Hijacking Simulators with Universal Probabilistic Programming

Paper ID: 1010

Abstract

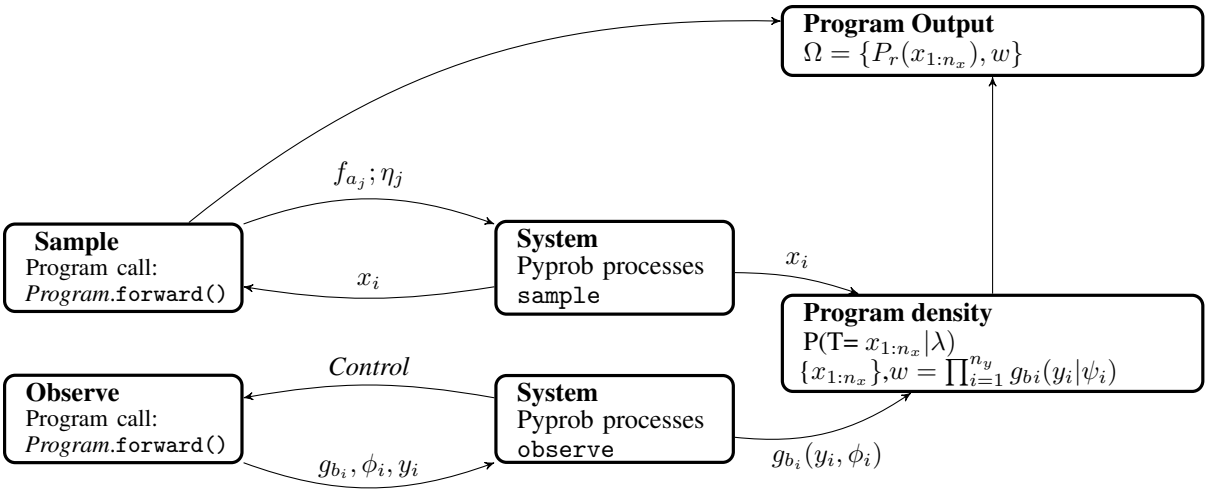
Notes on how to develop a formalism to perform inference in population based simulators.

1 Introduction

One challenge in performing inference in such models is the combinatorial increase in *paths* that can be taken within the given probabilistic model. This is because each member of the population has a bounded, but large number of decisions to make. In addition to this, each member of the population can interact with other members of the population. To model a single population member quantum-based methods leveraging linear superposition and entanglement provide an ability to explore multiple paths simultaneously maintaining the whole program, *state*, in one iteration. Unfortunately, analytically writing the functional form of the state is, in this instance, impossible. The usability of current classical inference methods in this context is also problematic. However, by combining hijacking and amortizing parts of the simulator, in particular rejection sampling loops, we can not only correctly perform inference in such models, as the rejection sampling loops are now replaced with learnt functions, but we can also perform inference much more efficiently. This is because the rejection sampling loops are now replaced with learnt functions which enables us to correctly construct the density and as they are amortized evaluation is cheap and only has to be performed once per call. This both reduces memory consumption and running time.

As updates are made sequentially MCMC methods typically are not-well suited this type of recursive estimation problem. As each time we get a new data point y_{t+1} we have to run new MCMC simulations for $P(x_{0:t_1}|y_{0:t+1})$, we also cannot directly reuse the samples $\{x^i_{0:t}\}$. Likewise, importance sampling is not well suited either, as we cannot reuse the samples and weights for a time t $\{x^i_{0:t}, \tilde{w}^i_t\}_{i=1}^N$ to sample from $(x_{0:t_1}|y_{0:t+1})$. However, as we can only run the simulator `forward()` we cannot return back to a previous state and re-run. So adapting existing inference techniques is the only viable option.

Preprint. Under review.



2 Limitations with different inference engines

2.1 Prior based sampling

In prior based sampling we take our existing program and look directly at the product of all the raw sample calls, denoted $f_{a_i}(x_i|\eta)$ and construct a proposal that is is dependent on those. In this sampling scheme our proposal is defined as:

$$q(x_{1:n_x}) = \begin{cases} \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta) \texttt{ if } \mathcal{B}(x_{1:n_x}) = 1 \\ 0 \end{cases} \tag{1}$$

where $\mathcal{B}(\cdot)$ is satisfied by the simulator and $f_{ai}(\cdot)$ represents a raw random sample, η is a subset of the variables in-scope at the point of sampling and n_x is the number of latent variables. This is true in all inference engines used in this hijacking framework. However, this makes use of no conditioning and is not sufficient for performing inference in complex, population-based simulations.

3 Non-prior Based Sampling

In non-prior based sampling we utilise conditioning too, which enables us to explore spaces more efficiently. In the current hijacking set-up, we can only do conditioning at the end of each `forward()` run of the simulator. This does inhibit certain inference engines and makes aspects of performing inference within this setting more complicated. Nonetheless, there are several inference schemes that we can exploit and utilise. We shall now give an Introduction to each of these schemes.

3.1 Importance Sampling

In the generalised probabilistic programming setting an importance sampling scheme over the program raw samples and observations can be defined as follows:

$$p(\mathcal{T} = x_{1:n_x}|\lambda) = \begin{cases} \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi) \texttt{if} B(x_{1:n_x}) = 1 \\ 0 \end{cases} \tag{2}$$

where the observe statements are defined by $g_{b_j}(.)$, ϕ plays the same role as η and n_y represents the number of observations. The expectation over the whole program can be defined as:

$$\mathbb{E}_{p(\Omega|\lambda)}[h(\Omega)] = \mathbb{E}_{p(x_{1:n_x}|\lambda)}[h(P_r(x_{1:n_x}))] \tag{3}$$

Note: It is a little weird how we define this, as the the program output Ω should also include the product of the weights

where r represents a run of the program, $P_r(\cdot)$ is the entirety of the program output, $h(\cdot)$ is just a deterministic function.

As the importance sampler updates the weights according to the following formula:

$$w(x_{1:n_x}) = \prod_{i=1}^{n_x} \frac{f_{ai}(x_i|\eta_i)}{q_i(x_i|f_{ai},\eta_i)} \prod_{j=1}^{n_y} g_{bj}(y_j|\phi_j) \tag{4}$$

When the number of observations is fixed, the combined weight is equal to $w(x_{1:n_x}) = \prod_{j=1}^{n_y} w_j$. To generate the weight up to each observe we can define $K(j)$ to be a function that returns the least i before hitting an observe j. This enables us to write the j-th weight as follows:

$$w_j(x1 : K(j)) = g_{bj}(y_j|\psi_j) \prod_{K(j-1)+1}^{K(j)} v_i \tag{5}$$

where we define $v_j = \frac{f_{aj}(x_j|\eta_j)}{q_j(x_j|f_{aj},\eta_j)}$ **Note: Ask Tom why the proposal is also dependent on f_{ai}.**

Taking these definitions we can construct a target density over the latents that is proportional to the products of `sample` and `observe` statements **Note: Add update posterior here** Thus, given the proposal $q(x_{1:K(t)})$ we define the importance weight as:

$$w(x_{1:t}) = \prod_{j=1}^{t} w_j(x_{1:K(j)}) = \\ \frac{p_t(x_{1:K(t)})}{q(x_{1:K(t)})} = \begin{cases} \prod_{j=1}^{t} g_{b_j}(y_j|w_i) \prod^{K}(j)_{K(j-1)+1} v_i \texttt{ if } B_t(x_1 : K(t)) = 1 \\ 0 \end{cases} \tag{6}$$

3.2 Sequential Monte Carlo

Given multiple observations, we can ake the inference more computationally efficient by employing sequential monte carlo (SMC), although as we can only condition at the end of one full run of the simulation, computational speed increases are minimal. SMC allows for conditioning on multiple observations, but the number of observations must be fixed. **Note: If we could condition on observations within the simulator, we could get dramatic speed increases over standard importance sampling.**

Algorithm 1 SMC for Population Based Simulators - well any general program - not new

procedure SMC(T, M,**x**, **w**,**v**, **g**)
  for $t = 1 : T$ **do**
    for $m = 1 : M$ **do**
      $w^m = 1$
      for $k = K(t-1)+1 : K(t)$ **do**
        $x_k^m \sim q_k(x_k^m) f^m_{a_k,\eta_j^m}$
        $w_t^m \leftarrow w_m^t v_j^m(x_k^m)$
      end for
    end for
    $w_t^m \leftarrow w_t^m g_b t^m(y_t^m|\psi_t^m)$
    $\{x_{1:t}^{1:m}\} \leftarrow$ `resample(` $\{x_{1:t}^{1:m}\}, \{w_{1:t}^{1:m}\}$)
  end for
end procedure

3.3 Random-walk Metropolis Hastings

Due to the current set-up performing Random-walk Metropolis Hastings (RMH) is not feasible as the rejection sampling blocks within the simulator mean that we cannot correctly perform RMH as we do not have a functional form for all parts of the simulator **Note: Re Word**

4 An Example: Modelling Parasite Densities in Simulation

4.1 The Model

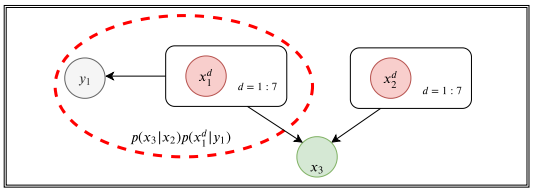


Figure 1: The DAG of the Parasite model.

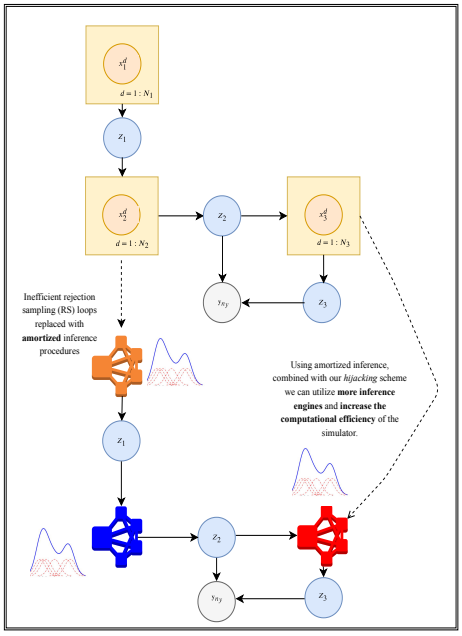


Figure 2: Using PyProb we can locate all regions of inefficient sampling and replace those points in the code with learnt functions that represent the distributions of the rejection sampling statements.