# Using probabilistic programs as proposals

**Marco F. Cusumano-Towner**
Probabilistic Computing Project
Massachusetts Institute of Technology
marcoct@mit.edu

**Vikash K. Mansinghka**
Probabilistic Computing Project
Massachusetts Institute of Technology
vkm@mit.edu

## Abstract

Monte Carlo inference has asymptotic guarantees, but can be slow when using generic proposals. Handcrafted proposals that rely on user knowledge about the posterior distribution can be efficient, but are difficult to derive and implement. This paper proposes to let users express their posterior knowledge in the form of *proposal programs*, which are samplers written in probabilistic programming languages. One strategy for writing good proposal programs is to combine domain-specific heuristic algorithms with neural network models. The heuristics identify high probability regions, and the neural networks model the posterior uncertainty around the outputs of the algorithm. Proposal programs can be used as proposal distributions in importance sampling and Metropolis-Hastings samplers without sacrificing asymptotic consistency, and can be optimized offline using inference compilation. Support for optimizing and using proposal programs is easily implemented in a sampling-based probabilistic programming runtime. The paper illustrates the proposed technique with a proposal program that combines RANSAC and neural networks to accelerate inference in a Bayesian linear regression with outliers model.

## 1 Introduction

Monte Carlo approaches to approximate inference include importance sampling, Markov chain Monte Carlo, and sequential Monte Carlo [1]. It is easy to construct Monte Carlo inference algorithms that have asymptotic guarantees, but constructing Monte Carlo inference algorithms that are accurate in practice often requires knowledge of the posterior distribution. Recent work in amortized inference [2–5] aims to acquire this knowledge by training neural networks on offline problem instances and generalizing to problem instances encountered at run-time. We propose to use probabilistic programming languages as a medium for users to encode their domain-specific knowledge about the posterior, and to optimize the program offline using an amortized inference approach to fill in gaps in this knowledge. In our setting, the model in which inference is being performed may or may not be itself represented by a probabilistic program.

This paper makes four contributions: First, we introduce the *proposal program* formalism. Proposal programs expose an interface that is stochastic generalization of the standard interface to proposal distributions. Instead of reporting proposal probabilities, proposal programs report estimates of these probabilities. We show that proposal programs can be used in place of regular proposal distributions within importance sampling and Metropolis-Hastings algorithms without sacrificing asymptotic consistency of these algorithms. Second, we give an amortized inference algorithm for offline optimization of proposal programs. Third, we propose a class of proposal programs based on combination of domain-specific heuristic randomized algorithms for identifying high-probability regions of the target distribution with neural networks that model the uncertainty in the target distibution around the outputs of the algorithm. Fourth, we discuss how proposal programs can be supported in a sampling-based probabilistic programming runtime, and give examples of proposal programs written in Gen.jl [6], a probabilistic language embedded in Julia [7].

## 2 Background and notation

### 2.1 Monte Carlo inference

Consider a target distribution $\pi(z)$ on latent variable(s) $z$, and let $\tilde{\pi}(z) := c\pi(z)$ be the unnormalized target distribution where $c > 0$ is a normalizing constant. Let $x$ denote the problem instance—this may include the observed data, or any other context relevant to defining the inference problem. In importance sampling, we sample many values $z$ from a proposal distribution $p(z; x)$ that is parameterized by $x$, and evaluate the importance weight $\tilde{\pi}(z)/p(z; x)$ for each sample, producing a weighted collection of samples that can be used to estimate expectations under $\pi$. In Markov chain Monte Carlo, we use one or more proposal distribution(s) $p_i(z; x)$ to construct a Markov chain that converges asymptotically to $\pi(z)$, and use iterates of one or more such chains to estimate expectations under $\pi$. Proposal distributions may be combined in cycles or mixtures [8], and each proposal distributions may only update a subset of the variables in $z$, leaving the remaining variables fixed.

### 2.2 Probabilistic programs

Consider a probabilistic programming framework that represents an execution history of a probabilistic program as a map $\rho : A \to V$ from the address (or 'name') of a random choices to its value, as introduced in [9], where $V$ is the set of all possible values taken by any random choice. We refer to such a map $\rho$ as a 'trace' of the program. In such a framework, a *probabilistic program*, denoted $\mathcal{P}$, is a program written in a probabilistic programming language endowed with an addressing scheme for random choices, along with a list of addresses $O \subseteq A$ indicating which random choices are the *outputs* of the program. We assume that the all of the output choices are made in all possible executions of the program. The restriction of the trace $\rho$ to the set of output addresses $O$, is called the 'output trace' and is denoted $z$. Let $I = A \setminus O$, and let $y$ denote the restriction of $\rho$ to the internal (i.e. non-output) random choices $I$, which is called the 'internal trace'. Let $x$ denote arguments of the program. We assume that all random choices are discrete—a rigorous treatment that includes continuous random choices is left for future work. Let $a(y) \subseteq I$ denote the names of random choices in an internal trace $y$, and let $p(y(i); y_{<i}, x)$ denote the probability of choice $i$ taking value $y(i)$ given the state of the program at that point in its execution. The probability $p(y; x)$ of an internal trace is the product of the probability of each internal random choice:

$$p(y; x) = \prod_{i \in a(y)} p(y(i); y_{<i}, x) \tag{1}$$

The probability of the output trace $p(z; y, x)$ given the input and internal trace is defined similarly. The joint probability on traces is then $p(y, z; x) = p(\rho; x) = p(y; x)p(z; y, x)$.

| | |
|---|---|
| Probabilistic program | $\mathcal{P}$ |
| Possible names of random choices | $A$ |
| Possible values for random choices | $V$ |
| Execution history (trace) | $\rho : A \to V$ |
| Output random choices | $O \subseteq A$ |
| Internal random choices | $I := A \setminus O$ |
| Arguments to program | $x$ |
| Internal trace | $y := \rho\|_I$ |
| Output trace | $z := \rho\|_O$ |
| Distribution on traces | $p(y, z; x) = p(y; x)p(z; y, x)$ |

**Figure 1:** Notation for probabilistic programs

## 3 Proposal programs: Using probabilistic programs to define proposals

A *proposal program* is a probabilistic program $\mathcal{P}$ whose output choices constitute a proposed value to some latent variable(s). The proposal distribution is the marginal distribution on the output trace of the program:

$$p(z; x) = \sum_y p(y; x)p(z; x, y) \tag{2}$$

In order to use a proposal program as a proposal distribution within a standard importance sampling (IS) algorithm, we require the ability to sample from the proposal distribution and evaluate the proposal probability $p(z; x)$. To use the proposal program inside standard Metropolis-Hastings (MH), we require the ability to sample $z'$ from the

**procedure** SIMULATE($\mathcal{P}$, $x$)
    $z \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$
    $\xi \leftarrow p(z; x)$ ▷ Compute output probability
    **return** $(z, \xi)$
**end procedure**

**procedure** ASSESS($\mathcal{P}$, $x$, $z$)
    $\xi \leftarrow p(z; x)$ ▷ Compute output probability
    **return** $\xi$
**end procedure**

**(a)** Standard interface for proposal distributions

**procedure** SIMULATE($\mathcal{P}$, $x$, $K$)
    $y_1, z \sim p(\cdot, \cdot; x)$ ▷ Execute $\mathcal{P}$
    **for** $k \leftarrow 2 \ldots K$ **do**
        $y_k \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$ (excluding outputs)
    **end for**
    $\hat{\xi} \leftarrow \frac{1}{K} \sum_{k=1}^{K} p(z; y_k, x)$
    **return** $\left(z, \hat{\xi}\right)$
**end procedure**

**procedure** ASSESS($\mathcal{P}$, $x$, $z$, $K$)
    **for** $k \leftarrow 1 \ldots K$ **do**
        $y_k \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$ (excluding outputs)
    **end for**
    $\hat{\xi} \leftarrow \frac{1}{K} \sum_{k=1}^{K} p(z; y_k, x)$
    **return** $\hat{\xi}$
**end procedure**

**(b)** Proposed interface for proposal programs

**Figure 2:** Primitive operations SIMULATE and ASSESS for a proposal program $\mathcal{P}$ which are sufficient for using $\mathcal{P}$ as a proposal in importance sampling and Metropolis-Hastings. (a) shows a standard implementation of the interface that is intractable in the general case, and (b) shows a novel relaxation of the interface that returns estimates in place of proposal probabilities.

proposal and evaluate $p(z'; x)$ for the resulting sample (where $x$ may contain the previous iterate $z_{t-1}$), as well as the ability to compute the proposal probability $p(z_{t-1}; x')$ (the probability that the proposal generates the reverse move, where $x'$ may contain the proposed iterate $z'$). Figure 2a summarizes an interface for proposal programs based on Equation (2) that is sufficient for using proposal programs with the standard IS and MH algorithms. The SIMULATE operation executes the proposal program, and returns the output trace $z$ and the marginal output probability $p(z; x)$. The ASSESS operation takes an output trace $z$ and returns $p(z; x)$.

However, evaluating $p(z; x)$ is intractable in the general case because Equation (2) contains a number of terms that is exponential in the number of random choices. Therefore, we propose an approximate implementation of the interface, shown in Figure 2b. The approximate SIMULATE operation executes the proposal program, and returns the output trace $z$ and an estimate $\hat{\xi}$ of the marginal output probability $p(z; x)$. The approximate ASSESS operation takes an output trace $z$ and returns an estimate $\hat{\xi}$ of $p(z; x)$.

The operations of Figure 2b can be easily implemented in a sampling-based probabilistic programming runtime that records a probability for each random choice in a random database [9]. The runtime needs to be able to identify certain choices for the program as 'output' choices, and needs to be able to execute a program with the output choices constrained to given values. We implemented such a runtime for Gen.jl [6], a probabilistic programming language embedded in Julia [7]. In Gen.jl, a probabilistic program is a Julia function that has been annotated with the `@probabilistic` keyword, and in which some random choices have been annotated with unique names using the `@choice` keyword. Gen.jl includes a lightweight probabilistic programming runtime based on a random database [9] but where names of random choices are explicitly defined in the program text.

### 3.1 Theoretical guarantees

The remainder of this section shows that substituting the operations in Figure 2b in place of the standard operations in Figure 2a within importance sampling (IS) and Metropolis-Hastings (MH) preserves the asymptotic consistency of both algorithms. The proofs are based on auxiliary-variable constructions.

Algorithm 1 shows an importance sampling algorithm that uses the SIMULATE operation of Figure 2b. Algorithm 1 can be interpreted as using unbiased estimates of the importance weights instead of the true importance weight, which would require a tedious or intractable marginalization over all internal traces $y$ for a proposal program.

**Theorem 1.** *For target distribution $\pi(z)$, proposal program $\mathcal{P}$ such that $\pi(z) > 0 \implies p(z; x) > 0$ for all $x$, and $p(z; x, y) > 0$ for all $x, y, z$, and function $f(z)$ with $\mathbb{E}_{z \sim \pi}[f(z)] = \mu$, the estimate $\hat{\mu}_N$ of Algorithm 1 converges almost surely to $\mu$ as $N \to \infty$.*

See Appendix A.1 for proof.

**Algorithm 1** Importance sampling using a proposal program

**Require:** Target distribution $\pi(z)$; unnormalized target distribution $\tilde{\pi}(z) = c\pi(z)$ for some $c > 0$; test function $f(z)$; proposal program $\mathcal{P}$; proposal arguments $x$; integers $N, K \geq 1$.

**for** $i \leftarrow 1, \ldots, N$ **do**

$(z^{(i)}, \hat{\xi}^{(i)}) \sim \text{SIMULATE}(\mathcal{P}, x, K)$

$w^{(i)} \leftarrow \tilde{\pi}(z^{(i)})/\hat{\xi}^{(i)}$

**end for**

$\hat{\mu}_N \leftarrow \dfrac{\sum_{i=1}^N w^{(i)} f(z^{(i)})}{\sum_{i=1}^N w^{(i)}}$

**return** $\hat{\mu}^N$

Algorithm 2 shows a Metropolis-Hastings transition operatior that uses the SIMULATE and ASSESS operations of Figure 2b. Note that this transition operator may be composed with other transition operators to construct an ergodic Markov chain. We consider the stationary distribution of the transition operator—the ergodicity of the Markov chain depends on the broader context including any other proposal kernels used.

**Algorithm 2** Metropolis-Hastings transition using a proposal program

**Require:** Unnormalized distribution $\tilde{\pi}(z)$; proposal program $\mathcal{P}$; integer $K \geq 1$; previous iterate $z_{t-1}$.

$(z', \hat{\xi}') \sim \text{SIMULATE}(\mathcal{P}, z_{t-1}, K)$

$\hat{\xi}_{t-1} \sim \text{ASSESS}(\mathcal{P}, z_{t-1}, z', K)$

$\alpha \leftarrow \dfrac{\pi(z')\hat{\xi}_{t-1}}{\pi(z_{t-1})\hat{\xi}'}$

$r \sim \text{Uniform}(0, 1)$

**if** $r \leq \alpha$ **then**

$z_t \leftarrow z'$

**else**

$z_t \leftarrow z_{t-1}$

**end if**

**Theorem 2.** *The transition operator of Algorithm 2 admits $\pi(x)$ as a stationary distribution.*

See Appendix A.2 for proof.

The non-asymptotic accuracies of Algorithm 1 and Algorithm 2 depend on the choice of $K$, the number of executions of the program used within each invocation of SIMULATE or ASSESS. As $K \to \infty$, the algorithms behave like the corresponding IS and MH algorithms using the exact proposal probabilities. For finite $K = 1$, the deviation of these algorithms from the $K \to \infty$ limit depends on the details of the proposal program. If $z$ is independent of $y$ or if $y$ is deterministic, $K = 1$ provides exact proposal probabilities. We expect that small $K$ may be sufficient for proposal programs which have low mutual information between the internal trace and the output trace. A more detailed analysis of how performance depends on $K$ is left for future work.

## 4 Offline optimization of proposal programs

This section describes a technique to optimize parameters of a proposal program during an offline 'inference compilation' phase, prior to use in importance sampling or Metropolis-Hastings. We consider proposal programs parameterized by $\theta$, with joint distribution on traces:

$$p(y, z; x, \theta) = q(y; x, \theta)q(z; y, x, \theta) \tag{3}$$

During the offline inference compilation phase, we assume that it is possible to sample from a training distribution on the arguments $x$ of the proposal program and the outputs $z$ of the proposal program, denoted $r(x, z)$. We factor the training distribution into a distribution on arguments $r(x)$, and a desired distribution on outputs given arguments $r(z|x)$. If the model $\pi$ in which we are doing inference is a generative model where $\pi(z, x)$ is a joint distribution on latent variables $z$ and data (observation) $x$, then we can define the training distribution as $r(x, z) := \pi(z, x)$, so that the argument distribution is $\pi(x)$ and the desired output distribution is the posterior $\pi(z|x)$. For this choice of $r$, sampling from the training distribution is achieved by ancestral sampling of the latents and observations from the generative model. This training distribution was previously used to train inference

networks in [2, 5, 10]. A different training distribution based on gold-standard SMC sampling instead of joint generative simulation was previously used in [4]. Given $r(x, z)$, we seek to find $\theta$ that solve the following optimization problem, where $\mathrm{D}_{\mathrm{KL}}$ denotes the Kullback-Leibler (KL) divergence:

$$\min_{\theta} \mathbb{E}_{x \sim r(\cdot)} \left[ \mathrm{D}_{\mathrm{KL}}(r(z|x) || p(z; x, \theta)) \right] \tag{4}$$

This is equivalent to the maximizing the expected conditional log likelihood of the training data:

$$\max_{\theta} J(\theta) = \max_{\theta} \mathbb{E}_{z,x \sim r(\cdot,\cdot)} \left[ \log p(z; x, \theta) \right] \tag{5}$$

Because the proposal program may use internal random choices, we do not assume that it is possible to evaluate $\log p(z; x, \theta)$, making direct optimization of $J(\theta)$ difficult. Therefore, we instead maximize the a lower bound on $J(\theta)$, denoted $J^K(\theta)$.

$$J^K(\theta) := \mathbb{E}_{x,z \sim r(\cdot,\cdot)} \left[ \mathbb{E}_{y_{1:K} \overset{\text{i.i.d.}}{\sim} p(\cdot;x,\theta)} \left[ \log \hat{\xi}(y_{1:K}, z, x, \theta) \right] \right] \leq J(\theta) \tag{6}$$

$$\tag{7}$$

where we have defined $\hat{\xi}(y_{1:K}, z, x, \theta) := \frac{1}{K} \sum_{k=1}^{K} p(z; y_k, x, \theta)$. We maximize $J^K(\theta)$ using stochastic gradient ascent. First, we introduce some notation. For some $j \in \{1, \ldots, K\}$, let $\hat{\xi}(y_{-j}, x, z, \theta) := \frac{1}{K-1} \sum_{k \neq j}^{K} p(z; y_k, x, \theta)$. Let $W_k := p(z; y_k, x, \theta) / (\sum_{j=1}^{K} p(z; y_j, x, \theta))$. Also, define functions $h$ and $g$:

$$g(x, z, y_{1:K}, \theta) := \sum_{k=1}^{K} \left( \log \hat{\xi}(y_{1:K}, x, z, \theta) - \log \hat{\xi}(y_{-k}, x, z, \theta)) \right) \nabla_\theta \log p(y_k; x, \theta) \tag{8}$$

$$h(x, z, y_{1:K}, \theta) := \sum_{k=1}^{K} W_k \nabla_\theta \log \hat{\xi}(y_{1:K}, z, x, \theta) \tag{9}$$

We then use the following unbiased estimator of the gradient $\nabla_\theta J^K(\theta)$ based on the 'per-sample' baseline of [11]:

$$g(x, z, y_{1:K}, \theta) + h(x, z, y_{1:K}, \theta) \text{ for } x, z \sim r(\cdot, \cdot) \text{ and } y_{1:K} \overset{\text{i.i.d.}}{\sim} p(\cdot; x, \theta)$$

The first term $g(x, z, y_{1:K}, \theta)$ accounts for the effect of $\theta$ on the distibution on internal random choices $y$, while the second term $h(x, z, y_{1:K}, \theta)$ accounts for the deterministic effect of $\theta$ on the output choices $z$. The resulting algorithm is shown below:

**Algorithm 3** Offline optimization of proposal program

**Require:** Training distribution $r(x, z)$; proposal program $\mathcal{P}$; integer $K \geq 1$, initial parameters $\theta_0$, mini-batch size $M$.

```
t ← 0
while not converged do
    t ← t + 1
    for m ← 1, ..., M do
        x^(m), z^(m) ~ r(·,·)                          ▷ Sample from training distribution
        for k ← 1, ..., K do
            y_k^(m) ~ p(·; x^(m), θ_{t-1})   ▷ Execute P (excluding output choices, which are constrained to z^(m))
            w_k^(m) ← p(z^(m); y_k^(m), x^(m), θ_{t-1})   ▷ Probability of output trace z^(m) given internal trace y_k^(m)
        end for
        W_{1:K}^(m) ← w_{1:K}^(m) / Σ_{k=1}^K w_k^(m)                          ▷ Normalize weights
        Δθ^(m) ← g(x^(m), z^(m), y_{1:K}^(m), θ_{t-1}) + h(x^(m), z^(m), y_{1:K}^(m), θ_{t-1})
    end for
    θ_t ← θ_{t-1} + ρ_t (1/M) Σ_{m=1}^M Δθ^(m)
end while
return θ_t
```

Algorithm 3 can be easily implemented on top of a sampling-based probabilistic programming runtime that includes reverse-mode automatic differentiation. The runtime must provide (1) the log probability of output random choices $\log p(z; x, y, \theta)$ and its gradient $\nabla_\theta \log p(z; x, y, \theta)$ and (2) the gradient of the log probability of internal random choices, $\nabla_\theta \log p(y; x, \theta)$. The gradient $\nabla_\theta \log \hat{\xi}(y_{1:K}, z, x, \theta)$ appearing in $h(x, z, y_{1:K}, \theta)$ can be computed from the collection of $\nabla_\theta \log p(z; x, y_k, \theta)$ for each $k$.

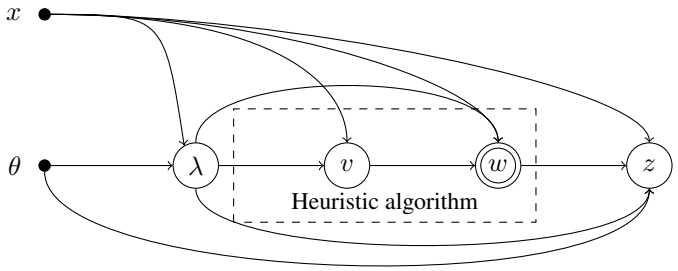



**Figure 3:** Proposed high-level data flow for a proposal program based on a heuristic randomized algorithm. $x$ is the input to the program, $\theta$ are the parameters of the program to be optimized, $\lambda$ are the parameters of the heuristic algorithm that are difficult to set manually, and $z$ is the output of the proposal program. The internal random choices $u$ of the proposal program include $\lambda$ and any internal random choices $v$ made by the heuristic algorithm, which need not be instrumented by the probabilistic runtime.

Note that the log probability of internal random choices $\log p(y; x, \theta)$ is not required (but its gradient is). Recall that the log probability of the internal random choices is the sum of the log probability of each internal random choice in the internal trace. For internal random choices that do not depend *deterministically* on $\theta$, the log probability contribution does not depend on $\theta$. Therefore, it suffices for the runtime to only instrument the internal random choices that deterministically depend on $\theta$. This permits use of black box code within the proposal program, which can reduce the overhead introduced by the probabilistic runtime in performance-critical sections of the proposal, and is important for our proposed methodology for using proposal programs in Section 5. We implemented Algorithm 3 on top of the Gen.jl runtime [6]. In Gen.jl, the user annotates random choices in a probabilistic program with a unique identifier. The user is free to use randomness in a probabilistic program without annotating it. In our implementation of Algorithm 3 we only accumulate the gradient $\nabla_\theta \log p(y; x, \theta)$ taking into account random choices that were annotated, relying on the user to ensure that there is no deterministic dependency of un-annotated random choices on $\theta$. In future work, automatic dependency analysis based on the approach of [12] could remove this reliance on the user.

## 5 Application: Proposals based on randomized mode-finding algorithms

Because Algorithm 3 minimizes the KL divergence from the posterior (or its proxy) to the proposal distribution, optimization will tend to force the proposal to have sufficient variability to cover the posterior mass, which provides a degree of robustness even when the heuristic is brittle. On problem instances where the heuristic gives wrong answers, the proposal variability should increase. On problem instances where the heuristic produces values near the posterior modes, the proposal variability should attempt to match the variability of each posterior mode.

Proposal programs permit the inference programmer to express their knowledge about the target distribution in the form of a sampling program. Consider the case when the user possesses a heuristic randomized algorithm that finds areas of high probability of the target distribution. Although such an algorithm contains some knowledge about the target distribution, the algorithm in isolation is not useful for peforming sound probabilistic inference—it does not necessarily attempt to represent the variability with the mode(s) of the target distribution, and its support may not even include the posterior support. The heuristic may also be brittle—it have parameters that need to be tuned for it to work on a particular problem instance, or it may fail completely on some problem instances.

To construct robust and sound probabilistic inference algorithms that can take advantage of heuristic randomized algorithms, we construct proposal programs that use the heuristic algorithm as a subroutine, and use the proposal program in importance sampling (Algorithm 1) or MCMC (Algorithm 2). Let $x$ denote the problem instance (e.g. the observed data or context variables); and let $\lambda$, $v$, and $w$ denote the parameters, internal random choices, and output of the algorithm, respectively. To construct the proposal program from the heuristic algorithm, we:

1. Make the parameters that are difficult to set manually into random choices $\lambda$ whose distribution may depend on $x$ (other parameters can be hardcoded or can have a hardcoded dependency on $x$).

2. Add random choices $z$ that are equal to the output $v$ of the heuristic algorithm plus some amount of noise, such that the support of $z$ given any $w$ and $x$ includes the support of the posterior $\pi(z)$. The choices $z$ are the output choices of the proposal program.

3. Parametrize the distribution of $\lambda$ given $x$ and the distribution of $z$ given $x$ and $w$ using the output of neural network(s) with parameters $\theta$. The networks take as input $x$ and generate parameter values $\lambda$ for the heuristic algorithm, as well as the amount of noise used to add to the output $w$ of the algorithm.

Figure 3 shows the high-level data flow of the resulting proposal program. We optimize the parameters $\theta$ during an offline inference compilation phase using Algorithm 3. Note that the internal random choices of the heuristic algorithm ($v$) do not depend deterministically on $\theta$. Therefore, to use Algorithm 3, the heuristic algorithm need not be instrumented by the probabilistic runtime, and can be fast black-box code.

## 6 Example: Proposal program combining RANSAC with a neural network

```
function ransac(xs, ys, params::RANSACParams)
    best_num_inliers::Int = -1
    best_slope::Float64 = NaN
    best_intercept::Float64 = NaN
    for i=1:params.num_iters

        # Select a random pair of points
        rand_ind = StatsBase.sample(1:length(xs), 2, replace=false)
        subset_xs = xs[rand_ind]
        subset_ys = ys[rand_ind]

        # Estimate slope and intercept using least squares
        A = hcat(subset_xs, ones(length(subset_xs)))
        slope, intercept = A\subset_ys

        # Count the number of inliers for this (slope, intercept) hypothesis
        ypred = intercept + slope * xs
        inliers = abs.(ys - ypred) .< params.epsilon
        num_inliers = sum(inliers)

        if num_inliers > best_num_inliers
            best_slope, best_intercept = slope, intercept
            best_num_inliers = num_inliers
        end
    end

    # return the hypothesis that resulted in the most inliers
    (best_slope, best_intercept)
end
```

**Figure 4:** A Julia implementation of a RANSAC-based heuristic algorithm for generating hypotheses (lines parameterized by a slope and intercept) that explain a given data set of x-y coordinates.

We illustrate Algorithm 1 and approach of Section 5, on a Bayesian linear regresssion with outliers inference problem. The data set is a set of x-y pairs $\{(x_i, y_i)\}_{i=1}^N$, the latent variables $z$ consist of the parameters of the line $(\alpha_1, \alpha_2)$ and binary variables $o_i$ indicating whether each data point $i$ is an outlier or not. The generative model $\pi(z, y)$, which is conditioned on the x-coordinates $x$, is defined below:

$$
\begin{aligned}
\alpha_1 &\sim \text{Normal}(0, 1) \quad \text{(slope)} \\
\alpha_2 &\sim \text{Normal}(0, 2) \quad \text{(intercept)} \\
o_i &\sim \text{Bernoulli}(0.1) \text{ for } i = 1 \ldots N \quad \text{(outlier or inlier)} \\
y_i | \alpha_1, \alpha_2, x_i, o_i &\sim \begin{cases} \text{Normal}(\alpha_1 x_i + \alpha_2, 1) & : o_i = 0 \quad \text{(inlier)} \\ \text{Normal}(\alpha_1 x_i + \alpha_2, 5.8) & : o_i = 1 \quad \text{(outlier)} \end{cases}
\end{aligned}
$$

The unnormalized target distribution $\tilde{\pi}(z)$ is the joint probability $\pi(z, y)$.

Random sample consensus (RANSAC, [13]) is an iterative algorithm that can quickly find lines near the posterior modes of the posterior on this problem, using deterministic least-squares algorithm within each iteration. Figure 4 shows a Julia implementation of a RANSAC algorithm for our linear-regression with outliers problem. However, the output distribution of the algorithm is an atomic set that does not sufficiently support the posterior distribution on lines, which is defined on $\mathbb{R}^2$. Also, the algorithm contains a parameter `epsilon` that needs to be roughly

```
@probabilistic function ransac_neural_proposal(xs, ys, params)

    # Generate parameters for RANSAC using learned parameters
    epsilon = @choice(gamma(exp(params.eps_alpha),
                            exp(params.eps_beta)), "epsilon")
    num_iters = @choice(categorical(params.iter_dist), "iters")
    ransac_params = RANSACParams(num_iters, epsilon)

    # Run RANSAC (uses many un-annotated random choices)
    slope_guess, intercept_guess = ransac(xs, ys, ransac_params)

    # Predict output variability using learned neural network
    nn_hidden = ewise(sigmoid, params.h_weights * vcat(xs, ys) + params.h_biases)
    nn_out = params.out_weights * nn_hidden + params.out_biases
    slope_scale, intercept_scale = (exp(nn_out[1]), exp(nn_out[2]))

    # Add noise
    slope = @choice(cauchy(slope_guess, slope_scale), "slope")
    intercept = @choice(cauchy(intercept_guess, intercept_scale), "intercept")

    # Generate outlier statuses from conditional distribution
    for (i, (x, y)) in enumerate(zip(xs, ys))
        p_outier = conditional_outlier(x, y, slope, intercept)
        @choice(flip(p_outlier), "outlier-$i")
    end
end
```

**Figure 5:** Proposal program in Gen.jl that uses RANSAC and a neural network to predict the line, followed by conditional sampling for the outlier variables

calibrated to the inlier variability in a particular data set, and it is not clear how many iterations of the algorithm to run.

To make use of RANSAC for robust and sound probabilistic inference, we wrote a proposal program in Gen.jl following the methodology proposed in Section 5, optimized the unknown parameters of the resulting proposal programs using Algorithm 3, and employed the optimized proposal programs within importance sampling (Algorithm 1). The proposal program, shown in Figure 5, (1) generates the parameter `epsilon` and the number of iterations of the RANSAC algorithm from distributions whose parameters are part of $\theta$, then (2) runs RANSAC, and (3) adds Cauchy-distributed noise to the resulting line parameters (slope and intercept), where the variability of the noise is determined by the output of a neural network whose parameters are part of $\theta$, and finally (4) samples the outlier binary variables from their conditional distributions given the line. Note that the random choices inside the `ransac` function are not annotated—the RANSAC procedure is executed as regular Julia code by Gen.jl's probabilistic runtime without incurring any overhead.

For comparison, we also implemented a proposal program (Figure 6) that does not use RANSAC, but instead generates the slope and intercept of the line from a distribution that depends on a neural network. We optimized both proposal programs using a modified version of Algorithm 3 that uses ADAM [14] instead of standard SGD. We used the generative model $\pi(z, y)$ as the training distribution, and we used $K = 100$ during training and minibatches of size 8, and 3000 iterations. The results are shown and discussed in Figure 7.

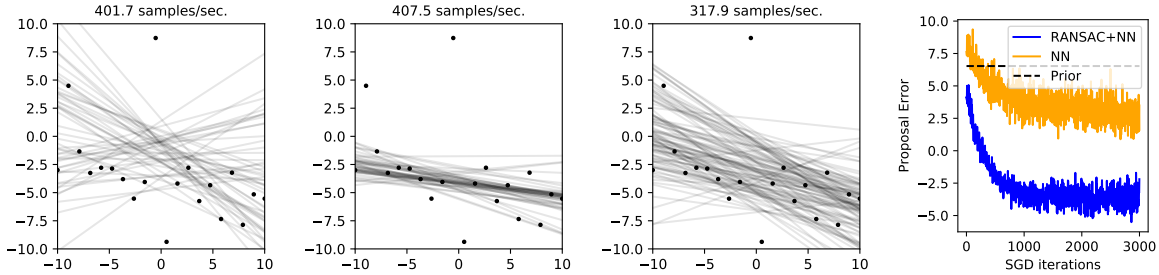
## 7 Related work

Some probabilistic programming systems support combining automated and custom inference strategies [15]. However, custom proposal distributions in [15] require user implementation of a density-evaluation function for the proposal (mirroring the interface of Figure 2a). Other probabilistic programming systems support the specification of 'guide programs' for use in importance sampling, sequential Monte Carlo, or variational inference [4, 16]. However, guide programs follow the control flow of a model program, or are restricted to using the same set of random choices. We focus on using general probabilistic programs to define proposal distributions, whether in the context of probabilistic programs for the model or not.

The Monte Carlo algorithms presented here derive their theoretical justification from auxiliary-variable constructions similar to those of pseudo-marginal MCMC [17] and random-weight particle filter [18]. In particular, our

```
@probabilistic function neural_proposal(xs, ys, params)

    # Generate randomness
    dim = 2
    latent = mvnormal(zeros(dim), eye(dim))

    # Construct feature vector
    features = vcat(latent, xs, ys)

    # Use neural network to predict parameters of line distribution
    hidden = ewise(sigmoid, params.hidden_weights * features + params.hidden_biases)
    output = params.output_weights * hidden + params.output_biases
    slope_mu = output[1]
    intercept_mu = output[2]
    slope_var = exp(output[3])
    intercept_var = exp(output[4])

    # Generate line
    slope = @choice(cauchy(slope_guess, slope_scale), "slope")
    intercept = @choice(cauchy(intercept_guess, intercept_scale), "intercept")

    # Generate outlier statuses from conditional distribution
    for (i, (x, y)) in enumerate(zip(xs, ys))
        p_outier = conditional_outlier(x, y, slope, intercept)
        @choice(flip(p_outlier), "outlier-$i")
    end
end
```

**Figure 6:** Proposal program in Gen.jl that a neural network to predict the line, followed by conditional sampling for the outlier variables




**(a)** IS (prior) **(b)** IS (RANSAC+NN) **(c)** IS (NN) **(d)** Training objective

**Figure 7:** (a) shows a dataset (points), and approximate posterior samples (lines) from an importance sampling algorithm using a prior proposal. (b) shows samples produced by Algorithm 1 using a proposal program (RANSAC+NN, Figure 5) that combines RANSAC with a neural network. (c) shows samples produced Algorithm 1 using a proposal program (NN, Figure 6) based on a neural network. Six particles ($N = 6$) were used for both proposals. Significantly more accurate samples are obtained with comparable throughput using the RANSAC+NN proposal program. (d) shows the estimated approximation error of the three proposals as the parameters of proposals are tuned offline using ADAM. Error is quantified using the expected KL divergence from the target distribution to the proposal distribution up to an unknown constant that does not depend on the proposal distribution, where the expectation is taken under datasets sampled from the model.

Metropolis-Hastings algorithm can be seen as an application of the general auxiliary-variable MCMC formalism of [19] to the setting where proposals are defined as probabilistic programs.

Recent work in 'amortized' or 'compiled' inference has studied techniques for offline optimizing of proposal distributions in importance sampling or sequential Monte Carlo [2–5]. Others have applied similar approaches to optimize proposal distributions for use in MCMC [10, 20]. However, the focus of these efforts is optimizing over neurally-parameterized distributions that inherit their structure from the generative model being targeted, do not contain their own internal random choices, and are therefore not suited to use with heuristic randomized algorithms. In contrast, we seek to allow the user to easily express and compute with custom proposal distributions that are

defined as arbitrary probabilistic programs that are independent of any possible structure in the probabilistic model being targeted, and may include 'internal' random choices not present in the target model.

There has been much recent work in probabilistic machine learning on training generative latent variable models using using stochastic gradient descent [21, 22, 11]. Our procedure for optimizing proposal programs is an application of the Monte Carlo variational objective approach of [11] to the setting where the generative model is itself a proposal distribution in a different probabilistic model. The observation that random variables can permit optimization of probabilistic computations that utilize black-box randomized code has been previously observed and used in reinforcement learning [23, 12].

## 8 Discussion

This paper proposed the concept of a proposal program, discussed how to implement proposal programs in a sampling-based probabilitsic runtime, showed how to use proposal programs within importance sampling and Metropolis-Hastings, how to optimize proposal programs offline, and suggested an application of the formalism to using randomized heuristics to accelerate Monte Carlo inference. Several directions for future work seem promising.

First, we note that the efficiency of Monte Carlo inference with proposal programs depends on the number of internal replicates $K$ used within the proposed SIMULATE and ASSESS operations. It is important to better characterize how the efficiency depends on $K$. Also, the proposed SIMULATE and ASSESS operations make use of forward execution of the proposal program to generate internal traces that are used for estimating the proposal probability. Following the 'probabilistic module' formalism [24], other distributions on internal traces that attempt to approximate $p(y|z;x)$ can be used instead for more accurate proposal probability estimates (and therefore more efficient Monte Carlo inference for a fixed proposal program). These distributions, which constitute nested inference samplers, should be able to make use of existing inference machinery in probabilistic programming languages that was originally developed for inference in model programs. It also seems promising to explore proposal programs that combine neural networks and domain-specific heuristic algorithms in different ways. For example, a proposal program may contain a switch statement that decides whether to propose using a heuristic (as in Section 5) or according to a pure-neural network proposal (as shown in Figure 6), where the probability of taking the two different paths can itself be predicted from the problem instance using a neural network. The proposal program formalism also suggests that other approaches for inference in probabilistic programs such as symbolic integration [25] could find applications in estimating or computing proposal probabilities. Finally, a rigorous theoretical analysis of importance sampling, Metropolis-Hastings, and offline optimization with proposal programs that contain continuous random variables would be useful.

#### Acknowledgements

This research was supported by DARPA (PPAML program, contract number FA8750-14-2-0004), IARPA (under research contract 2015-15061000003), the Office of Naval Research (under research contract N000141310333), the Army Research Office (under agreement number W911NF-13-1-0212), and gifts from Analog Devices and Google. This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a.

## References

[1] 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, 2006.

[2] Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. Learning stochastic inverses. In *Advances in neural information processing systems*, pages 3048–3056, 2013.

[3] Brooks Paige and Frank Wood. Inference networks for sequential monte carlo in graphical models. In *International Conference on Machine Learning*, pages 3040–3049, 2016.

[4] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances in Neural Information Processing Systems*, pages 622–630, 2016.

[5] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900*, 2016.

[6] Marco Cusumano-Towner, Vikash Mansinghka, et al. Gen.jl: A probabilistic meta-programming platform and compositional inference programming library. `https://github.com/probcomp/Gen.jl`, 2017.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[8] Luke Tierney. Markov chains for exploring posterior distributions. *the Annals of Statistics*, pages 1701–1728, 1994.

[9] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, pages 770–778, 2011.

[10] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977*, 2017.

[11] Andriy Mnih and Danilo Rezende. Variational inference for monte carlo objectives. In *International Conference on Machine Learning*, pages 2188–2196, 2016.

[12] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

[13] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[16] Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

[17] Christophe Andrieu and Gareth O. Roberts. The pseudo-marginal approach for efficient monte carlo computations. *Ann. Statist.*, 37(2):697–725, 04 2009.

[18] Paul Fearnhead, Omiros Papaspiliopoulos, and Gareth O Roberts. Particle filters for partially observed diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(4):755–777, 2008.

[19] Geir Storvik. On the flexibility of metropolis–hastings acceptance probabilities in auxiliary variable proposal generation. *Scandinavian Journal of Statistics*, 38(2):342–358, 2011.

[20] Tongzhou Wang, Yi Wu, David A Moore, and Stuart J Russell. Neural block sampling. *arXiv preprint arXiv:1708.06040*, 2017.

[21] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[22] Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030*, 2014.

[23] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[24] Marco F Cusumano-Towner and Vikash K Mansinghka. Encapsulating models and approximate inference programs in probabilistic modules. *arXiv preprint arXiv:1612.04759*, 2016.

[25] Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

# A Proofs

## A.1 Consistency of importance sampling using proposal program

Consider a proposal program $\mathcal{P}$ with probability distribution $p(y, z; x) = p(y; x)p(z; x, y)$ for interal trace $y$, output trace $z$, and input $x$, such that $p(z; x, y) > 0$ for all $x, y, z$. Consider a target distribution $\pi(z)$ such that $\pi(z) > 0 \implies p(z; x) > 0$. Consider the following extended target distribution:

$$\pi(z, y_{1:K}) = \pi(z) \prod_{k=1}^{K} p(y_k; x) \tag{10}$$

Consider the following extended proposal distribution:

$$q(z, y_{1:K}) = \left( \prod_{i=1}^{K} p(y_i; x) \right) \frac{1}{K} \sum_{k=1}^{K} p(z; x, y_k) \tag{11}$$

First we show that $\pi(z, y_{1:K}) > 0 \implies q(z, y_{1:K}) > 0$. Suppose $\pi(z, y_{1:K}) > 0$. Then $\pi(z) > 0$ and $p(y_k; x) > 0$ for each $k = 1 \dots K$. Since $p(z; x, y) > 0$ for all $x, y, z$, we have $q(z, y_{1:K}) > 0$. The importance weight for the extended target distribution and the extended proposal distribution is:

$$\frac{\pi(z, y_{1:K})}{q(z, y_{1:K})} = \frac{\pi(z) \prod_{k=1}^{K} p(y_k; x)}{\left( \prod_{i=1}^{K} p(y_i; x) \right) \frac{1}{K} \sum_{k=1}^{K} p(z; x, y_k)} = \frac{\pi(z)}{\frac{1}{K} \sum_{k=1}^{K} p(z; x, y_k)} \tag{12}$$

Note that invoking SIMULATE in Algorithm 1 is equivalent to sampling from the extended proposal distribution $q(z, y_{1:K})$ and that the importance weight in Algorithm 1 is equal to Equation (12). Therefore, Algorithm 1 is a standard self-normalized importance sampler for the extended target distribution $\pi(z, y_{1:K})$. By convergence of standard self-normalized importance sampling, $\hat{\mu}_N \overset{\text{a.s.}}{\to} \mathbb{E}_{(z, y_{1:K}) \sim \pi(\cdot)}[f(z)] = \mathbb{E}_{z \sim \pi(\cdot)}[f(z)] = \mu$.

## A.2 Stationary distribution for Metropolis-Hastings using proposal program

Algorithm 2 defines a transition operator that takes as input an output trace of proposal program $\mathcal{P}$, denoted $x = z_{t-1}$, and produces as output another output trace, denoted $z_t$. Note that other state besides $z_{t-1}$ (i.e. state that is not mutated by the transition operator) can be included in the input $x$ to the operator, but we write $x = z_{t-1}$ to simplify notation. Let $\delta$ denote the Kronecker delta. A rigorous treatment of continuous random choices is left for future work. Let $y_{1:K}$ denote the tuple $(y_1, \dots, y_K)$. To show that the this transition operator admits the target distribution $\pi(z)$ as a stationary distribution, we first define an extended target distribution on tuples $(z, \upsilon, \zeta, k, y_{1:K})$ where $\zeta$ and $z$ are both output traces of $\mathcal{P}$, where $k \in \{1 \dots K\}$, and where $\upsilon$ and each $y_j$ for $j = 1 \dots K$ are internal traces of $\mathcal{P}$:

$$\pi(z, \upsilon, \zeta, k, y_{1:K}) := \pi(z) p(\upsilon, \zeta; z) \frac{1}{K} \delta(y_k; \upsilon) \prod_{i \neq k}^{K} p(y_i; z) \tag{13}$$

We then define a proposal kernel on this extended space:

$$q(z', \upsilon', \zeta', k', y'_{1:K}; z, \upsilon, \zeta, k, y_{1:K}) := \delta(z'; \zeta) \delta(\zeta'; z) \left( \prod_{j=1}^{K} p(y'_j; \zeta) \right) \frac{p(z; \zeta, y'_{k'})}{\sum_{j=1}^{K} p(z; \zeta, y'_j)} \delta(\upsilon'; y'_{k'}) \tag{14}$$

Consider a Metropolis-Hastings (MH) move on the extended space, using $q$ as the proposal kernel and the extended target $\pi$ as the target. Assume that $z' = \zeta$, $\zeta' = z$, $\upsilon' = y'_{k'}$, and $y_k = \upsilon$. The MH acceptance ratio for this move

is:

$$\frac{\pi(z', \upsilon', \zeta', y'_{1:K}) q(z, \upsilon, \zeta, y_{1:K}; z', \upsilon', \zeta', y'_{1:K})}{\pi(z, \upsilon, \zeta, y_{1:K}) q(z', \upsilon', \zeta', y'_{1:K}; z, \upsilon, \zeta, y_{1:K})} \tag{15}$$

$$= \frac{\pi(z') p(\upsilon', \zeta'; z') \frac{1}{K} \left( \prod_{i \neq k'}^{K} p(y'_i; z') \right) \left( \prod_{j=1}^{K} p(y_j; \zeta') \right) \frac{p(z'; \zeta', y_k)}{\sum_{j=1}^{K} p(z'; \zeta', y_j)}}{\pi(z) p(\upsilon, \zeta; z) \frac{1}{K} \left( \prod_{i \neq k}^{K} p(y_i; z) \right) \left( \prod_{j=1}^{K} p(y'_j; \zeta) \right) \frac{p(z; \zeta, y'_{k'})}{\sum_{j=1}^{K} p(z; \zeta, y'_j)}} \tag{16}$$

$$= \frac{\pi(z') p(y'_{k'}, \zeta'; z') \frac{1}{K} \left( \prod_{i \neq k'}^{K} p(y'_i; z') \right) \left( \prod_{j=1}^{K} p(y_j; \zeta') \right) \frac{p(z'; \zeta', y_k)}{\sum_{j=1}^{K} p(z'; \zeta', y_j)}}{\pi(z) p(y_k, \zeta; z) \frac{1}{K} \left( \prod_{i \neq k}^{K} p(y_i; z) \right) \left( \prod_{j=1}^{K} p(y'_j; \zeta) \right) \frac{p(z; \zeta, y'_{k'})}{\sum_{j=1}^{K} p(z; \zeta, y'_j)}} \tag{17}$$

$$= \frac{\pi(z') p(\zeta'; z', y'_{k'}) \frac{1}{K} \left( \prod_{i=1}^{K} p(y'_i; z') \right) \left( \prod_{j=1}^{K} p(y_j; \zeta') \right) \frac{p(z'; \zeta', y_k)}{\sum_{j=1}^{K} p(z'; \zeta', y_j)}}{\pi(z) p(\zeta; z, y_k) \frac{1}{K} \left( \prod_{i=1}^{K} p(y_i; z) \right) \left( \prod_{j=1}^{K} p(y'_j; \zeta) \right) \frac{p(z; \zeta, y'_{k'})}{\sum_{j=1}^{K} p(z; \zeta, y'_j)}} \tag{18}$$

$$= \frac{\pi(z') p(\zeta'; z', y'_{k'}) \frac{1}{K} \frac{p(z'; \zeta', y_k)}{\sum_{j=1}^{K} p(z'; \zeta', y_j)}}{\pi(z) p(\zeta; z, y_k) \frac{1}{K} \frac{p(z; \zeta, y'_{k'})}{\sum_{j=1}^{K} p(z; \zeta, y'_j)}} \tag{19}$$

$$= \frac{\pi(z') \frac{1}{K} \frac{1}{\sum_{j=1}^{K} p(z'; \zeta', y_j)}}{\pi(z) \frac{1}{K} \frac{1}{\sum_{j=1}^{K} p(z; \zeta, y'_j)}} \tag{20}$$

$$= \frac{\pi(z') \frac{1}{K} \sum_{j=1}^{K} p(z; \zeta, y'_j)}{\pi(z) \frac{1}{K} \sum_{j=1}^{K} p(z'; \zeta', y_j)} \tag{21}$$

$$= \frac{\pi(z') \frac{1}{K} \sum_{j=1}^{K} p(z; \zeta, y'_j)}{\pi(z) \frac{1}{K} \sum_{j=1}^{K} p(\zeta; z, y_j)} \tag{22}$$

Consider a sampling process that begins with $z \sim \pi(\cdot)$ then executes Algorithm 2 on input $z_{t-1} = z$. Observe that the invocation of SIMULATE in Algorithm 2 is equivalent to sampling $(\upsilon, \zeta, k, y_{1:K})$ from the following distribution, conditioned on $z$:

$$\pi(\upsilon, \zeta, k, y_{1:K} | z) = p(\upsilon, \zeta; z) \frac{1}{K} \delta(y_k; \upsilon) \prod_{i \neq k}^{K} p(y_i; z) \tag{23}$$

Therefore, after SIMULATE, we have a sample from the extended target distribution:

$$(z, \upsilon, \zeta, k, y_{1:K}) \sim \pi(\cdot) \tag{24}$$

Next, note that the invocation of ASSESS in Algorithm 2, on input $z_{t-1} = z$ and $z' = \zeta$, is equivalent to sampling $(z', \upsilon', \zeta', k', y'_{1:K})$ from the extended proposal kernel on input $(z', \upsilon', \zeta', k', y'_{1:K})$ (note that $k'$ and $\upsilon'$ are not literally sampled in ASSESS but could be without changing the behavior of ASSESS). Finally, note that Equation (22) is the acceptance ratio used in the accept/reject step of Algorithm 2. Therefore, we can interpret Algorithm 2 as (1) extending the input sample $z$ onto the extended space by sampling from the conditional distribution $\pi(\upsilon, \zeta, k, y_{1:K} | z)$, and (2) performing an MH step on the extended space. Since $z \sim \pi(.)$, we have that $(z, \upsilon, \zeta, k, y_{1:K}) \sim \pi(.)$. Since the MH step on the extended space has the extended target distribution $\pi$ as a stationary distribution, we have $(z', \upsilon', \zeta', k', y'_{1:K}) \sim \pi(.)$. Then the marginal distribution on $z'$ is $\pi(z')$. Therefore $\pi(z)$ is a stationary distribution of Algorithm 2.