AURA: Precise Abstract Interpretation of Probabilistic Programs with Interval Data Uncertainty

Zixin Huang[1], Jacob Laurel[2], Saikat Dutta[3], and Sasa Misailovic[1]

[1] University of Illinois Urbana-Champaign, Urbana, IL, 61801, USA
[2] Georgia Institute of Technology, Atlanta, GA, 30332, USA
[3] Cornell University, Ithaca, NY, 14853, USA
{zixinh2,misailo}@illinois.edu, jlaurel6@gatech.edu✉, saikatd@cornell.edu

Abstract. We present AURA, a novel abstract interpretation for obtaining sound, precise bounds on the posterior distributions computed by probabilistic programs. AURA allows programmers to specify interval bounds that capture uncertainty or perturbations of the observed data. AURA abstractly computes the infinite set of posteriors that would result from performing inference for any possible data value in the specified perturbation range. AURA then certifies precise bounds on probabilistic queries over that set of posteriors. AURA's precision stems from a novel gradient-based optimization leveraging the structure of probabilistic programs. Our evaluation across 11 benchmarks with data perturbation shows that AURA improves precision by an order of magnitude (12.8x on average) over the interval-based abstract interpreter, within a run time of 3.1 seconds (geomean), using a GPU parallel implementation.

1 Introduction

Probabilistic programs (PPs) play an important role in many applications that make critical decisions such as security/privacy [13, 38, 47, 69], computer networks [17, 19, 67], analyzing hardware errors [15, 44], and pandemic modeling [6, 39]. In these applications, one often requires formal guarantees on the posterior probability distribution [65, 72]. Illustrating this critical need for formal assurances, prior work has shown Bayesian inference's fundamental susceptibility and brittleness to small perturbations [57], including adversarial perturbations to the observed data [24, 35, 57, 77]. Moreover, while adversarial attacks on datasets have been studied for other ML models, verifying robustness to data perturbations has been far less explored in probabilistic programming.

Verifying robustness of probabilistic programs to dataset perturbations encounters several core issues. First, these programs often involve many continuous distributions, which require symbolically evaluating possibly intractable integrals and highly non-linear probability densities. Second, for verification, one needs the analysis to be scalable *and* precise. Third, to reason about robustness to data perturbations, one must obtain guarantees on a *set* of possible posterior

distributions. Unlike bounds for a single posterior, the bounds must now enclose *any* possible posterior that could result from inference on perturbed data. Using these bounds, we can verify that an adversary making small perturbations to one or more observed data points can never change the posterior probability of an event by an unacceptable amount.

Our Work. AURA is a novel abstract interpretation of probabilistic programs that produces sound and precise bounds on the inferred posterior distributions. By evaluating a probabilistic program abstractly, the bounds AURA computes can be used to verify assertions over a program's posterior. Additionally, AURA is the first program analysis to efficiently compute precise and sound bounds on an *infinite set of possible posterior distributions* when the observed data is specified as bounds by the user. Lastly, AURA scales to data sizes that are an order of magnitude greater than the size handled in prior work [5, 20, 37].

AURA's key technical contribution is to reduce abstract interpretation to gradient-based optimization by leveraging a distribution-shape pattern commonly found in continuous PPs. Thus AURA constructs sound and optimally precise abstract transformers over the interval abstract domain. AURA's transformers are applicable to complex subprograms or even the entire program when their distributions are *pseudoconcave*. Pseudoconcavity [46] is a relaxation of the familiar notion of concavity, and many popular continuous distributions (e.g., Gaussian, Uniform, Exponential) and expressions over them satisfy this property. This insight helps AURA's abstract transformers achieve much higher precision than composing standard interval arithmetic operations for subexpressions. Moreover, we show how our abstraction can also be combined and composed with standard interval transformers to maintain the generality needed to analyze more complicated programs that may not be end-to-end pseudoconcave.

In addition to precise and scalable abstract transformers, AURA allows programmers to specify interval bounds on the observed data which AURA then propagates through the program. These results can be used to certify bounds on probabilistic queries in order to bound the probability of an event. These bounds hold for *all* possible posterior distribution that could result from data perturbations. Thus AURA uses abstract interpretation to verify properties for infinitely many probabilistic programs simultaneously. AURA also makes integration of the over- and under-approximated densities tractable and efficient during marginalization, normalization, or expectation calculation. Figures 1-3 illustrate AURA's abstraction for a *single posterior* and a *set of posteriors*.

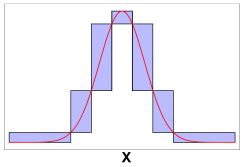


Fig. 1: Bound on a Single Posterior (no perturb.)

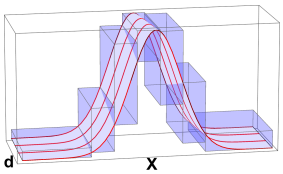


Fig. 2: Analysis of Data Perturbation (on 'd' axis)

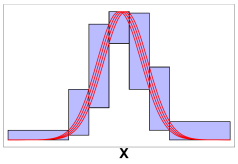


Fig. 3: Bound on the Set of Posteriors

Contributions. The paper presents the following core technical contributions:

- **Problem Formulation**. We introduce a novel formulation of the problem of certifying bounds on a set of posterior distributions resulting from (bounded) data perturbations.
- **Gradient-based Optimization**. We design a novel algorithm for obtaining precise abstract transformers for probabilistic programs that uses gradient-based optimization for tractably and optimally solving for precise bounds on the posterior distributions.
- **Soundness.** We show that our abstract interpretation can guarantee soundness for a broad class of probabilistic expressions and programs whose posteriors satisfy concavity or pseudoconcavity at each interval.
- **Evaluation**. We integrate AURA with PyTorch to leverage GPUs. Our evaluation of AURA across 11 programs with data perturbations shows that AURA improves precision by an order of magnitude (12.8x) over the interval-based abstract interpreter, within 3.1 seconds (geomean) on GPU. It also efficiently and precisely computes the probability of queries under perturbations.
  AURA is available at `https://github.com/uiuc-arc/AURA` .

2 Example

Probabilistic programming serves as a popular paradigm for encoding Bayesian probability models concisely as programs [26]. In addition, the probabilistic programming system automates Bayesian inference. Thus, the programmer only specifies the source program while the underlying inference details are abstracted away by the language. However, one may desire formal guarantees about the inference results. Hence our work aims to provide these guarantees with AURA.

Before describing AURA's approach, we first present our example probabilistic program, P in Figure 4. This program infers the ground braking force. The latent parameter x represents the braking force exerted on the vehicle, while data[i] stores the observed deceleration (m/s^2) under experimental conditions. In this model, an engineer uses the observed variable data[i] to infer the value of latent x. As an assumption, they model x's prior distribution as a uniform(0,100). The model assumes the observed variables are normally distributed.

```
1 data = [....] # numerical values
2 x ~ uniform(0, 100)
3
4 for i in 1..100:
5    observe(
6      normal(0.1*x+1, 1),
7      data[i]
8    )
```

Fig. 4: Example Code.

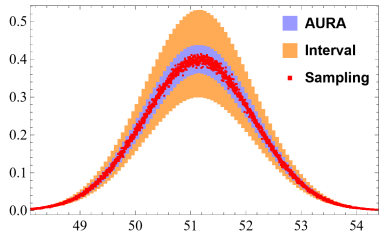


Fig. 5: Posterior x Result.

AURA's formalism allows one to express the scenario where the observed data values can suffer perturbations. We assume that multiple data observations can

encounter sensor defects, potentially altering an observed value data[i] by up to 0.05. Here, we analyze the scenario where five observed points are simultaneously perturbed from sensor defects.

AURA's bounds are shown as blue boxes in Figure 5 and soundly enclose the set of *all* possible posterior distributions that could arise from inference with perturbed data. The red dots in this figure represent a sampled histogram of x, based on different concrete observed data values data[i] within a specified interval $d^\sharp$, showing that AURA's bounds are precise. The orange boxes illustrate posterior bounds obtained by an analysis with standard interval abstract transformers and are much less precise than AURA at enclosing the sampled histograms. The reason for this trend is that composing over-approximate interval abstract transformers compounds imprecision. In contrast, AURA's abstraction improves precision by abstracting the entire unnormalized posterior at once. AURA can prove sound posterior bounds within just 0.115 seconds when using 200 partitions (i.e., subintervals) to represent the posterior.

Bounds on Probabilities of Queries. The certified bounds on the set of normalized posteriors can then be used for various queries. For instance, engineers may need to guarantee that the braking force x should not fall below a certain safety threshold, say 50 Newtons. Hence AURA's posterior bounds can be used to bound the posterior's probability of the query $Q \equiv x < 50$.

AURA provides a probability range of [0.11, 0.14] for the braking force falling below a critical threshold, which is twice as precise as the [0.09, 0.17] range obtained from interval analysis. AURA's narrower bounds indicate a smaller effect from data perturbations, namely a better *robustness*, which is the ability to infer reliable results even in the face of data perturbations.

3 Preliminaries

Language. We present our probabilistic programming language in Fig. 6. Our syntax is similar to the syntax of Stan [21], separating the block with priors over the latent variables M and the observations in the data block D. The programmer can also specify models where the parameters of one distribution are an arithmetic expression E of other latent parameters. The priors in M must be continuous with compact (truncated) support. Our data perturbation formalism requires the observed distribution be continuous. While the implementation optionally permits discrete observed variables, this setting is limited to only regular inference and not analysis of data perturbations.

$$
\begin{aligned}
P &::= M \mid M; D \\
M &::= \texttt{x}_i \sim \mathit{Dist} \mid M; M \mid \texttt{if flip}(p)\ M_1\ \texttt{else}\ M_2 \\
&\quad \mid \texttt{for j=}n_1\ \texttt{to}\ n_2\ \texttt{do}\ M \mid \texttt{let x}_i = E\ \texttt{in}\ M \\
D &::= \mathit{observe}(\mathit{Dist}, d_i) \mid D; D \mid \texttt{for j=}n_1\ \texttt{to}\ n_2\ \texttt{do}\ D \\
E &::= \texttt{x}_j \mid E + E \mid E - E \mid E * E \mid E/E \mid c \in \mathbb{R} \\
\mathit{Dist} &::= \mathit{dist}(E_1, \ldots, E_N),\ \mathit{dist} \in \{\mathit{uniform}, \ldots\}
\end{aligned}
$$

Fig. 6: AURA Language

In the $\texttt{flip}(p)$ primitive, p is a fixed constant between 0 and 1. The language is first-order (no recursion), however one can encode a broad class of popular

probabilistic models such as Linear and Logistic Regressions, Time-Series Models, Hierarchical Bayesian models, and others. This language is constrained for the purpose of identifying and optimally analyzing pseudoconcave programs, however our language could be used to define other subprograms in general PPLs such as for nested inference (example in Appendix A.4 [36]).

Concavity and Convexity. A core component of AURA's abstraction relies upon concavity and concavity-like properties of posterior distributions. A set $\mathcal{X} \subseteq \mathbb{R}^d$ is convex if for all $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$, then $\alpha x_1 + (1-\alpha)x_2 \in \mathcal{X}$. A function f is ***concave*** over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following inequality holds: $f(\alpha x_1 + (1-\alpha)x_2) \geq \alpha f(x_1) + (1-\alpha)f(x_2)$. The **Concave** functions remain closed under summation, which will be important for AURA's analysis. Additionally, we will see that AURA's analysis still supports weaker notions of concavity, which we describe next.

Definition 1. ***Quasiconcavity***. *A function f is quasiconcave over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following holds: $f(\alpha x_1 + (1-\alpha)x_2) \geq \min(f(x_1), f(x_2))$.*

Definition 2. ***Log-Concavity***. *A non-negative function f is logarithmically-concave over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following holds: $f(\alpha x_1 + (1-\alpha)x_2) \geq f(x_1)^{\alpha} \cdot f(x_2)^{1-\alpha}$.*

For a strictly positive function f (e.g., a probability density), the following implication holds: **Log-Concave** $f \implies$ **Concave** $\log(f)$. Many common probability densities are **Log-concave** (Appendix C; Table 6) and the **Log-concave** functions are closed under multiplication. Additionally, quasiconcavity is useful for defining pseudoconcavity.

Definition 3. ***Pseudoconcavity***. *A function $f(x)$ is pseudoconcave if and only if $f(x)$ is quasiconcave and for any x^*, $\nabla f(x^*) = 0 \implies x^* = arg\max f(x)$.*

Pseudoconcavity is similar to **Quasiconcavity**, however any stationary point of a pseudoconcave function is *necessarily* an optimum. Further, any quasiconcave function whose gradient is never zero is Pseudoconcave. Figure 7 shows example functions. One may notice the flat plateau of the quasiconcave function consists of stationary points which are *not* optimal. Hence in Section 6 we leverage pseudoconcavity to ensure a gradient ascent procedure does not become trapped in local extrema.

Fig. 7: Illustration of Different Notions of Concavity

Abstract Interpretation. AURA's analysis builds upon abstract interpretation [10] with the interval domain. The interval abstract domain was chosen because it has found widespread success in many program analysis tasks [42, 43, 66] due to its scalability. In the interval domain each variable is abstracted by an interval $[a, b] \in \mathbb{IR}$ where a is the lower bound and b is the upper bound and $a \leq b$. The set of all m-dimensional intervals will be denoted as $\mathbb{IR}^m$, and a given

$$[\![P]\!](x,d) = [\![M;D]\!](x,d) \qquad [\![M;D]\!](x,d) = [\![M]\!](x) \cdot [\![D]\!](x,d)$$

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!](x,d) = p[\![P_1]\!](x,d) + (1-p)[\![P_2]\!](x,d) \qquad [\![M_1;M_2]\!](x) = [\![M_1]\!](x) \cdot [\![M_2]\!](x)$$

$$[\![D_1;D_2]\!](x,d) = [\![D_1]\!](x,d) \cdot [\![D_2]\!](x,d)$$

$$[\![observe(Dist,d_i)]\!](x,d) = [\![Dist]\!](x,d) \circ d[i] \qquad [\![\mathtt{x}_i \sim Dist]\!](x,d) = [\![Dist]\!](x,d) \circ x[i]$$

$$[\![E_1 + E_2]\!](x,d) = [\![E_1]\!](x,d) + [\![E_2]\!](x,d) \qquad [\![E_1 - E_2]\!](x,d) = [\![E_1]\!](x,d) - [\![E_2]\!](x,d)$$

$$[\![E_1 * E_2]\!](x,d) = [\![E_1]\!](x,d) * [\![E_2]\!](x,d) \qquad [\![E_1/E_2]\!](x,d) \overset{\neq 0}{=} [\![E_1]\!](x,d)/[\![E_2]\!](x,d)$$

$$[\![\mathtt{x}_j]\!](x,d) = x[j] \qquad [\![cE]\!](x,d) = c \cdot [\![E]\!](x,d)$$

$$[\![dist(E_1,\ldots,E_N)]\!](x,d) = p_{dist}\left(u; [\![E_1]\!](x,d),\ldots,[\![E_N]\!](x,d)\right),\ dist \in \{uniform,\ \ldots\}$$

Fig. 8: Key Rules of Unnormalized Concrete Semantics.

element will be denoted as $x^\sharp \in \mathbb{IR}^m$ where $x^\sharp[i] = [a_i, b_i]$. The concretization $\gamma : \mathbb{IR}^m \to \mathcal{P}(\mathbb{R}^m)$ of a multidimensional interval is just the set of all points contained in that interval hence: $\gamma(x^\sharp) = \{(x_1, ..., x_m) : \forall i \in \{1, ..., m\}, a_i \leq x_i \leq b_i \text{ where } [a_i, b_i] = x^\sharp[i]\}$. For the interval domain, the basic abstract transformers are the standard interval arithmetic operations (we denote them as $+^\sharp$ and $.^\sharp$), which can easily be composed [50], however we will later see how AURA obtains precise abstract transformers by solving optimization problems.

4 Concrete Semantics

We now formalize our concrete semantics of probabilistic programs. The semantic interpretation of a probabilistic program is the normalized posterior distribution, which corresponds to the unnormalized likelihood defined by the statements in the probabilistic program divided by a normalizing constant.

Preliminary Transformation. To simplify the formalism description, we do three source-to-source transformations. The first is moving conditionals upward hence the production for P becomes $P ::= M \mid M; D \mid \texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2$. To move conditionals upward, it includes the code before and after the conditional into the branches. The second transformation is unrolling the `for` loops. Hence the productions for M becomes $M ::= \mathtt{x}_i \sim Dist \mid M; M$ and the production for D becomes $D ::= observe(Dist, d_i) \mid D; D$ since loops can be unrolled to sequencing: $M; M$ and $D; D$. The third transformation replaces all occurrences of (fresh) variables introduced by the `let` bindings with their original expressions (using capture-avoiding substitution), so that all the expressions and subexpressions only contain variables corresponding to sampled distributions, $\mathtt{x}_k$.

Unnormalized Concrete Semantics. We first formalize the concrete semantics of probabilistic programs in terms of their *unnormalized* likelihood. Because likelihoods are just density functions which map observations to scores, we formalize this mapping using a score-based functional semantics.

In the score-based semantics, the interpretation $[\![\cdot]\!]$ of a probabilistic program P is a function that scores the likelihood of a given trace $x \in \mathbb{R}^m$ (where m is the number of latents) and given data observations $d \in \mathbb{R}^n$. Hence $[\![\cdot]\!]$ is a

function of both x and d. The semantic signature for interpreting a program P is $[\![P]\!] : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$, and $[\![P]\!](x, d) : \mathbb{R}$. The full semantics are given in Fig. 8. For the distribution rule $[\![dist(E_1, \ldots, E_N)]\!](x, d)$, we take the probability density function (PDF) of that distribution, denoted as p_{dist}. Each statement multiplies the probability score by the corresponding probability density function of either the latent parameter or the observed data sample (see $[\![M; M]\!]$ and $[\![D; D]\!]$). For branches, we take the linear combination of the density functions.

Unnormalized Log-Likelihood Semantics, $[\![P]\!]_{\log}(x, d)$. For reasons of numerical stability it is often helpful to work with the logarithm of the likelihood instead of the original likelihood itself. Thus having defined the Unnormalized Concrete Semantics, we can now define the *log-likelihood* semantics by simply taking a logarithm. In particular: $[\![P]\!]_{\log}(x, d) = \log([\![P]\!](x, d))$ We can easily convert back to the original semantics: $[\![P]\!](x, d) = \exp([\![P]\!]_{\log}(x, d))$.

Normalized Concrete Semantics. We formalize the notion of a *normalized* probabilistic program, whose semantics is denoted $[\![\cdot]\!]_n$ and given by:

$$[\![P]\!]_n(x, d) = [\![P]\!](x, d) \,/ \int_x [\![P]\!](x, d)dx,$$

where $\int_x [\![P]\!](x, d)dx$ is the normalizing constant. Generally, computing the normalizing constant and hence $[\![P]\!]_n$ can be intractable.

5 Abstract Semantics for Data Perturbation

We now define an abstract semantics for over-approximating *entire sets* of posterior distributions, which intuitively encode *all* posteriors obtainable when the data d could be perturbed. However if one wishes to only bound a *single* posterior, they can still use AURA, the data interval will just be degenerate.

5.1 Unnormalized Abstract Semantics for Data Perturbation

In the data perturbation setting, the observed dataset is given by some interval, $d^\sharp \in \mathbb{IR}^n$. Thus for abstractly interpreting program P we have the signature $[\![P]\!]^\sharp(x^\sharp, d^\sharp) : \mathbb{IR}^m \times \mathbb{IR}^n \to \mathbb{IR}$. In essence, we prove guaranteed bounds on all possible posteriors obtained after a bounded (adversarial) perturbation on the data. Hence AURA can analyze and verify properties for an infinite number of probabilistic programs, a task which has not been studied in any prior work.

Optimization. The core idea of AURA is to compute precise lower and upper bounds by respectively solving minimization and maximization problems. Hence instead of computing lower and upper bounds with interval arithmetic, AURA reduces abstract interpretation to continuous optimization. This optimization formulation is defined as:

$$[\![P]\!]^\sharp(x^\sharp, d^\sharp) = [l, u], \quad \text{where}$$

$$l = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x, d), \qquad \text{and} \qquad u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d).$$

One can think of this formulation as defining an abstract transformer tailored for the *entire* program's (or subprogram's) expression instead of defining abstract transformers for individual primitive operations (as interval arithmetic does). Having an abstract transformer defined at this higher level of granularity allows AURA to improve precision greatly over interval arithmetic – and as we will show in Section 6, when the unnormalized likelihood $[\![P]\!](x, d)$ has a pseudo-concave structure, we can solve this optimization problem tractably.

However one can always use any $[l^*, u^*]$ where $l^* \leq l$ and $u \leq u^*$ as sound bounds. Hence, for programs which *lack* the necessary pseudoconcavity properties, one can always fallback to interval arithmetic to abstractly interpret the semantics of Fig. 8. This insight gives us the flexibility to analyze a (pseudo-concave) subprogram within P using AURA's precise optimization approach, while using interval arithmetic to bound other sub-expressions which may not be pseudoconcave and then compose the results. We can now state the following soundness result (the proofs are in Section 5.3):

Theorem 1. *The Unnormalized Abstract Semantics for data perturbation over-approximate the Unnormalized Concrete Semantics for fixed data observations. Equivalently for arbitrary program P, dataset $d^\sharp \in \mathbb{IR}^n$, and interval $x^\sharp \in \mathbb{IR}^m$, we have $\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![P]\!]^\sharp(x^\sharp, d^\sharp))$.*

Abstract Log-likelihood Semantics for Data Perturbations. We can take logarithmic transformations of the abstract unnormalized semantics, to define $[\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp) = [l', u']$, where $l' = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!]_{\log}(x, d)$, and also $u' = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!]_{\log}(x, d)$. Then, $[\![P]\!]^\sharp(x^\sharp, d^\sharp) = \exp([\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp))$.

5.2 Normalized Abstract Semantics for Data Perturbation

One of the most challenging parts of computing the abstract normalized semantics is performing the abstract integration $\int^\sharp$. The key intuition is that we partition the support of $[\![P]\!]$ into disjoint intervals and compute an interval bound of the unnormalized posterior over each partition. These bounds form lower and upper Riemann sums, which bound the value of the integral, and thus bound the integrating constant. Lastly, interval division of the previous bounds on the unnormalized posterior by the bounds on the integrating constant ultimately yields bounds on the normalized posterior, which we denote as $[\![P]\!]^\sharp_n$.

Definition 4 (Abstract Integral with Data Perturbation). *We let each $x_i^\sharp$ represent a multi-dimensional interval in $\mathbb{IR}^m$, such that* $\text{support}([\![P]\!]) = \cup_{i=1}^k \gamma(x_i^\sharp)$. *Thus each $x_i^\sharp$ is a subset of the posterior distribution's support. Each $x_i^\sharp$ can be denoted as a Cartesian product (denoted as $\otimes$) of the intervals as $x_i^\sharp = \otimes_{j=1}^m [x_{l_{ij}}, x_{u_{ij}}]$. Here m represents the dimension of the latent variables, and for each dimension $j \in \{1, ..., m\}$, the interval $[x_{l_{ij}}, x_{u_{ij}}] \in \mathbb{IR}$ corresponds to that dimension. The volume (or Lebesgue measure) of each $x_i^\sharp$ in the partition*

is $Vol(x_i^\sharp) = \prod_{j=1}^{k}(x_{u_{ij}} - x_{l_{ij}})$. *The abstract integral with data perturbation is:*

$$\int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)\, dx = \left[\sum_{i=1}^{k} l_i \cdot Vol(x_i^\sharp), \sum_{i=1}^{k} u_i \cdot Vol(x_i^\sharp)\right]$$

The bounds $\mathrm{Vol}(x_i^\sharp) = \prod_{j=1}^{m}(x_{u_{ij}} - x_{l_{ij}})$ and $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp) = [l_i, u_i]$ are defined in Section 5, however bounds obtained from *any* sound abstract transformer would also be valid. For distributions with compact support, there will be finitely many (non-zero) terms in the summation. We will see in Section 7 how this same idea can be used to abstractly integrate $[\![P]\!]_n^\sharp$ to formally bound probabilities of (measurable) events using the posterior bounds computed by $[\![P]\!]_n^\sharp$.

Lemma 1. *[Soundness of abstract integration for data perturbations] Given Theorem 1 and Definition 4, it follows that:*

$$\int [\![P]\!](x, d)\, dx \in \gamma\left(\int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)\, dx\right).$$

We now formally define the normalized abstract semantics for programs where the support has been partitioned as $\cup_{j=1}^{k} x_j^\sharp$ and where $x_i^\sharp \in \mathbb{IR}^m$ is the partition of interest, Here $\div^\sharp$ represents interval division.

$$[\![P]\!]_n^\sharp(x_i^\sharp, d^\sharp) = \frac{[\![P]\!]^\sharp(x_i^\sharp, d^\sharp)}{\int^\sharp [\![P]\!]^\sharp(\cup_{j=1}^{k} x_j^\sharp, d^\sharp)}^\sharp$$

Theorem 2. *The Normalized Abstract Semantics for data perturbation over-approximates the Normalized Concrete Semantics for sets of data observations. Formally, for a program* P, *dataset* $d^\sharp \in \mathbb{IR}^n$, *and interval* $x^\sharp \in \mathbb{IR}^m$, *we have:*

$$\{[\![P]\!]_n(x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![P]\!]_n^\sharp(x^\sharp, d^\sharp)).$$

5.3 Soundness Proofs

Proof of Theorem 1. Since $[\![P]\!]^\sharp(x^\sharp, d^\sharp) = [l, u]$ is a 1D interval in $\mathbb{IR}$, we have to show that

1. $l \leq \inf\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\}$ and
2. $u \geq \sup\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\}$.

However, by definition, $l = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x, d)$ and

$$\min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x, d) = \inf\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\}$$

Similarly, $u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d)$ and

$$\max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d) = \sup\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\}$$

Hence soundness follows (by construction). □

Proof of Lemma 1. To prove $\int [\![P]\!](x,d)\,dx \in \gamma\left(\int^\sharp [\![P]\!]^\sharp(x^\sharp,d^\sharp)\,dx\right)$, we need to show that $\sum_{i=1}^n l_i \cdot \text{Vol}(x_i^\sharp) \leq \int [\![P]\!](x,d)dx \leq \sum_{i=1}^n u_i \cdot \text{Vol}(x_i^\sharp)$

However the left hand side is a lower Riemann sum which is always less than the true integral, and likewise the right hand side is an upper Riemann sum which is always greater than the true integral. □

Proof of Theorem 2. Since the unnormalized bounds are sound from Theorem 1, the bounds on the normalizing constant are sound from Lemma 1 and interval division is sound, the normalized bounds are sound too. □

6 AURA Optimization Algorithm

Having defined the abstract semantics $[\![P]\!]^\sharp(x^\sharp,d^\sharp)$ in terms of interval bounds where the lower and upper bounds come from solutions to optimization problems, we now describe how AURA can precisely solve these optimization problems.

Pseudoconcave Probabilistic Programs. A probabilistic program P is *pseudoconcave* if the unnormalized density function defined by $[\![P]\!](x,d)$ is a pseudoconcave function of both x and d (a condition satisfied by many distributions; see Appendix C). We also have the following implications:

$$[\![P]\!](x,d) \textbf{ Log-Concave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Concave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Pseudoconcave}$$

$$[\![P]\!](x,d) \textbf{ Pseudoconcave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Pseudoconcave}$$

We choose **Pseudoconcavity**, because to the best of our knowledge it is the weakest condition that ensures the lower and upper bounds from gradient-based optimizations remain sound. Pseudoconcave functions have derivatives which exist everywhere except a measure zero set. Hence, we can use these derivatives for Gradient Ascent to solve the optimization problems of Section 5.1.

6.1 Computing Lower Bounds with AURA

The first step in computing the abstract semantics $[\![P]\!]^\sharp(x^\sharp,d^\sharp)$ needed to soundly bound posteriors involves computing the interval's lower bound. In the case of data perturbations one must compute $l = \min_{d\in\gamma(d^\sharp)} \min_{x\in\gamma(x^\sharp)} [\![P]\!](x,d)$. However, because of the pseudoconcavity requirements on $[\![P]\!](x,d)$ this minimization problem becomes tractable. In particular we only have to check the corner points of $x^\sharp$ and $d^\sharp$, denoted as $Corners$. For numerical stability, we use logarithms, hence we can solve the optimization problem by computing:

$$l = \exp\left(\min_{d\in Corners(d^\sharp)} \min_{x\in Corners(x^\sharp)} [\![P]\!]_{\log}(x,d)\right) \tag{1}$$

Theorem 3. *(Soundness) The lower bounds l computed above in Eq. 1 are sound when the log-likelihood $[\![P]\!]_{\log}(x,d)$ is pseudoconcave.*

Proof. (Sketch) Any pseudoconcave function is also quasiconcave, and quasiconcave functions over compact convex sets are minimized at corner points [45]. □

A key benefit of using the Interval domain instead of the Polyhedral or Zonotope [22, 23] domains is that checking extremal points of intervals is more scalable compared to checking extremal points of polyhedra or zonotopes. This lower bound is not just sound, it is **optimal**. We state this result below:

The lower bound computed in Eq. 1 is optimal – *the most precise bound possible*. Because $[\![P]\!]_{\log}(x, d)$ is continuous and the interval $d^\sharp \times x^\sharp$ is compact, the minimum will be attained on that interval. Since exp is monotonically increasing, the minimizer of $[\![P]\!]_{\log}(x, d)$ is also the minimizer of $\exp([\![P]\!]_{\log}(x, d))$.

6.2 Computing Upper Bounds with AURA

Similarly, to obtain sound enclosures, AURA computes the upper bound by solving the optimization problem $u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d)$. Our key technical insight is that this maximization can be solved directly by performing *Projected Gradient Ascent* on the log likelihood, $[\![P]\!]_{\log}(x, d)$. Our implementation uses automatic differentiation to efficiently compute the gradients. Further, since $x^\sharp$ and $d^\sharp$ define (multi-dimensional) intervals, they are convex sets, hence the constraints of this optimization problem are convex.

Definition 5. ***Projected Gradient Ascent.*** *Given a differentiable function $f(x) : \mathcal{X} \subset \mathbb{R}^m \to \mathbb{R}$, one iteratively computes:*

$$x_{n+1} = \Pi_{\mathcal{X}}\big(x_n + \eta \nabla_x f(x_n)\big)$$

with learning rate $\eta \in \mathbb{R}_{>0}$ until convergence where $\|x_{n+1} - x_n\| \leq \epsilon$. Here $\Pi_{\mathcal{X}}$ is the projection operator that takes a x_{n+1} that may lie outside $\mathcal{X}$, and returns the closest point inside $\mathcal{X}$. If the constraints are intervals: $\mathcal{X} = \otimes_{i=1}^m [l_i, u_i] \subseteq \mathbb{IR}^m$, the projection is $\Pi_{\mathcal{X}}(x) = \otimes_{i=1}^m \Pi_{\mathcal{X}}(x[i])$ where:

$$\Pi_{\mathcal{X}}(x[i]) = \begin{cases} l_i & \text{if } \; l_i > x[i]; \;\; \text{or} \\ x[i] & \text{if } \; x[i] \in [l_i, u_i]; \; \text{or} \\ u_i & \text{if } \; u_i < x[i] \end{cases} \tag{2}$$

AURA Gradient Optimization. AURA will run the following Gradient Ascent computations for the function $[\![P]\!]_{\log}(x, d) : \gamma(x^\sharp) \times \gamma(d^\sharp) \to \mathbb{R}_{\geq 0}$:

$$(x_{n+1}, d_{n+1}) = \Pi_{x^\sharp; d^\sharp}\big((x_n, d_n) + \eta \nabla [\![P]\!]_{\log}(x_n, d_n)\big) \tag{3}$$

until $x_{n+1} = x_n$ and $d_{n+1} = d_n$. The learning rate must satisfy $\eta \leq \frac{1}{\|\nabla [\![P]\!]_{\log}(x_n, d_n)\|}$ to ensure convergence. We further discuss the selection of the learning rate in Section 8. This optimization problem is constrained because the latent parameters x come from distributions with compact support (e.g., uniform). A key benefit of using the interval domain is that the projection function $\Pi_{x^\sharp; d^\sharp}$ in Eq. 3 reduces to the (efficiently computable) projection in Eq. 2 since $x^\sharp \times d^\sharp$ is just a multi-dimensional interval. Upon computing the $x_{n+1} = x_n$ and $d_{n+1} = d_n$ that the Projected Gradient Ascent converges to, we exponentiate the result to get:

$$u = \exp([\![P]\!]_{\log}(x_{n+1}, d_{n+1})) \tag{4}$$

Theorem 4. *(Soundness) The upper bounds u computed in Eq. 4 are sound when the log-likelihood $[\![P]\!]_{\log}(x, d)$ is pseudoconcave.*

Proof. (sketch) Projected gradient ascent/descent is guaranteed to converge to the *true* maxima (instead of a local one) for pseudoconcave/pseudoconvex functions [12, 14, 30]. Moreover, when projected gradient ascent applied to a pseudoconcave function finds a fixed point $x_{n+1} = x_n$, $d_{n+1} = d_n$, such a fixed point is guaranteed to the be the true optima ([14] Theorem 4.2) □

Corollary 1. *(Optimality) The upper bound computed in Eq. 4 is the most precise bound possible.*

Proof. (sketch) Since $[\![P]\!]_{\log}(x, d)$ is continuous and $d^\sharp \times x^\sharp$ is compact, the maximum is attained on that interval. Since exp is monotonically increasing, the maximizer of $[\![P]\!]_{\log}(x, d)$ is the maximizer of $\exp([\![P]\!]_{\log}(x, d))$ □

AURA runs the gradient ascent until a fixed point ($x_{n+1} = x_n$) is found. Alternatively, AURA can run the gradient ascent for fewer iterations (before hitting a fixed point), and add an error bound to the result to account for the distance to the true optimum value. For instance, prior works [14, 28] guaranteed for a pseudo-concave function $f(x) : \mathcal{X} \subset \mathbb{R}^m \to \mathbb{R}$, after T iterations of projected gradient ascent, the error between the true maximum and the current value will not exceed $E = \sqrt{\kappa^2 \|x_0 - x^*\|^2 / T}$, where x_0 denotes the starting point, x^* denotes the true maximum, and κ denotes the local Lipschitz constant over $\mathcal{X}$ that can be computed by AURA. Hence, even without a sufficient number of iterations to reach the fixed point, when adding tiny bound E, AURA still gives bounds which are sound. This strategy also applies if one wishes to account for numerical error, though prior work [2] shows that gradient ascent is already robust to floating point roundoff. Lastly, because of the optimality of the lower and upper bounds, *AURA achieves the most precise abstract transformer of pseudoconcave functions for the interval domain.*

6.3 Beyond Pseudoconcavity: Compositionality and Scaling

Supporting Branch Statements and Mixture Models. AURA supports mixture distributions that contain branches which can cause the likelihoods to no longer be pseudoconcave. The idea is that even if the entire posterior is multimodal and not pseudoconcave, each *component* when viewed in isolation could be pseudoconcave. For instance, we can define the abstraction of a branch as:

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]^\sharp(x^\sharp, d^\sharp) = p \cdot^\sharp [\![P_1]\!]^\sharp(x^\sharp, d^\sharp) +^\sharp (1-p) \cdot^\sharp [\![P_2]\!]^\sharp(x^\sharp, d^\sharp) \quad (5)$$

Thus by applying AURA's abstract interpreter, $[\![\cdot]\!]^\sharp$, to each component (which are pseudoconcave) and combining the results with standard interval arithmetic, we can still obtain sound posterior bounds. We present the proof in Appendix A.1.

Pseudoconcave Subexpressions when Programs are not Pseudoconcave. We can generalize the previous case and fallback to interval arithmetic for

any of the subexpressions in Fig. 8. Thus, even if the program P lacks a pseudoconcave posterior, we can compute optimized bounds on the largest subexpressions which *are* pseudoconcave, and then use standard interval arithmetic for the rest. Hence, Sections 6.1 and 6.2 provide sound interval domain abstract transformers for *any* pseudoconcave function, including subexpressions within P. We state this property formally and prove it in Appendix A.2.

Optimizing Programs with High-Dimensional Latents. A common method for writing robust models is to add many *local* latent parameters [71, 76], i.e., each observation depends on a fresh latent variable. Figure 9 gives an example model capturing this common pattern.

```
1 d = [1.3, 2.1, ... ]  #n data points
2 u ∼ normal(...)
        #global latent variable
3 for i in range(n):
4   v_i ∼ normal(u,1)
          #local latent variables
5   observe(d[i], normal(v_i,1))
```

Fig. 9: Robust Model with $n+1$ Latent Parameters

Line 5 encodes that each datapoint has its own i.i.d latent v_i, hence why there are $n+1$ latent variables. Given this pattern, we let u represent the *global* latent and v_i represent the *local* latents. Thus the latent vector is $x = (u, v_1, ..., v_n)$ which means that naively, we would solve an $\mathcal{O}(n)$-dimensional optimization problem. However, AURA automatically reduces this problem to n easy $\mathcal{O}(1)$-dimensional subproblems since each local latent's respective optimization subproblem is $\mathcal{O}(1)$ dimension and they can be solved in parallel (full proof in Appendix A.3).

7 AURA Verification Algorithm

Having described AURA's optimization routine, we next describe how to use this routine for end-to-end robustness verification of probabilistic programs. In particular, Equations 1 and 4 provide a strategy to solve the optimization problems of Section 5.1, thus giving a way to compute $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$. Hence we combine these insights together to give the full algorithm for AURA's abstract interpretation of normalized posterior distributions of probabilistic programs. The entire procedure is shown in Algorithm 1.

Partitioning (Splits). As input, AURA requires a partition of the support of the latent variables. The core intuition is that we partition the support of P's distribution into disjoint interval "*splits*". If $x^\sharp$ is the interval containing the entire (compact) support of P's (unnormalized) likelihood, then we take partitions $x_i^\sharp$ such that $x^\sharp = \bigcup_i x_i^\sharp$. We use an equal-area splitting strategy but support other strategies (see Section 8). For each split $x_i^\sharp$, AURA computes $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$ in lines 1-2. A key efficiency insight (inspired by DNN verification [80]) is that each split $x_i^\sharp$ can be processed in parallel on a GPU to compute $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$.

Abstract Dataset. Another input to AURA is the abstract dataset $d^\sharp$ representing the observed data points's bounded range. A novelty of AURA is that we can verify bounds for both a single posterior, *and* an infinite set of posteri-

Algorithm 1 AURA Core Verification Algorithm

Input: Probabilistic Program P; Abstract dataset $d^\sharp$; The partition of P's support $\bigcup_{i=1}^{m} x_i^\sharp$ (from implementation heuristic); The query Q (optional)

Output: (1) Posterior Bounds $[\![P]\!]_n^\sharp(x, d^\sharp)$;
(2) Query Bounds $[Q_l, Q_u]$ where $Pr(Q) \in [Q_l, Q_u]$ (only if Q provided)

1: **for** $x_i^\sharp$ in splits **do**
2: $l_i, u_i = [\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$ ▷ Unnormalized posterior bounds computed ▷ in Sections 6.1-6.3 (data parallel)
3: **end for**
4: $[c_l, c_u] = \int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)dx$ ▷ Normalizing Constant bound computed ▷ in Def. 4 (reduction)
5: **for** $x_i^\sharp$ in splits **do**
6: $[\![P]\!]_n^\sharp(x_i^\sharp, d^\sharp) = [l_i, u_i] \div^\sharp [c_l, c_u]$ ▷ Normalization (data parallel)
7: **end for**
8: **if** Query **then**
9: $[Q_l, Q_u] = \int^\sharp \mathbf{1}_Q \cdot [\![P]\!]_n^\sharp(x^\sharp, d^\sharp)dx$ ▷ Abstract integration of posterior ▷ from Def. 7 (reduction)
10: **end if**

ors. This set of posteriors is controlled by the width of $d^\sharp$, which represents the allowed dataset perturbation range.

Normalization. AURA finally outputs bounds on normalized posterior. To obtain bounds on the normalized posterior $[\![P]\!]_n^\sharp$ using unnormalized posterior bounds $[\![P]\!]^\sharp$, AURA performs abstract integration to bound the normalizing constant using the strategy in Def. 4. The partitions used in the previous step can be reused for lower and upper Riemann sums (line 4) as mentioned in Section 5.2. Upon computing the normalizing constant bound $[c_l, c_u]$, AURA performs interval division to normalize the posterior bound of each split (lines 5-6).

7.1 Certified Bounds on Probabilistic Queries

An optional input to AURA's algorithm is a query Q. AURA can use the normalized posterior bounds $[\![P]\!]_n^\sharp$ to certify bounds on the posterior probability of queries. In the data perturbation setting, the bounds computed by $[\![P]\!]_n^\sharp$ enclose not just a single posterior distribution (like in [5, 73]) but an *infinite* number of posteriors. Hence, AURA's bounds on a query's probability hold for an infinite set of posteriors.

Definition 6. ***Queries.*** *A query Q is a logical formula over the variables of P given by the following grammar:* $Q ::= \boldsymbol{x}_j \geq c \mid \boldsymbol{x}_j \leq c \mid Q \wedge Q \mid Q \vee Q$

The queries formally define measurable events, hence we can define the posterior distribution's probability of a query Q. This probability is defined as:

$$Pr_{x \sim [\![P]\!]_n(\cdot,d)}(Q) = \int_x \mathbf{1}_Q \cdot [\![P]\!]_n(x, d)\, dx, \tag{6}$$

where $\mathbf{1}_Q$ is the binary indicator function for the event Q. However, as in Section 4, in general this integral is not tractable, hence AURA *over-approximates* this probability. The over-approximation is computed using an abstract integration similar to Def. 4. The key difference is that we use the *normalized* posterior interval bounds, $[\![P]\!]^\sharp_n$, instead of the unnormalized bounds $[\![P]\!]^\sharp$ that Def. 4 uses. The new abstract integration is:

$$\int^\sharp \mathbf{1}_Q \cdot [\![P]\!]^\sharp_n(x, d^\sharp)dx = \sum_i [\![P]\!]^\sharp_n(x_i^\sharp, d^\sharp) \cdot Vol(x_i^\sharp \cap \{x : Q\}) = [l_Q, u_Q] \quad (7)$$

The summation ($\sum$) is interval addition and the $Vol(\cdot)$ function computes the Lebesgue measure of the input set. Since each $x_i^\sharp$ is an interval, and the set $\{x : Q\}$ is a union or intersection of finitely many intervals, the result of $x_i^\sharp \cap \{x : Q\}$ is itself a union or intersection of finitely many intervals and thus its Lebesgue measure can be computed easily. Hence Eq. 7 ultimately computes an interval that encloses the true integral. We can now state the soundness result:

Theorem 5. *(Soundness) For probabilistic program P, dataset $d \in \gamma(d^\sharp)$ and Query Q we have: $Pr_{x\sim[\![P]\!]_n(\cdot,d)}(Q) \in \int^\sharp \mathbf{1}_Q \cdot [\![P]\!]^\sharp_n(x, d^\sharp)dx$*

Proof.

$$\begin{aligned}
\int^\sharp \mathbf{1}_Q \cdot [\![P]\!]^\sharp_n(x, d^\sharp)dx &= \sum_i [\![P]\!]^\sharp_n(x_i^\sharp, d^\sharp) \cdot Vol(x_i^\sharp \cap \{x : Q\}) \\
&= \sum_i [l_i, u_i] \cdot Vol(x_i^\sharp \cap \{x : Q\}) \\
&= \sum_i \Big[l_i \cdot Vol(x_i^\sharp \cap \{x : Q\}),\ \ u_i \cdot Vol(x_i^\sharp \cap \{x : Q\})\Big] \\
&= \left[\sum_i l_i \cdot Vol(x_i^\sharp \cap \{x : Q\}),\ \ \sum_i u_i \cdot Vol(x_i^\sharp \cap \{x : Q\})\right]
\end{aligned}$$

Here the lower and upper bounds are just lower and upper Riemann sums, hence they enclose $\int \mathbf{1}_Q \cdot [\![P]\!]_n(x, d)dx$, which is exactly just $Pr_{x\sim[\![P]\!]_n(\cdot,d)}(Q)$. □

While the computational difficulty in evaluating this integral scales with the latent variable's dimension and the size of the query's predicate, the integration remains tractable for all our benchmarks.

7.2 Robustness

This soundness result directly implies the following proposition which relates soundness to a formal notion of robustness of probabilistic programs. Certification of this robustness property for a given program, is the most important output of AURA's verification approach.

Proposition 1 (Robustness). *For probabilistic program P, Query Q, data d_i and perturbation parameter ϵ_i then for any $d_i' \in [d_i - \epsilon_i, d_i + \epsilon_i]$*

$$Q_l \leq Pr_{x\sim[\![P]\!]_n(\cdot,d_i')}(Q) \leq Q_u$$

where Q_l and Q_u are the bounds computed by AURA in Algorithm 1

This proposition establishes that for a probabilistic program P, conditioning on observed data $d_i \in \mathbb{R}$, if an adversary can perturb each d_i within a range $[l_{d_i}, u_{d_i}]$, then the posterior probability of an event Q under *any such* adversarially perturbed posterior distribution of P is still guaranteed to lie between $[Q_l, Q_u]$ where Q_l, Q_u are the bounds computed by AURA. Thus AURA certifies the robustness of a posterior probability of a query subject to data perturbations.

7.3 Scope and Limitations

AURA only supports `for` loops with fixed number of iterations instead of `while` loops, since potentially unbounded loops can violate the pseudoconcavity properties. For data perturbations, AURA's support for discrete distributions remains limited to the `flip(p)` primitive which simulates a Bernoulli variable but is only used for encoding mixtures of continuous distributions. Similarly, AURA does not support conditioning on Boolean predicates (e.g., `observe( x > 1 ))`.

While pseudo-concavity checks exists (e.g., using Hladik et al. [31]), AURA's implementation already assumes the probabilistic program is pseudoconcave and thus lacks such automated checks.

In addition, while it is theoretically possible to run projected gradient descent on zonotopes or polyhedra, our implementation requires the interval domain.

Lastly, our data perturbation specifications support *local* robustness guarantees and not global robustness guarantees since our intervals only cover a local range (e.g., $\pm\epsilon$) around a given data point, d_i. In contrast to reasoning about values in a local range (e.g., $[d_i - \epsilon, d_i + \epsilon]$), global robustness requires logical reasoning over *all* values in $\mathbb{R}$, a task which is not yet tractable.

8 Implementation

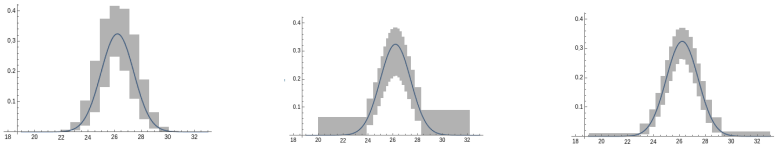
We implemented AURA to strike a balance between precision, efficiency and scalability. AURA supports a wide range of known distributions, including *normal*, *uniform*, *gamma*, *exponential*, *bernoulli*, *logistic*, *laplace*, *beta*, *bernoulli_logit*, and *bernoulli_probit*. For infinite support distributions (e.g., `normal`) AURA uses truncated versions in the priors to ensure compact support (as also done by GuBPI [5]), which we denote with a subscript t. The observed distributions need not be truncated. While AURA's implementation assumes ideal real arithmetic (as is common in ML verification [5, 40, 74]), our evaluation shows that numerical imprecision resulting from floating-point is negligible (Appendix E.3). Furthermore, our implementation can be directly extended to soundly account for floating-point roundoff error by using existing techniques [49] or by using arbitrary precision numerical libraries, e.g., NVIDIA XMP [79] for GPUs.

Parallelization. The steps of Algorithm 1 are parallelizable on CPU or GPU: the two for-loops are data-parallel, while abstract integration and computing bounds on queries are reductions. We implemented AURA using PyTorch, supporting both GPU and CPU backends. To scale to large datasets common in modern applications, which may exceed the memory of a single GPU, we used

software tiling and sharding to distribute computations across multiple GPUs. We implemented a method for integration of local posterior tiles, eliminating the need to store and communicate all posterior tiles between devices.

Efficient Lipschitz Analysis for Learning Rate Selection. To determine the learning rate η, PyTorch's Automatic Differentiation allows us to compute the Lipschitz constant (LC) of the unnormalized posterior $[\![P]\!]$. The LC is bounded by the largest gradient norm (for the local region of the split), and for known distributions, this maximum gradient will occur at the boundaries. Furthermore, using the rules from [4], we aggregate the LCs of individual distributions and estimate the one for the unnormalized posteriors.

Precision-Enhancing Splitting. AURA performs partitions the parameter space into splits $x_i^\sharp$, which can be analyzed in parallel. However one question we ask is what is the *best* strategy for this splitting. While different splitting heuristics do not affect the soundness, they *do* affect the precision.



(a) Equal-length (b) Equal-height (c) Equal-area

Fig. 10: Example of the Analysis Results by Different Splitting Strategies

As an illustration, Figure 10 shows the resulting bounds obtained from three different heuristics each using 20 splits on a simple regression model (lightspeed). The blue line represents the ground truth and the bounds AURA found are the gray boxes. Figure 10a shows the result from the equal-length strategy, which divides the variable interval in equal-length sub-intervals. Due to its simplicity, this strategy is widely used, e.g. by the baseline GuBPI. However, as the plot shows, while the bounds at the tails are reasonably precise, the bounds around the mode (middle part) of the curve are imprecise.

Another strategy is the equal-height splits (Figure 10b), which divides the intervals such that the resulting bounds have the same height. Without knowing the true bounds beforehand, AURA could run a separate analysis (like AQUA [34]) once with a small number of splits (e.g. 60 splits with equal-length) to estimate the shape of the curve, and then use those results to decide the splits with approximately equal height. Nevertheless, this strategy is imprecise for bounding the tails of the distribution.

Therefore, we designed the third strategy to strike a balance between precisely bounding the tails and precisely bounding the mode. In Figure 10c, we generate splits that results in similar the area for each bounding box. We estimate the shape of the curve, and use the results to generate splits with approximately equal area. We use this equal-area strategy as the default in AURA analysis, however AURA supports all three strategies and allows other customized splits.

Table 1: Benchmark program details. Symbols used: $\mathcal{B}$: Bernoulli, $\mathcal{U}$: Real Uniform, $\mathcal{N}$: Normal, $\mathcal{N}_t$: Truncated Normal, $\mathcal{S}$: Logistic. Operators: $+$: mix of distributions, $\times$: product of densities, $^{\alpha}$: number of priors/data. .

Program	Prior	Lik	Description	PC
human_height	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^3$	Learning height with mixture prior [56]	$\checkmark_{mix}$
reg_logistic	$\mathcal{N}_t$	$\mathcal{S}^{919}$	Linear regression with logistic likelihood [64]	$\checkmark_{LC}$
lightspeed	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Linear regression [68]	$\checkmark_{PC}$
anova_radon_n	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Hierarchical linear regression, non-predictive [68]	$\checkmark_{PC}$
reg_laplace	$\mathcal{U}^2 \times {\mathcal{L}_t}^2$	$\mathcal{N}^{919}$	Linear regression with Laplace priors [78]	$\checkmark_{LC}$
prior_mix	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^{10}$	Model with mixture prior [34]	$\checkmark_{mix}$
IQStan	$\mathcal{U}^3$	$\mathcal{N}^3 \times \mathcal{N}^3$	Regression on 2 datasets w. shared variance [41]	$\checkmark_{PC}$
timeseries	$\mathcal{U}^3$	$\mathcal{N}^{99}$	Timeseries model [68]	$\checkmark_{LC}$
unemployment	$\mathcal{U}^3$	$\mathcal{N}^{40}$	Linear Regression [68]	$\checkmark_{PC}$
altermu2	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Model with param symmetry [34]	$\checkmark_{LC}$
robust_model†	$\mathcal{N}^{101}$	$\mathcal{N}^{100}$	Robust model w. many local latent params. [71]	$\checkmark_{mix}$

† This benchmark is evaluated separately as it requires more splits.

9 Methodology

Benchmarks. We evaluated AURA on 11 benchmarks from existing literature including both PL works and real world end-user scientific applications with diverse program structure and distributions. We present the details of our selected benchmarks in Table 1. The studies of these models presented actionable insights to domain-experts across multiple communities (for discussion, see Appendix B). For each program, we manually verified its pseudoconcavity (the details of checking in Appendix C).

Baselines. We initially selected GuBPI [5], the start-of-the art tool for obtaining sound bounds on single posteriors and PSI [20], which leverages symbolic analysis to determine the exact posterior. However, our initial experiments for certifying bounds on ***single posteriors*** (without data perturbation) showed that: (1) GuBPI results in unacceptable imprecision even for simpler programs we studied here, is $> 70x$ slower than AURA on CPU, exhibits numerical instability, and the implementation does not support interval-valued data; (2) PSI can symbolically analyze programs with symbolic data noise, but its integration scales only to programs with a few observations. Hence, these tools cannot support computing bounds for ***a set of posteriors***, which is needed for verifying robustness of complex probabilistic programs benchmarks we consider. This reason also rules out works derived from those tools such as PSense [37] and [73]. Nevertheless, to provide evidence for this choice we present a detailed comparison of AURA with GuBPI and PSI for ***single posteriors*** in Appendix E.

As the baseline, we instead implemented interval-based abstract transformers within AURA, akin to GuBPI's interval abstraction, but we carefully enhanced its numerical stability. Evaluating this interval analysis version on the benchmarks achieves much higher precision than GuBPI.

Precision Metric. We define the lower p_l and upper p_u bound functions for marginal posterior of the parameter x: $p_l(x) = l$ and $p_u(x) = u$ where $[l, u] =$

$[\![P]\!]_n^\sharp(x_i^\sharp, d^\sharp)$ for $x \in \gamma(x_i^\sharp)$. Total Variation Distance (TVD) measures the discrepancy between the two bounds of a distribution: $\text{TVD}_x = \frac{1}{2}\int |p_l(x) - p_u(x)|\, dx$. For multiple parameters, TVD is averaged across each: $\text{TVD} = \frac{1}{M}\sum_{j=1}^{M} \text{TVD}_{x_j}$. Appendix D gives more detailed definitions of the metrics.

Setup for Adversarial Data Perturbation Analysis. We use AURA to find bounds on posteriors obtainable after data perturbations. We use 200 splits for the input domain. Perturbations involve 1-5 key data points per benchmark, identified by gradient magnitude $\frac{\partial}{\partial d}[\![P]\!](x, d)$. For datasets with $<$100 points, perturbations are capped at five points or 5% of the dataset. Perturbation intervals are computed by modifying the original data by $[0, 0.01\sigma]$ if $sign(\frac{\partial}{\partial d}[\![P]\!](x, d))$ is positive or $[-0.01\sigma, 0]$ if negative, where σ is the dataset's standard deviation. We adapt the adversarial data perturbation method from the Fast Gradient Signed Method (FGSM) [25], a prevalent method in machine learning, which is to add a small perturbation towards the direction increasing the loss (cf. decreasing likelihood). AURA and the baseline are enhanced with GPU acceleration and equal-area splits for efficiency.

Experimental Setup. We run AURA and all the other tools on a AMD 4.2 GHz machine with 32 cores and with 2 NVIDIA RTX A5000 GPUs.

10 Evaluation

We next describe the results that demonstrate AURA effectiveness and efficiency to compute posteriors under data perturbation and verify probabilistic queries.

10.1 Posteriors under Data Perturbation

Precision of Bounds. We use AURA to find bounds for a set of posteriors when subjected to data perturbation. Table 2 presents the precision (in TVD) alongside the run time for both AURA and a baseline interval analysis. We run both using a GPU (CPU shows the same time trend).

On average (geo-mean), AURA achieves a precision 12.8× better than that of the interval analysis. We observed across all benchmarks that the data perturbation causing the maximum posterior error (TVD) almost never occurs at the extremes of the data perturbation interval, indicating that the sound bounds cannot be simulated just from data values at perturbation extremes.

Execution Time. AURA's run time averages at 3.14s (geometric mean). The increase in run time when compared to AURA analysing a single posterior without perturbation is due to the additional complexity of high-dimensional optimization across both the parameters and the data dimensions subjected to perturbation. Compared to the interval analysis, AURA has an additional cost of iteratively evaluating the unnormalized posterior during gradient ascent.

10.2 Analysis Examples

Figures 11 and 12 illustrate the bounds computed by AURA for two example models under data perturbation, compared with the results from the interval

Table 2: AURA and Interval Analysis Results for Data Perturbation

Program	**TVD**		**Time (s)**	
	AURA	Interval	AURA (GPU)	Interval (GPU)
human_height	0.04	0.09 (2.3×)	0.25	0.02
reg_logistic	0.05	8.13 (175.8×)	3.49	1.39
lightspeed	0.07	0.56 (8.2×)	0.19	0.04
anova_radon_n	0.07	0.57 (8.2×)	1.52	0.05
reg_laplace	0.07	6.28 (87.6×)	10.44	1.18
prior_mix	0.07	0.23 (3.2×)	0.27	0.03
IQStan	0.07	0.20 (2.7×)	15.67	0.14
timeseries	0.18	1.04 (5.7×)	335.16	5.37
unemployment	0.23	4.10 (18.1×)	131.76	0.82
altermu2	0.28	16.43 (58.2×)	0.19	0.06
GeoMean	0.09	1.17 (12.8×)	3.14	0.19

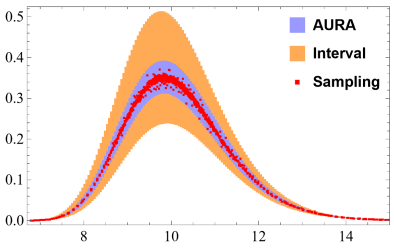


Fig. 11: lightspeed (param: σ)

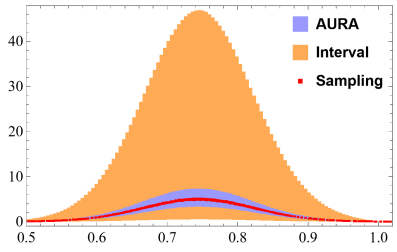


Fig. 12: unemployment (param:β_1)

analysis and reference distributions generated via Stan sampling. To generate the reference distributions, we simulated at least three concrete perturbations for each perturbed data point and used Stan's NUTS to collect 400,000 samples for each concrete perturbation.

The plots show the parameter value on the x-axis against the posterior probability density on the y-axis, with red dots representing the sampled reference distributions and blue and orange rectangles representing the bounds computed by AURA and interval analysis, respectively. Consistent with the findings reported in Table 2, AURA shows significantly tighter bounds than those from interval analysis. Further, the AURA bounds are close to the envelope created by observed samples (red dots), which give an under-approximation. This precision of AURA's analysis stems from its optimization-based abstraction, which is designed to find the narrowest bounds before normalization.

Table 3: Results for Queries under Data Perturbation ("|" refers to the same benchmark as in the previous row)

Program	**Query**	**Probability Bound**		
		AURA	Interval	Improv.
human_height	$\mu > 165$	$[0.93, 1.00]$	$[0.85, 1.00]$	2.1×
|	$170 < \mu < 172$	$[0.11, 0.13]$	$[0.10, 0.14]$	1.7×
reg_logistic	$\beta_0 \geq 1.35 \vee \beta_0 < 1.15$	$[0.05, 0.07]$	$[0.00, 0.98]$	63.6×
|	$\beta_0 \geq 1.25$	$[0.33, 0.40]$	$[0.02, 1.00]$	14.0×
lightspeed	$20 \leq \beta_0 < 40$	$[0.88, 1.00]$	$[0.47, 1.00]$	4.4×
|	$\beta_0 < 30 \wedge \sigma < 10$	$[0.42, 0.55]$	$[0.21, 1.00]$	6.1×
anova_radon_n	$0.9 \leq a_0 \leq 1$	$[0.28, 0.34]$	$[0.14, 0.66]$	7.6×
|	$\sigma_y \leq 1 \vee a_0 > 1$	$[0.86, 1.00]$	$[0.46, 1.00]$	3.9×
reg_laplace	$1.3 \leq \beta_1 \leq 1.35$	$[0.27, 0.36]$	$[0.02, 1.00]$	11.5×
|	$\beta_0 \geq -0.5$	$[0.08, 0.11]$	$[0.01, 1.00]$	34.1×
prior_mix	$\mu_0 \leq 0$	$[0.74, 0.97]$	$[0.58, 1.00]$	1.8×
|	$\mu_0 > -2 \wedge \mu_0 < 2$	$[0.08, 0.11]$	$[0.06, 0.12]$	1.4×
IQStan	$\mu_1 \geq 85 \vee \mu_2 \geq 85$	$[0.87, 1.00]$	$[0.71, 1.00]$	2.2×
|	$5 < \sigma < 10 \wedge \mu_1 > 95 \wedge \mu_2 > 95$	$[0.25, 0.32]$	$[0.21, 0.38]$	2.3×
timeseries	$\alpha < -1 \vee \beta < 1 \wedge lag < 0.8$	$[0.69, 1.00]$	$[0.31, 1.00]$	2.3×
|	$\alpha > -0.5 \wedge lag > 0.5$	$[0.02, 0.04]$	$[0.01, 0.09]$	4.3×
unemployment	$\sigma < 1.2 \wedge \beta_0 < 3 \wedge \beta_1 < 0.7$	$[0.19, 0.41]$	$[0.03, 1.00]$	4.5×
|	$0.95 < \beta_1 < 1$	$[0.00, 0.01]$	$[0.00, 0.04]$	10.4×
altermu2	$1 \leq \mu_0 \leq 1.1$	$[0.02, 0.04]$	$[0.00, 0.84]$	34.0×
|	$\mu_0 < 1.5 \wedge \mu_1 < 1.5$	$[0.47, 1.00]$	$[0.02, 1.00]$	1.9×

10.3 Queries under Data Perturbation

We demonstrate the use of AURA in evaluating the posterior probability of specific events when the input dataset is subject to perturbations. Intuitively, one provides a query, as described in Def. 6, and AURA then computes sound bounds on the posterior probability of the event defined by the query, where the probability bounds hold for *any* posterior that could result from the data perturbation.

Table 3 illustrates the results and the computation time AURA used to bound the posterior probability of each query. For each program amenable to data perturbation (i.e. those with continuous data), we formulated two distinct queries (shown in the **Query** column). The **Probability** columns show the bounds computed by AURA and the interval analysis we implemented as the baseline. The **Improv.** column shows the improvement by AURA, reflected in how many times smaller AURA's interval is compared to the results from interval analysis.

Across all programs with data perturbation, AURA's bounds on the queries are much more precise, being on geomean 5.35x narrower than the bounds from interval arithmetic. The time for each query is almost identical to the time for computing the posterior bounds under perturbation (geomean $\sim 3.1s$).

Table 4: AURA Scalability to Large Numbers of Local Latents

#Local Latents	#Splits	W.o. Perturbation		W. Perturbation	
		Time (s)	TVD	Time (s)	TVD
20	1600	3.1	0.07	5.6	0.11
50	1600	7.7	0.19	14.4	0.33
50	5000	41.0	0.05	150.0	0.12
100	5000	82.1	0.11	303.9	0.26
100	10000	239.5	0.05	1017.4	0.16

10.4 Scaling to a Large Number of Local Latent Parameters

We evaluated the robust linear regression model in Figure 9 by varying the number of latent parameters and splits. AURA's inference successfully scales to the large number of local latents in such a model. Table 4 presents the results for the model with up to 100 local latents/datapoints, with and without one datapoint subject to perturbation. This model requires more fine-grained splitting than our other benchmarks due to fine-grained integration of latents.

10.5 Ablation Studies

We performed several ablation studies of AURA's algorithm, detailed in Appendix E.3 and [33] and summarized next. (1) Increasing the number of observed data points to 5000 shows AURA's time increases linearly with data size. (2) AURA can extend computation across GPUs and obtian parallelization speeudp on CPU depends primarily on the size of the GPU memory. (3) Floating point imprecision vs double is small, with geomean errror $\sim 10^{-7}$ and double taking only 10% more time on a GPU. (4) Increasing the number of splits increases the precision of the result, but also increases run time; for our benchmarks 200 splits (used in the evaluation) gives near-optimal point in this tradeoff space.

11 Related Work

Exact Probabilistic Programming Systems. Despite recent progress, exact inference systems are limited for continuous distributions: e.g., many support only discrete models (DICE [32]), only handle Sum-Product networks [62] or cannot solve many complicated integrals (PSI [20], Hakaru [55]). Exact inference also faces obstacles with scaling to large numbers of data observations.

Interval-Based Abstractions for Probabilistic Inference. Closest in spirit to our work is GuBPI [5] which computes interval bounds on a *single* posterior. While GuBPI supports recursion it is much less precise than AURA and cannot scale to the datasizes AURA supports. Subsequent work [73] offered improvements to GuBPI, however like GuBPI, [73] also restricts to analyzing bounds for only a single posterior distribution. They *do not* study nor evaluate the problem of certifying bounds for *all* possible posteriors obtainable from a data

perturbation; neither [5, 73] can be parallelized and thus suffer from scalability concerns. Furthermore, while [7] studies Bayesian inference, in contrast to AURA, their bounds are confidence-interval based and tailored to cumulative distribution functions instead of probability density functions.

Several existing works study probabilistic abstract interpretations [11, 47] but they do not consider Bayesian inference and lack `observe` statements and distribution normalization, a limitation also present in other works [9, 44, 65]. Neither probability boxes [8, 16] nor Dempster-Shafer structures [16] can be used in our setting since they do not have a mechanism to reason about Bayesian inference. Additionally, [41, 51] compute heuristic bounds on likelihoods for fixed-point size selection, but they lack the formal verification guarantees that AURA provides and they do not study the data perturbation setting that AURA targets.

Sensitivity and Robustness Analysis of Probabilistic Programs. In addition, there are a few relevant works related to the sensitivity and robustness of probabilistic programs. PSense [37] performs sensitivity analysis on probabilistic programs but it builds directly on PSI [20] and thus inherits the same limitations regarding scalability. PSense also needs to analytically compute another integral characterizing the difference between the two normalized distributions with noise variables.

Additionally, AquaSense [81] is a tool that directly builds on Aqua [34], which provides only a weak asymptotic theorem that the computed distribution estimate converges to the true distribution when the number of splits is infinite (i.e., interval width tends to 0). Moreover, Aqua's distribution estimate provides a soundness guarantee only asymptotically – it has only a sound lower bound, which is equivalent to AURA's lower bound. In contrast, AURA provides both the sound lower and upper bounds for a finite number of splits.

Optimization-Based Abstract Interpretation. Using continuous optimization to perform abstract interpretation and sound bound computation has been extensively studied [1, 18, 40, 52, 53, 54, 61, 63, 66, 72, 75]. However these works target computations like DNNs [54, 66], security properties [63], automatic differentiation [40], and non-probabilistic programs [1, 18, 53, 75]. Unlike AURA, these applications do not typically obey the pseudoconcavity properties.

12 Conclusion

We presented AURA, a novel abstract interpretation of probabilistic programs that can certify bounds on posterior distributions under data perturbations, By designing custom, precise and scalable abstract transformers for probabilistic programming using optimization, AURA represents a first step towards making provably robust probabilistic programming a reality. We anticipate that AURA also opens the door to obtaining certified robustness for general probabilistic models (like Normalizing Flows and Probabilistic Circuits), and even abstracting other (non-probabilistic) pseudoconcave functions.

Acknowledgments This research was supported in part by NSF Grants No. CCF-1846354, CCF-2008883, and CCF-2313028.

Bibliography

[1] Adjé, A., Garoche, P.L., Werey, A.: Quadratic zonotopes: an extension of zonotopes to quadratic arithmetics. In: Programming Languages and Systems: 13th Asian Symposium, APLAS 2015. pp. 127–145 (2015)

[2] Ahn, K., Jain, P., Ji, Z., Kale, S., Netrapalli, P., Shamir, G.I.: Reproducibility in optimization: Theoretical framework and limits. Advances in Neural Information Processing Systems **35**, 18022–18033 (2022)

[3] Bagnoli, M., Bergstrom, T.: Log-concave probability and its applications. In: Rationality and Equilibrium: A Symposium in Honor of Marcel K. Richter. pp. 217–241 (2006)

[4] Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.Y.: Proving expected sensitivity of probabilistic programs. vol. 2 (2017)

[5] Beutner, R., Ong, C.H.L., Zaiser, F.: Guaranteed bounds for posterior inference in universal probabilistic programming. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 536–551 (2022)

[6] Bherwani, H., Anjum, S., Kumar, S., Gautam, S., Gupta, A., Kumbhare, H., Anshul, A., Kumar, R.: Understanding covid-19 transmission through bayesian probabilistic modeling and gis-based voronoi approach: a policy perspective. Environment, Development and Sustainability **23**(4), 5846–5864 (2021)

[7] Boreale, M., Collodi, L.: Bayesian parameter estimation with guarantees via interval analysis and simulation. In: International Conference on Verification, Model Checking, and Abstract Interpretation (2023)

[8] Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A generalization of p-boxes to affine arithmetic. Computing **94**, 189–201 (2012)

[9] Constantinides, G., Dahlqvist, F., Rakamarić, Z., Salvia, R.: Automated roundoff error analysis of probabilistic floating-point computations. ACM Trans. Probab. Mach. Learn. (2024)

[10] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pp. 238–252. POPL '77, ACM (1977)

[11] Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Programming Languages and Systems, pp. 169–193 (2012)

[12] Cruz, J.B., Pérez, L.L.: Convergence of a projected gradient method variant for quasiconvex objectives. Nonlinear Analysis: Theory, Methods & Applications **73**(9), 2917–2922 (2010)

[13] Darir, H., Dullerud, G.E., Borisov, N.: Probflow: Using probabilistic programming in anonymous communication networks. In: NDSS (2023)

[14] Dunn, J.C.: Global and asymptotic convergence rate estimates for a class of projected gradient processes. SIAM Journal on Control and Optimization **19**(3), 368–400 (1981)

[15] Fernando, V., Joshi, K., Laurel, J., Misailovic, S.: Diamont: Dynamic monitoring of uncertainty for distributed asynchronous programs. In: 21st International Conference on Runtime Verification, RV 2021. Springer (2021)
[16] Ferson, S., Kreinovich, V., Ginzburg, L., Sentz, F.: Constructing probability boxes and dempster-shafer structures. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia (2003)
[17] Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., Silva, A.: Probabilistic netkat. In: Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016. pp. 282–309 (2016)
[18] Gawlitza, T.M., Seidl, H., Adjé, A., Gaubert, S., Goubault, É.: Abstract interpretation meets convex optimization. Journal of Symbolic Computation **47**(12), 1416–1446 (2012)
[19] Gehr, T., Misailovic, S., Tsankov, P., Vanbever, L., Wiesmann, P., Vechev, M.: Bayonet: probabilistic inference for networks. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 586–602. ACM (2018)
[20] Gehr, T., Misailovic, S., Vechev, M.: PSI: Exact symbolic inference for probabilistic programs. In: International Conference on Computer Aided Verification (2016)
[21] Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. Journal of Educational and Behavioral Statistics (2015)
[22] Ghorbal, K., Goubault, E., Putot, S.: The zonotope abstract domain taylor1+. In: Computer Aided Verification: 21st International Conference, CAV 2009. pp. 627–633 (2009)
[23] Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22. pp. 212–226 (2010)
[24] Gloeckler, M., Deistler, M., Macke, J.H.: Adversarial robustness of amortized bayesian inference. In: Proceedings of the 40th International Conference on Machine Learning. pp. 11493–11524 (2023)
[25] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
[26] Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FoSE (2014)
[27] GuBPI – An Analyzer for Probabilistic Programs to Compute Guaranteed Bounds on the Posterior (2022), `https://github.com/gubpi-tool/gubpi`
[28] Hazan, E., Levy, K., Shalev-Shwartz, S.: Beyond convexity: Stochastic quasi-convex optimization. Advances in neural information processing systems **28** (2015)
[29] Heck, D.W., Thielmann, I., Moshagen, M., Hilbig, B.E.: Who lies? a large-scale reanalysis linking basic personality traits to unethical decision making. Judgment and Decision making **13**(4), 356–371 (2018)
[30] Higgins, J.E., Polak, E.: Minimizing pseudoconvex functions on convex compact sets. Journal of Optimization Theory and Applications **65**(1), 1–27 (1990)

[31] Hladík, M., Kolev, L.V., Skalna, I.: Linear interval parametric approach to testing pseudoconvexity. Journal of Global Optimization **79**, 351–368 (2021)
[32] Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–31 (2020)
[33] Huang, Z.: Enhancing trustworthiness in probabilistic programming: systematic approaches for robust and accurate inference. Ph.D. thesis, University of Illinois at Urbana-Champaign (2024)
[34] Huang, Z., Dutta, S., Misailovic, S.: Aqua: Automated quantized inference for probabilistic programs. In: International Symposium on Automated Technology for Verification and Analysis (2021)
[35] Huang, Z., Dutta, S., Misailovic, S.: Astra: understanding the practical impact of robustness for probabilistic programs. In: Uncertainty in Artificial Intelligence. pp. 900–910. PMLR (2023)
[36] Huang, z., Laurel, J., Dutta, S., Misailovic, S.: Appendix for aura: Precise abstract interpretation of probabilistic programs with interval uncertainty (2025), `https://github.com/uiuc-arc/AURA/AURAappendix.pdf`
[37] Huang, Z., Wang, Z., Misailovic, S.: Psense: Automatic sensitivity analysis for probabilistic programs. In: 16th International Symposium on Automated Technology for Verification and Analysis. ATVA (2018)
[38] Kučera, M., Tsankov, P., Gehr, T., Guarnieri, M., Vechev, M.: Synthesis of probabilistic privacy enforcement. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 391–408 (2017)
[39] Laurel, J., Misailovic, S.: Continualization of probabilistic programs with correction. In: European Symposium on Programming. pp. 366–393 (2020)
[40] Laurel, J., Qian, S.B., Singh, G., Misailovic, S.: Synthesizing precise static analyzers for automatic differentiation. Proceedings of the ACM on Programming Languages **7**(OOPSLA2) (2023)
[41] Laurel, J., Yang, R., Sehgal, A., Ugare, S., Misailovic, S.: Statheros: Compiler for efficient low-precision probabilistic programming. In: 58th ACM/IEEE Design Automation Conference (DAC). IEEE (2021)
[42] Laurel, J., Yang, R., Singh, G., Misailovic, S.: A dual number abstraction for static analysis of clarke jacobians. Proceedings of the ACM on Programming Languages **6**(POPL), 1–30 (2022)
[43] Laurel, J., Yang, R., Ugare, S., Nagel, R., Singh, G., Misailovic, S.: A general construction for abstract interpretation of higher-order automatic differentiation. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1007–1035 (2022)
[44] Lohar, D., Prokop, M., Darulova, E.: Sound probabilistic numerical error analysis. In: International Conference on Integrated Formal Methods. pp. 322–340. Springer (2019)
[45] Majthay, A., Whinston, A.: Quasi-concave minimization subject to linear constraints. Discrete Mathematics **9**(1), 35–59 (1974)
[46] Mangasarian, O.L.: Pseudo-convex functions. In: Stochastic optimization models in finance, pp. 23–32. Elsevier (1975)

[47] Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. Journal of Computer Security **21**(4), 463–532 (2013)
[48] Wikipedia: Mean absolute error (2023), `https://en.wikipedia.org/wiki/Mean_absolute_error`
[49] Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: European Symposium on Programming. pp. 3–17 (2004)
[50] Miné, A., et al.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Prog. Lang. **4**(3-4) (2017)
[51] Misra, A., Laurel, J., Misailovic, S.: Vix: analysis-driven compiler for efficient low-precision variational inference. In: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2023)
[52] Monniaux, D.: On using floating-point computations to help an exact linear arithmetic decision procedure. In: International Conference on Computer Aided Verification. pp. 570–583 (2009)
[53] Monniaux, D.P.: Automatic modular abstractions for linear constraints. SIGPLAN Not. p. 140–151 (Jan 2009)
[54] Müller, M.N., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.: Prima: General and precise neural network certification via scalable convex hull approximations. Proc. ACM Program. Lang. **6**(POPL) (2022)
[55] Narayanan, P., Carette, J., Romano, W., Shan, C.c., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: FLOPS 2016 (2016)
[56] Oberski, D.: Mixture models: Latent profile and latent class analysis. Modern statistical methods for HCI pp. 275–287 (2016)
[57] Owhadi, H., Scovel, C., Sullivan, T.: On the brittleness of bayesian inference. SIAM REVIEW **57**(4), 566–582 (2015)
[58] Pardo, R., Rafnsson, W., Probst, C.W., Wąsowski, A.: Privug: using probabilistic programming for quantifying leakage in privacy risk analysis. In: European Symposium on Research in Computer Security (2021)
[59] PSI Solver (2019), `https://github.com/eth-sri/psi/tree/e729dd7d68e23a4a75731b4bb800c95111a7a30b`
[60] Roberts, G.O., Rosenthal, J.S.: General state space markov chains and mcmc algorithms. Probability surveys **1**, 20–71 (2004)
[61] Rustenholz, L., López-García, P., Morales, J.F., Hermenegildo, M.V.: An order theory framework of recurrence equations for static cost analysis–dynamic inference of non-linear inequality invariants. In: International Static Analysis Symposium. pp. 352–385 (2024)
[62] Saad, F.A., Rinard, M.C., Mansinghka, V.K.: Sppl: probabilistic programming with fast exact symbolic inference. In: PLDI (2021)
[63] Saha, S., Ghentiyala, S., Lu, S., Bang, L., Bultan, T.: Obtaining information leakage bounds via approximate model counting. Proceedings of the ACM on Programming Languages **7**(PLDI), 1488–1509 (2023)
[64] Salimans, T., Karpathy, A., Chen, X., Kingma, D.P.: Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. arXiv preprint arXiv:1701.05517 (2017)

[65] Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths pp. 447–458 (2013)
[66] Singh, G., Laurel, J., Misailovic, S., Banerjee, D., Singh, A., Xu, C., Ugare, S., Zhang, H.: Safety and trust in artificial intelligence with abstract interpretation. Foundations and Trends in Prog. Lang. **8**(3-4) (2025)
[67] Smolka, S., Kumar, P., Kahn, D.M., Foster, N., Hsu, J., Kozen, D., Silva, A.: Scalable verification of probabilistic networks. In: PLDI (2019)
[68] Stan Examples (2018), `https://github.com/stan-dev/example-models`
[69] Sweet, I., Trilla, J.M.C., Scherrer, C., Hicks, M., Magill, S.: What's the over/under? probabilistic bounds on information leakage. In: International Conference on Principles of Security and Trust (2018)
[70] Thall, P.F., Ursino, M., Baudouin, V., Alberti, C., Zohar, S.: Bayesian treatment comparison using parametric mixture priors computed from elicited histograms. Statistical methods in medical research **28**(2), 404–418 (2019)
[71] Wang, C., Blei, D.M.: A general method for robust bayesian modeling. Bayesian Analysis **13**(4), 1159–1187 (2018)
[72] Wang, J., Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Quantitative analysis of assertion violations in probabilistic programs. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2021)
[73] Wang, P., Yang, T., Fu, H., Li, G., Ong, C.H.L.: Static posterior inference of bayesian probabilistic programming via polynomial solving. Proceedings of the ACM on Programming Languages (PLDI) (2024)
[74] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. Advances in Neural Information Processing Systems **34**, 29909–29921 (2021)
[75] Wang, T., Chen, L., Chen, T., Fan, G., Wang, J.: Making rigorous linear programming practical for program analysis. In: 27th International Conference on Principles and Practice of Constraint Programming (2021)
[76] Wang, Y., Kucukelbir, A., Blei, D.M.: Robust probabilistic modeling with bayesian data reweighting. In: Proceedings of the 34th International Conference on Machine Learning. pp. 3646–3655. ICML'17 (2017)
[77] Wicker, M., Laurenti, L., Patane, A., Chen, Z., Zhang, Z., Kwiatkowska, M.: Bayesian inference with certifiable adversarial robustness. In: International Conference on Artificial Intelligence and Statistics. PMLR (2021)
[78] Williams, P.M.: Bayesian regularization and pruning using a laplace prior. Neural computation **7**(1), 117–143 (1995)
[79] XMP Library (2016), `https://github.com/NVlabs/xmp/tree/master`
[80] Yang, R., Laurel, J., Misailovic, S., Singh, G.: Provable defense against geometric transformations. In: 11th International Conference on Learning Representations (2023)
[81] Zhou, Z., Huang, Z., Misailovic, S.: Aquasense: Automated sensitivity analysis of probabilistic programs via quantized inference. In: International Symposium on Automated Technology for Verification and Analysis. pp. 288–301 (2023)

Appendix

A AURA Extensions Beyond Pseudoconcavity: Compositionality and Interval Abstract Domain

A.1 Supporting Branch Statements and Mixture Models.

While pseudoconcave likelihoods lead to precise and tractable gradient-based abstract interpretation, the question arises of how to support posteriors that are *not* pseudoconcave. This scenario is encountered in popular mixture distributions that result from the branching primitive, `if flip`(p) P_1 `else` P_2, in our language.

We will show that AURA still supports mixture distributions that contain such branches which can cause the likelihoods to no longer be pseudoconcave. The key technical insight is that even if the entire posterior is multi-modal and thus not pseudoconcave, each *component* when viewed in isolation could be pseudoconcave. Thus by applying AURA's abstract interpreter, $[\![\cdot]\!]^\sharp$, to each component (which *will* be pseudoconcave) and then combining the results with standard interval arithmetic, we can still obtain sound posterior bounds. Indeed, we recall from Fig. 8 that:

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!](x,d) = p \cdot [\![P_1]\!](x,d) + (1-p) \cdot [\![P_2]\!](x,d) \quad (8)$$

Thus for this primitive, we will define the unnormalized *abstract* semantics as:

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]^\sharp(x^\sharp,d^\sharp) = p \cdot^\sharp [\![P_1]\!]^\sharp(x^\sharp,d^\sharp) +^\sharp (1-p) \cdot^\sharp [\![P_2]\!]^\sharp(x^\sharp,d^\sharp) \quad (9)$$

Hence by applying the abstract interpreter to P_1 and P_2, each of which *will* be pseudoconcave, the computation of $[\![P_1]\!]^\sharp(x^\sharp,d^\sharp)$ and $[\![P_2]\!]^\sharp(x^\sharp,d^\sharp)$ can use AURA's gradient based optimization to obtain tight bounds. Given the soundness of interval addition $+^\sharp$ and multiplication $\cdot^\sharp$, we obtain:

Lemma 2. *When P_1 and P_2 are pseudoconcave, based on Theorem 1, the preservation of soundness by interval addition and interval multiplication, AURA's abstraction for a branching program is sound:*

$$\{[\![\textit{if flip}(p)\ P_1\ \textit{else}\ P_2]\!](x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![\textit{if flip}(p)\ P_1\ \textit{else}\ P_2]\!]^\sharp(x^\sharp,d^\sharp))$$

Once we have these sound bounds on the *unnormalized* posterior, we can pass them as input to the abstract integration, and thus bound the *normalized* posterior for programs with branches.

Lemma 3. *Under data perturbation, the soundness Theorem 2, still holds for a program P that consists of a mixture of pseudoconcave branches .*

$$\{[\![\textit{if flip}(p)\ P_1\ \textit{else}\ P_2]\!]_n(x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![\textit{if flip}(p)\ P_1\ \textit{else}\ P_2]\!]_n^\sharp(x^\sharp,d^\sharp))$$

A.2 Pseudoconcave Subexpressions when Programs are not Pseudoconcave.

Beyond using interval arithmetic for the addition and multiplication operations in Eq. 9, *since AURA uses the interval domain, we can fallback to interval arithmetic for* any *of the subexpressions in Fig. 8.* Indeed, the baseline uses a standard interval abstract interpretation for all expressions. Thus, even if the program P lacks a pseudoconcave posterior, we can compute optimized bounds on the largest subexpressions which *are* pseudoconcave, and then use standard interval arithmetic for the rest. Hence Sections 6.1 and 6.2 provide sound interval domain abstract transformers for *any* pseudoconcave function, including subexpressions within P. We state this property in Theorem 6:

Theorem 6. *Let M be a prior subexpression in program P. If $[\![M]\!](x, d)$ is a pseudoconcave function and if the lower bounds are computed as in Section 6.1 and upper bounds computed as in Section 6.2, then the following bounds are sound*

$$[\![M]\!]^\sharp(x^\sharp, d^\sharp) = [l, u] \quad \textit{where} \quad l = \min_{x \in \gamma(x^\sharp)} \min_{d \in \gamma(d^\sharp)} [\![M]\!](x, d) \qquad u = \max_{x \in \gamma(x^\sharp)} \max_{d \in \gamma(d^\sharp)} [\![M]\!](x, d)$$

Similarly let D be an observation subexpression in program P. If $[\![D]\!](x, d)$ is pseudoconcave, then the bounds computed as in Sections 6.1 and 6.2 are sound:

$$[\![D]\!]^\sharp(x^\sharp, d^\sharp) = [l, u] \quad \textit{where} \quad l = \min_{x \in \gamma(x^\sharp)} \min_{d \in \gamma(d^\sharp)} [\![D]\!](x, d) \qquad u = \max_{x \in \gamma(x^\sharp)} \max_{d \in \gamma(d^\sharp)} [\![D]\!](x, d)$$

The soundness of the lower bounds for these subexpressions follows identically to Theorem 3 and for upper bounds identically to Theorem 4. We can now define $[\![M]\!]^\sharp_{best}$ and $[\![D]\!]^\sharp_{best}$ as abstract transformers that use the optimized bounds of Theorem 6 for the entire subexpression M or D if it is pseudoconcave, otherwise they will default to recursively evaluating the subexpression in interval arithmetic $\mathbb{IR}$ for subexpressions which are not pseudoconcave.

$$[\![M]\!]^\sharp_{best}(x^\sharp, d^\sharp) = \begin{cases} [\![M]\!]^\sharp_{\mathbb{IR}} & [\![M]\!](x, d) \text{ is not pseudoconcave} \\ [\![M]\!]^\sharp & [\![M]\!](x, d) \text{ is pseudoconcave} \end{cases}$$

where $[\![M]\!]^\sharp_{\mathbb{IR}}$ evaluates $[\![M]\!]$ using standard interval arithmetic.

$$[\![D]\!]^\sharp_{best}(x^\sharp, d^\sharp) = \begin{cases} [\![D]\!]^\sharp_{\mathbb{IR}} & [\![D]\!](x, d) \text{ is not pseudoconcave} \\ [\![D]\!]^\sharp & [\![D]\!](x, d) \text{ is pseudoconcave} \end{cases}$$

where similarly, $[\![D]\!]^\sharp_{\mathbb{IR}}$ evaluates $[\![D]\!]$ using standard interval arithmetic.

In light of this definition, we can now reformalize AURA's abstract interpreter (for unnormalized semantics) to use the optimized bounds for pseudoconcave sub-expressions, even when the full program's expression $[\![P]\!](x, d)$ is not pseudoconcave. The interval results of the sub-expressions can then be combined

$$[\![P]\!]^\sharp(x,d) = [\![M;D]\!]^\sharp(x^\sharp,d^\sharp) \qquad [\![M;D]\!]^\sharp(x^\sharp,d^\sharp) = [\![M]\!]^\sharp_{best}(x^\sharp) \cdot^\sharp [\![D]\!]^\sharp_{best}(x^\sharp,d^\sharp)$$
$$[\![D;D]\!]^\sharp(x^\sharp,d^\sharp) = [\![D]\!]^\sharp_{best}(x^\sharp,d^\sharp) \cdot^\sharp ([\![D]\!]^\sharp_{best}(x^\sharp,d^\sharp)) \qquad [\![M;M]\!]^\sharp(x^\sharp) = [\![M]\!]^\sharp_{best}(x^\sharp) \cdot^\sharp [\![M]\!]^\sharp_{best}(x^\sharp)$$

Fig. 13: Reformalization of AURA's unnormalized abstract semantics for programs which are not completely pseudoconcave

using interval arithmetic, particularly interval multiplication $\cdot^\sharp$. This reformalization is shown in Fig. 13.
Thus AURA supports a compositional analysis even when the full program P is not pseudoconcave. Indeed, a key benefit of our abstract interpretation-based approach is that it allows us to compose our precise, optimized bounds (for subexpressions that are pseudoconcave), with standard interval arithmetic (for subexpressions that are not pseudoconcave).

Theorem 7. *(Soundness) The unnormalized posterior bounds computed in Fig. 13 are sound.*

This soundness follows because compositions of sound abstract transformers are still sound [50].

Example 1. Let $P \equiv M;D$, as in Fig. 6 where M and D are arbitrary subexpressions. To bound $[\![P]\!](x,d)$ we bound $[\![M;D]\!](x,d)$. However when $[\![M]\!](x,d)$ is pseudoconcave but $[\![D]\!](x,d)$ is not, we compute $[\![M]\!]^\sharp(x^\sharp,d^\sharp) \cdot^\sharp [\![D]\!]_{\mathbb{IR}}(x^\sharp,d^\sharp)$ where $[\![M]\!]^\sharp(x^\sharp,d^\sharp)$ uses the bounds of Theorem 6 and $[\![D]\!]_{\mathbb{IR}}(x^\sharp,d^\sharp)$ is the standard interval arithmetic abstraction and $\cdot^\sharp$ is interval multiplication.

A.3 Scaling to High-Dimensional Latents

Prior work has shown that models with more local latent parameters, (meaning a large latent variable dimension) can be more robust. For example, as summarized in [71, 76] a common technique for writing robust models is to add a large number of *local* latent parameters i.e., each observation depends on a fresh latent variable. Thus, *the question of scalability to large numbers of latent variables* arises. Despite the challenges introduced by a large latent parameter dimension, AURA uses a technique shown below to scale linearly with increasing number of latent parameters.

```
1 d = [1.3, 2.1, ... ] #n data points
2 u ~ normal(...)
      #global latent variable
3 for i in range(n):
4   v_i ~ normal(u,1)
        #local latent variables
5   observe(d[i], normal(v_i,1))
```

Fig. 14: Robust Model with $n+1$ Latent Parameters

Figure 14 gives an example model capturing this common pattern. Line 5 encodes that each datapoint has its own i.i.d latent v_i, hence why there are $n+1$ latent variables. In light of this pattern, we let u represent the *global* latent and v_i represent the *local* latents. Thus the latent parameter vector is $x = (u, v_1, ..., v_n)$ which means that naively, we would solve $\mathcal{O}(n)$-dimensional optimization and

integration problems. However, AURA reduces these problems to n easy $\mathcal{O}(1)$-dimensional subproblems (full proof in Appendix A.3 below). Models with this pattern have the concrete semantics:

$$[\![P]\!](x,d) = [\![M]\!](x,d) \cdot \prod_{i=1}^{n} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!](x,d) dv_i,$$

Noting that because each $d[i]$ only depends on the corresponding v_i and $[\![M]\!]$ is only a function of u, we can simplify as

$$[\![P]\!](x,d) = [\![M]\!](u,d) \cdot \prod_{i=1}^{n} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u,v_i),d[i]) dv_i,$$

Efficient Optimization. AURA splits the expression as above and takes the logarithm to obtain:

$$[\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp) = [\![M]\!]^\sharp_{\log}(u^\sharp, d^\sharp) +^\sharp \sum_{i=1}^{n}{}^\sharp [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}((u^\sharp, v_i^\sharp), d[i]^\sharp)$$

Bounding $[\![M]\!]^\sharp_{\log}(u^\sharp, d^\sharp)$ does not depend on $d^\sharp$, thus the issue of $d^\sharp$ having large dimension n due to the n latents is circumvented. Each $[\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}((u^\sharp, v_i^\sharp), d[i]^\sharp)$ summand can also be abstractly interpreted using $d[i]^\sharp$ instead of $d^\sharp$, since each only depends on the i^{th} entry of $d^\sharp$. Thus the dimension of each optimization problem needed to bound $[\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}$ is 3 instead of the original $2n+1$. Hence we can reduce the dimension of the optimization problems from $\mathcal{O}(n)$ to $\mathcal{O}(1)$.

Efficient Integration We further leverage this insight to perform the integration for the normalizing constant C (full proof in Appendix A.3 below). We use the fact that for functions defined over disjoint variables s and t, then $\int_s \int_t f(s) \cdot g(t)\ dsdt = \int_s f(s)ds \cdot \int_t g(t)dt$. The normalizing constant C is:

$$C = \int_u du \int_{v_1} dv_1 ... \int_{v_n} dv_n [\![P]\!](x,d)$$

$$= \prod_{i=1}^{n} \int_u \int_{v_i} [\![M]\!](u,d) \cdot [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u,v_i),d[i]) du dv_i,$$

Which we can then over approximate using the abstract integration (Def. 4).

$$C \in \prod_{i=1}^{n}{}^\sharp \int_u^\sharp \int_{v_i}^\sharp [\![M]\!]^\sharp(u^\sharp, d^\sharp) \cdot [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp((u^\sharp, v_i^\sharp), d[i]^\sharp) du dv_i,$$

Hence we (abstractly) compute n easy $\mathcal{O}(1)$-dimensional integrals instead of a challenging $n+1$ dimensional integral. Thus AURA bypasses the curse of dimensionality for this pattern.

Derivations for Scalability to Large Numbers of Latent Variables Efficient Optimization. In abstract domain instead of abstractly evaluating the entire expression as AURA normally would (for lower dimensional problems), we could split this expression up as:

$$[\![P]\!]^\sharp(x^\sharp, d^\sharp) = [\![M]\!]^\sharp(u^\sharp, d^\sharp) \cdot^\sharp \prod_{i=0}^{n\ \sharp} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp((u^\sharp, v_i^\sharp), d[i]^\sharp)$$

and then take the logarithm:

$$[\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp) = [\![M]\!]^\sharp_{\log}(u^\sharp, d^\sharp) +^\sharp \sum_{i=0}^{n\ \sharp} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}((u^\sharp, v_i^\sharp), d[i]^\sharp)$$

Bounding $[\![M]\!]^\sharp_{\log}(u^\sharp, d^\sharp)$ does not actually depend on $d^\sharp$, thus the issue of $d^\sharp$ having high dimension due to the large number of latents is circumvented. Similarly each $[\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}((u^\sharp, v_i^\sharp), d[i]^\sharp)$ summand can be abstractly interpreted using $d[i]^\sharp$ instead of $d^\sharp$, since each one only depends on the i^{th} entry of $d^\sharp$. Thus the dimension of each optimization problem needed to bound $[\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^\sharp_{\log}$ is 3 instead of the original $2n + 1$. The resulting bounds of these intermediate subexpressions are recombined using interval arithmetic like in Fig. 13. Hence by splitting the expression this way and calling AURA on the sub-expressions, we can effectively reduce the dimension of the optimization problems to enable scalability to probabilistic programs with large numbers of latent variables and observations.

Efficient Integration We use this insight further when we have to perform the integration for the normalizing constant C. We leverage the fact that for functions defined over disjoint variables s and t, then $\int_s \int_t f(s) \cdot g(t) = \int_s f(s) \cdot \int_t g(t)$. The normalizing constant C is:

$$C = \int_u du \int_{v_1} dv_1 ... \int_{v_n} dv_n [\![P]\!](x, d)$$

$$= \int_u du \int_{v_1} ... \int_{v_n} [\![M]\!](u, d) \cdot \prod_{i=0}^{n} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u, v_i), d[i]) dv_i,$$

$$= \int_u [\![M]\!](u, d) du \cdot \int_{v_1} ... \int_{v_n} \prod_{i=0}^{n} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u, v_i), d[i]) dv_i,$$

$$= \int_u [\![M]\!](u, d) du \cdot \prod_{i=1}^{n} \int_{v_i} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u, v_i), d[i]) dv_i,$$

$$= \prod_{i=1}^{n} \int_u [\![M]\!](u,d)du \cdot \int_{v_i} [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u,v_i), d[i])dv_i,$$

$$= \prod_{i=1}^{n} \int_u \int_{v_i} [\![M]\!](u,d) \cdot [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]((u,v_i), d[i])dudv_i,$$

Which we can then over approximate using the abstract integration (Def. 4).

$$C \in \prod_{i=1}^{n\ \sharp} \int_u^{\sharp} \int_{v_i}^{\sharp} [\![M]\!]^{\sharp}(u^{\sharp}, d^{\sharp}) \cdot [\![v_i \sim dist_1(u); d[i] \sim dist_2(v_i)]\!]^{\sharp}((u^{\sharp}, v_i^{\sharp}), d[i]^{\sharp})dudv_i,$$

Hence we only have to (abstractly) compute n two-dimensional integrals instead of a complicated $n+1$ dimensional integral.

A.4 Nested Inference with AURA

AURA's normalized posterior bounds could be consumed by another tool (or interval analysis) to give certified bounds on *nested* inference posteriors. For simplicity, we assume no data perturbation, however everything follows identically in the case of data perturbations.

```
1
2  d = 1.8  # data point (outer inference)
3  m ~ { #posterior of inner is prior of outer inference
4      d = 1.3  # data point (inner inference)
5      m ~ uniform(0,1)  #prior
6      observe(d, normal(m,1))
7      return m
8  }
9
10 observe(d, normal(m,1))
```

Fig. 15: Nested Inference Model

Here we will denote P_{in} as the inner inference program:

$d = 1.3$; $m \sim$ uniform(0,1); observe(d, normal(m,1)); **return** m;

The unnormalized semantics of the inner posterior are:

$$[\![P_{in}]\!](m, d = 1.3) = f_{uniform}(m; 0, 1) \cdot f_{normal}(1.3; m, 1) = f_{uniform}(m; 0, 1) \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{(1.3-m)^2}{2}}$$

AURA can easily handle this inner posterior to compute sound (unnormalized) bounds $[\![P_{in}]\!]^\sharp(m, 1.3)$ which can then be (abstractly) integrated (as in Def. 4) to obtain the normalized posterior bounds $[\![P_{in}]\!]^\sharp_n(m, 1.3)$.

Here we will denote P_{out} as the *outer* inference program:

$$d = 1.8;\ m \sim [\![P_{in}]\!]_n(m, 1.3)\ ;\ \text{observe}(d,\ \text{normal}(m,1));\ \textbf{return}\ m;$$

The unnormalized semantics of the outer posterior distribution are:

$$[\![P_{out}]\!](m, d = 1.8) = [\![P_{in}]\!]_n(m, 1.3) \cdot f_{normal}(1.8; m, 1) = [\![P_{in}]\!]_n(m, 1.3) \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{(1.8-m)^2}{2}}$$

Which follows from the fact that for the outer inference problem, the prior distribution over m is the posterior distribution from the inner inference problem which is $[\![P_{in}]\!]_n(m, 1.3)$.

While in general we do know the exact form of $[\![P_{in}]\!]_n(m, 1.3)$, we have AURA's bounds $[\![P_{in}]\!]^\sharp_n$, which can then be used to compute bounds $[\![P_{out}]\!]^\sharp$ and ultimately bounds on $[\![P_{out}]\!]^\sharp_n$. For instance one can compute the normal distribution's expressions in interval arithmetic to obtain the following sound abstraction:

$$[\![P_{out}]\!]^\sharp([m_l, m_u], d = 1.8) = [\![P_{in}]\!]^\sharp_n([m_l, m_u], 1.3) \cdot^\sharp \frac{[1,1]}{[\sqrt{2\pi}, \sqrt{2\pi}]} [e^{-u}, e^{-l}]$$

where $[l, u] = -\frac{([1.8,1.8]-[m_l,m_u])^2}{[2,2]}$. These bounds, $[\![P_{out}]\!]^\sharp$, can then be integrated over the support of m (which is 0 to 1) using the abstract integration of Def. 4 to give:

$$[\![P_{out}]\!]^\sharp_n([m_l, m_u], 1.8) = \frac{[\![P_{out}]\!]^\sharp([m_l, m_u], 1.8)}{\int^\sharp [\![P_{out}]\!]^\sharp(m^\sharp, 1.8)}$$

Thus for $m \in [m_l, m_u]$, by our soundness result (Theorem 2) we know that:

$$[\![P_{out}]\!]_n(m, 1.8) \in [\![P_{out}]\!]^\sharp_n([m_l, m_u], 1.8).$$

Hence, AURA's bounds can be consumed by a separate analysis (such as an interval arithmetic analysis) for the purpose of computing nested inference bounds.

B Full Probabilistic Benchmark Set

Table 5: Benchmark program details. Symbols used: $\mathcal{B}$: Bernoulli, $\mathcal{B}_{\log}$: Bernoulli-Logit, $\mathcal{B}_{pro}$: Bernoulli-Probit, $\mathcal{U}$: Real Uniform, $\mathcal{N}$: Normal, β: Beta, $\mathcal{L}$: Laplace, $\mathcal{S}$: Logistic. Operators: $+$: mix of distributions, $\times$: product of densities. E.g., human_height's **Prior** is $\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$: a mixture of truncated normal distributions chosen from a Bernoulli distribution; likelihoods (**Lik**) are $\mathcal{N}^3$: three observations from normal distribution. In **PC** column, $\checkmark_{PC}$ shows Pseudoconcavity, $\checkmark_{LC}$ shows Log-Concavity, $\checkmark_{mix}$ for a mixture of Pseudoconcave functions.

Program	Prior	Lik	Description	PC
exponential	Exponential_t	N/A	An Exponential distribution	$\checkmark_{LC}$
gamma	Gamma_t	N/A	A Gamma distribution	$\checkmark_{LC}$
gaussian	$\mathcal{N}_t$	N/A	A Normal distribution	$\checkmark_{LC}$
coinBias	β	$\mathcal{B}^5$	Bias of coin using Beta-Bernoulli model [5]	$\checkmark_{LC}$
human_height	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^3$	Learning height with mixture prior [56]	$\checkmark_{mix}$
clinicalTrial	$\mathcal{B} \times (\beta + \beta)$	$\mathcal{B}_{\log}^{10}$	Logistic regression with mixed prior [70]	$\checkmark_{mix}$
altermu2	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Model with param symmetry [34]	$\checkmark_{LC}$
personality	$\mathcal{S}_t$	$\mathcal{B}_{\log}{}^{1000}$	Logistic regression for cheating study [29]	$\checkmark_{LC}$
reg_logistic	$\mathcal{N}_t$	$\mathcal{S}^{919}$	Linear regression with logistic likelihood [64]	$\checkmark_{LC}$
privacy	$\mathcal{N}_t^2$	$\mathcal{N}$	Regression estimates age from group mean [58]	$\checkmark_{LC}$
logistic	$\mathcal{U}^2$	$\mathcal{B}_{\log}^{100}$	Logistic regression [68]	$\checkmark_{LC}$
lightspeed	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Linear regression [68]	$\checkmark_{PC}$
anova_radon_n	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Hierarchical linear regression, non-predictive [68]	$\checkmark_{PC}$
IQStan	$\mathcal{U}^3$	$\mathcal{N}^3 \times \mathcal{N}^3$	Regression on two datasets with shared variance [41]	$\checkmark_{PC}$
reg_laplace	$\mathcal{U}^2 \times \mathcal{L}_t{}^2$	$\mathcal{N}^{919}$	Linear regression with Laplace priors [78]	$\checkmark_{LC}$
prior_mix	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^{10}$	Model with mixture prior [34]	$\checkmark_{mix}$
wells_probit	$\mathcal{U}^2$	$\mathcal{B}_{pro}{}^{500}$	Logistic model w. probit activation [68]	$\checkmark_{LC}$
timeseries	$\mathcal{U}^3$	$\mathcal{N}^{99}$	Timeseries model [68]	$\checkmark_{LC}$
unemployment	$\mathcal{U}^3$	$\mathcal{N}^{40}$	Linear Regression [68]	$\checkmark_{PC}$
robust_model†	$\mathcal{N}^{101}$	$\mathcal{N}^{100}$	Robust model with many local latent parameters [71]	$\checkmark_{mix}$

Table 5 presents the extended set of benchmarks. In addition to the benchmarks we studied in the main paper, it contains several other benchmarks that help us test alternative tools that compute posterior distributions. The studies of these models presented actionable insights to domain-experts in multiple communities:

- personality model and data come from a large-scale analysis performed by social scientists involving over 1000 data observations on students cheating on tests. That study has been cited over 130 times (GoogleScholar).
- anova_radon_n models the distribution of the level of radon gas found in homes across different counties based on real data collected by environmental scientists (details: `https://mc-stan.org/users/documentation/case-studies/radon_cmdstanpy_plotnine.html`).
- wells_probit model comes from a study that makes decisions about whether to change their source of drinking water for households in Bangladesh. The original study `http://www.stat.columbia.edu/~gelman/research/published/risk.pdf` was cited over 30 times.
- unemployment comes from a statistical study that aimed to precisely estimate the employment trends in US from the census data (the original study `https://www.jstor.org/stable/2669921` cited over 100 times).

In addition to those above, our additional benchmarks come from the Stan developers repository [62], which in turn took those benchmarks from various real-world statistics applications described in "Data Analysis Using Regression and Multilevel/Hierarchical Models" book by Gelman and Hill. Finally, several of our benchmarks present models known to challenge probabilistic inference (e.g., altermu2) and/or are difficult for tools that do inference with guarantees.

C Pseudo-Concavity of Unnormalized Posteriors for Benchmarks

In this section, we prove the pseudo-concavity of all our benchmarks from Appendix B. Namely, for each benchmark P, the function $[\![P]\!](x, d)$ is pseudo-concave with respect to their parameters and data. Users of AURA can apply the same conclusions or adopt the general proof strategy if their programs are similar to ours.

Our benchmarks are categorized into four distinct classes:

1. Benchmarks named "exponential", "gamma", and "gaussian" correspond to individual distributions. Their pseudo-concavity is given in Lemma 4, with details in Table 6.
2. Benchmarks such as "lightspeed", "anova_radon_n", "IQStan", and "unemployment" have a linear regression model structure. The pseudo-concavity of these benchmarks is established by Theorem 8.
3. Benchmarks "personality", "reg_logistic", "privacy", "logistic", "reg_laplace", "altermu2", "wells_probit", and "timeseries" are log-concave. Their log-concavity is established in Theorem 9, result from their composition of individual log-concave functions, which is rigorously proven using structural induction.
4. Benchmarks such as "human_height", "clinicalTrial", and "prior_mix" are composed of mixture models. The unnormalized posterior of these models is a summation of pseudo-concave functions. The pseudo-concavity of each component is given by one of the first three points.

Table 6 shows the details of log-concavity (LC) and pseudo-concavity (PC) of each distribution with respect to their parameters or data.

We first outline several essential lemmas that underpin the subsequent proof and formulation of the theorems:

Lemma 4. *(Log-Concavity and Pseudoconcavity of the Individual Distributions in Table 6) The Log-Concavity and Pseudoconcavity properties of the individual distributions (normal, uniform, beta, bernoulli, bernoulli-logit, bernoulli-probit, laplace, logistic, gamma, exponential) shown in Table 6 are well-known facts, as summarised in [3].*

Lemma 5. *(Product of Log-Concave Functions) Let $f_i(x_i)$ be a set of functions where each f_i is log-concave, then $g(x_1, ..., x_n) = \prod_i f_i(x_i)$ is LC w.r.t $x_1, \ldots, x_N$.*

Table 6: Log-Concavity and Pseudo-Concavity of Individual Distributions and Likelihoods. $\mathscr{P}$ denotes the power set. For example, $p_{\text{normal}}(x, \mu, \sigma)$ is LC w.r.t μ and x when both of them are variables, or w.r.t μ when x being constant, or w.r.t x with μ being constant.

Distribution	**Log-Concavity (LC) w.r.t**	**Pseudo-Concavity (PC) w.r.t**
$\prod_i p_{\text{normal}}(x_i \mid \mu, \sigma)$	$\mathscr{P}(\{x_1, ..., x_N, \mu\})$	$\mathscr{P}(\{x_1, ..., x_N, \mu, \sigma\})$
$p_{\text{normal}}(x \mid \mu, \sigma)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, \sigma\})$
$p_{\text{uniform}}(x \mid a, b)$	x	x
$p_{\text{beta}}(x \mid \alpha, \beta)$	x if $\alpha \geq 1 \wedge \beta \geq 1$	x if $\alpha \geq 1 \vee \beta \geq 1$
$p_{\text{bernoulli}}(x \mid p)$	p	p
$p_{\text{bernoulli-logit}}(x \mid \theta)$	θ	θ
$p_{\text{bernoulli-probit}}(x \mid \theta)$	θ	θ
$p_{\text{laplace}}(x \mid \mu, b)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, b\})$
$p_{\text{logistic}}(x \mid \mu, s)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, s\})$
$p_{\text{gamma}}(x \mid k, \theta)$	x if $k \geq 1$	x for any k
$p_{\text{exponential}}(x \mid \lambda)$	x	x

Lemma 6. *(Product of Pseudoconcave Functions) Let $f_i(x_i)$ be a set of functions where each f_i is log-concave, then $g(x_1, ..., x_n) = \prod_i f_i(x_i)$ is also pseudoconcave w.r.t $x_1, \ldots, x_N$.*

Lemma 7. *(Composition of a Log-Concave function with a linear function) If $f(x) : \mathbb{R}^m \to \mathbb{R}$ is Log-Concave and $A \in \mathbb{R}^{m \times n}$, then $g(y) : \mathbb{R}^n \to \mathbb{R}$ defined by $f(A(y))$ is Log-Concave.*

Lemma 8. *(Composition of a Quasiconcave function with a linear function) If $f(x) : \mathbb{R}^m \to \mathbb{R}$ is Quasiconcave and $A \in \mathbb{R}^{m \times n}$, then $g(y) : \mathbb{R}^n \to \mathbb{R}$ defined by $f(A(y))$ is Quasiconcave.*

Lemma 9. *(Composition of a Pseudoconvex function with a monotonic function) Let $f(x) : \mathbb{R}^m \to \mathbb{R}$ be Pseudoconvex, then for any non-decreasing function $g : \mathbb{R} \to \mathbb{R}$, then $g(f(x))$ is Pseudoconvex. Similarly is f is pseudoconcave, then $g(f(x))$ is pseudoconcave*

Lemma 10. *Let $f(x) : \mathbb{R}^m \to \mathbb{R}$ be Quasiconvex, then if $\nabla f \neq 0$, then f is Pseudoconvex. Likewise if f is quasiconcave and $\nabla f \neq 0$, then f is Pseudoconcave.*

Lemma 11. *Let $\alpha : \mathbb{R}^2 \to \mathbb{R}$ be defined as $\alpha(\sigma, c) = sqrt(-2\sigma^2(ln(\sigma^n(\sqrt{2\pi})^n)))$. α is concave with respect to σ for $\sigma > 0$, $n \in \mathbb{N}_+$, and $0 < c < \frac{1}{(\sqrt{2\pi}\sigma)^n}$.*

Proof. The second derivative of $\alpha(\sigma, c)$ is

$$\alpha''(\sigma, c) = \frac{-n\sigma^2(n - 2ln(c \cdot (\sqrt{2\pi}\sigma)^n))}{2\sqrt{2}(-\sigma^2 \cdot ln(c \cdot (\sqrt{2\pi}\sigma)^n))^{\frac{3}{2}}}$$

But the numerator is strictly negative while the denominator is strictly positive, hence

$$\alpha''(\sigma, c) < 0$$

□

Lemma 12. *If $f(x) : \mathbb{R}^m \to \mathbb{R}$ has convex upper contour sets for all levels, then f is quasiconcave.*

Lemma 13. *(Quasiconcavity of Composed Multi-variant Normal Likelihood) For simplicity, denote*

$$f_{normal}(\mu, \sigma, y_1, ..., y_n) = \prod_i p_{normal}(y_i|\mu, \sigma) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i-\mu)^2}{\sigma^2}}$$

Then $f_{normal}(\mu, \sigma, y_1, ..., y_n)$ is Quasiconcave on $[l_\sigma, u_\sigma]\times[l_{y_1}, u_{y_1}]\times...[l_{y_n}, u_{y_n}]$ where $l_\sigma > 0$.

Proof. We first define

$$h(\sigma, y_1, ..., y_n) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i)^2}{\sigma^2}}$$

and we prove it is quasiconcave. We algebraically convert the product of the gaussian pdfs (one for each observed data point) into a single exponential function:

$$\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i)^2}{\sigma^2}} = \frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}\frac{\sum_i(y_i^2)}{\sigma^2}}$$

Define the upper contour set at level c as

$$S_c = \{(\sigma, y_1, ..., y_n)|\frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}\frac{\sum_i(y_i^2)}{\sigma^2}} \geq c\}$$

We do the following algebraic rearrangements:

$$\frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}\frac{\sum_i(y_i^2)}{\sigma^2}} \geq c$$

$$\Rightarrow e^{-\frac{1}{2}\frac{\sum_i(y_i^2)}{\sigma^2}} \geq c \cdot \sigma^n(\sqrt{2\pi})^n$$

$$\Rightarrow -\frac{1}{2}\frac{\sum_i(y_i^2)}{\sigma^2} \geq ln(c\sigma^n(\sqrt{2\pi})^n)$$

$$\Rightarrow \sum_i(y_i^2) \leq -2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n))$$

$$\Rightarrow sqrt(\sum_i(y_i^2)) \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))$$

$$\Rightarrow \|y\|_2 \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))$$

Hence $S_c = \{(\sigma, y_1, ..., y_n) \ : \ \|y\|_2 \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))\}$.
We will actually use the following shorthand notation

$$\alpha(\sigma, c) = sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))\}$$

Hence $S_c = \{(\sigma, y_1, ..., y_n) \ : \ \|y\|_2 \leq \alpha(\sigma, c)\}$.

We now prove the convexity of S_c for $0 < c < \frac{1}{(\sqrt{2\pi}\sigma)^n}$. If $c \leq 0$, then any $(\sigma, y_1, ..., y_n)$ satisfies the constraint since $\frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}\frac{\sum_i (y_i^2)}{\sigma^2}} > 0$, and the set of *all* $(\sigma, y_1, ..., y_n)$ is a convex set. Likewise if $c \geq \frac{1}{(\sqrt{2\pi}\sigma)^n}$, then S_c is either a singleton or empty, both of which are convex sets.

Let $v, w \in S_c$, where by notational convenience $v = (v_1, ..., v_n, \sigma_1)$ and $w = (w_1, ..., w_n, \sigma_2)$. We will prove that their convex combination $\lambda v + (1-\lambda)w \in S_c$ for any $\lambda \in [0, 1]$.

Since $v \in S_c$ we know that $\|(v_1, ..., v_n)\|_2 \leq \alpha(\sigma_1, c)$, and furthermore $\lambda\|(v_1, ..., v_n)\|_2 \leq \lambda \cdot \alpha(\sigma_1, c)$. Similarly since $w \in S_c$, we know that $\|(w_1, ..., w_n)\|_2 \leq \alpha(\sigma_2, c)$ and likewise $(1-\lambda)\|(w_1, ..., w_n)\|_2 < (1-\lambda)\alpha(\sigma_2, c)$. Hence

$$\lambda\|(v_1, ..., v_n)\|_2 + (1-\lambda)\|(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c)$$

Since $\lambda, (1-\lambda) \geq 0$

$$\|\lambda(v_1, ..., v_n)\|_2 + \|(1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c)$$

By triangle inequality (since $\|\cdot\|_2$ is a norm)

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \|\lambda(v_1, ..., v_n)\|_2 + \|(1-\lambda)(w_1, ..., w_n)\|_2$$

Hence

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c)$$

By the concavity of $\alpha(\sigma, c)$ (Lemma 11) we know by Jensen's inequality that

$$\lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c)$$

Hence

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c)$$

Or just

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c)$$

This implies the point $(\lambda v_1 + (1-\lambda)w_1, ..., \lambda v_n + (1-\lambda)w_n, \lambda\sigma_1 + (1-\lambda)\sigma_2)$ is in S_c, hence the upper contour sets S_c are convex, thus by Lemma 12 we have quasiconcavity of h.

Furthermore, since f_{normal} is $h(A(x_1, ..., x_n, \mu, \sigma)$ where $A(x_1, ..., x_n, \mu, \sigma)$ is the linear function defined as

$$A(x_1, ..., x_n, \mu, \sigma) = (x_1 - \mu, ..., x_n - \mu, \sigma)$$

And since h is already proved to be quasiconcave and quasiconcave functions are closed under composition with linear functions (Lemma 8), then f_{normal} is quasiconcave. □

Lemma 14. $f_{normal}(\mu, \sigma, y_1, ..., y_n)$ *is* ***Pseudoconcave****.*

Proof. By Lemma 13, we know that f_{normal} is at least quasiconcave, but since we have that $\frac{\partial}{\partial\sigma} f_{normal}(\mu, \sigma, y_1, ..., y_n) \neq 0$, then $\nabla f_{normal}(\mu, \sigma, y_1, ..., y_n)$ is never zero, thus by Lemma 10 it is actually pseudoconcave (though not fully concave) □

Lemma 15. $\log(f_{normal}(\mu, \sigma, y_1, ..., y_n))$ *is* ***Pseudoconcave****.*

Proof. Since log is a monotonic, non-decreasing function, the composition of log with a pseudoconcave function (such as f_{normal}) is still pseudoconcave by lemma 9 □

Theorem 8 (Pseudoconcavity of Linear Regression Programs). *The posterior distribution of Bayesian Linear Regression with the general pattern shows in Figure 16 is Pseudoconcave.*

```
1 m ~ uniform(l_m, u_m)
2 b ~ uniform(l_b, u_b)
3 s ~ uniform(l_s, u_s)
4 y1 ~ normal(m*v1+b, s)
5 ...
6 yn ~ normal(m*vn+b, s)
7 observe(y1, d1)
8 ...
9 observe(yn, dn)
```

Fig. 16: Linear Regression Code

Proof. All of our linear regression benchmarks have the code form of Fig. 16 and thus have uniform priors over all parameters (slope, intercept, σ), hence the expression for $[\![P]\!](x, d)$ will be

$$[\![P]\!](x, d) = \frac{1}{u_m - l_m}\frac{1}{u_b - l_b}\frac{1}{u_s - l_s} f_{normal}(b, s, d_1 - mv_1, ..., d_n - mv_n)$$

where $x = (b, s)$ and $d = (d_1, ..., d_n)$. Since all v_i are constants, this is just the composition of a linear transformation with f_{normal} (which is already pseudo-concave) hence $[\![P]\!](x, d)$ is pseudoconcave and thus so is $[\![P]\!]_{\log}(x, d)$. □

Theorem 9 (Log-Concavity in Branch-free Programs with Log-Concave Distributions). *For a program P written in our language (as shown in Figure 6), if P does not contain branching, and assuming that each individual distribution in the program is Log-Concave as classified in Table 6, then P is also Log-Concave.*

Proof. We prove Theorem 9 using structural induction on the concrete semantics rules (Figure 8). The rules can be classified into three categories: statements, arithmetic expressions, and distribution expressions. The proof has three parts:

1. For arithmetic expressions $[\![E + E]\!]$, $[\![E - E]\!]$, $[\![cE]\!]$, $[\![\mathrm{x}_j]\!]$: if all operands are linear, and since these expressions are linear with respect to their operands, the resulting expression will also be linear. In structural induction, the base cases are a singleton parameter or a data point, both of which are directly linear functions w.r.t parameters or data themselves. Then, for each of these rules, by assuming their operand sub-expressions evaluate to linear functions, the $+$, $-$, and constant factor result in linear functions.
2. For the distribution expression $[\![dist(E_1, \ldots, E_N)]\!](x, d) = p_{dist}\,(u; [\![E_1]\!](x, d), \ldots, [\![E_N]\!](x, d))$: by the assumption of this lemma, p_{dist} is Log-Concave w.r.t its parameters/data. Then by Lemma 7, and the conclusion on arithmetic expressions that E_i must be linear functions, we have $[\![dist(E_1, \ldots, E_N)]\!](x, d)$ being Log-Concave.
3. For statements $[\![M; M]\!]$, $[\![D; D]\!]$, $[\![M; D]\!]$, $[\![\mathrm{observe}(\mathrm{Dist}, d_i)]\!](x, d)$ or $[\![\mathtt{x}_i \sim \mathrm{Dist}]\!](x)$, we prove they result in Log-Concave functions given that the sub-statements give Log-Concave functions. Again by structural induction:
   - The base cases are the single statements for likelihood or prior: $[\![\mathrm{observe}(\mathrm{Dist}, d_i)]\!](x, d) = [\![\mathrm{Dist}]\!](x, d) \circ d[i]$ or $[\![\mathtt{x}_i \sim \mathrm{Dist}]\!](x) = [\![\mathrm{Dist}]\!](x, d) \circ x[i]$. Both statements evaluate to the distribution expression $[\![dist(E_1, \ldots, E_N)]\!](x, d)$, which is Log-Concave as shown above.
   - Then, the inductive step utilizes the two compositional properties of Log-Concave functions outlined in Lemma 5. For sequencing statements $[\![M; M]\!]$, $[\![D; D]\!]$, and $[\![M; D]\!]$, if a preceding density is Log-Concave, and it is multiplied with a subsequent prior/likelihood function that is also Log-Concave (by inductive hypothesis), the resulting function remains Log-Concave (by Lemma 5). Furthermore, for expressions (Lemma 7).

This confirms that the Log-Concavity is preserved under all the statement rules in Figure 8, except for the branching rule. Therefore, $[\![P]\!](x, d)$ is Log-Concave. □

Lemma 16 (Pseudo-Concavity of Log Unnormalized Posteriors). *Given a set of distributions where their corresponding prior or likelihood PDFs are either LC or PC, the log unnormalized posterior $[\![P]\!]_{\log}(x, d)$, which is the sum of log PDFs of these distributions, is pseudo-concave.*

Proof. Let $[\![P]\!]_{\log}(x,d) = \sum_i \log f_i(x,d)$ where each $f_i(x,d)$ is the PDF of a distribution that is either LC or PC. By Lemma 6, the sum $\sum_i \log f_i(x,d)$ retains the property of being LC or PC. Then $[\![P]\!]_{\log}(x,d)$ is pseudo-concave. □

D Experimental Setup Details

D.1 Precision Metrics

We define two precision metrics for the analysis of program P. From the analysis, we first establish a lower bound function p_l and an upper bound function p_u for any value of the latent parameter x: For a fixed dataset:$p_l(x) = l$ and $p_u(x) = u$ are obtained from the analysis as $[l,u] = [\![P]\!]^\sharp_n(x_i^\sharp, d)$ for x within any $\gamma(x_i^\sharp)$. For perturbed datasets:$p_l(x) = l$ and $p_u(x) = u$ are defined as $[l,u] = [\![P]\!]^\sharp_n(x_i^\sharp, d^\sharp)$ for x within any $\gamma(x_i^\sharp)$.

Total Variation Distance (TVD). TVD [60] is a widely-used metric that intuitively measures the *area* between two distribution density functions. For two univariate probability density functions, p and q for a continuous random variable $x \in \mathbb{R}$, $\text{TVD}_{pq} = \frac{1}{2}\int |p(x) - q(x)|\, dx$. To measure the precision of the bounds on posterior distributions, we define the TVD for the bounds as:

$$\text{TVD}_x = \frac{1}{2}\int |p_l(x) - p_u(x)|\, dx.$$

The TVD between the lower and upper bounds also represents the maximum TVD for any two probability density functions confined within these bounds. If the program contains multiple parameters, the overall TVD is reported as: $\text{TVD} = \frac{1}{M}\sum_{j=1}^{M} \text{TVD}_{x_j}$, averaged across the parameters $x_1, x_2, \ldots, x_M$, where each TVD_{x_j} is computed based on the marginal density function of each x_j after computing the bounds on their joint density function.

Absolute Difference on Parameter Means (ADM). ADM [48, 71] measures the absolute difference between the expected values of parameters within two distributions. For two probabilistic density functions p and q on a random variable $x \in \mathbb{R}$, ADM is $ADM_{pq} = |\mathbb{E}_p(x) - \mathbb{E}_q(x)|$, where $\mathbb{E}_p(x) = \int x \cdot p(x) dx$ and similarly for q. We use ADM to quantify the precision of the bounds. Formally, given the bounds $p_l(x)$ and $p_u(x)$ on the posterior density function, we consider all the posterior density functions between these bounds, denoted as $\mathcal{P} = \{p' : \forall x. p_l(x) \le p'(x) \le p_u(x)\}$. The ADM then becomes the maximal absolute difference in the expectations between any function in $\mathcal{P}$ and the true expectation:

$$ADM_x = \max_{P \in \mathcal{P}} \left|\mathbb{E}_p(x) - \mathbb{E}_{p_{truth}}(x)\right|.$$

To get $\mathbb{E}_{p_{truth}}(x)$, we use Stan's NUTS sampling to obtain 400,000 samples and take the sample mean as the true mean. We report the program's $ADM = \frac{1}{M}\sum_i^M ADM_{x_j}$, averaged across all parameters.

The TVD and ADM for bounds obtained from GuBPI and the interval analysis are analogous.

E Comparing AURA with GubPI and PSI on Bounding Single Posteriors

Table 5 in Appendix B presents the extended set of benchmarks. In addition to the benchmarks we studied in the main paper, it contains several other benchmarks that help us test alternative tools that compute posterior distributions.

Baselines. For certifying bounds on posterior distributions (without data perturbation), we compare AURA with two baselines: GuBPI [5], the start-of-the art tool for obtaining sound bounds on single posteriors and PSI [20], which leverages symbolic analysis to determine the exact posterior. For both baselines we use their most recent versions. For GuBPI we report the most precise result and their computation times, based on a grid search across all configurable GuBPI parameters[1]. We run GuBPI and AURA with the same number of splits.

.

Metrics. See Section D.1.

Baseline Setup Details. We use the most recent version of GuBPI [27] and report the most precise solutions and their computation times, based on a grid search across all configurable GuBPI parameters and we run GuBPI with the same number of splits as AURA. The parameters include method ("boxes","linear"), scoring precision (0.001-0.1), variable precision (0.01-1), the depth of the symbolic exploration (10-1000) and splits in the "boxes" method (200-800000). We omit configurations under which GuBPI implementation is not sound due to disconnected bounding boxes on continuous curves (Appendix F presents an example). Since GuBPI does not work with infinite support, we use the precision enhancing splitting strategy to compute the same bounded interval for GuBPI and AURA. The time for this step is negligible ($<$0.01s) and is included in AURA's run time but not GuBPI's. We exclude the one-time cost to initialize the GPU from AURA's run time. For PSI, we use the most recent version [59] with default configurations. We run AURA's abstract interpretation until the gradient ascent converges, which we define as the point at which the density value is repeated within the machine epsilon.

Precision Metrics Experimental Setup. We run AURA and all the other tools on a AMD 4.2 GHz machine with 32 cores with 2 NVIDIA RTX A5000 GPUs. For the experiments where we compare to GubPI and PSI, a fair comparison since GuBPI and PSI which only utilize a single core, we present the AURA timing on a single core as well.

[1] The parameters include method ("boxes","linear"), scoring precision (0.001-0.1), variable precision (0.01-1), the depth of the symbolic exploration (10-1K) and splits in the "boxes" method (200-800K). We omit the configurations under which GuBPI implementation is unsound due to disconnected bounding boxes on continuous curves (Appendix F presents an example).

Table 7: AURA Execution Time and Precision of AURA compared to baselines (using 200 quantization splits). We run AURA and other tools on a single core CPU. For PSI, we denote timeout (>90 min) as "t.o.", unevaluated integrals as "inte". We compute the geometric mean of AURA's speedup over the baselines (as $\times^*$) only on the benchmarks that work for the baseline (on the single-core CPU).

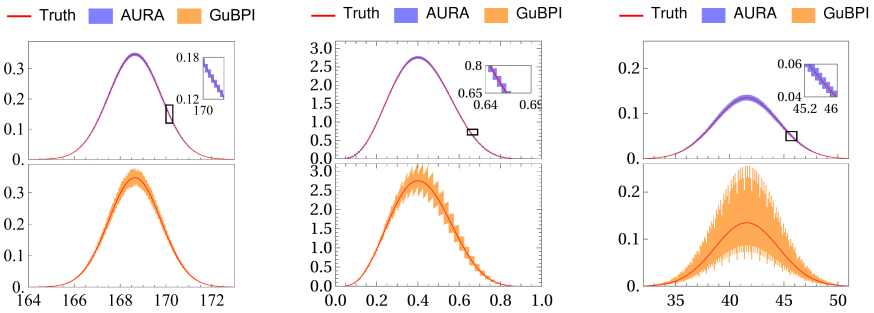
Program	**TVD**		**ADM**		**Time (s)**			
	AURA	GuBPI	AURA	GuBPI	AURA (GPU)	AURA (CPU)	GuBPI	PSI
exponential	0.02	-	0.05	-	0.04	0.01	-	0.02
gamma	0.02	-	0.03	-	0.10	0.03	-	0.03
gaussian	0.02	0.02	0.03	0.04	0.03	0.01	0.10 (9.8×)	0.02
coinBias	0.01	0.06	0.01	0.06	0.04	0.01	170.93 ($1.3\times10^4\times$)	0.15
human_height	0.02	0.04	5.14	13.97	0.06	0.04	1.33 (29.7×)	0.13
clinicalTrial	0.02	∞	0.02	∞	0.10	0.03	-	1.53
altermu2	0.11	16.33	0.20	88.01	0.04	0.20	132.84 (668.0×)	12.17
personality	0.02	-	0.00	-	0.04	0.05	-	-
reg_logistic	0.02	-	0.04	-	2.47	0.94	-	-
privacy	0.03	0.38	2.57	31.40	0.26	0.23	18.69 (81.0×)	inte
logistic	0.03	-	0.09	-	0.13	2.77	-	t.o.
lightspeed	0.03	5.0×10^5	1.28	2.7×10^7	0.07	0.76	93.40 (122.6×)	inte
anova_radon_n	0.03	1.1×10^8	0.06	1.2×10^8	1.66	1.55	93.98 (60.5×)	inte
IQStan	0.04	3.3×10^5	5.43	7.0×10^7	3.85	183.48	71.17 (0.4×)	inte
reg_laplace	0.04	-	0.08	-	4.05	5.02	-	t.o.
prior_mix	0.04	1.0×10^6	1.10	2.5×10^6	0.16	0.07	12.40 (178.6×)	inte
wells_probit	0.06	-	0.05	-	0.36	29.67	-	-
timeseries	0.13	∞	0.21	∞	33.83	873.96	-	inte
unemployment	0.16	∞	0.33	∞	28.90	833.43	-	inte
Geometric Mean	0.03	267.15	0.14	3334.35	0.32	0.66	19.56 ($77.9\times^*$)	0.17 ($6.6\times^*$)

E.1 AURA Precision and Execution Time Comparison

Table 7 presents the results of AURA and two other baseline tools, GuBPI and PSI. We run AURA on both a single core CPU and a GPU, and all the other tools on a single core CPU. Columns 2-5 present the *precision* of the bounds of AURA and GuBPI with Total Variation Distance (**TVD**) and Absolute Difference on Parameter Means (**ADM**). Columns 6-9 (**Time (s)**) show the run times.

Precision of Bounds. AURA obtains highly precise bounds for all benchmarks on average (geomean) 0.03 in TVD and 0.14 in ADM (lower is better). AURA outperforms GuBPI in precision across all benchmarks: only 3 GuBPI instances have lower error than AURA's worst error in either metric. These results highlight the difference between AURA's gradient-based method and GuBPI's interval abstraction for programs with non-trivial number of data points (10-1000). PSI is guaranteed to produce exact results, but is only able to compute results for seven simple benchmarks, all of which have under 40 data points and contain only simple posterior distribution expressions.

Execution Time. Using a GPU, AURA solved 13 out of 20 benchmarks within 1 second, and all the benchmarks within one minute. On a single core CPU, AURA solves 13 out of 20 benchmarks within 1s, and is faster on 10 benchmarks than AURA on GPU. This speed difference is because these programs are too



(a) human_height (b) coinBias (c) privacy

Fig. 17: Visualization of AURA and GuBPI Bounds

small to take advantage of the GPU: the GPU version of AURA is 25-50x faster than the CPU version for the four largest programs.

On the programs GuBPI can solve, AURA is much faster than GuBPI, with geometric mean 125.7× on GPU, and 77.9× on the single core CPU.

Analysis Examples. Figure 17 presents the posterior density bounds derived by AURA and GuBPI for three benchmarks. The x-axis shows the parameter values; the y-axis shows the posterior probabilistic density. We show the ground truth with a red line and bounded area derived from AURA and GuBPI with blue and orange rectangles, respectively. The ground truth is derived manually and symbolically with the aid of Mathematica's numerical integration. From the plots, AURA is significantly more precise than GuBPI. AURA's bounds closely follow the ground truth line (see the enlarged area). This precision is mainly because AURA uses optimization to find the tightest abstractions. AURA also improves numerical stability by using the log density, avoiding the over/underflow issues that cause GuBPI to report extremely large numbers (TVD $> 10^5$) in five benchmarks. Unlike GuBPI, which uniformly splits intervals, AURA creates nearly equal-area bounding boxes, adjusting splits based on density function shape (details in Appendix 8).

Illustration of Unsound Results from GuBPI. GuBPI, under certain configurations, is not sound. In our experience, our hyper-parameter sweeping will ignore those unsound results. Figure 18 shows an example when GuBPI become unsound. For illustration purpose, we run GuBPI with 20 splits, while the same unsoundness would occur with any number of splits. On the plot, each orange box shows GuBPI's bounds on that interval, and the red line shows the ground truth. At several places the orange bounding boxes failed to cover the ground truth. For example, the point highlighted with a blue dot is (mu $= -1.02$) which is in the interval of (-1.5,-1) (i.e. the box to the left), but its ground truth is above the upper bound GuBPI derived for this interval.

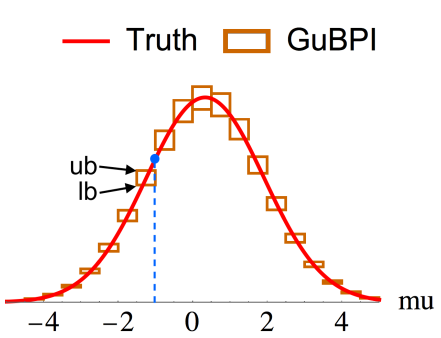


Fig. 18: Unsound Result from GuBPI

```
# splits 1000
# depth 1000
# discretization -5 5 0.500000
# epsilonScore 0.05
# epsilonVar 0.05

let data = [-5.4, 2.1, 6.7, 0.6, -1.1,
      -1.3, 0.02, 1, -3.7, 4.4] in
let mu = sample uniform(-5, 5) in
letrec iterate xs = match xs
  | [] -> mu
  | [x | xs] ->
      score(pdfnormal(mu, 5, x)); iterate
          xs
    in iterate data
```

Fig. 19: GuBPI Program which results in Figure 18

E.2 Scaling to Larger Datasets

Figure 20 presents the impact of data size on AURA results. We increase the data size for all 15 applicable programs from 200 to 5000 (simulated from the same distribution as the original data), without applying data perturbation. We run AURA on a GPU with 200 splits. The left y-axis shows AURA's execution time in blue; the right y-axis shows precision in orange, both representing the geometric mean across benchmarks. AURA maintains nearly constant precision across varying data sizes, while GuBPI fails to scale and already gives imprecise bounds (TVD $> 10^5$) at around 10 data points. Also, AURA's time increases linearly with data size, and the number of data points AURA can compute with depends primarily on the size of the GPU memory. Beyond 2000 data points, the memory of a single A5000 GPU is insufficient for a few benchmarks, and thus we distributed the computation across two GPUs for data sizes ranging from 2000-5000 for all benchmarks. Spliting to two GPUs introduced a small shift in the execution time around 2000 data points, but execution time increase remained consistently linear before and after this split, which also demonstrates AURA's scalability in leveraging multiple GPUs. In all cases, the impact on the precision is minimal.

E.3 Other Ablation Studies

Varying Quantization Splits. Figure 21 presents the geometric mean precision and performance trade-off, as a function of the number of quantization splits (#splits). AURA's error decreases exponentially, while the computation time initially decreases before starting to increase. More splits help the optimization converge in fewer steps since the interval regions that gradient ascent explores are smaller, albeit at the cost of increased computation per step from having more intervals. For our benchmarks, 200 splits gives a reasonable balance between run time and precision.

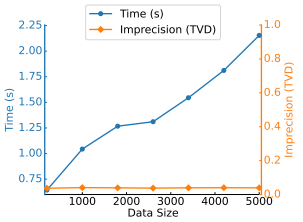


Fig. 20: Varying #Data (no Pert.)

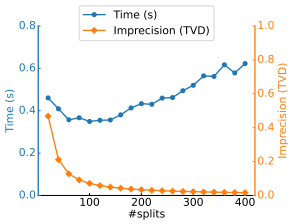


Fig. 21: Varying #Splits

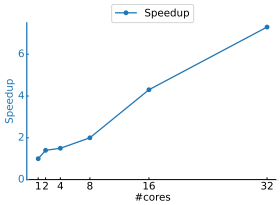


Fig. 22: Varying #CPUs

Scaling to Multiple CPUs. Fig. 22 shows the average speedup when running AURA on different number of cores using original data. The speedup of our PyTorch-based implementation (with no additional performance-enhancing optimizations) is approximately a linear function of the number of cores. AURA is the first tool for PP inference with guaranteed bounds to run in parallel.

Numerical data formats. For each benchmark, we calculated the posterior bounds using both FP32 and FP64 precision. The overall numerical discrepancy between FP32 and FP64 was less than 10^{-6}. Specifically, the geometric mean of differences in the posterior bounds across all intervals and benchmarks was $4.7 \cdot 10^{-7}$. The geomean execution time overhead when using FP64 compared to FP32 on a GPU is only 1.1x. Thus, our analysis confirms that using FP32 precision gives accurate outcomes across all benchmarks.