Automated Quantized Inference for Probabilistic Programs with AQUA

Zixin Huang[1*], Saikat Dutta[1] and Sasa Misailovic[1]

[1*]Department of Computer Science, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, 61801, IL, USA.

*Corresponding author(s). E-mail(s): zixinh2@illinois.edu;
Contributing authors: saikatd2@illinois.edu; misailo@illinois.edu;

Abstract

We present AQUA, a new probabilistic inference algorithm that operates on probabilistic programs with continuous posterior distributions. AQUA approximates programs via an efficient quantization of the continuous distributions. It represents the distributions of random variables using quantized value intervals (Interval Cube) and corresponding probability densities (Density Cube). AQUA's analysis transforms Interval and Density Cubes to compute the posterior distribution with bounded error. We also present an adaptive algorithm for selecting the size and the granularity of the Interval and Density Cubes. We evaluate AQUA on 24 programs from the literature. AQUA solved all of 24 benchmarks in less than 43s (median 1.35s) with a high-level of accuracy. We show that AQUA is more accurate than state-of-the-art approximate algorithms (Stan's NUTS and ADVI) and supports programs that are out of reach of exact inference tools, such as PSI and SPPL.

1 Introduction

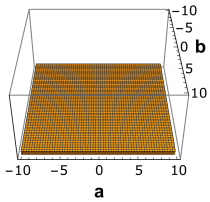
Many modern applications (e.g., in machine learning, robotics, autonomous driving, medical diagnostics, and financial forecasting) need to make decisions under uncertainty. Probabilistic programming languages (PPLs) offer an intuitive way to model uncertainty by representing complex probabilistic models as simple programs [1]. They expose randomness and Bayesian inference as first-class abstractions by extending standard languages with statements for sampling from probability distributions and probabilistic conditioning. The underlying programming system then automates the intricate details of the probabilistic inference.

Probabilistic inference is a computationally hard problem. Most current approaches that emerged from the statistics and machine learning communities applied aggressive numeric approximations, such as Markov Chain Monte Carlo sampling (MCMC) or Variational Inference (VI). However, these approaches often cannot obtain the level of accuracy that is required in applications such as algorithmic fairness [2], security/privacy [3, 4], sensitivity analysis [5, 6], or software testing [7].

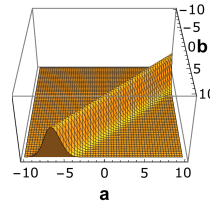
Symbolic techniques for inference have been resurging as a more accurate alternative. They use a symbolic representation of the model's state (e.g., elementary functions, piecewise-linear functions, or hypercubes), and compute the posterior distribution algebraically [7–9] or closely approximate programs using volume computation [2, 3, 10]. However, these approaches are often limited by the classes of programs they can solve. For instance, continuous programs pose a major challenge for these

```
1 D=40
2 Y=[3.4,0.3,...]
3 a~Uniform(-10,10)
4 b~Uniform(-10,10)
5 for (i in 1:D)
6  Y[i]~Normal(a+b,1)
7 return a,b
```

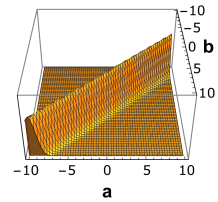
(a) Example program



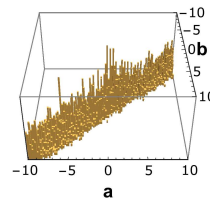
(b) AQUA prior



(c) AQUA observe



(d) AQUA result



(e) NUTS result

Fig. 1: Example program and AQUA estimated probabilistic density v.s. NUTS histogram

approaches due to integrals in posterior calculation. State-of-the-art symbolic solvers cannot solve many integrals exactly (often, the integrals do not have a closed form). Similarly, volume computation approaches have a limited support for continuous distributions (e.g., do not allow for conditioning on continuous random variables) and/or compute the probability of a single event, not the entire posterior distribution. *An intriguing research question is how to approximate multi-dimensional continuous distributions in a principled manner that allows for more expressive programs and can solve programs that are out of reach of existing tools for exact inference.*

AQUA. We present AQUA, a novel system for symbolic inference that uses quantization of probability density function for delivering scalable and precise solutions for a broad range of probabilistic programs. AQUA's inference algorithm approximates the original continuous program via an efficient quantization of the continuous distributions by using multi-dimensional tensor representations that we call *Interval Cube and Density Cube*. The Interval Cube stores the quantized value ranges of variables in the probabilistic program. The Density Cube approximates the joint posterior distribution by recording the probability of each hypercube contained in the interval cube.

AQUA's analysis transforms the symbolic state to compute quantized approximate posterior distribution.We derive the bounds for the approximation error (due to the quantization and integration) and show that our inference converges in distribution to the true posterior. We also present an adaptive algorithm for automatically selecting the granularity of the Interval and Density Cubes.

Example. Figure 1a presents a probabilistic program that represents the distribution of two random variables. In the program, we have two random variables `a` and `b`, each having `Uniform` *prior* distribution (Lines 3-4). We then condition the model on 40 data points `Y`, assuming that each point is normally distributed with the mean `a+b` (Lines 5-6). We finally query for the joint *posterior* distribution (i.e., the distribution of latent variables `a` and `b` after observing the data on Line 6).

Figure 1 presents AQUA's results: 1b shows the prior of the two variables, 1c shows the likelihood (observation) on a single data point, and 1d shows the posterior distribution. On each plot, the X-axis and Y-axis represent `a` and `b` values, and the Z-axis values are the probability densities computed by AQUA. AQUA computes the result in 0.76s, whereas an MCMC based inference algorithm (NUTS) produces a less accurate posterior within the same amount of time (Figure 1e).

Evaluation. We evaluate our implementation of AQUA on a set of 24 probabilistic programs from the literature. We compare AQUA with exact inference – PSI [7] and SPPL [9] – and approximate inference – MCMC and VI implemetations in Stan [1]. We show that AQUA can solve programs that are out of reach for PSI and SPPL. Our results show AQUA solved all benchmarks in less than 43s (median 1.35s). It is significantly more accurate than VI for all programs (for the Kolmogorov-Smirnov metric). AQUA is substantially more accurate than MCMC for 10 programs, even when MCMC is given substantially more time to complete. We also present a case study that shows AQUA can precisely capture the tails of the distribution of robust models.

Contributions. This paper makes the following contributions:

- **Inference Algorithm:** We present AQUA, a novel inference algorithm that works on general, real-world probabilistic programs with continuous distributions based on quantization and symbolic computation.
- **Quantization with Interval and Density Cubes:** Our analysis defines symbolic transformers on the abstract state consisting of the Interval and Density Cubes. We also present theoretical bounds on the quality of approximation.

- **Inference Algorithm Optimizations:** We present algorithm extensions that automatically refine the size/granularity of the analysis to satisfy a given precision threshold and aggressively reduce the analysis overhead of local variables.
- **Evaluation:** Our experiments show that AQUA is more accurate than approximate inference algorithms (Stan's MCMC/VI) and supports programs with conditioning on continuous distributions that are out of reach of exact inference tools (PSI and SPPL).

2 Preliminaries

Language Syntax and Semantics. Figure 2 describes the syntax of a probabilistic program using an imperative, first-order intermediate representation, drawing from Storm-IR [11, 12]. It has statements for sampling from distributions[1] and conditioning on data with `factor` and `observe`.

The language semantics are standard, inspired by those presented by Gorinova et al. [13] (We present the detailed semantics rules in the Appendix, available in the full version of the paper). In summary, a probabilistic program evaluates the *posterior probability density function*. Our operational semantics for a program defines its effect on the program state, σ, which maps variables to values. A value V can either be a constant c or an array of values $[c_1,c_2,...]$. The notations $\sigma(x)$ and $\sigma(x \mapsto V)$ denote accessing and updating a variable x respectively. We refer to the return variables of the program as the *global variables*, and the others as *local variables*. We allow local variables to have discrete distributions (e.g. Bernoulli), as long as the density of the global variables are Lipschitz continuous. We define a special variable $\mathcal{L} \in \mathbb{R}^+$ which tracks the *unnormalized posterior density* of the probabilistic program. We initialize $\sigma(\mathcal{L})$ to 1.0 at the start of the program.

Probability Density. We review several basic terms from the probability theory. Let $\boldsymbol{x}$ be the set of variables with values in V, and $\mathcal{D}$ be the set of observed data points. Then, the posterior probability density function is $p(\boldsymbol{x} \mid \mathcal{D}) : V \to \mathbb{R}$, such that $\int_{\boldsymbol{x} \in V} p(\boldsymbol{x} \mid \mathcal{D}) d\boldsymbol{x} = 1$. The distribution $p(\boldsymbol{x} \mid \mathcal{D})$ can be calculated from the *unnormalized probability density function* $f(\boldsymbol{x}, \mathcal{D}) : V \to \mathbb{R}$, by $p(\boldsymbol{x} \mid \mathcal{D}) = \frac{1}{z} f(\boldsymbol{x}, \mathcal{D})$, where z is the normalizing constant: $z = \int f(\boldsymbol{x}, \mathcal{D}) d\boldsymbol{x}$. If $\boldsymbol{x}_{-i}$ contains all the variables in $\boldsymbol{x}$ excluding x_i, we define the *marginal probability density function* of x_i as $p(x_i \mid \mathcal{D}) = \int p(\boldsymbol{x} \mid \mathcal{D}) \, d\boldsymbol{x}_{-i}$. Hereon, we omit data symbol $\mathcal{D}$ to write $p(\boldsymbol{x})$ and $f(\boldsymbol{x})$ when clear from the context. In the semantics, $f(\boldsymbol{x})$ is represented by $\sigma(\mathcal{L})$.

[1] We support common continuous distributions including Normal, Uniform, Exponential, Beta, Gamma, Student-T, Laplace, Triangular, and any mixture of the above distributions.

3 Inference with Density Cubes

3.1 Notation and Basic Definitions

We represent the closed, bounded set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ with its lower-bound $a \in \mathbb{R}$ and upper-bound $b \in \mathbb{R}$. We denote this abstraction as an **interval** $I = [a,b] \in \mathbb{R}^2$. We refer to the lower and upper bound of I as $\underline{I}$ and $\overline{I}$, respectively ($\underline{I}, \overline{I} \in \mathbb{R}$).

A probabilistic program lifts a normal program operating on single values to a *distribution* over values. Hence, a probabilistic program represents a joint distribution over its variables. For our symbolic analysis, to represent the quantized values of variables, we define tensors of intervals which we will refer to as *Interval Cube*. We also assign a probability density to each interval in the Interval Cube. We will refer to this assignment of densities as *Density Cube*. If there are N variables in the program, the Density Cube will be an N-dimensional tensor.

Definition 1 (Interval Cube) We represent the value of a variable x with Interval Cube, $\boldsymbol{I}^x_{M_1,M_2,...,M_N}$ where $[M_1, M_2, \ldots, M_N]$ represents the shape of the Interval Cube and each $M_i \in \mathbb{N}$ is the *number of intervals* (*splits*) along the i-th dimension. Each element of $\boldsymbol{I}^x_{M_1,M_2,...,M_N}$ is a single interval. We let $\mathbb{I}$ be the set of all Interval Cubes. For a constant c, we denote its Interval Cube as $[c]$, meaning a singleton interval with both lower and upper bounds being c.

Example 1 Suppose a program has $x \sim$ Beta(2,2), meaning that x following a Beta distribution. Beta distribution has bounded support $0 \leq x \leq 1$, and thus we consider splitting the possible values into, say, 10 equal-length intervals: $\boldsymbol{I}^x_{10} = [[0,0.1],[0.1,0.2],...,[0.9,1]]$.

If the support is unbounded, e.g. $x \sim$ Normal(0,1), where the Normal distribution has infinite support, we will truncate the support into a bounded interval, and ensure the probability that x being out of this interval is small. For example, we may consider $-12 \leq x \leq 12$ with $\Pr(x < -12 \text{ or } x > 12) = 3.6 \cdot 10^{-33}$, and then split the interval into 10 equal-length intervals: $\boldsymbol{I}^x_{10} =$

```
x    ∈   Vars
c    ∈   Consts
op   ∈   {+,−,∗,>,...}
d    ∈   {Normal,Uniform,...}

E    :=  c | x | E[E*] | E op E |d(E*).pdf(E*) |f(E*)
S    :=  x = E |x ∼ d(E*) |factor(E) |observe(d(E*),x)
         | if (E) S* else S* | for x ∈ 1..N; {S*}
P    :=  S+; return x+
```

Fig. 2: Syntax of AQUA's language

Table 1: Correspondence of Symbolic and Concrete Analysis

Concrete	Symbolic
Value $\sigma(x):\mathbb{R}$	Interval Cube $\sigma^{\#}(x):\mathbb{I}$
Density $\sigma(\mathcal{L}):\mathbb{R}^+$	Density Cube $\sigma^{\#}(P):(\mathbb{R}^+)^{\prod_{i=1}^{N} M_i}$
State$:\mathcal{P}(\mathbf{Vars}\mapsto\mathbb{R})\mapsto\mathbb{R}^+$	**Astate**$:\mathcal{P}(\mathbf{Vars}\mapsto\mathbb{I})\mapsto(\mathbb{R}^+)^{\prod_{i=1}^{N} M_i}$
$[\![E]\!]:$**State**$\mapsto\mathbb{R}$	$[\![E]\!]^{\#}:$**Astate**$\mapsto\mathbb{I}$
$[\![S]\!]:$**State**$\mapsto$**State**	$[\![S]\!]^{\#}:$**Astate**$\mapsto$**Astate**

[[−12,−9.6],[−9.6,−7.2],...,[9.6,12]]. In practice, AQUA will adaptively select the bounded interval (Algorithm 2).

Definition 2 (Shape of the Interval Cube) To simplify the notation, we hereon denote the shape of the hypercube as $\boldsymbol{M}=[M_1,M_2,...,M_N]$ and each index in the hypercube is $\boldsymbol{m}\in\mathbb{M}$, $\mathbb{M}=\{[m_1,...,m_N]\mid m_i\in[1,...,M_i], i\in\{1,...,N\}\}$. We write $\boldsymbol{K}=\boldsymbol{M}_1\odot\boldsymbol{M}_2$ as the element-wise product (Hadamard product) operation on two shape vectors, namely $K_i=M_{1i}\times M_{2i}$, $i\in\{1,...,N\}$. We will use the operation in the computation of multi-dimensional Interval Cubes (see Example 3). We use $\boldsymbol{m}_1$ to denote the index of a Interval Cube with shape $\boldsymbol{M}_1$, $\boldsymbol{m}_1=[m_1,...,m_N]$, $m_i\in\{1,...,M_{1i}\}$, and similarly we use $\boldsymbol{m}_2$ for index in $\boldsymbol{M}_2$, and $\boldsymbol{k}$ for index in $\boldsymbol{K}$.

Definition 3 (Density Cube) For a given probabilistic program *Prog* with N sampled variables, $\boldsymbol{x}=\{x_1,...,x_N\}$, we define the Density Cube with shape $\boldsymbol{M}=[M_1,...,M_N]$ as $\boldsymbol{P}_{\boldsymbol{M}}^{Prog}$, where

$$\boldsymbol{P}_{\boldsymbol{M}}^{Prog}(\boldsymbol{m})=p_{\boldsymbol{m}}, \text{ for each index } \boldsymbol{m}\in\mathbb{M},$$

and $p_{\boldsymbol{m}}$ denotes the value of the unnormalized probability density function at the lower bound of the corresponding interval in the Interval Cube. The densities at the lower bound of intervals will help us do numerical integration for posterior calculation. Here we use the density at the lower bound for convenience. Using the upper bound or the mid-point will give the same accuracy guarantee (Theorem 3). Further, $\boldsymbol{P}_{\boldsymbol{M}}^{Prog}\in(\mathbb{R}^+)^{\prod_{i=1}^{N} M_i}$, and $p_{\boldsymbol{m}}\in\mathbb{R}^+$.

Example 2 (Density Cube for a Single Variable) In Example 1 where $\boldsymbol{I}_{10}^{x}=[[0,0.1],[0.1,0.2],...,[0.9,1]]$, the corresponding Density Cube is $\boldsymbol{P}_{10}^{x}=[0,0.54,...,0.54]$, which are the densities of Beta(2,2) at each interval's lower bound. When there are sufficiently many splits, the discretized $\boldsymbol{P}_{10}^{x}$ will converge to the true density.

Example 3 (Density Cube for Multiple Variables) Suppose we have a program defining two variables: $x_1\sim$ Beta(2,2), $x_2\sim$ Beta(2,2). If we apply 10 equal-length splits for x_1 and 5 equal-length splits for x_2, two Interval Cubes will be defined: $\boldsymbol{I}_{10}^{x_1}=[[0,0.1],[0.1,0.2],...,[0.9,1.0]]$ and $\boldsymbol{I}_{5}^{x_2}=[[0,0.2],[0.2,0.4],...,[0.8,1.0]]$. The corresponding Density Cube $\boldsymbol{P}_{10,5}^{P}$ will have the shape 10 by 5, so that each element $\boldsymbol{P}_{10,5}^{P}([m_1,\ m_2])$, $m_1\in\{1,...,10\}$, $m_2\in\{1,...,5\}$, stores the approximate joint density when $x_1\in\boldsymbol{I}_{10}^{x_1}(m_1)$ and $x_2\in\boldsymbol{I}_{5}^{x_2}(m_2)$. For example, $\boldsymbol{P}_{10,5}^{P}([10,5])=0.5184$ corresponds to $x_1\in\boldsymbol{I}_{10}^{x_1}(10)=[0.9,1.0]$ and $x_2\in\boldsymbol{I}_{5}^{x_2}(5)=[0.8,1.0]$. The value 0.5184 is the exact joint density for $x_1=0.9$ and $x_2=0.8$, and is the approximate joint density for other x_1 and x_2 in their intervals.

For easy implementation and explanation, in AQUA we represent x_1's Interval Cube as $\boldsymbol{I}_{10,1}^{x_1}$ and x_2's Interval Cube as $\boldsymbol{I}_{1,5}^{x_2}$, by reshaping their intervals along different dimensions. The shape of $\boldsymbol{P}_{10,5}^{P}$ will simply be element-wise product (Hadamard product) of the individual shape vectors, $[10,1]\odot[1,5]$. In fact, we represent the intervals of all the *Sampled Variables* on different dimensions. Sampled Variables are variables initialized by sampling statements (e.g. $x_1\sim$Beta(2,2)), not deterministic assignments (e.g. $x_3=x_1+x_2$).

Definition 4 (Symbolic Domain) Our *symbolic state* has two components, a map from variables to Interval Cubes, and a Density Cube representing the joint density approximation. Let **Var** denote the set of variables, and $\mathcal{P}$ be the power set, the domain of the symbolic state is $\Sigma=\mathcal{P}(\mathbf{Var}\mapsto\mathbb{I})\mapsto(\mathbb{R}^+)^{\prod_{i=1}^{N} M_i}$ a symbolic state $\sigma^{\#}\in\Sigma$ will have the form

$$\sigma^{\#}=\left\langle\{x_1\mapsto\boldsymbol{I}_{\boldsymbol{M}_1}^{x_1},x_2\mapsto\boldsymbol{I}_{\boldsymbol{M}_2}^{x_2},...\},P\mapsto\boldsymbol{P}_{\boldsymbol{M}}^{Prog}\right\rangle.$$

The symbolic domain expresses a piecewise constant interpolation of the joint probability density at a program point. Hereon, we refer to the set of all the variables in the state $\sigma^{\#}$ as $\boldsymbol{x}=\{x_1,x_2,...,x_N\}$. The shape vectors $\boldsymbol{M}_1$, $\boldsymbol{M}_2$,... will remain the same throughout the program.

3.2 Analysis

We approximate the posterior density function of variables in our symbolic states. Table 1 presents the correspondence of the objects in concrete semantics

to symbolic states. While a concrete state has a single valuation of variables and evaluates to a single density value, our symbolic state stores all possible variable values in Interval Cube and the corresponding joint probability density in Density Cube. As the concrete semantics for a expression maps state to values, the symbolic semantics map symbolic state to Interval Cube; and as the concrete semantics for a statement map state to state, our symbolic semantics map symbolic state to symbolic state.

Analysis of Expressions. The symbolic transformer $[\![E]\!]^\#$ on an expression E takes a symbolic state $\sigma^\# : \textbf{Astate}$ as input, and outputs an Interval Cube. Figure 3 presents the rules. We explain two important cases in detail:

- $[\![E_1 \ op \ E_2]\!]^\#$: For the arithmetic/boolean operation on two Interval Cubes, which may not always have the same shape, the resulting Interval Cube needs to contain all possible value combinations. Specifically, for $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ with shape $\boldsymbol{M}_1 = [M_{11},...,M_{1N}]$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$ with shape $\boldsymbol{M}_2 = [M_{21},...,M_{2N}]$, the result $\boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}}$ has shape $\boldsymbol{K} = [K_1,...,K_N]$ with $K_i = M_{1i} \times M_{2i}$ to capture all the combinations of elements from $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$. If $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are not of the same length, we reshape both $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$ to have the same dimension, by letting some M_{1i} or M_{2i} to have value 1. We let the arithmetic or boolean operation on the interval pairs be $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2) \coloneqq [\underline{\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1)} \ op \ \underline{\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2)}, \overline{\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1)} \ op \ \overline{\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2)}]$. We handle the case with multiple intervals analogously. This operation on multiple Interval Cubes can be implemented efficiently with the *broadcast* function in tensor libraries.
- $[\![d(E_1,...,E_{n-1}).\text{pdf}(E_n)]\!]$: Similar to arithmetic operator, we apply the mathematical density d_pdf(_) of the distribution d whose parameters (e.g., mean, location, shape or variance) are intervals obtained by evaluating $E_1,...,E_{n-1}$, and it takes the intervals of E_n for which we seek the density. We denote the shape of the result Interval Cube as $\boldsymbol{K}$, which is computed from the element-wise product of the shapes of the input Interval Cubes.

Analysis of Statements. Figure 4 presents the transformers $[\![S]\!]^\#$ on statements S, which takes an abstract state $\sigma^\# : \textbf{Astate}$ as input, and outputs an abstract state. We explain two important rules where we modify Density Cube (the remaining statements are standard or rely on these two rules):

- $[\![x \sim d(E_1,...,E_n)]\!]^\#$, $[\![\text{factor}(E)]\!]^\#$: We first evaluate d.pdf(_) of the expressions into an Interval Cube, and multiply the current Density Cube with the lower bound of intervals from the Interval Cube. Then at the lower bound of each interval, the density is the same as the one from concrete semantics (Lemma 1). Intuitively, we discretize the density function and use the density at the lower bound to represent each interval. For convenience, our discretization uses the density at the lower bound. Using the density at the upper bound or the midpoint is also possible, and our accuracy guarantee (Theorem 3) still holds.
- $[\![\text{if } (E) \text{ then } \{S_1\} \text{ else } \{S_2\}]\!]^\#$: We first solve the results from two branches one conditioning on E and the other on $1-E$. The true boolean expressions evaluate to 1 and false to 0 in our analysis, and we get the interval cubes for E and $1-E$ from expression rules (Figure 3). We then *Join* the result states by adding up the Density Cubes from both branches.

Definition 5 (Joins) *Join* ($\sqcup$) adds the Density Cubes from two states. Formally, $\sigma^\#_1 \sqcup \sigma^\#_2 = \sigma^\#_1(P \mapsto \boldsymbol{P}^{Prog}_{\boldsymbol{M}})$, where each element in $\boldsymbol{P}^{Prog}_{\boldsymbol{M}}$ at location $\boldsymbol{m}$ is $\sigma^\#_1(P)(\boldsymbol{m}) + \sigma^\#_1(P)(\boldsymbol{m})$, with $\boldsymbol{m} = [m_1,m_2,...,m_N]$, $m_i \in \{1,...,M_i\}$, $\boldsymbol{M} = [M_1,M_2,...,M_N]$. Since we already initialized the global variables with their Interval Cube, $\sigma^\#_1$ and $\sigma^\#_2$ should have the same variables and Interval Cubes. Then the joint probability density P is changed to the sum of the densities from both states. Similarly, we can define *Meet* ($\sqcap$) by product of $\sigma^\#_1(P)(\boldsymbol{m})$ and $\sigma^\#_1(P)(\boldsymbol{m})$.

Algorithm. Algorithm 1 takes as input a probabilistic program *Prog*, the shape vector $\boldsymbol{M}$ where each element M_i is the number of intervals for variable x_i, and the interval bounds $\boldsymbol{C}$ (optional). In Section 4, we describe an adaptive scheme to automatically search for a proper $\boldsymbol{C}$ for the analysis.

First, it initializes the joint probability density P with the single interval [1.0] (Line 2). Then, it splits the value domain for each x_i in *SampledVars*, which are variables sampled from a prior distribution $x_i \sim d(E_1,...,E_n)$ and not from deterministic assignments, into M_i equi-length intervals in C_i (in the function *GetInitIntervals*, Line 3-5). M_i is the i-th element in $\boldsymbol{M}$, and C_i is the i-th element in $\boldsymbol{C}$.

The algorithm follows the analysis rules to get the state at the end of the program (Line 6). Then it computes the joint probability density estimation $\hat{f}$, as a piecewise function of $\sigma^\#(P)$ (Line 7).

$$
\begin{aligned}
[\![E]\!]^{\#} &\mapsto (\textbf{Astate} \mapsto \text{Interval Cube}) \\
[\![x]\!]^{\#} &:= \lambda\sigma^{\#}.\sigma^{\#}(x) \\
[\![c]\!]^{\#} &:= \lambda\sigma^{\#}.[c] \\
[\![E_1[E_2]]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let } [c,c] = [\![E_2]\!]^{\#}\sigma^{\#} \text{ in } [\![E_1[c]]\!]^{\#}\sigma^{\#} \\
[\![E_1 \ op \ E_2]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let } \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1} = [\![E_1]\!]^{\#}\sigma^{\#},\ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2} = [\![E_2]\!]^{\#}\sigma^{\#},\ \boldsymbol{K} = \boldsymbol{M}_1 \odot \boldsymbol{M}_2 \\
&\qquad \text{in } \boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}}, \text{ where } \boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}}(\boldsymbol{k}) = \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_1}(\boldsymbol{m}_2) \\
[\![d(E_1,\ldots,E_{n-1}).\text{pdf}(E_n)]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let } \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1} = [\![E_1]\!]^{\#}\sigma^{\#},\ldots,\boldsymbol{I}^{E_n}_{\boldsymbol{M}_n} = [\![E_n]\!]^{\#}\sigma^{\#},\ \boldsymbol{K} = \bigodot_{i=1}^{n} \boldsymbol{M}_i, \\
&\qquad \text{in } \boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}}, \text{ where } \boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}}(\boldsymbol{k}) = d\_\text{pdf}(\boldsymbol{I}^{E_n}_{\boldsymbol{M}_n}(\boldsymbol{m}_n), \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1),\ldots,\boldsymbol{I}^{E_{n-1}}_{\boldsymbol{M}_{n-1}}(\boldsymbol{m}_{n-1}))
\end{aligned}
$$

Fig. 3: Analysis of Expressions

$$
\begin{aligned}
[\![S]\!]^{\#} &\mapsto (\textbf{Astate} \mapsto \textbf{Astate}) \\
[\![\text{skip}]\!]^{\#} &:= \lambda\sigma^{\#}.\sigma^{\#} \\
[\![S_1;S_2]\!]^{\#} &:= \lambda\sigma^{\#}.[\![S_2]\!]^{\#}([\![S_1]\!]^{\#}\sigma^{\#}) \\
[\![x = E]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let } \boldsymbol{I} = [\![E]\!]^{\#}\sigma^{\#} \text{ in } \sigma^{\#}(x \mapsto \boldsymbol{I}) \\
[\![x \sim d(E_1,\ldots,E_n)]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let } \boldsymbol{P}_{\boldsymbol{M}_0} = \sigma^{\#}(P),\ \boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}} = [\![d(E_1,\ldots,E_n).\text{pdf}(x)]\!]^{\#}\sigma^{\#}, \text{ in} \\
&\qquad \text{let } \boldsymbol{M} = \boldsymbol{M}_0 \odot \boldsymbol{K}, \text{ in } \sigma^{\#}(P \mapsto \boldsymbol{P}'_{\boldsymbol{M}}), \\
&\qquad \text{where } \boldsymbol{P}'_{\boldsymbol{M}}(\boldsymbol{m}) = \boldsymbol{P}_{\boldsymbol{M}_0}(\boldsymbol{m}_0) \cdot \underline{\boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}}(\boldsymbol{k})}, \text{ for all } \boldsymbol{m} = \boldsymbol{m}_0 \odot \boldsymbol{k}, \\
&\qquad\ \boldsymbol{m}_0 \in \{[m_{01},\ldots,m_{0N}] \mid m_{0i} \in \{1,\ldots,M_{0N}\}\}, [M_{01},\ldots,M_{0N}] = \boldsymbol{M}_0, \\
&\qquad\ \boldsymbol{k} \in \{[k_1,\ldots,k_N] \mid\ k_i \in \{1,\ldots,K_N\}\},\ [K_1,\ldots,K_N] = \boldsymbol{K} \\
[\![\text{factor}(E)]\!]^{\#} &:= \lambda\sigma^{\#}.\text{let} \boldsymbol{P}_{\boldsymbol{M}_0} = \sigma^{\#}(P),\ \boldsymbol{I}_{\boldsymbol{K}} = [\![E]\!]^{\#}\sigma^{\#},\ \boldsymbol{M} = \boldsymbol{M}_0 \odot \boldsymbol{K} \\
&\qquad \text{in } \sigma^{\#}(P \mapsto \boldsymbol{P}'_{\boldsymbol{M}}), \text{ where } \boldsymbol{P}'_{\boldsymbol{M}}(\boldsymbol{m}) = \boldsymbol{P}_{\boldsymbol{M}_0}(\boldsymbol{m}_0) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}(\boldsymbol{k})} \\
&\qquad \text{where } \boldsymbol{P}'_{\boldsymbol{M}}, \boldsymbol{P}_{\boldsymbol{M}_0} \text{and } \boldsymbol{P}_{\boldsymbol{K}} \text{ are as above} \\
[\![\text{observe}(d(E_1,\ldots,E_{n-1}),E_n)]\!]^{\#} &:= \lambda\sigma^{\#}.[\![\text{factor}(d(E_1,\ldots,E_{n-1}).\text{pdf}(E_n))]\!]^{\#}\sigma^{\#} \\
[\![\text{if } (E) \text{ then } \{S_1\} \text{ else } \{S_2\}]\!]^{\#} &:= \lambda\sigma^{\#}.\left([\![\text{factor}(E);S_1]\!]^{\#}\sigma^{\#}\right) \sqcup \left([\![\text{factor}(1\text{-}E);S_2]\!]^{\#}\sigma^{\#}\right) \\
[\![\text{for } (i \text{ in } E_1..E_2)\ S]\!]^{\#} &:= \lambda\sigma^{\#}.[\![i = E_1; \text{if } (i \le E_2) \text{then} \{S; \text{for } (i \text{ in } E_1+1..E_2)S\} \text{else} \{\text{skip}\}]\!]^{\#}\sigma^{\#}
\end{aligned}
$$

Fig. 4: Analysis of Statements

Definition 6 (Concretization of Symbolic States) Define γ as the concretization function, s.t. $\gamma(\sigma^{\#}) = \hat{f}$, where $\hat{f}(\boldsymbol{x}) = \sigma^{\#}(P)(\boldsymbol{m})$ if $\boldsymbol{x} \in \bigotimes_{i=1}^{N}[\underline{\boldsymbol{I}^{x_i}_{M_i}(\boldsymbol{m})}, \overline{\boldsymbol{I}^{x_i}_{M_i}(\boldsymbol{m})}) \subset \mathbb{R}^N$ for any $\boldsymbol{m}$, and 0 otherwise. Intuitively, $\hat{f}$ is a piecewise constant interpolation of $\sigma^{\#}$.

The result $\hat{f}(\boldsymbol{x})$ is an approximation of the true unnormalized probability density function $f(\boldsymbol{x})$. In the concrete domain, the posterior probabilistic density function is calculated as $p(\boldsymbol{x}) = \frac{1}{z}f(\boldsymbol{x})$, but the integration $z = \int f(\boldsymbol{x})d\boldsymbol{x}$ is often intractable. We compute our approximation $\hat{z}$ using integration on the piecewise function:

Definition 7 (Integration for Normalizing Constant) Suppose there are N sampled variables $\boldsymbol{x}$ in the program, and let $\boldsymbol{C} = \bigotimes_{i=1}^{N}[a_i,b_i] \subset \mathbb{R}^N$ for each $x_i \in [a_i,b_i] \subset \mathbb{R}$ be the bounded domain used in the analysis ($\bigotimes$ represents the Cartesian Product on intervals on $\mathbb{R}$). We initialize $\sigma^{\#}[\boldsymbol{x}] = \boldsymbol{C}$ in the analysis. Then $z = \int_{\boldsymbol{C}} f(\boldsymbol{x})\ d\boldsymbol{x}$ is approximated with $\hat{z} = \int_{\sigma^{\#}[\boldsymbol{x}]} \hat{f}(\boldsymbol{x})\ d\boldsymbol{x}$ $= \sum_{\boldsymbol{m} \in \mathbb{M}} (\prod_{i=1}^{N} (\overline{\boldsymbol{I}^{x_i}_{M_i}(\boldsymbol{m})} - \underline{\boldsymbol{I}^{x_i}_{M_i}(\boldsymbol{m})}) \cdot \boldsymbol{P}^{P}_{\boldsymbol{M}}(\boldsymbol{m}))$.

The algorithm finally computes the posterior and the marginals for every variable (Lines 8-11). When the program has N variables, and each variable has the same number of intervals M, Algorithm 1 has the time complexity $\mathcal{O}(N \cdot M^N)$ and space complexity $\mathcal{O}(M^N)$.

Example 4 (Analysis Example) We use the following example to show how analysis works. It is a simplified version of the example in Section 1.

Algorithm 1 Posterior Interval Analysis Algorithm

procedure PosteriorAnalysis(*Prog*, $\boldsymbol{M}$, $\boldsymbol{C}$)
  $\sigma^{\#}_{init} \leftarrow \{P \mapsto [1]\}$ ▷ Initialize
  for $x_i \in SampledVars(Prog)$ **do**
    $\sigma^{\#}_{init}[x_i] \leftarrow GetInitIntervals(x_i, M_i, C_i)$
  end for
  $\sigma^{\#} \leftarrow [\![Prog]\!]\sigma^{\#}_{init}$ ▷ Apply analysis rules
  $\hat{f}(\boldsymbol{x}) \leftarrow PiecewiseFunc(\sigma^{\#}(P))$
  $\hat{z} \leftarrow \int_{\sigma^{\#}[\boldsymbol{x}]} \hat{f}(\boldsymbol{x})\, d\boldsymbol{x}$; $\hat{p}(\boldsymbol{x}) \leftarrow \frac{1}{\hat{z}} \hat{f}(\boldsymbol{x})$ ▷ Posterior
  for $x_i \in SampledVars(Prog)$ **do**
    $Marginal[x_i] \leftarrow \frac{1}{z} \int_{\sigma^{\#}[\boldsymbol{x}_{-i}]} \hat{p}(\boldsymbol{x})\, d\boldsymbol{x}_{-i}$ ▷ Marginalize
  end for
return ($\hat{p}$, *Marginal*)
end procedure

```
1 a ~ Uniform(0,4)
2 b ~ Uniform(0,4)
3 c = a + b
4 observe(Normal(c, 1), 5)
5 return a, b
```

Fig. 5: Analysis Example

Before the analysis starts, we initialize the Density Cube as $\sigma^{\#}_{init}(P) = [1.0]$, which is a scalar. We also initialize the Interval Cubes of a and b as:

$\sigma^{\#}_{init}(a) =$

[0, 1]	[1, 2]	[2, 3]	[3, 4]

$\sigma^{\#}_{init}(b) =$

[0, 1]
[1, 2]
[2, 3]
[3, 4]

In this example we use 4 equal-length splits per sampled variable. AQUA infers the variable support [0,4] from the Uniform priors. The user can specify the different number of intervals or interval bounds. Also, note that the Interval Cubes for $\sigma^{\#}(a)$ and $\sigma^{\#}(b)$ are on different dimensions. This will later us calculate the joint density or dependent expression.

Then we go over the program to apply the analysis rules:
Line 1. `a ~ Uniform(0,4)` defines the prior distributions for variables a. It times the initial Density Cube $\sigma^{\#}_{init}(P)$ with the density of a being at the lower bounds of the 4 intervals [[0, 1], [1, 2], [2, 3],[3, 4]] respectively:

$([\![\texttt{a~Uniform(0,4)}]\!]\sigma^{\#}_{init})(P) =$

1/4	1/4	1/4	1/4

The first `1/4` approximates the probability density when $a \in [0,1]$ and the second `1/4` approximates the density when $a \in [1,2]$, and so on. Denote the result state as $\sigma^{\#}_1$. Besides the Density Cube $\sigma^{\#}_1(P)$, $\sigma^{\#}_1$ also contains the Interval Cubes $\sigma^{\#}_1(a) = \sigma^{\#}_{init}(a)$ and $\sigma^{\#}_1(b) = \sigma^{\#}_{init}(b)$.
Line 2. `b ~ Uniform(0,4)` defines the prior distributions for variables b. Denote the density of b at the lower bounds of the intervals as

$P^b =$

1/4
1/4
1/4
1/4

then the result Density Cube representing the joint density of a and b will be:

$([\![\texttt{b~Uniform(0,4)}]\!]\sigma^{\#}_1)(P) =$

1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16

where the element at column i and row j is calculated from $(\sigma^{\#}_1(P))(i) \cdot P^b(j) =$ `1/4*1/4`. It represents the density when a is in the i-th interval of $\sigma^{\#}_1(a)$ and b is in the j-th interval of $\sigma^{\#}_1(b)$. Intuitively the rows correspond to the intervals of a and the columns correspond to the intervals of b. We let different sampled variables occupy different dimensions, e.g. $\sigma^{\#}_1(a)$ has shape $\boldsymbol{M}_a = (4,1)$ while $\sigma^{\#}_1(b)$ has shape $\boldsymbol{M}_b = (1,4)$. Then the shape of the Density Cube for the joint density is simply $\boldsymbol{M}_a \odot \boldsymbol{M}_b = (4,4)$. Let the result state be $\sigma^{\#}_2$ with $\sigma^{\#}_2(P)$ given above and $\sigma^{\#}_2(a) = \sigma^{\#}_1(a)$ and $\sigma^{\#}_2(b) = \sigma^{\#}_1(b)$.
Line 3. `c = a + b` specifies a dependent variable c. The Interval Cube of c and the corresponding Density Cube after the statement will be:

$([\![\texttt{c=a+b}]\!]\sigma^{\#}_2)(c) =$

[0, 2]	[1, 3]	[2, 4]	[3, 5]
[1, 3]	[2, 4]	[3, 5]	[4, 6]
[2, 4]	[3, 5]	[4, 6]	[5, 7]
[3, 5]	[4, 6]	[5, 7]	[6, 8]

$([\![\texttt{c=a+b}]\!]\sigma^{\#}_2)(P) =$

1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16

where in Interval Cube the interval at column i and row j is calculated from the i-th interval of $\sigma^{\#}_2(a)$ and j-th interval of $\sigma^{\#}_2(b)$. For example, `[2, 4]` at column 3 row 1 is from $a \in$ `[2, 3]` and $b \in$ `[0, 1]`, by adding the lower and upper bounds respectively. Then, each element of $([\![\texttt{c=a+b}]\!]\sigma^{\#}_2)(c)$ has its corresponding probability density in $([\![\texttt{c=a+b}]\!]\sigma^{\#}_2)(P)$ at the same position: for example, `[2, 4]` at column 3 row 1 has the corresponding density in $([\![\texttt{c=a+b}]\!]\sigma^{\#}_2)(P)$ at column 3 row 1 being `1/16`. Let the result state be $\sigma^{\#}_3$ with $\sigma^{\#}_3(c)$ given above and $\sigma^{\#}_3(P) = \sigma^{\#}_2(P)$, $\sigma^{\#}_3(a) = \sigma^{\#}_2(a)$, and $\sigma^{\#}_3(b) = \sigma^{\#}_2(b)$.
Line 4. `observe(Normal(c, 1), 5)` means that the observed data (which is `5`) follows a Normal distribution with mean being c and variance 1. In Bayesian terminology, it defines a likelihood function as *Normal*_pdf(5,c,1). To simplify the notation, we let

$$lik(c) = Normal\_\text{pdf}(5,c,1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(5-c)^2}{2}}$$

Since c can take multiple intervals, the result Density Cube is

$([\![\texttt{observe(Normal(c, 1), 5)}]\!]\sigma_3^\#)(P) =$

1/16*lik(0)	1/16*lik(1)	1/16*lik(2)	1/16*lik(3)
1/16*lik(1)	1/16*lik(2)	1/16*lik(3)	1/16*lik(4)
1/16*lik(2)	1/16*lik(3)	1/16*lik(4)	1/16*lik(5)
1/16*lik(3)	1/16*lik(4)	1/16*lik(5)	1/16*lik(6)

,

where each entry is by replacing c with the lower bound of each interval in $\sigma_3^\#(c)$. Note that the density is accurate at the lower bound of each interval, e.g. when $a=2$ (the lower bound of the third interval in $\sigma_3^\#(a)$) and $b=0$ (the lower bound of the first interval in $\sigma_3^\#(b)$) the result density will be exactly 1/16*lik(2) (the element in the third column and first row in the result Density Cube). Using the density for other values in the interval may result in a bounded error. The error can go to 0 when the number of splits goes to infinity (Theorem 3). Let the result state be $\sigma_4^\#$ with $\sigma_4^\#(P)$ given above and other variables unchanged. **Line 5.** `return a, b` will let AQUA output the normalized joint density and marginalized densities of a and b. AQUA will first approximate the unnormalized joint density with a function $\hat{f}(a,b)$, which is constructed by the piecewise constant interpolation (Definition 6) of $\sigma_4^\#(P)$. Recall the columns and rows of $\sigma_4^\#(P)$ correspond to the values of a and b stored in $\sigma_4^\#(a)$ and $\sigma_4^\#(b)$. Then AQUA normalize $\hat{f}(a,b)$ to get the normalized joint density. To get the marginalized densities of a, AQUA integrates $\hat{f}(a,b)$ over b. This is equivalent to summing up $\sigma_4^\#(P)$ along the rows, interpolating and normalizing. Similarly, to get the marginalized densities of b, AQUA integrates $\hat{f}(a,b)$ over a, which is equivalent to summing up $\sigma_4^\#(P)$ along the columns, interpolating and normalizing.

3.3 Formal Guarantee of Accuracy

In this section we formally derive how well the symbolic state $\sigma^\#$ approximates the joint unnormalized density function f and the posterior density function p. To simplify the presentation, we use $\underline{\boldsymbol{x}}^{(\boldsymbol{m})} = [\underline{\boldsymbol{I}^{x_1}_{M_1}(\boldsymbol{m})}, \ldots, \underline{\boldsymbol{I}^{x_N}_{M_N}(\boldsymbol{m})}]$ for all variables, and analogously for $\overline{\boldsymbol{x}}^{(\boldsymbol{m})}$.

Definition 8 We write $\gamma(\sigma^\#) =^\mu_Q f$ if $\sigma^\#(P)(\boldsymbol{m}) = f(\boldsymbol{x})$ when $\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, which means the abstract transformers are exact at the lower bounds. Further, if f is μ-Lipschitz continuous, namely $|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)| \le \mu\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$, we write $\gamma(\sigma^\#) =^\mu_Q f$.

In Lemma 1, to allow the posterior distributions to be Lipschitz continuous, we put an additional restriction on our programs: if an expression follows a discrete distribution, it must be a branching condition, e.g. `b ~ Uniform(0,1); if (b > 0.5){...}` where `b > 0.5` is discrete. The result posterior distribution of the statement will be a mixture of conditional distributions and thus is continuous. For example, it is acceptable to have the conditional statement `if (b > 0.5) {x ~ Normal(0,1)} else {x ~ Normal(1,1)}`. We do not provide formal guarantee for the programs with discrete distributions elsewhere, nevertheless, AQUA may still run on those programs and give results with small error.

Lemma 1 (Discretization Error) The error of discretization is $|\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})| \le \mu \cdot \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$ if $\boldsymbol{x} \neq \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, and if $\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$ the error is 0.

Lemma 1 shows that at any program point, the error is bounded if we use the analysis result $\gamma(\sigma^\#) = \hat{f}$ as an approximation of joint density function f, and the error will reduce when the number of intervals is increased.

Proof of Lemma 1 We need to show that $\gamma(\sigma^\#) =^\mu_Q f$ at any program point. The proof is by structural induction on Expressions and Statements.

First we prove for all expressions E, $([\![E]\!]^\#\sigma^\#)(\boldsymbol{m}) = [([\![E]\!]\sigma)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), ([\![E]\!]\sigma)(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$, and further $[\![E]\!]\sigma$ about variable $\boldsymbol{x}$ is μ_E-Lipschitz continuous if $[\![E]\!]\sigma$ is not the condition in if-then-else. We assume the function $[\![\_]\!]$ binding σ and $[\![\_]\!]^\#$ binding $\sigma^\#$ have the highest precedence, so hereon we omit the parentheses around them. The proof is by structural induction on ***expressions:***

Base case for expressions: for constants, $[\![c]\!]^\#\sigma_0^\#(\cdot) = [c,c] = [[\![c]\!]\sigma_0, [\![c]\!]\sigma_0]$, where $[\![c]\!]\sigma_0 = c$ is a constant and thus is 0-Lipschitz continuous. For variables, $[\![x]\!]^\#\sigma_0^\#(\boldsymbol{m}) = [\underline{I^x}, \overline{I^x}] = [[\![x]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), [\![x]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$ and $[\![x]\!]\sigma_0$ about $\boldsymbol{x}$ is 1-Lipschitz continuous.

Inductive steps for expressions:

1. $[\![E_1[E_2]]\!]^\#$: E_2 must be evaluated to a constant, in the form of $[c,c]$. Then $[\![E_1[E_2]]\!]^\# = [\![E_1[c]]\!]^\#$, and we evaluate $E_1[c]$ in $\sigma^\#$ as an individual variable.

2. $[\![E_1\ op\ E_2]\!]^\#\sigma_0^\#$: each element in the result Interval Cube is $[\![E_1\ op\ E_2]\!]^\#\sigma_0^\#(\boldsymbol{m}) = [\underline{\boldsymbol{I}^{E_1}}\ op\ \underline{\boldsymbol{I}^{E_2}}, \overline{\boldsymbol{I}^{E_1}}\ op\ \overline{\boldsymbol{I}^{E_2}}] = [[\![E_1]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})\ op\ [\![E_2]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), [\![E_1]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})\ op\ [\![E_2]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$. Because $[\![E_1]\!]\sigma_0$ and $[\![E_2]\!]\sigma_0$ about $\boldsymbol{x}$ are Lipschitz continuous (by inductive hypothesis), and op is one of $+,-,*$, $[\![E_1\ op\ E_2]\!]\sigma_0$ about $\boldsymbol{x}$ is Lipschitz continuous. If op is $/$, we only allow E_2 to be a non-zero constant. If op is $>$, we represent the result *True* and *False* with 0 and 1 respectively. If $[\![E_1 > E_2$ is not the condition in the conditional statements, we require the operands of $>$ to be independent of $\boldsymbol{x}$ and thus be Lipschitz continuous.

3. $[\![d(E_1,...).\text{pdf}(E_n)]\!]^\# \sigma_0^\# : \text{Let } \boldsymbol{I}_{M_i}^{E_i} = [\![E_i]\!]^\# \sigma_0^\#$ for $i \in \{1,2,...,n\}$, and $[\![d(E_1,...).\text{pdf}(E_n)]\!]^\# \sigma_0^\# = \boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}$. Then each element in the Interval Cube $\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k}) = [d\_\text{pdf}\,(\underline{\boldsymbol{I}_{M_n}^{E_n}(\boldsymbol{m}_n)}, \underline{\boldsymbol{I}_{M_1}^{E_1}(\boldsymbol{m}_1)}, ...),\ d\_\text{pdf}\,(\overline{\boldsymbol{I}_{M_n}^{E_n}(\boldsymbol{m}_n)}, \overline{\boldsymbol{I}_{M_1}^{E_1}(\boldsymbol{m}_1)}, ...)] = [d\_\text{pdf}\,([\![E_n]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![E_1]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), ...),\ d\_\text{pdf}\,([\![E_n]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![E_1]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})}), ...)] = [[\![d\_\text{pdf}(E_n,E_1,...)]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![d\_\text{pdf}(E_n,E_1,...)]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})})]$. And for all the distributions in this language, d_pdfs about $\boldsymbol{x}$ is Lipschitz continuous.

Then we prove the lemma for ***statements:*** as the *base case*, we initialize $f(\cdot) = 1$, and $\sigma^\#(P)(\cdot) = 1$, so $\gamma(\sigma^\#) =_Q^0 f$ (f is 0-Lipschitz continuous). By applying the function *GetInitIntervals* in Algorithm 1, we have the initial splits for all variables $\boldsymbol{x}$, and $\sigma^\#(P)(\boldsymbol{m}) = 1$ for any $\boldsymbol{m}$ as the index of density cube. Then $(\sigma^\#) =_Q^0 f$. Suppose $\gamma(\sigma_0^\#) =_Q^{\mu_0} f_0$ holds before statement S. In inductive steps we prove $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$ where $\sigma_1^\# = [\![S]\!]^\# \sigma_0^\#$, and $f_1 = ([\![S]\!]\sigma)(\mathcal{L})$ is true density function after the statement.

We apply structural induction on statements:

1. $[\![\text{skip}]\!]^\#$: $\sigma_1^\#(\boldsymbol{x}) = \sigma_0^\#(\boldsymbol{x})$, $f_0(\boldsymbol{x}) = f_1(\boldsymbol{x}) \Longrightarrow \gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$ where $\mu_1 = \mu_0 < \infty$.

2. $[\![x = E]\!]^\#$: $\sigma_1^\#(P) = \sigma_0^\#(P)$ since this assign is deterministic and does not change the density cube, so $f_0 = f_1$. Because $\gamma(\sigma_0^\#) =_Q^{\mu_0} f_0$, $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$.

3. $[\![S_1; S_2]\!]^\#$: Let σ_0 be the concrete state with variable $\boldsymbol{x}$ before the statement, and let σ_1 be the concrete state after the statement, namely $\sigma_0(\mathcal{L}) = f_0(\boldsymbol{x})$ and $\sigma_1(\mathcal{L}) = f_1(\boldsymbol{x})$. Then, since $\gamma(\sigma_0^\#) =_Q^{\mu_0} f_0$, by inductive hypothesis, $([\![S_1]\!]^\# \sigma_0^\#)(P)(\boldsymbol{m}) = f_1(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) = ([\![S_1]\!]\sigma_0)(\mathcal{L})(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$, and f_1 is μ_1-Lipschitz continuous. Let $\sigma_1^\# = [\![S_1]\!]^\# \sigma_0^\#$, then $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$. Apply inductive hypothesis again, $([\![S_2]\!]^\# \sigma_1^\#)(P)(\boldsymbol{m}) = f_2(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) = ([\![S_2]\!]\sigma_1)(\mathcal{L})(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$ and f_2 is μ_2-Lipschitz continuous. So $\gamma(\sigma_2^\#) =_Q^{\mu_2} f_2$.

4. $[\![\text{factor}(E)]\!]^\#$: In concrete semantics, let $([\![E]\!]\sigma_0)$ be the result of evaluating E. The true density function is derived as $f_1(\boldsymbol{x}) = f_0(\boldsymbol{x}) \cdot (([\![E]\!]\sigma_0)(\boldsymbol{x}))$. In symbolic semantics, $[\![E]\!]^\# \sigma_0^\#$ is the interval cube where $([\![E]\!]^\# \sigma_0^\#)(\boldsymbol{m}) = [([\![E]\!]\sigma_0)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), ([\![E]\!]\sigma_0)(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$. Then $\sigma_1^\#(P)(\boldsymbol{m}) = \sigma_0^\#(P)(\boldsymbol{m}) \cdot (([\![E]\!]\sigma_0)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})) \Longrightarrow \sigma_1^\#(P)(\boldsymbol{m}) = f_1(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. To show f_1 is μ_1-Lipschitz continuous, $|\, f_1(\boldsymbol{x}_1) - f_1(\boldsymbol{x}_2) \,| \le \mu_0 \|\boldsymbol{x}_2 - \boldsymbol{x}_1\| \,|\, f_1(\boldsymbol{x}_1) \,| + |\, f_0(\boldsymbol{x}_1) - f_0(\boldsymbol{x}_2) \,|\, |\, f_0(\boldsymbol{x}_2) \,|) \,\|\boldsymbol{x}_2 - \boldsymbol{x}_1\| = \mu_1 \|\boldsymbol{x}_2 - \boldsymbol{x}_1\|$, where μ_E is the Lipschitz constant of the expression $[\![E]\!]$ (see the proof for expressions below). This implies $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$.

5. $[\![\text{observe}(d(E_1,...,E_{n-1}),E_n)]\!]^\#$: by $[\![\text{factor}(E)]\!]^\#$ rule above.

6. $[\![x \sim d(E_1,...,E_n)]\!]^\#$: First, we evaluate the expression $[\![d(E_1,\dots,E_n).\text{pdf}(x)]\!]^\# \sigma_0^\#$ to get $\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}$ (see the proof for expressions below). According to the analysis rule in Figure 4, $\sigma_1^\#(P)(\boldsymbol{m}) = \sigma_0^\#(P)(\boldsymbol{m}_0) \cdot \boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k})$. Also, $f_1(\boldsymbol{x}) = f_0(\boldsymbol{x}) \cdot ([\![d(E_1,\dots,E_n).\text{pdf}(x)]\!]\sigma_0)(\boldsymbol{x})$. By inductive hypothesis, $\sigma_0^\#(P)(\boldsymbol{m}) = f_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$, so $\sigma_1^\#(P)(\boldsymbol{m}) = \sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k})} = f_1(\boldsymbol{x})$, which implies $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$, where μ_1 is the Lipschitz constant of f_1. To prove f_1 is μ_1-Lipschitz continuous, $|\ f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\ | \le |\ f(\boldsymbol{x}_1) \cdot \text{d\_pdf}(\boldsymbol{x}_1) - f(\boldsymbol{x}_2) \cdot \text{d\_pdf}(\boldsymbol{x}_2)\ | \le \mu_0 \cdot |\, \text{d\_pdf}(\boldsymbol{x}_1)\,| + |\, \text{d\_pdf}(\boldsymbol{x}_1) - \text{d\_pdf}(\boldsymbol{x}_2)\,|\,|\, f_0(\boldsymbol{x}_2)\,| = \mu_1 < \infty$. Therefore, $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$.

7. $[\![\text{if } (E) \text{ then } S_1 \text{ else } S_2]\!]^\# \sigma_0^\# = ([\![\text{factor}(E);\ S_1]\!]\sigma_0^\#) \sqcup ([\![\text{factor}(1-E);\ S_2]\!]\sigma_0^\#)$. After evaluating the statement in the true branch we get $\gamma(\sigma_T^\#) =_Q^{\mu_T} f_T$, and after evaluating the false branch we get $\gamma(\sigma_F^\#) =_Q^{\mu_F} f_F$. Then $(\sigma_T^\# \sqcup \sigma_F^\#)(P)(\boldsymbol{m}) = \sigma_T^\#(P)(\boldsymbol{m}) + \sigma_F^\#(P)(\boldsymbol{m}) = f_T(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) + f_F(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Also, $|\, f_1(\boldsymbol{x}_1) - f_1(\boldsymbol{x}_2)\ | = |\ f_T(\boldsymbol{x}_1) + f_F(\boldsymbol{x}_1) - (f_T(\boldsymbol{x}_2) + f_F(\boldsymbol{x}_2))\ | \le \mu_T |\, \boldsymbol{x}_1 - \boldsymbol{x}_2\,| + \mu_F |\, \boldsymbol{x}_1 - \boldsymbol{x}_2\,| = (\mu_T + \mu_F)\|\boldsymbol{x}_1 - \boldsymbol{x}_2\| = \mu_1 \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$. Therefore, $\gamma(\sigma_1^\#) =_Q^{\mu_1} f_1$.

8. $[\![\text{for } (i \text{ in } E_1..E_2)\ S]\!]^\#$ is reduced to if-then-else and sequencing. Thus the property holds. □

The error of AQUA's approximation to the normalizing constant z is also bounded:

Lemma 2 (Integration Error) Let $U = \prod_{i=1}^N (b_i - a_i)$ be the volume of $\boldsymbol{C}$. For all the probability distributions supported in our language, the error is $|z - \hat{z}| \le U\mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. If we use M equal-length intervals for each variable, $|z - \hat{z}| \le U\mu \frac{1}{M} (\sum_{i=1}^N (b_i - a_i)^2)^{\frac{1}{2}}$. Then $|z - \hat{z}| \to 0$ as $M \to \infty$.

Proof of Lemma 2 Recall, all posteriors f in our language (Section 2) are Lipschitz continuous. We derive the error bound by applying the Lipschitz continuous property of f and the triangle inequality. First, $\hat{z} = \int_{\boldsymbol{C}} \hat{f}(\boldsymbol{x}) d\boldsymbol{x} = \sum_{\boldsymbol{m}}^{\boldsymbol{M}} \left(\prod_i^N (\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}) \cdot p^{(\boldsymbol{m})} \right)$. According to Lemma 8, $\hat{f}$ is a quantization of f, meaning $\hat{f}(\boldsymbol{x}) = f(\boldsymbol{x})$ at the points $\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, while for other $\boldsymbol{x} \in (\underline{\boldsymbol{x}}^{(\boldsymbol{m})}, \overline{\boldsymbol{x}}^{(\boldsymbol{m})})$, $\hat{f}(\boldsymbol{x}) = f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Then

$$
\begin{aligned}
|z - \hat{z}| &= \left| \int_{\boldsymbol{C}} f(\boldsymbol{x}) d\boldsymbol{x} - \int_{\boldsymbol{C}} \hat{f}(\boldsymbol{x}) d\boldsymbol{x} \right| && \text{(Error term)} \\
&\le \int_{\boldsymbol{C}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})| d\boldsymbol{x} && \text{(Triangle ineq.)} \\
&\le U \cdot \max_{\boldsymbol{x}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})| && (U \text{ is volume of } \boldsymbol{C})
\end{aligned}
$$

Then we prove $\max_{\boldsymbol{x}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})| \le \mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. For each interval box $[\boldsymbol{x}^{(\boldsymbol{m})}]$, $f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}) = f(\boldsymbol{x}) - f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Because f is μ-Lipschitz

continuous, $|f(\boldsymbol{x}) - f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})| \leq \mu\|\boldsymbol{x} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\| \leq \mu\max_{\boldsymbol{m}}\|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. □

Moreover, the integration error bound above will decrease when we decrease the interval length, or increase the number of intervals. Then at the end of the analysis, we approximate the *posterior probability density function* $p(\boldsymbol{x})$ on $\boldsymbol{C}$ as:

Definition 9 (Posterior Probability Density Approximation) Define $\hat{p}(\boldsymbol{x}) = \frac{1}{\hat{z}}\hat{f}(\boldsymbol{x})$ as the approximation of $p(\boldsymbol{x})$.

Now we show the end-to-end error of the analysis. As Theorem 3 states, by applying sufficiently many intervals, the random variables following AQUA's posterior estimation in $\boldsymbol{C}$ will *converge in distribution* to the true posterior in $\boldsymbol{C}$. Without loss of generality, suppose we apply at least M equal-length intervals for each variable in its domain $[a_i,b_i]$, i.e. $M = \min\{M_1,M_2,...,M_N\}$. And we refer $\hat{p}_M(\boldsymbol{x})$ as AQUA's approximation of $p(\boldsymbol{x})$ by applying at least M equal-length intervals for each variable.

Theorem 3 (Convergence of Posterior Density Approximation) *Define* $F_{\boldsymbol{C}}(\boldsymbol{x}) = \frac{1}{z}\int_{-\infty}^{\boldsymbol{x}} \mathbf{1}_{[\boldsymbol{u}\in\boldsymbol{C}]}\cdot p(\boldsymbol{u})d\boldsymbol{u}$ *as the true cumulative distribution function (CDF) on* $\boldsymbol{C}$*, where* $z = \int_{\boldsymbol{C}} p(\boldsymbol{x})d\boldsymbol{x}$*, and* $\hat{F}_{\boldsymbol{C},M}(\boldsymbol{x}) = \int_{-\infty}^{\boldsymbol{x}} \hat{p}_M(\boldsymbol{u})d\boldsymbol{u}$ *as the approximate CDF. Then*

$$\lim_{M\to\infty} \hat{F}_{\boldsymbol{C},M}(\boldsymbol{x}) = F_{\boldsymbol{C}}(\boldsymbol{x}).$$

Proof of Theorem 3 Recall $\boldsymbol{C} = \bigotimes_{i=1}^{N}[a_i,b_i]$, so $\boldsymbol{a}$ is the lower bound of $\boldsymbol{C}$. Given Lemma 1 for the quantization error $|\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})| \leq \mu\max_{\boldsymbol{m}}\|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$, we know $|\int_{\boldsymbol{a}}^{\boldsymbol{x}} \hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})d\boldsymbol{u}| \leq \int_{\boldsymbol{a}}^{\boldsymbol{x}}|\hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})|d\boldsymbol{u} \leq \theta\mu\max_{\boldsymbol{m}}\|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$, where $\theta = \|\boldsymbol{x} - \boldsymbol{a}\|$. Using M equal-length splits for each variables, we can write $\max_{\boldsymbol{x}}\|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\| = \sqrt{\sum_{i=1}^{N}\left(\frac{b_i-a_i}{M}\right)^2} = \frac{h}{M}$ where $h = \sqrt{\sum_{i=1}^{N}(b_i-a_i)^2}$ is a constant.

By combining the error bounds in Lemma 1 and Lemma 2 and applying triangle inequality (Tri. ineq.), we show the convergence of the end-to-end error:

$$\begin{aligned}
&|\hat{F}_{\boldsymbol{C},t}(\boldsymbol{x}) - F_{\boldsymbol{C}}(\boldsymbol{x})| \\
&= |\frac{\int_{-\infty}^{\boldsymbol{x}}\hat{p}(\boldsymbol{u})d\boldsymbol{u}}{\int_{\boldsymbol{C}}\hat{p}(\boldsymbol{x})d\boldsymbol{x}} - \frac{\int_{-\infty}^{\boldsymbol{x}}\mathbf{1}_{[\boldsymbol{u}\in\boldsymbol{C}]}p(\boldsymbol{u})d\boldsymbol{u}}{\int_{\boldsymbol{C}}p(\boldsymbol{x})d\boldsymbol{x}}| && \text{(Definition 9)} \\
&= |\frac{\int_{\boldsymbol{a}}^{\boldsymbol{x}}\hat{f}(\boldsymbol{u})d\boldsymbol{u}}{\hat{z}} - \frac{\int_{\boldsymbol{a}}^{\boldsymbol{x}}f(\boldsymbol{u})d\boldsymbol{u}}{z}| && \text{(Definition 7)} \\
&\leq \frac{z|\int_{\boldsymbol{a}}^{\boldsymbol{x}}\hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})d\boldsymbol{u}| + \int_{\boldsymbol{a}}^{\boldsymbol{x}}f(\boldsymbol{u})d\boldsymbol{u}|z-\hat{z}|}{\hat{z}z} && \text{(Tri. ineq.)} \\
&\leq \frac{z(\theta\mu h/M) + \int_{\boldsymbol{a}}^{\boldsymbol{x}}f(\boldsymbol{u})d\boldsymbol{u}|z-\hat{z}|}{\hat{z}z} && \text{(Lemma 1)} \\
&\leq \frac{z(\theta\mu h/M) + \int_{\boldsymbol{a}}^{\boldsymbol{x}}f(\boldsymbol{u})d\boldsymbol{u}(U\mu h/M)}{\hat{z}z} && \text{(Lemma 2)} \\
&\leq \frac{z\theta\mu h + F_{\boldsymbol{C}}(\boldsymbol{x})U\mu h}{M\cdot\hat{z}z} \\
&\to 0 \qquad \text{as } M\to\infty
\end{aligned}$$

Then θ, μ (Lipschitz constant of f), z (normalizing constant), U (volume of $\boldsymbol{C}$), and $F_{\boldsymbol{C}}(\boldsymbol{x})$ are all constants regarding M, and $\hat{z}\to z>0$ as $M\to\infty$. Hence $|\hat{F}_{\boldsymbol{C},t}(\boldsymbol{x}) - F_{\boldsymbol{C}}(\boldsymbol{x})|\to 0$ as $M\to\infty$. □

We allow a user to provide a bounded domain $\boldsymbol{C}$, or infer it with automatically with a heuristic (Section 4). Although AQUA's formal guarantee is in a bounded domain, it can give runtime warnings when any prior or likelihood has probability greater than a given threshold on the rest of the domain $\mathbb{R}^N - \boldsymbol{C}$. If AQUA does not give any warning, the final error caused by truncating infinite domain into $\boldsymbol{C}$ will be smaller than the threshold.

4 AQUA Optimizations

Adaptive Intervals. To find the suitable bounded intervals $\boldsymbol{C} = [C_1,C_2,...,C_N]$ that cover most probability, we design a adaptive algorithm (Algorithm 2) to adjust $\boldsymbol{C}$ the based on the result from last run.

Algorithm 2 takes as inputs the program, the vector of number of intervals, and two thresholds t_0 and t_{dist} for deciding the interval bounds $\boldsymbol{C}$. Increasing C_i or increasing the number of intervals in C_i will help reduce the approximation error.

The function *GetInitBounds* (Line 2) takes the prior distribution of each x_i as a rough estimate of the distribution to determine an initial interval split. If the domain of the prior distribution is

Algorithm 2 Posterior Interval Analysis with Adaptive Interval

```
1:  procedure PosteriorAdaptiveAnalysis(Prog,M,t0,tdist)
2:      C ← GetInitBounds(Prog,t0)        ▷ C = [C1,C2,...,CN]
3:      changed ← True
4:      while changed do                  ▷ Stop if C no longer changes
5:          (p̂,Marginal) ← PosteriorAnalysis(Prog,M,C)
6:          changed ← False
7:          for xi ∈ SampledVars(Prog) do     ▷ Adapt each Ci
8:              p̂i(xi) ← Marginal[xi]
9:              if ∃xi ∈ Ci, p̂i(xi) < tdist then
10:                 ai ← inf{xi | p̂i(xi) > tdist}
11:                 bi ← sup{xi | p̂i(xi) > tdist}
12:                 Ci ← [ai,bi]
13:                 changed ← True
14:             end if
15:         end for
16:     end while
17: return (p̂, Marginal)
18: end procedure
```

Table 2: Program Description and Characteristics

	Description	Distributions	#D	#N
prior_mix	Mixture model[14]	$B\times(N+N)\times T^{10}$	10	1
zeroone	Bayesian neural network[15]	$U^2\times M^{20}$	20	2
tug	Causal cognition model[16]	$U^2\times(N+N)^2\times B^{40}$	40	2
altermu	Model with param symmetry[17]	$N^3\times N^{40}$	40	3
altermu2	Model with param symmetry[17]	$U^2\times N^{40}$	40	2
neural	Bayesian neural network[18]	$U^2\times(B\times M)^{39}$	39	2
normal_mixture	Mixture model with mixing rate[19]	$N^2\times Be\times(B\times(N+N))^{63}$	63	3
mix_asym_prior	Mixture model with scale params[19]	$N^2\times G^2\times(B\times(N+N))^{40}$	40	4
logistic	Logistic regression[19]	$U^2\times(B\times M)^{100}$	100	2
logistic_RW	Reweighted logistic regression[19, 20]	$U^2\times Be^{100}\times(B\times M)^{100}$	100	102
anova	Linear regression [19]	$U^2\times N^{40}$	40	2
anova_RP	Localized linear regression[19, 21]	$U^2\times G^{40}\times N^{40}$	40	42
anova_RW	Reweighted linear regression[19, 20]	$U^2\times Be^{40}\times N^{40}$	40	42
lightspeed	Linear regression[19]	$N\times U\times N^{66}$	66	2
lightspeed_RP	Localized linear regression[19, 21]	$N\times U\times G^{66}\times N^{66}$	66	68
lightspeed_RW	Reweighted linear regression[19, 20]	$N\times U\times Be^{66}\times N^{66}$	66	68
unemployment	Linear regression[19]	$N^2\times U\times N^{40}$	40	3
unemployment_RP	Localized linear regression[19, 21]	$N^2\times U\times G^{40}\times N^{40}$	40	43
unemployment_RW	Reweighted linear regression[19, 20]	$N^2\times U\times Be^{40}\times N^{40}$	40	43
timeseries	Timeseries analysis[19]	$U^3\times N^{39}$	39	3
gammaTransform	Transformed param[9]	G	0	3
GPA	Hybrid continuous & discrete distr.[22]	$B\times(B\times(A+U)+B\times(A+U))$	1	3
radar_query1	Bayesian network in robotics[7]	$B\times(A+B)\times U\times N\times(Tr+Tr)$	2	6
radar_query2	Bayesian network in robotics[7]	$B\times(A+B)\times U^2\times N\times Tr$	1	6

Distributions: A: Atomic, B: Bernoulli, Be: Beta, G: Gamma, M: Softmax, N: Normal, T: Student-T, Tr: Triangular, U: Uniform. '+' represents the mixture of two distributions, and '×' represents the product of the individual density functions in the joint probability density function.

bounded in $[a_i,b_i]$ where $-\infty<a_i<b_i<\infty$, e.g. $\texttt{x}_\texttt{i}\sim\texttt{Uniform(a,b)}$, AQUA divides $[a_i,b_i]$ into M_i equi-length intervals, each with length $(b_i-a_i)/M_i$, where M_i is given in $\boldsymbol{M}$ by the user. If the distribution is not bounded, e.g. $\texttt{x}_\texttt{i}\sim\texttt{Normal(0,1)}$, the user can specify a threshold t_0 for AQUA to infer C_is such that values from the prior being out of C_is has probability smaller than t_0. Otherwise by default we set $t_0=4\cdot10^{-32}$.

In each iteration, the algorithm applies the analysis on the current $\boldsymbol{C}$ (line 5) and check if we need to adapt $\boldsymbol{C}$. We adapt $\boldsymbol{C}$ when any variable x_i has density value $\hat{p_i}(x_i)$ being almost about 0 – smaller than the user provided threshold t_{dist} (e.g. 10^{-8}) (line 8-12). We shrink C_i to focus on the smallest area with density greater than a given threshold t_{dist}. With the same number of intervals M_i, the smaller C_i will produce thiner intervals and result in more accurate results. Practically, this adaptive algorithm is as accurate but is much more efficient than naively increasing the number of intervals M_i on the whole initial domain C_i. Suppose the program takes A adaptive iterations, and it has N variables and each variable has the same number of intervals M, Algorithm 2 has the time complexity $\mathcal{O}(A\cdot N\cdot M^N)$ and the space complexity $\mathcal{O}(M^N)$. In our experiments, A is usually less than 5.

Improving Inference for Many Local Variables. In this optimization we change the analysis of statements in Section 3 to marginalize the local variables as soon as possible. Local variables are those defined and only used in local blocks (e.g. in for-loop and if-then-else from Figure 4).

By marginalizing out the local variables, we avoid repeatedly computing the joint density on the unused variables. For example, in a robust model one may naively calculate the joint density via $\hat{f}(x)=\prod_{i=1}^{D}\text{d\_pdf}(x,\text{w}_i)$, where w_is are local variables defined in each loop body. This requires keeping a $(D+1)$-dimensional density cube to capture all the variables x and w_is. Instead, our optimization divides the above product into calculating the individual $\text{d\_pdf}(x,\text{w}_i)$, when w_i leaves its scope, so we do not carry the current w_i to the next iteration. In each iteration we only operate on a 2-dimensional Density Cube for variables x and a single w_i. If out of N variables in the program D are local variables we will have a time complexity $\mathcal{O}(N\cdot M^{N-D})$ for Algorithm 1 (while the original is $\mathcal{O}(N\cdot M^N)$).

5 Methodology

We evaluate AQUA on 24 probabilistic programs collected from existing literature. We compare the execution time of AQUA on these programs with other probabilistic programming languages: Stan [1], PSI [7], and SPPL [9]. We implement AQUA in Java using ND4J library for tensor computation,

and run all experiments on Intel Xeon 3.6 GHz machine with 6 cores and 32 GB RAM. For numerical stability, we use log probability/density (instead of original probability/density) for Density Cube.

Benchmarks. Table 2 presents the benchmarks obtained from the literature. Column **Description** summarizes the task of each program. Column **Distributions** shows the distributions of observable and latent variables. For example, the distributions in program "prior_mix" are one Bernoulli (B), one Mixture of two Normals (N+N), and 10 Student-T distributions (T^{10}). All posterior distributions are continuous. Column **#D** shows the number of data observations, **#N** shows the number of random variables in the program. The benchmarks "prior_mix", "normal_mixture", "mix_asym_prior", "GPA", "radar_query1" and "radar_query2" have discrete distributions in the if-then-else conditions, but the result posterior density functions are continuous and thus AQUA's formal guarantee holds. "zeroone" and "tug" have discrete distributions in the intermediate results and posterior densities, where AQUA cannot provide the formal guarantee. However, Section 6.1 gives empirical evidence that AQUA's result error is very small.

Comparing Posterior Distributions. The Kolmogorov-Smirnov (KS) statistic measures the distance between two probability distributions. We use the KS statistic for the accuracy evaluation in the analysis. Let F_{truth} and $\hat{F}$ denote the cumulative density functions of the posterior distributions of the variable x from the original input data and the noisy data respectively, the *KS statistic* is defined as $KS = \sup_x |F_{truth}(x) - \hat{F}(x)|$, namely, the maximum difference in the cumulative distribution functions. The KS statistic takes a value between 0 (most close distributions) and 1 (most different distributions). Therefore, smaller KS statistic implies better accuracy.

Experimental Setup. We manually derived the *ground truth* posterior distributions for all the programs. We run AQUA with the adaptive algorithm described in Section 4. We use the equal number of $M = \max\{60, \lceil 40000^{(1/N)} \rceil\}$ intervals for each variable, where N is the number of sampled variables, so that the total number intervals $M^N \geq 40000$. Rounding up the total number of intervals to 40000 does not significantly affect time but will guarantee more accurate results. We test Stan on its two major inference algorithms, NUTS (a variant of MCMC) and ADVI (a variant of variational inference). For fair comparison, we allow running VI/NUTS until it reaches the same accuracy level (in KS statistic) as AQUA and report the average time, or until it reaches the maximum iterations (fixed at 400000 for both VI and NUTS). We set the timeout to be 20 minutes for all the inference tools.

6 Evaluation

6.1 Runtime/Accuracy Comparison

Table 3 presents the runtime and accuracy comparison of AQUA with Stan, PSI, and SPPL. Column **Program** shows the name of the probabilistic program. Columns **Time (s)** show the execution time (in seconds) of each tool, averaged across 5 runs. We report the total time for computing joint density and marginals for all sampled variables. Columns **Error** show the error (KS statistic, Section 5) of each tool vs. the ground truth when run for the same time, averaged across 5 runs.

Overall, **AQUA** (Column 2-3) solves the probabilistic programs with average time 5.08s, median time 1.35s. For 20 out of 24 programs, it takes less than two seconds to compute the results. AQUA results in average error 0.01, median error 0.01, and maximum error 0.02. With our optimization on local variables (Section 4), we are able to handle the 7 robust programs which have 42-102 variables, which might timeout with a naive approach.

Stan VI (Column 4-5) finishes fast but results in significantly larger error than AQUA or Stan NUTS. The average error from VI is 0.15, minimum error is 0.03 and maximum error is 0.34. For all cases, VI cannot reach the same accuracy level as AQUA. While VI often fits the posterior means correctly but it is not able to capture the joint distribution shape especially when it is non-Gaussian (it is a well known characteristic of VI). **Stan NUTS** (Column 6-7) takes more time than AQUA to reach the same level of accuracy of AQUA, although in theory NUTS will converge to the true distribution with enough iterations. AQUA provides the similar (with difference < 0.01) or even better accuracy (with smaller KS statistic) in all cases for NUTS and NUTS fails to reach the same accuracy level by the maximum number of iterations in 12 cases.

PSI (Column 8) and **SPPL** (Column 9) are not able to give result in many cases. PSI does not finish running within 20 minutes in 11 cases, or evaluates

Table 3: Runtime Comparison for AQUA, Stan, PSI, and SPPL. Stan column shows time needed reach AQUA's accuracy.

Program	AQUA		Stan VI		Stan NUTS		PSI	SPPL
	Time(s)	Error	Time(s)	Error	Time(s)	Error	Time (s)	Time (s)
prior_mix	4.77	0.02	0.53	0.31	5.67	0.19	inte	⊘
zeroone	0.98	0.00	0.44	0.21	630.73	0.21	91.16	⊘
tug	0.83	0.01	1.20	0.25	519.94	0.06	inte	⊘
altermu	1.35	0.00	0.96	0.31	29.46	0.03	inte	⊘
altermu2	0.76	0.00	0.75	0.34	25.98	0.07	inte	⊘
neural	0.85	0.01	0.82	0.03	5.10	0.01	t.o.	⊘
normal_mixture	1.19	0.02	1.02	0.12	25.67	0.04	t.o.	⊘
mix_asym_prior	24.63	0.02	1.04	0.09	16.41	0.03	t.o.	⊘
logistic	0.99	0.02	0.74	0.07	17.31	0.02	t.o.	⊘
logistic_RW	1.87	0.01	15.37	0.09	72.45	0.02	t.o.	⊘
anova	0.90	0.01	0.75	0.07	6.72	0.02	inte	⊘
anova_RP	1.55	0.01	6.89	0.07	77.48	0.02	t.o.	⊘
anova_RW	1.40	0.01	6.93	0.06	24.67	0.02	t.o.	⊘
lightspeed	0.74	0.00	0.71	0.04	3.56	0.00	inte	⊘
lightspeed_RP	1.37	0.01	6.18	0.06	61.37	0.02	t.o.	⊘
lightspeed_RW	1.09	0.02	6.19	0.05	61.37	0.05	t.o.	⊘
unemployment	1.44	0.02	0.64	0.21	5.07	0.01	inte	⊘
unemployment_RP	42.34	0.01	6.78	0.25	12.46	0.01	t.o.	⊘
unemployment_RW	27.41	0.02	7.07	0.23	2.53	0.01	t.o.	⊘
timeseries	1.55	0.01	0.87	0.23	12.66	0.01	inte	⊘
gammaTransform	0.72	0.00	0.62	0.05	3.01	0.01	inte	1.30
GPA	0.46	0.02	⊘	⊘	⊘	⊘	0.12	0.05
radar_query1	0.87	0.01	⊘	⊘	⊘	⊘	inte	⊘
radar_query2	1.82	0.02	⊘	⊘	⊘	⊘	inte	⊘
Avg	5.08	0.01	3.17	0.15	77.12	0.04	⊘	⊘
Median	1.35	0.01	0.99	0.10	20.99	0.02	⊘	⊘

[*time*] : VI or NUTS takes more time than AQUA, or AQUA take more time than VI and NUTS.
[*error*] Has the error (in terms of a KS statistic) larger than 0.01 from the best solution.
"⊘": the PPL cannot work on the program. "t.o.": timeout, "inte": evaluates to unsolved integrals.

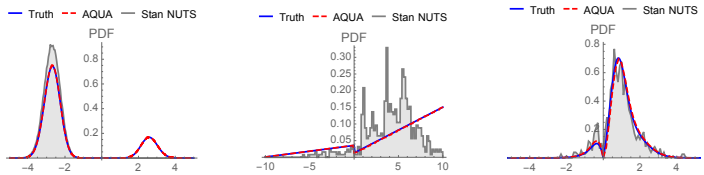
to unsolved integrals in 11 cases, since the exact integration in posterior calculation is often intractable. SPPL does not allow transformed variables in `factor` statements, which is essential to specify the likelihood of the variables given observed data, and thus is inapplicable to most of the programs.

Figure 6 presents the posterior densities from six programs where Stan NUTS was not able to reach the same accuracy level of AQUA, within maximum iterations. X-axis shows the value of a variable in the program, Y-axis shows the posterior probability density of the variable. A solid blue line shows the ground truth, a dashed red line shows the density function computed from AQUA, the gray histogram shows the density estimated with samples from Stan NUTS after running for the same time as AQUA. For each program we present the posterior from one variable (the first one in alphabetical order); the posteriors from other variables show a similar pattern.

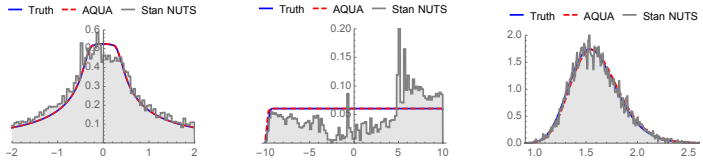
These examples show that AQUA is able to closely track the density of mixture models with large difference in densities ("prior_mix"), non-differentiable distributions ("zeroone" and "tug"), models with variable symmetries (in "altermu" and "altermu2" such symmetries can cause non-identifiability of variables from data), and some robust models with strong correlation between variables that can form complicated posterior geometries ("anova_RP").

6.2 Estimating the Posterior Tails

We illustrate AQUA's ability to capture tails on several robust models. The distribution for robust models are often more spread-out than the original model, as they are designed to capture outliers in the data. We consider two different robust models: (1) Reparameterized-Localization (RP) [21], which assumes that each data point is from its distribution with a local variance variable; (2) Reweighting (RW) [20], which down-weights potential outliers in the data. We show the results from AQUA and NUTS running for the same amount of time, together with

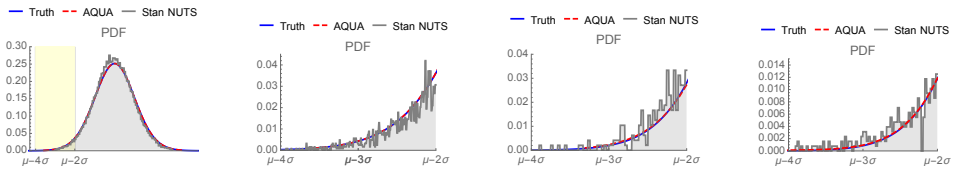


(a) prior_mix (b) zeroone (c) tug

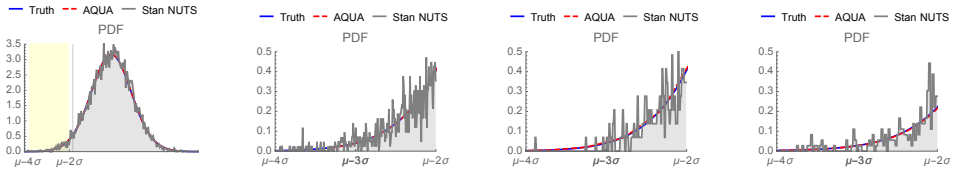


(d) altermu (e) altermu2 (f) anova_RP

Fig. 6: Programs handled by AQUA for which Stan NUTS is imprecise.



(a) lightspeed full (b) lightspeed (c) lightspeed_RP (d) lightspeed_RW



(e) anova full (f) anova (g) anova_RP (h) anova_RW

Fig. 7: Capturing tails by AQUA and Stan NUTS

the ground truth. We omit VI since its accuracy is significantly worse.

Figure 7 presents the comparison of AQUA and NUTS. Plots (a),(e) are the full posterior distributions of original distribution. We highlight the left tail $[\mu-4\sigma, \mu-2\sigma]$, where μ is the posterior mean of and σ its standard deviation. Plots (b),(f) show the magnified tails from original distribution, plots (c),(g) show the tails from the RP transformation, and (d),(h) show the tails from RW transformation. AQUA is able to capture the tails precisely for both original and robust models, while Stan NUTS is less precise on the robust models (e.g., its KS statistic is 0.05 compared to AQUA's 0.02).

7 Related Work

Probabilistic Programming Languages. In recent times, Probabilistic programming languages have gained traction in both academic communities and industry. Most of these languages are thightly coupled with specific algorithms for approximate probabilistic inference. The majoristy of the languages are sampling-based [1, 23–29], and several recent languages support variational inference [1, 30–32]. These languages support a rich set of features including general loops and some have support for higher-order inference. These languages are Turing complete and some of them also support advanced features like higher order functions and composability with neural networks. More recently, languages like Edward and Pyro began combining Bayesian reasoning with deep learning [30, 31]. However, they are inherently approximate: sampling-based approaches can reach accurate solution only in the limit, while the variational inference-based may not have theoretical guarantees (except for specific distributions). Recently, there has been interest in expanding the reach of exact or near-exact inference methods. Although these methods are computationally intractable in general, they can

solve many practical problems. We next discuss these techniques falling in the domain of symbolic inference and volume computation.

Symbolic Inference. Researchers have proposed several symbolic inference techniques in recent years [7–9, 33]. Each of these techniques have limitations which AQUA improves upon. DICE [33] performs fast and exact inference by reducing discrete probabilistic programs to weighted model counting. It supports only programs with discrete distributions.

Hakaru [8] and PSI [7] languages have constructs for both discrete and continuous distributions. They perform exact inference using computer algebra (e.g., Hakaru usues Maple and PSI uses Mathematica). However, they often cannot solve integrals for even modestly complicated probabilistic programs with continuous distributions (as our evaluation has also demonstrated for PSI).

SPPL [9] performs exact inference and supports programs with discrete and continuous distributions. Its inference is enabled by translating programs into sum-product expressions. Nevertheless, it does not allow users to specify the likelihood on transformed variables with continuous distributions, which is a necessity for many real-world models like hierarchical regression or time-series models (such as those used in our evaluation).

QCoral [34] and SYMPAIS [35] combine symbolic execution with sampling to solve the satisfaction probability of constraints. They aim to quantify the probability of a target event by representing the path conditions of the event with symbolic expressions. They apply constraint solver and sampling to compute the probability under the symbolic conditions. However, they only compute the probability of a given event and do not output the whole posterior.

Earlier works [36–40] perform symbolic inference on graphical models. The distributions they support in the graphical models are limited and are not able to represent many models allowed in probabilistic programs. Shachter et al. [36] works on discrete distributions. Chang et al. [37] allows both discrete and continuous distributions, but restricts continuous distributions to be linear-Gaussian related. Other works [38–40] use easy-to-integrate approximations to replace the continuous distributions, which include gaussian mixtures, truncated exponentials, and polynomials. These approximations may introduce inevitable error, and these works do not provide any formal guarantee of the error bound. On the contrary, AQUA avoids distorting the distribution by using quantizations of the original distribution. AQUA's quantizations have the guarantee to converge to the true distributions when using sufficiently many splits (Theorem 3).

In contrast to these existing approaches, AQUA supports a wide range of probabilistic models with continuous distribution, involving transformed or correlated random variables, and provides scalable, exact (or approximately exact), and interpretable solutions.

Volume Computation. Several works use volume computation methods to make a precise approximation of probabilistic inference [2, 3, 10]. These approaches have constraints on the form of programs they support, regarding conditioning and continuous distributions. None of these systems can support conditioning on continuous variables, and thus we have not used them in our evaluation.

Sankaranarayanan et al. [10] estimates the probability upper and lower bounds for properties of probabilistic programs. It applies symbolic execution, bounds the path probabilities with hypercubes and does volume bound computation to estimate the probability of the given property. FairSquare [2] verifies the fairness property of probabilistic programs. It generates probabilistic verification conditions and computes the weighted volume described by the conditions. In the volume computation, it presents an approach to discretize some continuous distributions, but it is inflexible. For instance, it approximates Gaussians with only five intervals. These techniques compute only the probability of an event, not the entire posterior. Sweet et al. [3] estimates the probability of information leakage of a query. It ensures a sound over-estimation combining sampling with concolic execution. They support only discrete models.

8 Conclusion

AQUA is a new inference algorithm which works on general, real-world probabilistic programs with continuous distributions. By using quantization with symbolic inference, AQUA solved all benchmarks in less than 43s (median 1.35s). Our evaluation shows that AQUA is more accurate than approximate algorithms and supports programs that are out of reach of state-of-the-art exact inference tools.

Supplementary information. AQUA is avaliable at https://github.com/uiuc-arc/AQUA.

Acknowledgments. This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, and Facebook PhD Fellowship. The previous version of this work appeared in ATVA 2021 [41].

References

[1] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., et al.: Stan: A probabilistic programming language. JSTATSOFT **20**(2) (2016)

[2] Albarghouthi, A., D'Antoni, L., Drews, S., Nori, A.: Fairsquare: probabilistic verification of program fairness (OOPSLA) (2017)

[3] Sweet, I., Trilla, J.M.C., Scherrer, C., Hicks, M., Magill, S.: Whats the over/under? probabilistic bounds on information leakage. POST (2018)

[4] Pardo, R., Rafnsson, W., Probst, C.W., Wasowski, A.: Privug: using probabilistic programming for quantifying leakage in privacy risk analysis. In: European Symposium on Research in Computer Security, pp. 417–438 (2021). Springer

[5] Huang, Z., Wang, Z., Misailovic, S.: PSense: automatic sensitivity analysis for probabilistic programs. ATVA (2018)

[6] Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J.-P., Matheja, C.: A pre-expectation calculus for probabilistic sensitivity. POPL (2021)

[7] Gehr, T., Misailovic, S., Vechev, M.: PSI: Exact symbolic inference for probabilistic programs. CAV (2016)

[8] Narayanan, P., Carette, J., Romano, W., Shan, C.-c., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). FLOPS (2016)

[9] Saad, F.A., Rinard, M.C., Mansinghka, V.K.: SPPL: a probabilistic programming system with exact and scalable symbolic inference. PLDI (2021)

[10] Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. PLDI (2013)

[11] Dutta, S., Legunsen, O., Huang, Z., Misailovic, S.: Testing probabilistic programming systems. In: FSE (2018)

[12] Dutta, S., Zhang, W., Huang, Z., Misailovic, S.: Storm: program reduction for testing and debugging probabilistic programming systems. In: FSE (2019)

[13] Gorinova, M.I., Gordon, A.D., Sutton, C.: Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. POPL (2019)

[14] Gelman, A., Stern, H.S., Carlin, J.B., Dunson, D.B., Vehtari, A., Rubin, D.B.: Bayesian Data Analysis. Chapman and Hall/CRC, United States (2013)

[15] Bissiri, P., Holmes, C., Walker, S.: A general framework for updating belief distributions. Journal of the Royal Stat. Soc. Series B, Statistical Methodology **78**(5), 1103 (2016)

[16] Goodman, N., Tenenbaum, J.: Probabilistic Models of Cognition. probmods.org

[17] Nishihara, R., Minka, T., Tarlow, D.: Detecting parameter symmetries in probabilistic models. arXiv preprint arXiv:1312.5386 (2013)

[18] Neal, R.M.: Bayesian Learning for Neural Networks. Springer, Toronto (2012)

[19] https://github.com/stan-dev/example-models (2018)

[20] Wang, Y., Kucukelbir, A., Blei, D.M.: Robust probabilistic modeling with bayesian data reweighting. ICML (2017)

[21] Wang, C., Blei, D.M.: A general method for robust bayesian modeling. Bayesian Analysis **13**(4), 1159–1187 (2018)

[22] Laurel, J., Misailovic, S.: Continualization of probabilistic programs with correction. ESOP (2020)

[23] Goodman, N., Mansinghka, V., Roy, D.M.,

Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv preprint arXiv:1206.3255 (2012)

[24] Gilks, W.R., Thomas, A., Spiegelhalter, D.J.: A language and program for complex bayesian modelling. The Statistician (1994)

[25] Pfeffer, A.: Ibal: a probabilistic rational programming language. In: Proceedings of the 17th International Joint Conference on Artificial intelligence-Volume 1, pp. 733–740 (2001). Morgan Kaufmann Publishers Inc.

[26] Nori, A.V., Hur, C.-K., Rajamani, S.K., Samuel, S.: R2: An efficient mcmc sampler for probabilistic programs. In: AAAI (2014)

[27] Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)

[28] Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint 1404.0099 (2014)

[29] Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. Retrieved 2015/1/16, from http://dippl. org (2014)

[30] Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv (2016)

[31] Pyro. http://pyro.ai (2018)

[32] Minka, T., Winn, J.M., Guiver, J.P., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5. Microsoft Research Cambridge. http://research.microsoft.com/infernet (2013)

[33] Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. OOPSLA (2020)

[34] Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S., Visser, W.: Compositional solution space quantification for probabilistic software analysis. PLDI (2014)

[35] Luo, Y., Filieri, A., Zhou, Y.: Sympais: Symbolic parallel adaptive importance sampling for probabilistic program analysis. arXiv preprint arXiv:2010.05050 (2020)

[36] Shachter, R.D., D'Ambrosio, B., Del Favero, B.: Symbolic probabilistic inference in belief networks. In: AAAI, vol. 90, pp. 126–131 (1990)

[37] Chang, K.-C., Fung, R.: Symbolic probabilistic inference with both discrete and continuous variables. Systems, Man and Cybernetics, IEEE Transactions on **25**(6), 910–916 (1995)

[38] Moral, S., Rumí, R., Salmerón, A.: Mixtures of truncated exponentials in hybrid bayesian networks. In: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, pp. 156–167. Springer, Barcelona, Spain (2001)

[39] Shenoy, P.P., West, J.C.: Inference in hybrid bayesian networks using mixtures of polynomials. International Journal of Approximate Reasoning **52**(5) (2011)

[40] Sanner, S., Abbasnejad, E.: Symbolic variable elimination for discrete and continuous graphical models. In: AAAI (2012)

[41] Huang, Z., Dutta, S., Misailovic, S.: Aqua: Automated quantized inference for probabilistic programs. In: International Symposium on Automated Technology for Verification and Analysis, pp. 229–246 (2021). Springer