Particle Gibbs with Ancestor Resampling for Probabilistic Programs

Jan-Willem van de Meent
Dept. of Statistics
Columbia University

Hongseok Yang
Dept. of Computer Science
University of Oxford

Frank Wood
Dept. of Engineering
University of Oxford

1 Introduction

In this paper we develop an implementation of a recently proposed inference algorithm known as particle Gibbs with ancestral resampling (PGAS) [Lindsten et al., 2012] for higher-order probabilistic programming languages. Higher-order probabilistic languages such as Church [Goodman et al., 2008], Venture [Mansinghka et al., 2014] and Anglican [Wood et al., 2014] allow statistical models to be represented as programs. Evaluation of a program F, which we will here represent as a sequence of statements $F = f_{1:N}$, instantiates random values $\boldsymbol{x}$ for some subset of the expressions, which can be thought of as a sample from a prior $p(\boldsymbol{x} \,|\, F)$. A program $F[\boldsymbol{x}]$ where all sampled values are substituted as constants is once again deterministic. This substitution also forms the basis for posterior inference, which samples the unconstrained variables from $p(\boldsymbol{x} \,|\, F[\boldsymbol{y}])$, where constants $\boldsymbol{y}$ are substituted for some random values in $F[\boldsymbol{y}]$. In higher-order languages it is straightforward to define non-parametric models, which can instantiate a arbitrary numbers of variables, or specify model structures recursively in terms of a generative grammar. This offers a greater flexibility relative to declarative systems such as Infer.NET [Minka et al., 2010] and STAN [STAN Development Team, 2014], which restrict the model definition syntax in order to omit higher-order functions and recursion. At the same time this expressivity makes it difficult to characterize the support $\mathcal{X}_F = \{\boldsymbol{x} : p(\boldsymbol{x} \,|\, F) > 0\}$, since neither the number of random values or their conditional dependencies are necessarily fixed for all possible execution histories.

Arguably the simplest inference algorithms for probabilistic languages use extensions of Sequential Monte Carlo (SMC) methods [Del Moral et al., 2006]. These "forward" techniques sequentially approximate the posterior $p(\boldsymbol{x}_n \,|\, F_n[\boldsymbol{y}_n])$ on a series of partial programs $F_n = f_{1:n}$ with particle sets $\{\boldsymbol{x}_n^{1:L}, w_n^{1:L}\}$. At each *generation*, $\boldsymbol{x}_n^{1:L}$ is instantiated by running multiple copies of F_n relative to ancestor states $\boldsymbol{x}_{n-1}^{\boldsymbol{a}_n}$, where $\boldsymbol{a}_n = a_n^{1:L}$ denotes a set of ancestor indices. The new particle set is assigned importance weights $w_n^l = p(\boldsymbol{y}_n \,|\, F[\boldsymbol{x}_n^l])$, which are used to sample a_n^l with probability proportional to w_n^l [Wood et al., 2014, Paige and Wood, 2014]. A more general framework for MCMC inference is offered by Venture [Mansinghka et al., 2014], which represents the execution history of a program as a graph, where each evaluated expression constitutes a node. A graph walk can then obtain the subset of expressions influenced by a proposed change, allowing the program to be re-executed conditioned on all unaffected nodes. This strategy allows formulation of a variety of MCMC methods as well SMC variants.

PGAS [Lindsten et al., 2012] is a particle MCMC method [Andrieu et al., 2010] where a particle set is sampled conditioned on a retained particle. This retained particle is itself resampled during as part of a SMC sweep that considers all possible *concatenations* of a newly sampled program (which we here call a *prefix*) to a retained partial program (which we call a *suffix*). This operation requires that we solve two nontrivial technical problems. The first is the reconstruction of a valid set of variables in the global environment, where bindings in the newly generated particle replace bindings in the retained particle, and dependent variables are updated accordingly. The second is updating the probability for each retained sample and conditioning expression in light of the updated variables, where one would like to avoid recomputation of expressions that are unaffected.

2 Particle Gibbs with Ancestor Resampling

Formally, SMC sampling is an auxiliary variable scheme that associates a set of ancestor lineages $a_{2:N}^{1:L}$ with the random variables $x_{1:N}^{1:L}$ and targets a density on an extended space

$$\psi(x_{1:N}^{1:L}, a_{2:N}^{1:L}) = \prod_{l=1}^{L} \rho_1(x_1^l) \prod_{n=2}^{N} \prod_{l=1}^{L} \bar{w}_{n-1}^{a_n} \rho_n(\boldsymbol{x}_n^l \,|\, \boldsymbol{x}_{n-1}^{a_n}) \;,$$

where $\bar{w}_{n-1}^l = w_{n-1}^l / \sum_k w_{n-1}^k$. In a probabilistic programming setting, this density expands to

$$\psi(x_{1:N}^{1:L}, a_{2:N}^{1:L}) = \quad \prod_{l=1}^{L} p(x_1^l \,|\, F_1) \prod_{n=2}^{N} \prod_{l=1}^{L} \bar{w}_{n-1}^{a_n} p(\boldsymbol{x}_n^l \,|\, F_n[\boldsymbol{x}_{n-1}^{a_n}]) \;.$$

A PGAS sampler initializes a particle set via SMC, and then selects a retained particle lineage $\boldsymbol{x}_N^k$ with probability $\bar{w}_N^k$. Subsequent sampler iterations target the density $\phi(\boldsymbol{x}^{1:L}, \boldsymbol{a}^{1:L}, k)$ with updates

1. $\{\boldsymbol{x}^{*,-k}, \boldsymbol{a}^*\} \sim \phi(\boldsymbol{x}^{-k}, \boldsymbol{a} \,|\, \boldsymbol{x}^k, k)$
2. $k^* \sim \phi(k \,|\, \boldsymbol{x}^{*,-k}, \boldsymbol{a}^*, \boldsymbol{x}^k)$

The first step is a SMC sweep, conditioned on a retained lineage $\boldsymbol{x}_N^k$ with indices $b_{1:N}$ recursively defined via $b_N = k$ and $b_{n-1} = a_n^{b_n}$. For each $n = 1, \ldots, N$ we sample ancestor indices $a_n^{*,-b_n}$ and values $\boldsymbol{x}_n^{*,-b_n}$ for all but the retained particle, and update retained index $a_n^{*,-b_n}$

1a. For all but the retained particle: $a_n^{*,l} \sim \mathscr{R}(a_n \,|\, w_{n-1}^*)$; $\boldsymbol{x}_n^{*,l} \sim p(\boldsymbol{x}_n \,|\, F_n[\boldsymbol{x}_{n-1}^{*,a_n^{*,l}}])$.

1b. For the retained particle: $a_n^{*,b_n} \sim \mathscr{R}(a_n \,|\, w_{n-1|N}^*)$; $\boldsymbol{x}_n^{*,b_n} \leftarrow \boldsymbol{x}_n^{b_n}[\boldsymbol{x}_{n-1}^{a_n^{*,b_n}}]$.

Here, the notation $\boldsymbol{x}_n^{b_n}[\boldsymbol{x}_{n-1}^{a_n^{*,b_n}}]$ refers to a substitution operation where all random values in $\boldsymbol{x}_{n-1}^{a_n^{*,b_n}}$ replace values in $\boldsymbol{x}_n^{b_n}$. The ancestral weight $w_{n|N}$ is defined as

$$w_{n|N}^l = p(\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_{n-1}^l] \,|\, F[\boldsymbol{x}_{n-1}^l]). \tag{1}$$

3 Implementation

Computing $w_{n|N}^l$ amounts to updating the probabilities associated with each sampled and observed value in the suffix relative to the environment defined by the prefix. This operation is complicated by the fact that the suffix was sampled from the distribution $p(\boldsymbol{x}_N \,|\, F_N[\boldsymbol{x}_n^{b_n}])$, which may not have the same support as the distribution $p(\boldsymbol{x}_N \,|\, F_N[\boldsymbol{x}_n^l])$. This means we must provide a mechanism for *regeneration* of a consistent program state conditioned on the values $\boldsymbol{x}_N^k[\boldsymbol{x}_{n-1}^l]$. This regeneration procedure can either be *strict* in the sense that it fails when $p(\boldsymbol{x}_N^k[\boldsymbol{x}_{n-1}^l] \,|\, F_N) = 0$ or *non-strict* in the sense that regeneration reruns the program, reusing values from $\boldsymbol{x}_N^k[\boldsymbol{x}_{n-1}^l]$ where possible.

A conceptually simple forward approach to non-strict regeneration is offered by lightweight Metropolis Hastings (LMH) implementations [Wingate et al., 2011]. The main drawback of such approaches is that they require LN^2 full evaluations of the program, which quickly becomes prohibitively expensive as the dataset size increases. It is similarly straightforward to implement PGAS within Venture with order LN^2 asymptotic cost, but a more efficient implementation is made difficult by the fact that Venture (in it its current form) does not provide a mechanism to regenerate$F_N[\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_{n-1}^l]]$ in an asymptotically efficient manner.

Here we consider an implementation of strict regeneration. The steps needed to perform this operation are (1) the regeneration of a consistent set of global environment variables, which includes any bindings in the prefix, augmented with any bindings defined in the suffix (some of which may require re-evaluation as a result of changes to bindings in the prefix) (2) a verification that the flow control path in the suffix is consistent with the environment bindings in the prefix and (3) the recomputation of the probabilities of any sampled and observed values for which the probability depends on bindings in the prefix.

3.1 Traced Evaluation

We begin by introducing a set of type annotations for each value v that is returned upon evaluation of an expression e. In practical terms, each value in the language is *boxed* into a data structure which we call a trace. We represent a trace τ as tuple $(\mathtt{v}, \epsilon, l, \rho, \omega, \sigma, \phi)$. v is the value of the expression. ϵ is a partially evaluated expression, whose sub-expressions are themselves represented as traces. l is the accumulated log-density of `observe` evaluated within the expression. ρ is a mapping $\{\mathtt{s} \to \tau\}$ containing the subset of the global environment variables that were referenced in the evaluation of the e. ω is a mapping $\{\alpha \mapsto (\tau, \mathtt{v}, l)\}$. It contains an entry at address α for each `observe` that was evaluated in e and its sub-expressions. This entry is represented as a tuple $(\tau, \mathtt{v}, l)$ containing a trace of the first argument to the `observe` (which must be of the `distribution` type), the observed value v, and the associated log-density l. Similarly σ is a mapping $\{\alpha \mapsto (\tau, \mathtt{v})\}$ that contains an entry for each evaluated `sample` expression (which omits the associated log-density). The last component ϕ is also a mapping $\{\alpha \mapsto \tau\}$ and records all the traces that appear as conditions in the `if` expressions and, thus, influence the control flow of program execution.

3.2 Regeneration

We now define an operation $\mathcal{R}(\tau, R) = \tau'$ that regenerates a traced value relative to an environment R. This operation performs the following steps:

1. Re-evaluate the predicates: Compare $\tau = \phi(\alpha)$ to $\tau' = \mathcal{R}(\tau, R)$ for each α. Abort if τ' and τ have different v components. Otherwise update $\phi[\alpha \mapsto \tau']$.
2. Re-score `observe` expressions and statements: Let $(\tau, \mathtt{v}, l) = \omega(\alpha)$, and $\tau' = \mathcal{R}(\tau, R)$. If τ' and τ have different v components, recalculate $l' = \mathcal{L}(\tau', \mathtt{v})$ and update $\omega[\alpha \to (\tau', \mathtt{v}, l')]$. Otherwise, update $\omega[\alpha \to (\tau', \mathtt{v}, l)]$.
3. Re-score samples: Let $(\tau, \mathtt{v}) = \sigma(\alpha)$, and $\tau' = \mathcal{R}(\tau, R)$. Calculate $l' = \mathcal{L}(\tau', \mathtt{v})$ and update $\omega[\alpha \mapsto (\tau', \mathtt{v}, l')]$.
4. Regenerate the environment bindings: For all symbols s that do not exist in R, let $\tau' = \mathcal{R}(R, \rho(\mathtt{s}))$ and update $R[\mathtt{s} \mapsto \tau']$ and $\rho[\mathtt{s} \mapsto \tau']$. For existing symbols update $\rho[\mathtt{s} \mapsto R(\mathtt{s})]$. We for convenience assume that R is updated in place, though this may be avoided by having $\mathcal{R}$ return a tuple (R', τ').
5. If any bindings were given new values in step 4, regenerate all τ_i in ϵ to reconstruct ϵ'.
6. If ϵ' was constructed in 5, evaluate ϵ' and update v.

Rescoring an individual trace may be performed as part of the regeneration sweep by calculating a difference in log-density Δl. This Δl is the sum of all terms $l' - l$ in step 3 and all terms l' in step 4, and any $\Delta l'$ values return from recursive calls to $\mathcal{R}$. In practice build a map $\mathcal{C} = \{\tau \to \tau', \ldots\}$ on a call to $\mathcal{R}$, which is passed as an additional argument in recursive calls, effectively memoizing the computation relative to a given initial environment R. While the procedure here does not provide asymptotic guarantees, since regeneration of the environment will generally require many recursive lookups, this memoization will mitigate the computational cost of recursive calls to some extent.

3.3 Ancestor Resampling

Given an implementation of a traced evaluator and a regenerate/rescore procedure $\mathcal{R}$, an implementation for PGAS in probabilistic programs becomes straightforward. We represent the programs $F[\boldsymbol{x}_n^l]$ evaluated up to the n top-level statements as pairs (R_n^l, τ_n^l). We now construct a concatenated suffix $\mathcal{T}_{1:N}$ by recursively evaluating $(\mathtt{cons}\ \tau_n^{b_n}\ \mathcal{T}_{n+1:N}) \to \mathcal{T}_{n:N}$. For $n = 1, \ldots, N$, we now define $p(y_{n:N}, x_{n:N}^k \mid F_N[\boldsymbol{x}_{n-1}^{*,l}])$ as the log-density of the rescored trace $\mathcal{R}(T_{n:N}, R_{n-1}^{*,l}) = T_{n:N}^l$. Note here that by construction, we may extract $\mathcal{T}_{n+1:N}$ from $\mathcal{T}_{n:N}$ without additional computation.

4 Experiments

Figure 1 shows posterior inference results in a linear Gaussian model with 100 time points. As is to be expected, the ICSMC results show a high effective number of samples near the end of the

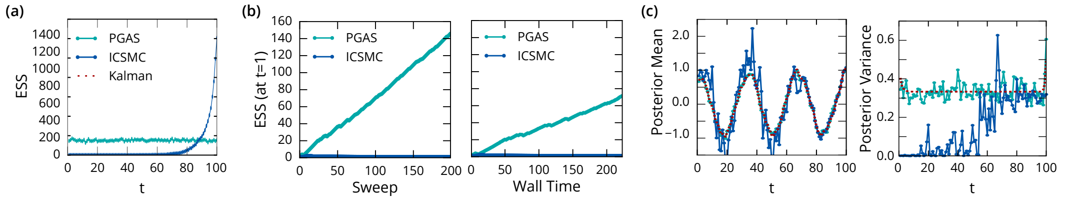

Figure 1: Posterior estimation in a linear Gaussian model with 100 time points. Sampling was performed with a iterative conditional SMC (ICSMC, i.e. particle Gibbs) method, using 2000 sweeps with 10 particles, and with PGAS using 200 sweeps, also with 10 particles for PGAS. **(a)** Effective sample size (ESS) as a function of time point. **(b)** ESS at the first time point as a function of sampler sweep and wall time. **(c)** Posterior mean and variance estimates, compared to the exact values calculated using Kalman smoothing.

sequence, but completely fail to mix at early time points. PGAS, by contrast maintains a completely homogeneous ESS of about approximately 180, implying that a new value is selected at every time point in 9 out of 10 sweeps (i.e. the maximum possible, given that the retained particle would be selected once every 10 sweeps in expectation). The effect of this is further underscored in the posterior estimates shown in figure 1c for the state mean and variance as compared to the exact estimates obtained via Kalman smoothing, which shows the effective sample size as a function of position t in the state sequence of the model. As is to be expected, the ICSMC results show a high effective number of samples near the end of the sequence, but completely fail to mix at early time points. These results illustrate that while PGAS certainly incurs a significant computational cost, it will easily yield better results in cases models where the parameters space is large and ICSMC sampling fails to mix.

References

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3): 269–342, 2010.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, June 2006. ISSN 1369-7412. doi: 10.1111/j.1467-9868.2006.00553.x. URL `http://doi.wiley.com/10.1111/j.1467-9868.2006.00553.x`.

Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

Fredrik Lindsten, Michael I Jordan, and Thomas B. Schön. Ancestor Sampling for Particle Gibbs. In *Advances in Neural Information Processing Systems*, pages 2591–2599, October 2012. URL `http://arxiv.org/abs/1210.6911`.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv*, page 78, March 2014. URL `http://arxiv.org/abs/1404.0099`.

T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

Brooks Paige and Frank Wood. A Compilation Target for Probabilistic Programming Languages. *International Conference on Machine Learning (ICML)*, 32, March 2014. URL `http://arxiv.org/abs/1403.0504v2http://arxiv.org/abs/1403.0504`.

STAN Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4. 2014.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 770_778, 2011.

F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014. URL `http://jmlr.org/proceedings/papers/v33/wood14.pdf`.