Delayed Sampling and Automatic Rao–Blackwellization of Probabilistic Programs

Lawrence M. Murray
Uppsala University

Daniel Lundén
KTH Royal Institute of Technology

Jan Kudlicka
Uppsala University

David Broman
KTH Royal Institute of Technology

Thomas B. Schön
Uppsala University

Abstract

We introduce a dynamic mechanism for the solution of analytically-tractable substructure in probabilistic programs, using conjugate priors and affine transformations to reduce variance in Monte Carlo estimators. For inference with Sequential Monte Carlo, this automatically yields improvements such as locally-optimal proposals and Rao–Blackwellization. The mechanism maintains a directed graph alongside the running program that evolves dynamically as operations are triggered upon it. Nodes of the graph represent random variables, edges the analytically-tractable relationships between them. Random variables remain in the graph for as long as possible, to be sampled only when they are used by the program in a way that cannot be resolved analytically. In the meantime, they are conditioned on as many observations as possible. We demonstrate the mechanism with a few pedagogical examples, as well as a linear-nonlinear state-space model with simulated data, and an epidemiological model with real data of a dengue outbreak in Micronesia. In all cases one or more variables are automatically marginalized out to significantly reduce variance in estimates of the marginal likelihood, in the final case facilitating a random-weight or pseudo-marginal-type importance sampler for parameter estimation. We have implemented the approach in Anglican and a new probabilistic programming language called Birch.

Proceedings of the 21^{st} International Conference on Artificial Intelligence and Statistics (AISTATS) 2018, Lanzarote, Spain. PMLR: Volume 84.

1 INTRODUCTION

Probabilistic programs extend graphical models with support for stochastic branches, in the form of conditionals, loops, and recursion. Because they are highly expressive, they pose a challenge in the design of appropriate inference algorithms. This work focuses on Sequential Monte Carlo (SMC) inference algorithms [4], extending an arc of research that includes probabilistic programming languages (PPLs) such as Venture [14], Anglican [26], Probabilistic C [18], WebPPL [8], Figaro [20], and Turing [7], as well as similarly-motivated software such as LibBi [16] and BiiPS [25].

The simplest SMC method, the bootstrap particle filter [9], requires only simulation—not pointwise evaluation—of the prior distribution. While widely applicable, it may be suboptimal with respect to Monte Carlo variance in situations where, in fact, pointwise evaluation is possible, so that other options are viable. One way of reducing Monte Carlo variance is to exploit analytical relationships between random variables, such as conjugate priors and affine transformations. Within SMC, this translates to improvements such as the locally-optimal proposal, variable elimination, and Rao–Blackwellization (see [5] for an overview). The present work seeks to automate such improvements for the user of a PPL.

Typically, a probabilistic program must be run in order to discover the relationships between random variables. Because of stochastic branches, different runs may discover different relationships, or even different random variables. While an equivalent graphical model might be constructed for any single run, it would constitute only partial observation. It may take many runs to observe the full model, if this is possible in finite time at all. We therefore seek a runtime mechanism for the solution of analytically-tractable substructure, rather than a compile-time mechanism of static analysis.

A general-purpose programming language can be augmented with some additional constructs, called *checkpoints*, to produce a PPL (see e.g. [26]). Two checkpoints are usual, denoted sample and observe. The first suggests that a value for a random variable needs to be sampled, the second that a value for a random variable is given and needs to be conditioned upon. At these checkpoints, random behavior may occur in the otherwise-deterministic execution of the program, and intervention may be required by an inference algorithm to produce a correct result.

The simplest inference algorithm instantiates a random variable when first encountered at a sample checkpoint, and updates a weight with the likelihood of a given value at an observe checkpoint. This produces samples from the prior distribution, weighted by their likelihood under the observations. It corresponds to importance sampling with the posterior as the target and the prior as the proposal. A more sophisticated inference algorithm runs multiple instances of the program simultaneously, pausing after each observe checkpoint to resample amongst executions. This corresponds to the bootstrap particle filter (see e.g. [26]).

These are *forward* methods, in the sense that checkpoints are executed in the order encountered, and sampling is myopic of future observations. The present work introduces a mechanism to change the order in which checkpoints are executed so that sampling can be informed by future observations, exploiting analytical relationships between random variables. This facilitates more sophisticated *forward-backward* methods, in the sense that information from future observations can be propagated backward through the program.

We refer to this new mechanism as *delayed sampling*. When a sample checkpoint is reached, its execution is delayed. Instead, a new node representing the random variable is inserted into a graph that is maintained alongside the running program. This graph resembles a directed graphical model of those random variables encountered so far that are involved in analytically-tractable relationships. Each node of the graph is marginalized and conditioned by analytical means for as long as possible until, eventually, it must be instantiated for the program to continue execution. This occurs when the random variable is passed as an argument to a function for which no analytical overload is provided. It is at this last possible moment that sampling is executed and the random variable instantiated.

Operations on the graph are forward-backward. The forward pass is a filter, marginalizing each latent variable over its parents and conditioning on observations, in all cases analytically. The backward pass produces a joint sample. This has some similarity to belief propagation [19], but the backward passes differ: belief propagation typically obtains the marginal posterior distribution of each variable, not a joint sample. Furthermore, in delayed sampling the graph evolves dynamically as the program executes, and at any time represents only a fraction of the full model. This means that some heuristic decisions must be made without complete knowledge of the model structure.

For SMC, delayed sampling yields locally-optimal proposals, variable elimination, and Rao–Blackwellization, with some limitations, to be detailed later. At worst, it provides no benefit. There is little intrusion of the inference algorithm into modeling code, and possibly no intrusion with appropriate language support. This is important, as we consider the user experience and ergonomics of a PPL to be of primary importance.

Related work has considered analytical solutions to probabilistic programs. Where a full analytical solution is possible, it can be achieved via symbolic manipulations in Hakaru [23]. Where not, partial solutions using compile-time program transformations are considered in [17] to improve the acceptance rate of Metropolis–Hastings algorithms. This compile-time approach requires careful treatment of stochastic branches, and even then it may not be possible to propagate analytical solutions through them. Delayed sampling instead operates dynamically, at runtime. It handles stochastic branches without problems, but may introduce some additional execution overhead.

The paper is organized as follows. Section 2 introduces the delayed sampling mechanism. Section 3 provides a set of pedagogical examples and two empirical case studies. Section 4 discusses some limitations and future work. Supplementary material includes further details of the case studies and implementations.

2 METHODS

As a probabilistic program runs, its memory state evolves dynamically and stochastically over time, and can be considered a stochastic process. Let $t = 1, 2, \ldots$ index a sequence of checkpoints. These checkpoints may differ across program runs (this is one of the challenges of inference for probabilistic programs, see e.g. [27]). In contrast to the two-checkpoint sample-observe formulation, we define three checkpoint types:

- assume$(X, p(\cdot))$ to initialize a random variable X with prior distribution $p(\cdot)$,
- observe$(x, p(\cdot))$ to condition on a random variable X with likelihood $p(\cdot)$ having some value x,
- value(X) to realize a value for a random variable X previously encountered at an assume checkpoint.

We use the statistics convention that an uppercase character (e.g. X) denotes a random variable, while the corresponding lowercase character (e.g. x) denotes an instantiation of it.

An assume checkpoint does not result in a random variable being sampled: its sampling is delayed until later. A value checkpoint occurs the first time that a random variable, previously encountered by an assume, is used in such a way that its value is required. At this point it cannot be delayed any longer, and is sampled.

Denote the state of the running program at checkpoint t by $X_t \in \mathbb{X}_t$. This can be interpreted as the current memory state of the program. Randomness is exogenous and represented by the random process $U_t \in \mathbb{U}_t$. This may be, for example, random entropy, a pseudorandom number sequence, or uniformly distributed quasirandom numbers.

The program is a sequence of functions f_t that each maps a starting state $X_{t-1} = x_{t-1}$ and random input $U_t = u_t$ to an end state $X_t = x_t$, so that $x_t = f_t(x_{t-1}, u_t)$. Note that f_t is a deterministic function given its arguments. It is not permitted that f_t has any intrinsic randomness, only the extrinsic randomness provided by U_t.

The target distribution over X_t is $\pi_t(\mathrm{d}x_t)$, typically a Bayesian posterior. In general, the program cannot sample from this directly. Instead, it samples x_t from some proposal distribution $q_t(\mathrm{d}x_t)$, which in many cases is just the prior distribution $p_t(\mathrm{d}x_t)$. Then, assuming that both π_t and q_t admit densities, it computes an associated importance weight $w_t \propto \pi_t(x_t)/q_t(x_t)$. Assuming U_t is distributed according to $\xi_t(\mathrm{d}u_t)$, we have

$$q_t(\mathrm{d}x_t) = \int_{\mathbb{X}_{t-1}} \int_{\mathbb{U}_t} \delta_{f_t(x_{t-1},u_t)}(\mathrm{d}x_t)\xi_t(\mathrm{d}u_t)q_{t-1}(\mathrm{d}x_{t-1}),$$

where δ is the Dirac measure. For brevity, we omit the subscript t henceforth, and simply update the state for the next time, as though it is mutable.

2.1 Motivation

We are motivated by variance reduction in Monte Carlo estimators. Consider some functional $\varphi(X)$ of interest. We wish to compute expectations of the form:

$$\mathbb{E}_\pi[\varphi(X)] = \int_{\mathbb{X}} \varphi(x)\pi(\mathrm{d}x) = \int_{\mathbb{X}} \varphi(x)\frac{\pi(x)}{q(x)}q(\mathrm{d}x).$$

Self-normalized importance sampling estimates can be formed by running the program N times and computing (where superscript n indicates the nth program run):

$$\hat{\varphi} := \sum_{n=1}^{N} \bar{w}^n \varphi(x^n), \quad \bar{w}^n = w^n \Big/ \sum_{n=1}^{N} w^n.$$

A classic aim is to reduce mean squared error:

$$\mathrm{MSE}(\hat{\varphi}) = \mathbb{E}_q\left[(\hat{\varphi} - \mathbb{E}_\pi[\varphi(X)])^2\right].$$

One technique to do so is *Rao–Blackwellization* (see e.g. [21, §4.2]). Assume that, amongst the state X, there is some variable X_v which has been observed to have value x_v, some set of variables X_M which can be marginalized out analytically, and some other set of variables X_R which have been instantiated previously. The functional of interest is the incremental likelihood of x_v. An estimator would usually require instantiation of $X_M^n \sim p(\mathrm{d}x_M^n \mid x_R^n)$ for $n = 1, \ldots, N$, and computation of:

$$\hat{Z} := \sum_{n=1}^{N} \bar{w}^n p(x_v \mid x_M^n, x_R^n).$$

The Rao–Blackwellized estimator does not instantiate X_M, but rather marginalizes it out:

$$\hat{Z}_{RB} := \sum_{n=1}^{N} \bar{w}^n \int p(x_v \mid x_M^n, x_R^n)p(\mathrm{d}x_M^n \mid x_R^n).$$

By the law of total variance, $\mathrm{var}(\hat{Z}_{RB}) \leq \mathrm{var}(\hat{Z})$, and as $\hat{Z}$ and $\hat{Z}_{RB}$ are unbiased [3], $\mathrm{MSE}(\hat{Z}_{RB}) \leq \mathrm{MSE}(\hat{Z})$.

This form of Rao–Blackwellization is local to each checkpoint. While X_M is marginalized out, it may require instantiation at future checkpoints, and so it must also be possible to simulate $p(\mathrm{d}x_M \mid x_v, x_R)$.

2.2 Delayed sampling

Delayed sampling uses analytical relationships to reorder the execution of checkpoints and reduce variance. Each observe is executed as early as possible, and the sampling associated with assume is delayed for as long as possible, to be informed by observations in between.

Alongside the state X, we maintain a graph $G = (V, E)$. This is a directed graph consisting of a set of nodes V and set of edges $E \subset V \times V$, where $(u, v) \in E$ indicates a directed edge from a parent node u to a child node v. For $v \in V$, let $\mathrm{Pa}(v) = \{u \in V \mid (u, v) \in E\}$ denote its set of parents, and $\mathrm{Ch}(v) = \{u \in V \mid (v, u) \in E\}$ its set of children. Associated with each $v \in V$ is a random variable X_v (part of the state, X) and prior probability distribution $p_v(\mathrm{d}x_v \mid x_{\mathrm{Pa}(v)})$, now using the subscript of X to select that part of the state associated with a single node, or set of nodes. We partition V into three disjoint sets according to three states. Let

- $I \subseteq V$ be the set of nodes in an *initialized* state,
- $M \subseteq V$ be the set of nodes in a *marginalized* state,
- $R \subseteq V$ be the set of nodes in a *realized* state.

At some checkpoint, the program would usually have instantiated all variables in V with a simulated or observed value, whereas under delayed sampling only those in R are instantiated, while those in $I \cup M$ are delayed.

We will restrict the graph G to be a forest of zero or more disjoint trees, such that each node has at most one parent. This condition is easily ensured by construction: the implementation makes anything else impossible, i.e. only relationships between pairs of random variables are coded. There are some interesting relationships that cannot be represented as trees, such as a normal distribution with conjugate prior over both mean and variance, or multivariate normal distributions. We deal with these as special cases, collecting multiple nodes into single supernodes and implementing relationships between pairs of supernodes, much like the structure achieved by the junction tree algorithm [11].

The following invariants are preserved at all times:

1. If a node is in M then its parent is in M. (1)
2. A node has at most one child in M. (2)

These imply that the nodes of M form marginalized paths: one in each of the disjoint trees of G, from the root node to a node (possibly itself) in the same tree. We will refer to the unique such path in each tree as its *M-path*. The node at the start of the M-path is a root node, while the node at the end is referred to as a *terminal* node. Terminal nodes have a special place in the algorithms below, and are denoted by the set T.

By the invariants, each $v \in M \setminus T$ has a child $u \in M$; let $\mathrm{Fo}(v)$ denote the entire subtree with this child u as its root (the *forward* set). Otherwise let $\mathrm{Fo}(v)$ be the empty set. The graph G then encodes the distribution

$$\left(\prod_{v \in I} q_v(\mathrm{d}x_v \mid x_{\mathrm{Pa}(v)})\right)\left(\prod_{v \in M \setminus T} q_v(\mathrm{d}x_v \mid x_{R \setminus \mathrm{Fo}(v)})\right) \times \left(\prod_{v \in T} q_v(\mathrm{d}x_v \mid x_R)\right), \quad (3)$$

where q_v equals the prior for nodes in I, some updated distribution for nodes in M, and all nodes in R are instantiated. The distribution suggests why terminals (in the set T) are important: they are the nodes informed by all instantiated random variables up to the current point in the program, and can be immediately instantiated themselves. Other nodes in M await information to be propagated backward from their forward set before they, too, can be instantiated.

When the program reaches a checkpoint, it triggers operations on the graph (details follow):

- For assume$(X_v, p(\cdot))$, call Initialize$(v, p(\cdot))$, which inserts a new node v into the graph.
- For observe$(x_v, p(\cdot))$, call Initialize$(v, p(\cdot))$, then Graft(v), which turns v into a terminal node, then Observe(v), which assigns the observed value to v and updates its parent by conditioning.
- For value(X_v), call Graft(v), then Sample(v), which samples a value for v.

Figure 1 provides pseudocode for all operations; Figure 2 illustrates their combination. Operations are of two types: *local* and *recursive*. Local operations modify a single node and possibly its parent:

- Initialize$(v, p(\cdot))$ inserts a new node v into the graph. If v requires a parent, u (implied by $p(\cdot)$ having a conditional form, i.e. $p(\mathrm{d}x_v \mid x_u)$ not $p(\mathrm{d}x_v)$), then v is put in I and the edge (u, v) inserted. Otherwise, it is a root node and is put in M, with no edges inserted.
- Marginalize(v), where v is the child of a terminal node, moves v from I to M and updates its distribution by marginalizing over its parent.
- Sample(v) or Observe(v), where v is a terminal node, assigns a value to the associated random variable by either sampling or observing, moves v from M to R, and updates the distribution of its parent node by conditioning. Both Sample(v) and Observe(v) use an auxiliary function Realize(v) for their common operations.

As shown in the pseudocode, these local operations have strict preconditions that limit their use to only a subset of the nodes of the graph, e.g. only terminal nodes may be sampled or observed. As long as these preconditions are satisfied, the invariants (1) and (2) are maintained, and the graph G encodes the representation (3). This is straightforward to check.

The recursive operations realign the M-path to establish the preconditions for any given node, so that local operations may be applied to it. These have side effects, in that other nodes may be modified to achieve the realignment. The key recursive operation is Graft, which combines local operations to extend the M-path to a given node, making it a terminal node. Internally, Graft may call another recursive operation, Prune, to shorten the existing M-path by realizing one or more variables.

3 EXAMPLES

We have implemented delayed sampling in Anglican (see also [13]) and a new PPL called Birch. Details are

Program	**Checkpoint**	**Local operations**	**Commentary**
`x ~ N(0,1);`	assume(X)	Initialize(X)	Named `delay_triplet` in supplementary material.
`y ~ N(x,1);`	assume(Y)	Initialize(Y)	
`z ~ N(y,1);`	observe(z)	Initialize(Z)	
		Marginalize(Y)	No Marginalize(X) is necessary: X, as a root node, is initialized in the marginalized state.
		Marginalize(Z)	
		Observe(z)	
`print(x);`	value(X)	Sample(Y)	Samples $Y \sim p(\mathrm{d}y \mid z)$.
		Sample(X)	Samples $X \sim p(\mathrm{d}x \mid y, z)$.
`print(y);`			A value $Y = y$ is already known.
`x ~ N(0,1);`	assume(X)	Initialize(X)	Named `delay_iid` in supplementary material. It encodes multiple i.i.d. observations with a conjugate prior distribution over their mean.
`for (t in 1..T) {`			
`  y[t] ~ N(x,1);`	observe(y_t)	Initialize(y_t)	
		Marginalize(y_t)	
		Observe(y_t)	
`}`			
`print(x);`	value(X)	Sample(X)	Samples $X \sim p(\mathrm{d}x \mid y_1, \ldots, y_T)$.
`x ~ Bernoulli(p);`	assume(X)	Initialize(X)	Named `delay_spike_and_slab` in supplementary material. It encodes a spike-and-slab prior [15] often used in Bayesian linear regression.
`if (x) {`	value(X)	Sample(X)	
`  y ~ N(0,1);`	assume(Y)	Initialize(Y)	
`} else {`			
`  y <- 0;`			Used as a regular variable, no graph operations are triggered.
`}`			Y is marginalized or realized as some $Y = y$ by the end, according to the stochastic branch.
`x[1] ~ N(0,1);`	assume(X_1)	Initialize(X_1)	Named `delay_kalman` in supplementary material. It encodes a linear-Gaussian state-space model, for which delayed sampling yields a forward Kalman filter and backward simulation.
`y[1] ~ N(x[1],1);`	observe(y_1)	Initialize(y_1)	
		Marginalize(y_1)	
		Observe(y_1)	
`for (t in 2..T) {`			
`  x[t] ~ N(a*x[t-1],1);`	assume(X_t)	Initialize(X_t)	After each tth iteration of this loop, the distribution $p(\mathrm{d}x_t \mid y_1, \ldots, y_t)$ is obtained; the behavior corresponds to a Kalman filter.
`  y[t] ~ N(x[t],1);`	observe(y_t)	Initialize(y_t)	
		Marginalize(X_t)	
		Marginalize(y_t)	
		Observe(y_t)	
`}`			
`print(x[1]);`	value(X_1)	Sample(X_T)	Samples $X_T \sim p(\mathrm{d}x_T \mid y_1, \ldots, y_T)$.
		...	Recursively samples $X_t \sim p(\mathrm{d}x_t \mid x_{t+1}, y_1, \ldots, y_t)$ and computes $p(\mathrm{d}x_{t-1} \mid x_t, y_1, \ldots, y_{t-1})$.
		Sample(X_1)	Samples $X_1 \sim p(\mathrm{d}x_1 \mid x_2, y_1)$.

Table 1: Pedagogical examples of delayed sampling applied to four probabilistic programs, showing the programs themselves (first column), the checkpoints reached as they execute linearly from top to bottom (second column), the sequence of local operations that these trigger on the graph (third column), and commentary (fourth column). The programs use a Birch-like syntax. Random variables with given values (from earlier assignment) are annotated by underlining. The function `print` is assumed to accept real-valued arguments only, so may trigger a value checkpoint when used.

INITIALIZE($v, p(\cdot)$)
1 **if** p includes a parent node, u
2 $\quad I \leftarrow I \cup \{v\}$
3 $\quad E \leftarrow E \cup \{(u, v)\}$
4 $\quad q_v(\mathrm{d}x_v) \leftarrow p(\mathrm{d}x_v \mid x_u)$
5 **else**
6 $\quad M \leftarrow M \cup \{v\}$
7 $\quad q_v(\mathrm{d}x_v) \leftarrow p(\mathrm{d}x_v)$

MARGINALIZE(v)
1 **assert** $v \in I$ and v has a parent $u \in T$
2 $q_v(\mathrm{d}x_v) \leftarrow \int_{\mathbb{X}_u} p(\mathrm{d}x_v \mid x_u) q_u(\mathrm{d}x_u)$
3 $I \leftarrow I \setminus \{v\}$
4 $M \leftarrow M \cup \{v\}$

SAMPLE(v)
1 **assert** $v \in T$
2 draw $x_v \sim q_v(\mathrm{d}x_v)$
3 REALIZE(v)

OBSERVE(v)
1 **assert** $v \in T$
2 $w \leftarrow q_v(x_v) w$
3 REALIZE(v)

REALIZE(v)
1 **assert** $v \in T$
2 $M \leftarrow M \setminus \{v\}$
3 $R \leftarrow R \cup \{v\}$
4 **if** v has a parent u // condition parent
5 $\quad q_u(\mathrm{d}x_u) \leftarrow \frac{p(x_v|x_u) q_u(\mathrm{d}x_u)}{\int_{\mathbb{X}_u} p(x_v|x'_u) q_u(\mathrm{d}x'_u)}$
6 $\quad E \leftarrow E \setminus \{(u, v)\}$
7 **for** $u \in \mathrm{Ch}(v)$ // new roots from children
8 $\quad$ MARGINALIZE(u)
9 $\quad E \leftarrow E \setminus \{(v, u)\}$

GRAFT(v)
1 **if** $v \in M$
2 $\quad$ **if** v has a child $u \in M$
3 $\quad\quad$ PRUNE(u)
4 **else**
5 $\quad$ GRAFT(u) where u is the parent of v
6 $\quad$ MARGINALIZE(v)
7 **assert** $v \in T$

PRUNE(v)
1 **assert** $v \in M$
2 **if** v has a child $u \in M$
3 $\quad$ PRUNE(u)
4 SAMPLE(v)

Figure 1: Operations on the graph. The left arrow ($\leftarrow$) denotes assignment. Assigning to a distribution is interpreted as updating its hyperparameters.

given in Appendices C and D.

Table 1 provides pedagogical examples using a Birch-like syntax, showing the sequence of checkpoints and graph operations triggered as some simple programs execute. They show how delayed sampling behaves through programming structures such as conditionals and loops, including stochastic branches.

In addition, we provide two case studies where delayed sampling improves inference, firstly a linear-nonlinear state-space model with simulated data, secondly a vector-borne disease model with real data from an outbreak of dengue virus in Micronesia. We use a simple random-weight or pseudo-marginal-type importance sampling algorithm for both of these examples:

1. Run SMC on the probabilistic program with delayed sampling enabled, producing N number of samples $x^1, \ldots, x^N$ with associated weights $w^1, \ldots, w^N$ and a marginal likelihood estimate $\hat{Z}$.

2. Draw $a \in \{1, \ldots, N\}$ from the categorical distribution defined by $P(a) = w^a / \sum_{n=1}^{N} w^n$.

3. Output x^a with weight $\hat{Z}$.

This produces one sample with associated weight, but may be repeated as many times as necessary—in parallel, even—to produce an importance sample as large as desired. The success of the approach depends on the variance of $\hat{Z}$. This variance can be reduced by marginalizing out one or more variables (recall Section 2.1). This is what delayed sampling achieves, and so we compare the variance of $\hat{Z}$ with delayed sampling enabled and disabled. When disabled, the SMC algorithm is simply a bootstrap particle filter. When enabled, it yields a Rao–Blackwellized particle filter. Where parameters are involved (as in the second case study), the diversity of parameter values depletes through the resampling step of SMC. This has motivated more sophisticated methods for parameter estimation such as particle Markov chain Monte Carlo methods [1], also applied to probabilistic programs [28]. Particle Gibbs is an obvious candidate here. We find, however, that the reduction in variance afforded by marginalizing out one or more variables with delayed sampling is sufficient to enable the above importance sampling algorithm for the two case studies here.

3.1 Linear-nonlinear state-space model

The first example is that of a mixed linear-nonlinear state-space model. For this model, delayed sampling yields a particle filter with locally-optimal proposal and Rao–Blackwellization.

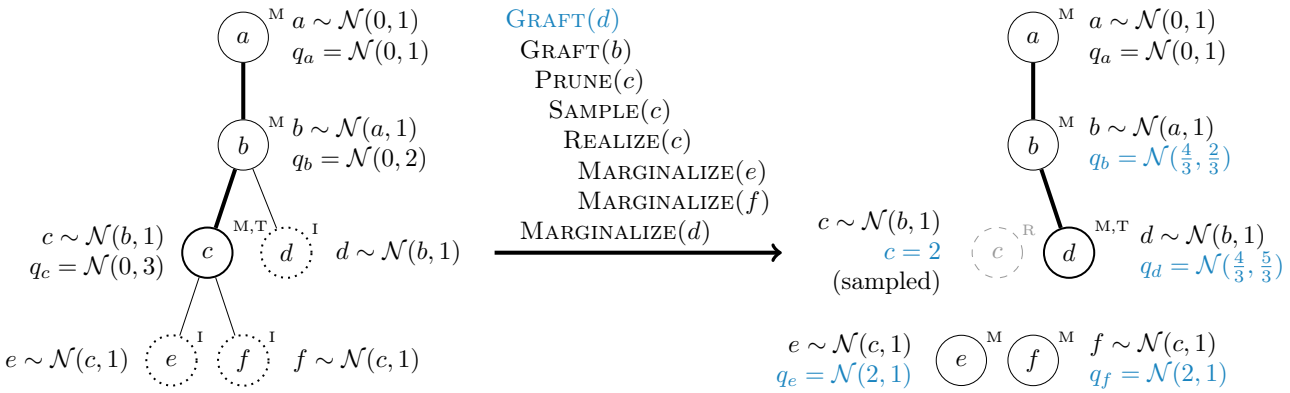

Figure 2: Demonstration of the M-path and operations on the graph. On the left, the M-path reaches from the root node, a, to the terminal node, c, marked in bold lines. The Graft operation is called for d. This requires a realignment of the M-path around b, pruning the previous M-path at c, then extending it through to d. The stack trace of operations is in the center, and the final state on the right. Descendants of c that were not on the M-path are now the roots of separate, disjoint trees.

The model is given by [12] and repeated in Appendix A. It consists of both nonlinear and linear-Gaussian state variables, as well as nonlinear and linear-Gaussian observations. Parameters are fixed. Ideally, the linear-Gaussian substructure is solved analytically (e.g. using a Kalman filter), leaving only the nonlinear substructure to sample (e.g. using a particle filter). The Rao–Blackwellized particle filter, also known as the marginalized particle filter, was designed to achieve precisely this [2, 22].

Delayed sampling automatically yields this method for this model, as long as analytical relationships between multivariate Gaussian distributions are encoded. In Birch these are implemented as supernodes: single nodes in the graph that contain multiple random variables. While the relationships between individual variables in a multivariate Gaussian have, in general, directed acyclic graph structure, their implementation as supernodes maintains the required tree structure.

The model is run for 100 time steps to simulate data. It is run again with SMC, conditioning on this data. For various numbers of particles, it is run 100 times to estimate $\hat{Z}$, with delayed sampling enabled and disabled. Figure 3 (left) plots the distribution of these estimates. Clearly, with delayed sampling enabled, fewer particles are needed to achieve comparable variance in the log-likelihood estimate.

3.2 Vector-borne disease model

The second example is an epidemiological case study of an outbreak of dengue virus: a mosquito-borne tropical disease with an estimated 50-100 million cases and 10000 deaths worldwide each year [24]. It is based on the study in [6], which jointly models two outbreaks of dengue virus and one of Zika virus in two separate locations (and populations) in Micronesia. Presented here is a simpler study limited to one of those outbreaks, specifically that of dengue on the Yap Main Islands in 2011. The data used consists of 172 observations of reported cases, on a daily basis during the main outbreak, and on a weekly basis before and after.

The model consists of two components, representing the human and mosquito populations, coupled via cross-infection. Each population is further divided into subpopulations of susceptible, exposed, infectious and recovered individuals. At each time step a binomial transfer occurs between subpopulations, parameterized with conjugate beta priors. Details are in Appendix B.

The task is both parameter and state estimation. For this model, delayed sampling produces a Rao–Blackwellized particle filter where parameters, rather than state variables, are marginalized out. While the state variables are sampled immediately, the parameters are maintained in a marginalized state, conditioned on the samples of these state variables. This is a consequence of conjugacy between the beta priors on parameters and the binomial likelihoods of the state variables (as pseudo-observations).

For various numbers of particles, SMC is run 100 times to estimate $\hat{Z}$, with delayed sampling enabled and disabled. Figure 3 (right) plots the distribution of these estimates. Clearly, with delayed sampling enabled, fewer particles are needed to achieve comparable variance in the log-likelihood estimate. Some posterior results are given in Appendix B.

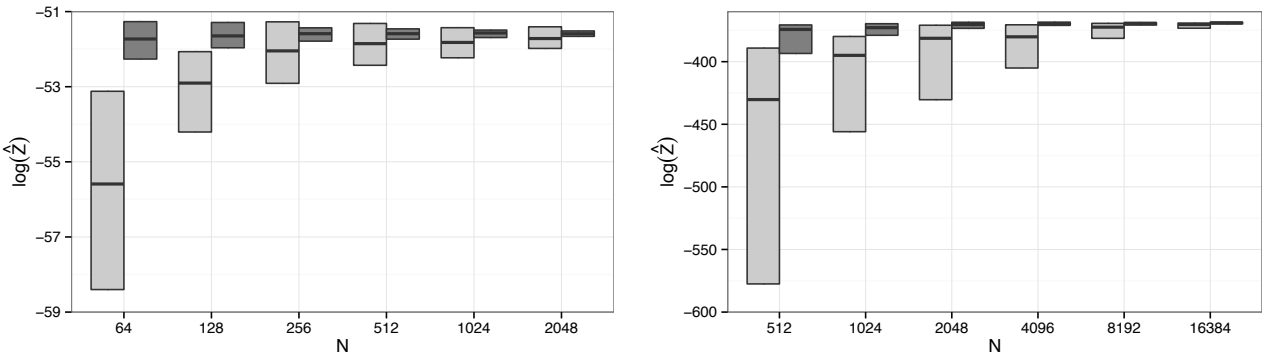

Figure 3: Distribution of the marginal log-likelihood estimate ($\log \hat{Z}$) for different numbers of particles (N) over 100 runs for (left) the linear-nonlinear state-space model, and (right) the vector-borne disease model, with (light gray) delayed sampling disabled, corresponding to a bootstrap particle filter, and (dark gray) delayed sampling enabled, corresponding to a Rao–Blackwellized particle filter. All runs use systematic resampling [10] when effective sample size falls below $0.7N$. Boxes indicate the interquartile range, midline the median. In both cases, significantly fewer particles are required to achieve comparable variance when delayed sampling is enabled.

4 DISCUSSION AND CONCLUSION

Table 1 demonstrates how delayed sampling operates through typical program structures such as conditionals and loops, including stochastic branches as encountered in probabilistic programs. Figure 3 demonstrates the potential gains. These are particularly encouraging given that the mechanism is mostly automatic.

Some limitations are worth noting. The graph of analytically-tractable relationships must be a forest of disjoint trees. It is unclear whether this is a significant limitation in practice, but support for more general structures may be desirable. It is worth emphasizing that this relates to the structure of analytically-tractable relationships and the ability of the mechanism to utilize them, not to the structure of the model as a whole. At present, for more general structures, some opportunities for variance reduction are missed. One remedy is to encode supernodes, as for the multivariate Gaussian distributions in Section 3.1.

Delayed sampling potentially reorders the sampling associated with assume checkpoints, and the interleaving of this amongst observe checkpoints, but does not reorder the execution of observe checkpoints. There is an opportunity cost to this. Consider the final example in Table 1: move the observations $y_1, \ldots, y_T$ into a second loop that traverses time backward from T to 1. Delayed sampling now draws each x_t from $p(\mathrm{d}x_t \mid x_{t+1}, y_t)$, not $p(\mathrm{d}x_t \mid x_{t+1}, y_1, \ldots, y_t)$. This is suboptimal but not incorrect: whatever the distribution, importance weights correct for its discrepancy from the target. It is again unclear whether this is a significant limitation in practice; examples seem contrived and easily fixed by reordering code.

While delayed sampling may reduce the number of samples required for comparable variance, it does require additional computation per sample. For univariate relationships (e.g. beta-binomial, gamma-Poisson), this overhead is constant and—we conjecture—likely worthwhile for any fixed computational budget. For multivariate relationships the overhead is more complex and may not be worthwhile (e.g. multivariate Gaussian conjugacies require matrix inversions that are $\mathcal{O}(N^3)$ in the number of dimensions). A thorough empirical comparison is beyond the scope of this article.

Finally, while the focus of this work is SMC, delayed sampling may be useful in other contexts. With undirected graphical models, for example, delayed sampling may produce a collapsed Gibbs sampler. This is left to future work.

Acknowledgements

This research was financially supported by the Swedish Foundation for Strategic Research (SSF) via the project *ASSEMBLE*. Jan Kudlicka was supported by the Swedish Research Council grant 2013-4853.

Supplementary material

Appendix A details the linear-nonlinear state-space model, and Appendix B the vector-borne disease model. Appendix C details the Anglican implementation, and Appendix D the Birch implementation. Code is included for the pedagogical examples in both Anglican and Birch, and for the empirical case studies, along with data sets, in Birch only.

References

[1] C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society B*, 72:269–302, 2010. doi: 10.1111/j.1467-9868.2009.00736.x.

[2] R. Chen and J. S. Liu. Mixture Kalman filters. *Journal of the Royal Statistical Society B*, 62:493–508, 2000.

[3] P. Del Moral. *Feynman-Kac Formulae: Genealogical and Interacting Particle Systems with Applications*. Springer–Verlag, New York, 2004.

[4] P. Del Moral, A. Doucet, and A. Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society B*, 68:441–436, 2006. doi: 10.1111/j.1467-9868.2006.00553.x.

[5] A. Doucet and A. M. Johansen. *A tutorial on particle filtering and smoothing: fifteen years later*, chapter 24, pages 656–704. Oxford University Press, 2011.

[6] S. Funk, A. J. Kucharski, A. Camacho, R. M. Eggo, L. Yakob, L. M. Murray, and W. J. Edmunds. Comparative analysis of dengue and Zika outbreaks reveals differences by setting and virus. *PLOS Neglected Tropical Diseases*, 10(12):1–16, 12 2016. doi: 10.1371/journal.pntd.0005173.

[7] H. Ge, A. Ścibior, K. Xu, and Z. Ghahramani. Turing: A fast imperative probabilistic programming language. Technical report, June 2016.

[8] N. D. Goodman and A. Stuhlmüller. The design and implementation of probabilistic programming languages. http://dippl.org, 2014.

[9] N. Gordon, D. Salmond, and A. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings-F*, 140:107–113, 1993. doi: 10.1049/ip-f-2.1993.0015.

[10] G. Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5:1–25, 1996. doi: 10.2307/1390750.

[11] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society B*, 1988.

[12] F. Lindsten and T. B. Schön. Identification of mixed linear/nonlinear state-space models. In *49th IEEE Conference on Decision and Control (CDC)*, pages 6377–6382, 2010.

[13] D. Lundén. Delayed sampling in the probabilistic programming language Anglican. Master's thesis, KTH Royal Institute of Technology, School of Computer Science and Communication, 2017.

[14] V. K. Mansinghka, D. Selsam, and Y. N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv abs/1404.0099*, 2014.

[15] T. J. Mitchell and J. J. Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83:1023–1032, 1988. doi: 10.2307/2290129.

[16] L. M. Murray. Bayesian state-space modelling on high-performance hardware using LibBi. *Journal of Statistical Software*, 67(10):1–36, 2015. doi: 10.18637/jss.v067.i10.

[17] A. Nori, C.-K. Hur, S. Rajamani, and S. Samuel. R2: An efficient MCMC sampler for probabilistic programs. *AAAI Conference on Artificial Intelligence (AAAI)*, 2014.

[18] B. Paige and F. Wood. A compilation target for probabilistic programming languages. *31st International Conference on Machine Learning (ICML)*, 2014.

[19] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[20] A. Pfeffer. *Practical Probabilistic Programming*. Manning, 2016.

[21] C. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer-Verlag New York, 2004. doi: 10.1007/978-1-4757-4145-2.

[22] T. Schön, F. Gustafsson, and P. Nordlund. Marginalized particle filters for mixed linear/nonlinear state-space models. *IEEE Transactions on Signal Processing*, 53:2279–2289, 2005. doi: 10.1214/193940307000000518.

[23] C. Shan and N. Ramsey. Exact Bayesian inference by symbolic disintegration. *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[24] J. D. Stanaway, D. S. Shepard, E. A. Undurraga, Y. A. Halasa, L. E. Coffeng, O. J. Brady, S. I. Hay, N. Bedi, I. M. Bensenor, C. A. Castañeda Orjuela, T.-W. Chuang, K. B. Gibney, Z. A. Memish, A. Rafay, K. N. Ukwaja, N. Yonemoto, and C. J. L. Murray. The global burden of dengue: an analysis from the Global Burden of Disease Study 2013. *The Lancet Infectious Diseases*, 16(6):712–723, 2016. doi: 10.1016/s1473-3099(16)00026-8.

[25] A. Todeschini, F. Caron, M. Fuentes, P. Legrand, and P. Del Moral. Biips: Software for Bayesian inference with interacting particle systems. *arXiv abs/1412.3779*, 2014.

[26] D. Tolpin, J. van de Meent, H. Yang, and F. Wood. Design and implementation of probabilistic programming language Anglican. *arXiv abs/1608.05263*, 2016.

[27] D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. *14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 770–778, 2011.

[28] F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2014.