[\![m\_model(\alpha, obs)$ `where rec init` α `=` `sample(`m`_prior)` `and` α `=` `last` $\alpha]\!]\}$ pour une bisimulation $\mathcal{P}$. Comme $H(m) = \{\theta \leftarrow d\}$, alors $C \vdash e : \emptyset$ car le seul paramètre constant, θ, est défini à l'extérieur de e et donc $\mathcal{C}_\emptyset(e) = e$. On pose $\mathcal{P}\ \sigma_1\ \sigma_2$ ssi il existe σ tel que $\sigma_1 = \lambda U.\ \int \sigma(dm, ds)\delta((), (m, (), s))(U)$ et $\sigma_2 = \lambda U.\ \int \sigma(dm, ds)\delta(m, (), (((), ()), s))(U)$. On vérifie alors que $\mathcal{P}$ relie les états initiaux et se propage à travers les fonctions de transitions ce qui suppose de dérouler et faire commuter les intégrales imbriquées introduites par les environnements locaux. □

5 Implémentation

Il reste maintenant à implémenter la construction `APF.infer(m_prior, m_model, obs)`. Dans cette section, nous décrivons d'abord comment implémenter les méthodes de Monte Carlo séquentielles pour les machines à états produites par la compilation d'un programme synchrone. Nous détaillons ensuite l'implémentation de *Assumed Parameter Filter*.

5.1 Méthodes de Monte Carlo séquentielles

Zelus compile les nœuds en machines à états. Une machine est une structure qui définit les méthodes suivantes : `alloc` alloue la mémoire nécessaire pour stocker l'état, `reset` réinitialise l'état en place, et `step` implémente la fonction de transition qui met à jour l'état en place. La mémoire est donc allouée statiquement avant l'exécution. Pour implémenter le filtrage, le compilateur génère également une méthode `copy` qui copie l'état d'une particule source dans celui d'une particule cible. En OCaml, on a ainsi [1] :

```
type (α, β) cnode = Cnode: { alloc : unit → 's;
                             reset : 's → unit;
                             step : 's → α → β;
                             copy : 's → 's → unit; } → (α, β) cnode
```

Les nœuds probabilistes sont également des machines à état mais la fonction `step` prend un argument supplémentaire `prob` contenant les informations nécessaires à l'inférence :

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

L'opérateur `infer` est un nœud d'ordre supérieur qui transforme une machine probabiliste de type `(`α`, `β`) pnode` en une machine déterministe de type `(`α`, `β` dist) cnode` qui calcule un flot de distributions.

L'implémentation de `infer` avec une méthode de Monte Carlo séquentielle, y compris *Assumed Parameter Filter*, approxime les distributions recherchée à l'aide d'un ensemble de particules indépendantes. L'état mémoire de `infer` est donc un tableau de particules qui sont des paires

1. Voir le fichier `lib/std/ztype.ml` du compilateur Zelus `https://github.com/inria/zelus`.

(état du modèle × score). La fonction de transition de `infer` exécute la fonction de transition du modèle sur chacun des états du tableau de particules en ajoutant l'argument `prob` qui contient en particulier le score de la particule qui peut être mis à jour par effet de bord. À la fin de l'instant, `infer` tire un nouvel ensemble de particules en fonction des scores obtenus : c'est le filtrage. Le tableau de particules est mis à jour à l'aide de la méthode `copy` du modèle qui permet de dupliquer certains états, et les scores sont réinitialisés.

Les constructions probabilistes `sample` et `observe` prennent aussi l'argument supplémentaire `prob` et implémentent l'échantillonneur décrit dans la Section 2.2 : `sample(prob, d)` tire un échantillon aléatoire dans la distribution `d`, et `observe(prob, (d, x))` incrémente le score de la particule avec la vraisemblance de l'observation $\mathtt{d}_{\mathrm{pdf}}(\mathtt{x})$.

5.2 *Assumed Parameter Filter*

Assumed Parameter Filter est une méthode de Monte Carlo séquentielle. L'implémentation de `APF.infer` suit donc de près le schéma présenté ci-dessus. Pour chaque particule, en plus de l'état et du score, `APF.infer` maintient une distribution de paramètres constants `d_theta`. La fonction de transition de `APF.infer` implémente l'Algorithme 1 décrit dans la Section 2.3 qui fait progresser les particules avant de mettre à jour la distribution `d_theta`.

Pour mettre à jour `d_theta`, il faut être capable de mesurer la vraisemblance d'une exécution où seule la valeur de `theta` a changé : $\mathit{Udpate}(\Theta_i^{t-1}, \lambda \mathit{theta}.\ \mathtt{model}(y_t \mid \theta, x_i^{t-1}, x_i^t))$ dans l'Algorithme 1. Il faut donc trouver un moyen de rejouer un pas d'exécution en fixant la valeur de tous les paramètres d'état.

Mémoriser les échantillons. On peut implémenter l'opérateur `sample` comme une machine à état et utiliser l'état mémoire pour mémoriser les valeurs échantillonnées lors de la première exécution du modèle : $x_i^t, w_i^t = \mathtt{model}(y_t \mid \theta_i, x_i^{t-1})$ dans l'Algorithme 1. Malheureusement, l'exécution du modèle modifie l'état du programme en place. Si on ne fait pas attention, la fonction *Udpate* utilise donc $\lambda\theta.\ \mathtt{model}(y_t \mid \theta, x_i^t, x_i^t)$ au lieu de $\lambda\theta.\ \mathtt{model}(y_t \mid \theta, x_i^{t-1}, x_i^t)$

Pour résoudre ce problème, nous proposons de stratifier la mémoire. Avant l'exécution du modèle, on copie superficiellement l'état qui contient des pointeurs vers l'état *profond* de chaque `sample`. Après l'exécution, on peut ainsi réutiliser cette copie de l'état où seul l'état des `sample` a été modifié. L'état de `sample` ne stocke donc pas directement la valeur échantillonnée, mais un pointeur vers une mémoire partagée. On distingue alors deux modes d'exécution gardés par une variable booléenne `replay`. Si `replay` est faux, l'appel à `sample(d)` tire un nouvel échantillon dans `d` et met à jour la mémoire partagée. Si `replay` est vrai, la valeur est déjà connue et l'appel à `sample(d)` se comporte comme `observe(d, m)` où `m` est le contenu de la mémoire partagée. En OCaml on a donc :

```
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 = Distribution.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; }
```

La méthode `copy` implémente la copie superficielle qui ne duplique que les pointeurs en préservant l'état profond.

L'exécution de chaque particule dans la fonction de transition de `infer` se décompose donc de la manière suivante :

1. Copier superficiellement l'état de départ `state` dans une variable `start_state`.
2. Échantillonner `theta` dans `d_theta`.
3. Exécuter `f_model.step(state, (prob, (theta, obs)))` avec `replay = false` pour mettre à jour l'état et mémoriser les valeurs de tous les paramètres d'état.
4. Mettre à jour la distribution `d_theta` en répétant plusieurs fois les deux étapes suivantes avec des valeurs différentes de `theta` :
   — Copier superficiellement `start_state` dans une variable `tmp_state`,
   — Exécuter `f_model.step(tmp_state, (prob, (theta, obs)))` avec `replay = true` pour réutiliser les valeurs des paramètres d'état.

Calcul des sorties. Contrairement à ce qui est présenté dans l'Algorithme 1 les sorties d'un modèle probabiliste ne correspondent pas forcément aux paramètres du modèle. Un modèle probabiliste peut renvoyer une expression arbitraire qui peut dépendre, ou non, de ces paramètres.

On ajoute donc une étape de génération à l'Algorithme 1. À la fin de l'instant, chaque particule tire un ensemble d'échantillons dans `d_theta` pour calculer un ensemble de sorties possibles. Chaque valeur de sortie est associée au score de la particule. Cet ensemble d'échantillons pondéré approxime la distribution recherchée.

Modularité. Notre implémentation ne fait pas d'hypothèse sur la fonction *Update* de l'Algorithme 1 qui peut donc implémenter diverses heuristiques. L'article original [16] se concentre sur la méthode de correspondance des moments. À chaque instant, on tire un ensemble d'échantillons aléatoires dans `d_theta` et on mesure les scores obtenus. `d_theta` devient alors la distribution Gaussienne qui approxime le mieux ces échantillons pondérés.

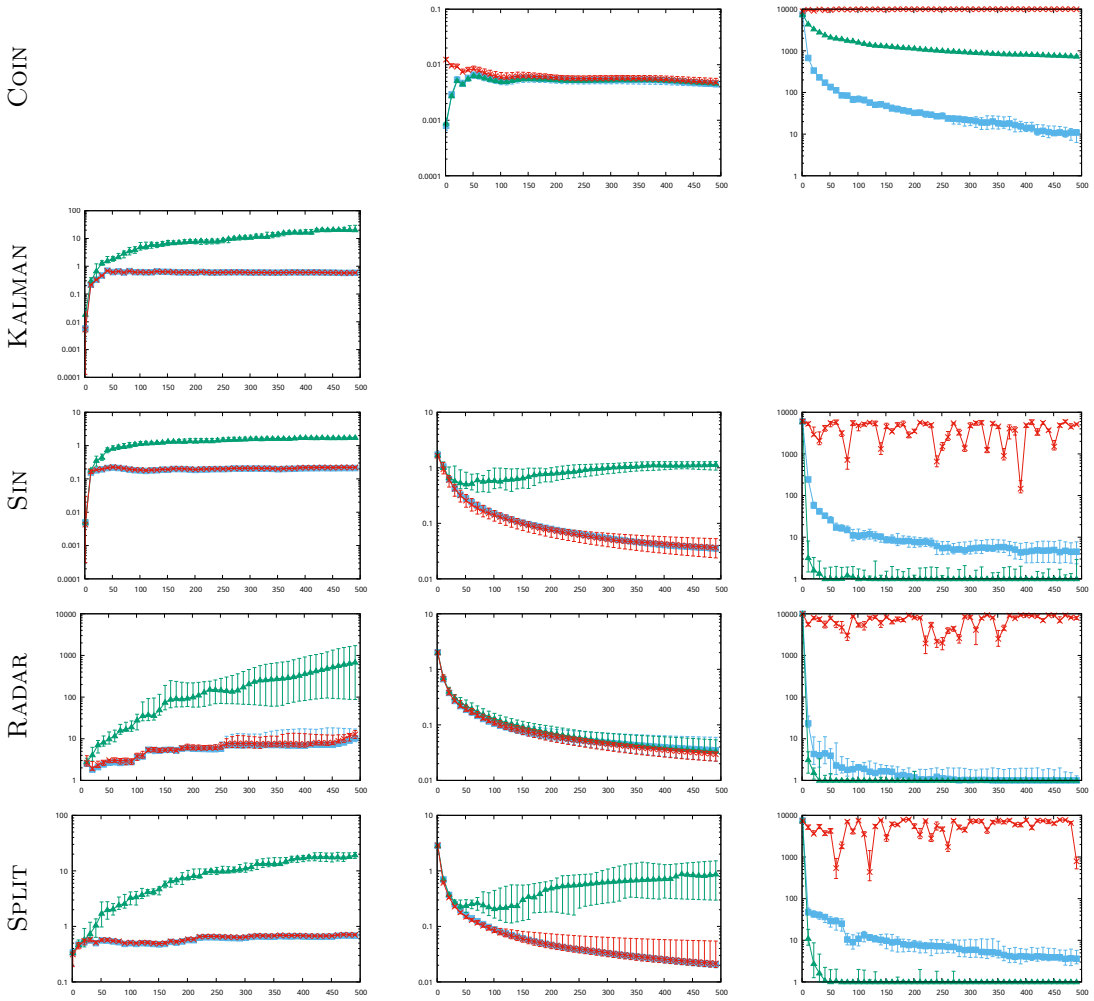
Nous avons également implémenté l'échantillonnage préférentiel, une méthode d'inférence sans filtrage. Au premier instant, on tire un ensemble de valeurs aléatoires pour les paramètres constants. Puis, à chaque instant, on met à jour le score associé à chacune de ces valeurs.

6 Évaluation

Dans cette section, nous validons expérimentalement notre implémentation d'APF sur un ensemble de modèles ProbZelus. Notre évaluation répond aux questions suivantes : (**QR1**) Quel est l'impact d'APF sur la précision des estimations à ressources fixées ? (**QR2**) APF résout-il le problème d'appauvrissement sur les estimations de paramètres constants ?

Méthode expérimentale. Pour chacun des exemples, on mesure l'évolution sur 500 pas de l'erreur quadratique moyenne (*Mean Square Error*, MSE) pour les paramètres d'état et les paramètre constants. Pour mesurer l'appauvrissement, on regarde également l'évolution de la taille effective d'échantillonnage (*Effective Sample Size*, ESS), une métrique qui indique le nombre de particules utiles. Les expériences ont été réalisées avec un processeur Intel Core i5-6200U (2.30GHz) avec 7.4GiO de mémoire vive.

On compare les performances d'APF avec deux autres méthodes d'inférence : l'échantillonnage préférentiel (*Importance Sampling*, IS), une méthode de Monte Carlo élémentaire sans filtrage, et le filtre particulaire (*Particle Filter*, PF), une méthode de Monte Carlo séquentielle avec filtrage à chaque instant. Pour chacune des méthodes d'inférence, on fixe les ressources à 10^4 particules. Sauf mention contraire, APF repartit les ressources disponibles avec 100 particules de filtrage et 100 particules d'échantillonnage pour calcul de la correspondance des moments.



(a) MSE paramètres d'état (b) MSE paramètres constants (c) ESS paramètres constants

FIGURE 8 – ■ PF, ▲ IS, × APF. Pour chacun des 5 modèles on mesure l'évolution sur 500 pas d'exécution du *mean square error* (MSE) pour les paramètres d'état et les paramètres constants, et le *effective sample size* (ESS) des paramètres constants. Les marqueurs indiquent les valeur médianes et les les barres d'erreurs correspondent aux 90% et 10% quantiles sur 10 exécutions.

Nous avons sélectionné des modèles qui illustrent plusieurs types de dépendances entre paramètres d'état et paramètres constants. **Coin** infère le biais d'une pièce à partir d'une série de jets [2]. Ce modèle n'a pas de paramètre d'état. APF utilise 10^4 particules d'échantillonnage. **Kalman** estime la position d'un agent à partir d'observations bruitées [2]. Ce modèle n'a pas de paramètre constant. APF utilise 10^4 particules de filtrage. **Sin** est similaire à Kalman, mais le modèle de mouvement fait intervenir une relation non-linéaire entre un paramètre constant et la position. C'est le modèle utilisé pour motiver APF dans l'article original [16]. **Split** fusionne deux problèmes indépendants : l'inférence d'un paramètre constant à partir d'observations bruitées et un filtre de Kalman. **Radar** est le modèle de radar de la Section 2.

QR1 : Précision Les résultats de précision sont présentés Figures 8a et 8b. On remarque que, sur tous les exemples, la précision d'APF est équivalente à celle de PF pour l'estimation des paramètres d'état et des paramètres constants. On note cependant une plus grande variabilité dans les résultat d'APF sur les paramètres constant, notamment sur les derniers pas d'exécution. Ce comportement s'explique par le phénomène d'appauvrissement : après autant de filtrages successifs, l'estimation de PF pour les paramètres constants ne repose plus que sur une unique valeur et on a perdu toute mesure sur l'incertitude de l'estimation.

QR2 : Appauvrissement. Les résultats d'appauvrissement sont présentés Figure 8c. L'ESS pour APF reste quasiment constant autour du nombre d'échantillons choisi pour générer les sorties. Tous les échantillons générés ajoutent de l'information sur la distribution recherchée.

Sans surprise, on observe que l'ESS pour PF décroît strictement à chaque nouveau filtrage. L'ESS pour IS s'effondre lui aussi mais pour une raison différente. Aucune particule n'est éliminée, mais la plupart des valeurs tirées initialement deviennent tellement improbables que leur score tombe à 0. Ces particules n'ajoutent plus d'information pour le calcul de la distribution et ne sont donc plus comptées dans l'ESS.

Coût. Sur les trois exemples précédents où APF repartit les ressources entre filtrage et échantillonnage APF est en moyenne 1.43x plus lent que PF (Sin : 1.32x, Radar : 1.65x, Split : 1.33x). Pour les modèles avec un seul type de paramètre où APF utilise toutes les ressources soit en filtrage, soit en échantillonnage, APF est en moyenne 2.63x plus lent que PF. Ces performances sont correctes si l'appauvrissement n'est pas acceptable et que l'on cherche à estimer précisément l'incertitude sur la valeur des paramètres constants.

7 Conclusion

Nous avons montré comment estimer les paramètres constants de modèles ProbZelus en évitant le problème d'appauvrissement grâce à l'algorithme d'inférence *Assumed Parameter Filter*. Notre implémentation repose sur une nouvelle analyse statique, une nouvelle passe de compilation, et un nouveau moteur d'inférence pour ProbZelus. Nos résultats expérimentaux montrent que cette méthode d'inférence permet d'éviter le problème d'appauvrissement pour les paramètres constants sans pénaliser la précision des estimations pour les paramètres d'état.

Travaux connexes L'analyse statique de la Section 3 est inspirée d'analyses spécifiques aux langages synchrones flot de données, en particulier l'analyse d'initialisation [11], qui vérifie que tous les flots sont bien définis au premier instant, et le typage des arguments statiques de Zelus, qui vérifie que certaines valeurs peuvent être calculées dès la compilation [7].

La compilation présentée en Section 4 se concentre uniquement sur les paramètres constants. C'est le moteur d'exécution qui se charge de fixer la valeur des paramètres d'état à l'aide d'une mémoire partagée pour rejouer un instant. Une alternative possible est de se rapprocher de la compilation des modèles hybrides de Zelus [4] et ajouter des entrées/sorties pour toutes les variables aléatoires introduites dans le modèle. On pourrait ainsi simplifier le moteur d'exécution au prix d'une compilation plus avancée.

La méthode d'inférence privilégiée de ProbZelus est *Delayed Sampling* [2], une méthode de Monte Carlo séquentielle qui calcule autant que possible des solutions analytiques exactes et n'échantillonne une variable aléatoire que lorsque le calcul symbolique échoue [21]. *Delayed Sampling* peut ainsi éviter l'appauvrissement si les paramètres constants n'ont jamais besoin d'être échantillonnés. Malheureusement, le moteur symbolique ne sait exploiter que des relations de conjugaison entre variables aléatoires, ou des relations affines [1]. On pourrait cependant combiner APF et *Delayed Sampling* pour améliorer la précision de ces deux méthodes.

Références

[1] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and M. Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. In *OOPSLA*, 2022.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.

[3] Guillaume Baudart, Louis Mandel, and Reyyan Tekin. Jax based parallel inference for reactive probabilistic programming. In *LCTES*, San Diego, USA, June 2022.

[4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle : types and compilation for a hybrid synchronous language. In *LCTES*, April 2011.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1) :64–83, 2003.

[6] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro : Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20 :28 :1–28 :6, 2019.

[7] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A synchronous look at the simulink standard library. *ACM Trans. Embed. Comput. Syst.*, 16(5s) :176 :1–176 :24, 2017.

[8] Timothy Bourke and Marc Pouzet. Zélus : a synchronous language with ODEs. In *HSCC*, pages 113–118. ACM, 2013.

[9] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *CMCS*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[10] Nicolas Chopin and Omiros Papaspiliopoulos. *An introduction to sequential Monte Carlo*. Springer, 2020.

[11] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer*, 6(3) :245–255, 2004.

[12] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *EMSOFT*, 2006.

[13] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6 : A formal language for embedded critical software development. In *TASE*, pages 1–11. IEEE Computer Society, 2017.

[14] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen : a general-purpose probabilistic programming system with programmable inference. In *PLDI*, pages 221–236. ACM, 2019.

[15] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *J. Royal Statistical Society : Series B (Statistical Methodology)*, 68(3) :411–436, 2006.

[16] Yusuf Bugra Erol, Yi Wu, Lei Li, and Stuart Russell. A nearly-black-box online algorithm for joint parameter and state estimation in temporal models. In *AAAI*, 2017.

[17] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing : A language for flexible probabilistic inference. In *Proceedings of Machine Learning Research*, 2018.

[18] Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014. Accessed April 2020.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[20] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam : A factored solution to the simultaneous localization and mapping problem. In *AAAI*, 2002.

[21] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *AISTATS*, 2018.

[22] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, 1981.