RESEARCH-ARTICLE

Piecewise Analysis of Probabilistic Programs via #-Induction

TENGSHUN YANG, Chinese Academy of Sciences, Beijing, Beijing, China

SHENGHUA FENG, Chinese Academy of Sciences, Beijing, Beijing, China

HONGFEI FU, Shanghai Jiao Tong University, Shanghai, China

NAIJUN ZHAN, Peking University, Beijing, China

JINGYU KE, Shanghai Jiao Tong University, Shanghai, China

SHIYANG WU, Shanghai Jiao Tong University, Shanghai, China

Published: 08 January 2026
Accepted: 06 November 2025
Received: 10 July 2025

Piecewise Analysis of Probabilistic Programs via k-Induction

TENGSHUN YANG*, Institute of Software Chinese Academy of Sciences, China
SHENGHUA FENG*, Institute of Software Chinese Academy of Sciences, China
HONGFEI FU†, Shanghai Jiao Tong University, China
NAIJUN ZHAN, Peking University, China and Zhongguancun Laboratory, China
JINGYU KE, Shanghai Jiao Tong University, China
SHIYANG WU, Shanghai Jiao Tong University, China

In probabilistic program analysis, quantitative analysis aims at deriving tight numerical bounds for probabilistic properties such as expectation and assertion probability. Most previous works consider numerical bounds over the whole program state space monolithically and do not consider piecewise bounds. Not surprisingly, monolithic bounds are either conservative, or not expressive and succinct enough in general. To derive better bounds, we propose a novel approach for synthesizing piecewise bounds over probabilistic programs. First, we show how to extract useful piecewise information from latticed k-induction operators, and combine the piecewise information with Optional Stopping Theorem to obtain a general approach to derive piecewise bounds over probabilistic programs. Second, we develop algorithms to synthesize piecewise polynomial bounds, and show that the synthesis can be reduced to bilinear programming in the linear case, and soundly relaxed to semidefinite programming in the polynomial case. Experimental results show that our approach generates tight piecewise bounds for a wide range of benchmarks when compared with the state of the art.

CCS Concepts: • **Theory of computation** → **Probabilistic computation**; • **Mathematics of computing** → **Markov processes**; • **Security and privacy** → **Logic and verification**.

Additional Key Words and Phrases: probabilistic programs, k-induction, quantitative analysis

ACM Reference Format:
Tengshun Yang, Shenghua Feng, Hongfei Fu, Naijun Zhan, Jingyu Ke, and Shiyang Wu. 2026. Piecewise Analysis of Probabilistic Programs via k-Induction. *Proc. ACM Program. Lang.* 10, POPL, Article 67 (January 2026), 31 pages. https://doi.org/10.1145/3776709

1 Introduction

Probabilistic programming [29, 36, 51] is a programming paradigm that extends classical programming languages with probabilistic statements such as sampling and probabilistic branching, and provides a powerful modelling mechanism for randomized algorithms [6], machine learning [12], reliability engineering [14], etc. Therefore, analysis of probabilistic programs is becoming increasingly significant, and attracting more and more attention in recent years.

*Equal Contribution
†The corresponding author

Authors' Contact Information: Tengshun Yang, Institute of Software Chinese Academy of Sciences, Beijing, China, yangts@ios.ac.cn; Shenghua Feng, Institute of Software Chinese Academy of Sciences, Beijing, China, fengshenghua@iscas.ac.cn; Hongfei Fu, Shanghai Jiao Tong University, Shanghai, China, jt002845@sjtu.edu.cn; Naijun Zhan, School of Computer Science & Key Laboratory of High Confidence Software Technology, Peking University, Beijing, China and Zhongguancun Laboratory, Beijing, China, znj@ios.ac.cn; Jingyu Ke, Shanghai Jiao Tong University, Shanghai, China, windocotber@gmail.com; Shiyang Wu, Shanghai Jiao Tong University, Shanghai, China, bravendergclio@gmail.com.

ACM 2475-1421/2026/1-ART67
https://doi.org/10.1145/3776709

In this work, we consider the quantitative analysis problem that aims at automated approaches that derive quantitative bounds for probabilistic programs. Common quantitative properties include expected runtime [1, 27, 33, 34], expected resource consumption [44, 52, 54], sensitivity [2], assertion probabilities [19, 50, 53], and so forth. Most existing works focus on deriving numerical bounds instead of solving the semantic equations exactly, as the latter is impossible theoretically in general. In the literature, various approaches have been proposed to address the quantitative analysis problem, including template-based constraint solving [15, 16, 18, 30], trace abstraction [49], sampling [46], etc. Most of these approaches consider to synthesize a monolithic bound over the whole state space of a probabilistic program of interest, and have the following disadvantages: First, a monolithic bound is either too conservative (e.g., only very coarse bounds exist) or not succinct enough (e.g., although tight monolithic bounds exist, the tightness usually requires complicated polynomials with higher degree). Second, it may be even worse that no monolithic polynomial bounds exist.

It is straightforward to observe that piecewise bounds are more accurate than monolithic bounds. Moreover, a recent work [9] demonstrates that probabilistic program analysis requires piecewise feature. However, the synthesis of piecewise bounds for probabilistic programs is not well investigated in the literature. To our best knowledge, a handful relevant work is by [10]. They propose an approach for generating (piecewise) invariants to *verify* user-provided linear bounds for probabilistic programs with discrete probabilistic choices, which is based on Counterexample-Guided Inductive Synthesis (CEGIS) and template refinement. Another relevant work is [5] that proposes a data-driven approach that can synthesize piecewise (sub-)invariants over probabilistic programs with discrete probabilistic choices. Their approach prefers a suitable list of numerical program features (such as multiplication expressions over variables), which requires prior knowledge of the program or user's assistance. Both of these related works require a bound to be verified as an additional program input when synthesizing (super-/sub-) invariants.

In this work, we propose a novel automated approach that synthesizes piecewise polynomial bounds for probabilistic programs with discrete probability choices without user-provided bounds or piecewise features to assist the derivation of the piecewise bound. The challenges are that (a) We need to resolve a good criterion to partition the state space of a probabilistic program into multiple parts in order to derive the form of the target piecewise bound. (b) We need to devise efficient algorithms to synthesize piecewise bounds given the criterion. Our detailed contributions to address these challenges are as follows.

To address the first challenge, we consider latticed k-induction operators [11, 39]. k-induction is a powerful proof tactic in software and hardware verification that generalizes normal inductive reasoning [22, 23, 37, 48]. While standard induction assumes the property holds after a single step, k-induction extends this approach by considering sequences of k steps, allowing properties to be established even when a single-step induction is impossible. Latticed k-induction [11, 39] further adapts k-induction to lattices and has been applied to probabilistic program analysis [11]. In this work, we leverage latticed k-induction as the central criterion for partitioning the whole state space into multiple parts, and the Optional Stopping Theorem (see the classical Optional Stopping Theorem (OST) [56, Chapter 10]) provides the theoretical foundation of our methods. We develop a novel combination of operators from latticed k-induction and OST, which enables the synthesis of both upper and lower bounds for quantitative properties for probabilistic programs without requiring a global bound of program values (such as non-negativity in [10, 11, 39]). Importantly, the combination is non-trivial: the classical OST is insufficient in our setting, and we rely on an extended version of OST [55] to establish our results. Additionally, as a by-product, we slightly extend existing latticed k-induction operators.

To address the second challenge, we propose novel algorithms for synthesizing piecewise linear and polynomial bounds w.r.t our combination of latticed k-induction and OST. It is important to observe that the latticed k-induction involves *minimum/maximum* operation, and therefore increases the difficulty to synthesize a bound algorithmically. We first introduce a key improvement in time efficiency on the unrolling of the k-induction operators. Then, we show that the synthesis of piecewise linear bounds can be equivalently transformed into a bilinear programming problem. A bilinear programming problem is that the variables can be decomposed into two groups so that within each group of variables the constraints are linear, and is a special non-convex programming that admits efficient constraint solving [40]. Finally, since even on the linear benchmarks we require piecewise polynomials to upper/lower bound the quantitative properties, we show that the synthesis of the more general piecewise polynomial bounds can be soundly relaxed to semidefinite programming. Experimental results over an extensive set of benchmarks that includes various benchmarks from the literature show that our approach is capable of generating tight or even accurate piecewise bounds and can solve benchmarks that previous approaches could not handle.

Technical Contributions. Approaches with latticed k-induction has inherent combinatorial explosion [11, 39]. To address the difficulty, we propose two techniques. The first is a heuristic selection of a small part of the functions in the minimum operation of latticed k-induction. The second is the sound relaxation that over-approximates the minimum operation with convex combination.

2 Preliminaries

In this section, we briefly review probability theory, define the k-induction operators, present the probabilistic loops under consideration, and finally formulate the problem of interest.

2.1 Probability Theory and Martingales

Consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the sample space, $\mathcal{F}$ is a σ-algebra on Ω and $\mathbb{P} : \mathcal{F} \to [0,1]$ is a probability measure on the measurable space $(\Omega, \mathcal{F})$. A *random variable* is an $\mathcal{F}$-measurable function $X : \Omega \to \mathbb{R} \cup \{+\infty, -\infty\}$, i.e., a function satisfying that for all $d \in \mathbb{R} \cup \{+\infty, -\infty\}$, $\{\omega \in \Omega : X(\omega) \leq d\} \in \mathcal{F}$. The *expectation* of a random variable X, denoted by $\mathbb{E}(X)$, is the Lebesgue integral of X w.r.t. $\mathbb{P}$, i.e., $\mathbb{E}(X) = \int X d\mathbb{P}$. A *filtration* of the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an infinite sequence $\{\mathcal{F}_n\}_{n=0}^{\infty}$ such that for every n, the triple $(\Omega, \mathcal{F}_n, \mathbb{P})$ is a probability space and $\mathcal{F}_n \subseteq \mathcal{F}_{n+1} \subseteq \mathcal{F}$. A *stopping time* w.r.t. $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a random variable $\tau : \Omega \to \mathbb{N} \cup \{0, \infty\}$ such that for every $n \geq 0$, the event $\{\tau \leq n\} \in \mathcal{F}_n$, i.e., $\{\omega \in \Omega : \tau(\omega) \leq n\} \in \mathcal{F}_n$. Intuitively, τ is interpreted as the time at which the stochastic process shows a desired behavior. A *discrete-time stochastic process* is a sequence $\Gamma = \{X_n\}_{n=0}^{\infty}$ of random variables in $(\Omega, \mathcal{F}, \mathbb{P})$. The process Γ is adapted to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$, if for all $n \geq 0$, X_n is a random variable in $(\Omega, \mathcal{F}_n, \mathbb{P})$. A discrete-time stochastic process $\Gamma = \{X_n\}_{n=0}^{\infty}$ adapted to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a *martingale* (resp. supermartingale, submartingale) if for all $n \geq 0$, $\mathbb{E}(|X_n|) < \infty$ and it holds almost surely that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$ (resp. $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n$, $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \geq X_n$). See [56] for more details about martingale theory. Applying martingales for probabilistic programs analysis is well-studied [15, 16, 19].

2.2 k-Induction Operators

To present k-induction operators, we briefly review lattice theory. Informally, a lattice is a partially ordered set $(E, \sqsubseteq)$ (where E is a set and $\sqsubseteq$ is a partial order on E) equipped with a *meet* operation $\sqcap$ and a *join* operation $\sqcup$. Given two elements $u, v \in E$, the meet $u \sqcap v$ is defined as the infimum of $\{u, v\}$ and dually the join $u \sqcup v$ is defined as the supremum of $\{u, v\}$. A partially ordered set $(E, \sqsubseteq)$ is a *lattice* if for any $u, v \in E$, we have that both $u \sqcap v$ and $u \sqcup v$ exist. Given a lattice $(E, \sqsubseteq)$, we say

$$C ::= \text{skip} \mid x := e \mid x :\approx \mu \mid C;C \mid \{C\}\ [p]\ \{C\} \mid \text{if } (\varphi)\ \{C\} \text{ else } \{C\}$$
$$\varphi ::= e < e \mid \neg\varphi \mid \varphi \wedge \varphi \qquad e ::= c \mid x \mid e \cdot e \mid e + e \mid e - e$$

Fig. 1. Syntax of Loop Guard and Body in the form (1)

that an operator $\Phi : E \longrightarrow E$ is *monotone* if for all $u, v \in E$, $u \sqsubseteq v$ implies $\Phi(u) \sqsubseteq \Phi(v)$. Throughout this section, we fix a lattice $(E, \sqsubseteq)$ and a monotone operator $\Phi : E \longrightarrow E$.

We recall the k-induction operator given in [11] as follows, which we refer to as the *upper* k-induction operator.

Definition 2.1 (Upper k-Induction Operator [11]). Given any element $u \in E$, the upper k-induction operator Ψ_u w.r.t. u and the monotone operator Φ is defined by: $\Psi_u : E \longrightarrow E, v \mapsto \Phi(v) \sqcap u$.

Below we propose a dual version for the upper k-induction operator. The intuition is simply to replace the meet operation with join. We call this dual operator as the *lower* k-induction operator.

Definition 2.2 (Lower k-Induction Operator). Let $u \in E$. The *dual* k-induction operator Ψ'_u w.r.t. u and the aforementioned monotone operator Φ is defined by: $\Psi'_u : E \longrightarrow E, v \mapsto \Phi(v) \sqcup u$.

Remark 1. *Alterative formulation of the k-induction operators have also been proposed in [39]. In our extended version [58, Appendix A], We show that these formulations are essentially equivalent to the definitions adopted in this work. Therefore, in the rest of this paper, we focus exclusively on the upper and lower k-induction operators defined above.* □

2.3 Probabilistic Loops

In this work, we use simple probabilistic while loops of the form (1) for easing the explanation of our basic idea, and will discuss how to extend our approach to general probabilistic while loops like nested loops without substantial changes in Section 5.2. Below we define the class of single probabilistic loops.

Syntax. A probabilistic while loop takes the form

$$\textbf{while } (\varphi)\ \{C\} \tag{1}$$

where φ is the loop guard and C is the loop body without loops. Formally, the loop guard φ and loop body C are generated by the grammar in Figure 1, where x is a program variable taken from a countable set Vars of variables, $c \in \mathbb{R}$ is a real constant, e is an arithmetic expression that involves addition and multiplication, φ is a formula over program variables that is a Boolean combination of arithmetic inequalities, and μ is a predefined probability distribution. In this work, we consider μ to be a finite discrete probability distribution (i.e., distributions with a finite support) such as Bernoulli distribution and discrete uniform distribution. The semantics of skip, assignment, sequential composition, conditional, and while statement can be understood as their counterparts in imperative programs. The semantics of a probabilistic choice $\{C_1\}[p]\{C_2\}$ is that flips a coin with bias $p \in [0, 1]$ and executes the statement C_1 if the coin yields head, and C_2 otherwise. The semantics of a sampling statement $x :\approx \mu$ samples a value according to the predefined distribution μ and assigns the value to the variable x.

Given a probabilistic while loop, a *program state* is a function that maps every program variable to a real number. We denote by S the set of program states. The initial state for a probabilistic while loop is denoted by s^*. The evaluation $\varphi(s)$ of a logical formula φ and the evaluation $e(s)$ of an arithmetic expression e over a program state s are defined in the standard way. $\varphi(s) = \texttt{true}$ is denoted by $s \models \varphi$.

Semantics. The semantics of a probabilistic loop of the form (1) can be interpreted as a discrete-time Markov chain, where the state space is the set of all program states S, and the transition probability function $\mathbf{P}$ is given by the loop body C and determines the probability $\mathbf{P}(s, s')$ for $s, s' \in S$, meaning the probability producing output state s' from input state s. If the loop guard $\varphi(s)$ evaluates to false, then we treat the program state s as a sink state, that is $\mathbf{P}(s, s) = 1$ and $\mathbf{P}(s, s') = 0$ for $s \neq s'$.

Given the Markov chain of a probabilistic while loop as described above, a *path* is an infinite sequence $\pi = s_0, s_1, \ldots, s_n, \ldots$ of program states such that $\mathbf{P}(s_n, s_{n+1}) > 0$ for all $n \geq 0$. Intuitively, each s_n corresponds to the state right before the $(n+1)$-th loop iteration. A program state s is *reachable* from an initial program state s^* if there exists a path $\pi = s_0, s_1, \ldots$ such that $s_0 = s^*$ and $s_n = s$ for some $n \geq 0$, and define $\mathit{Reach}(s^*)$ as the set of reachable states starting from the initial state s^*. By the standard cylinder construction (see e.g. [4, Chapter 10]), the Markov chain with a designated initial program state s^* for the probabilistic loop induces a probability space over paths and reachable states. We denote the probability measure in this probability space by $\mathbb{P}_{s^*}$ and its related expectation operator by $\mathbb{E}_{s^*}$.

Problem formulation. Given a probabilistic loop P in the form (1), assuming that P terminates with probability 1, a *return function* f is a function $f : S \to \mathbb{R}$ that is used to specify the output of the loop P in the sense that when the loop P terminates at a program state s, then the return value is given as $f(s)$. A return function is *piecewise polynomial* if it can be expressed as a piecewise polynomial expression in program variables. We denote by X_f the random variable for the return value of the loop given a return function f. In this work, we consider the following problem: Given a probabilistic while loop P in the form (1) and a piecewise polynomial return function f, synthesize *piecewise upper and lower bounds* on the expected value of X_f.

3 An Overview of Our Approach

Our approach falls in the background of (latticed) k-induction [11, 39]. k-induction is an induction principle that generalizes the standard induction by considering k consecutive transitions together in the inductive condition. Roughly speaking, given a predicate P to be proved via induction, the k-induction principle considers the inductive condition as $(P(x_1) \wedge \cdots \wedge P(x_k)) \to P(x_{k+1})$, for which the premise $P(x_1) \wedge \cdots \wedge P(x_k)$ means that the predicate P holds for k consecutive transitions, and the whole condition states that if P holds for k consecutive transitions, then P holds after these consecutive transitions. In particular, 1-induction coincides with the usual inductive condition.

Latticed k-induction [11] adapts the idea of k-induction to lattices for deriving bounds of fixed points. It considers k consecutive applications of a monotone operator over a lattice and applies the *meet/join* operations iteratively in the k consecutive applications. The parameter k here does not matter in the monotone operator (see Definitions 2.1 and 2.2), but is the number of iterative applications (see Definition 4.5) when the operator is applied. In this work, we propose a novel combination of latticed k-induction operators and Optional Stopping Theorem (OST), and propose novel algorithms for deriving piecewise linear and polynomial bounds on probabilistic programs.

We illustrate the main idea of our approach via the following example, which is a discretized version of the Growing Walk in [12]:

$$\textsc{Growing Walk}: \quad \texttt{while}\,(\,0 \leq x\,)\,\{\{x := x + 1; y := y + x\}\ [0.5]\ \{x := -1\}\}$$

The example models a simple random walk where the step size x is increased by 1 with one half probability, and set to -1 with the other half probability. The program terminates when x becomes negative. The objective is to analyze the expected value of the return function $f(x, y) = y$, which corresponds to the total traveled distance y, after the program terminates. We take the synthesis of piecewise linear upper bound as an example.

Step 1: Establishing k-induction operators. Let $\overline{\Phi}_f\colon (\mathbb{R}\times\mathbb{R}\to\mathbb{R})\to(\mathbb{R}\times\mathbb{R}\to\mathbb{R})$ be the operator

$$\overline{\Phi}_f(h) := \lambda(x,y).[x<0]\cdot y + [x\geq 0](0.5\cdot h(x+1,y+x+1) + 0.5\cdot h(-1,y))$$

for function $h:\mathbb{R}\times\mathbb{R}\to\mathbb{R}$, and $[x\geq 0]$ denotes the Iverson-bracket of the predicate $x\geq 0$, which evaluates to 1 if $x\geq 0$ holds at state s and 0 otherwise. Intuitively, $\overline{\Phi}_f$ outputs y if the loop guard $x\geq 0$ is violated, and the expected value of $h(x,y)$ after the execution of the loop body $\{x := x+1; y := y+x\}$ $[0.5]$ $\{x := -1\}$ otherwise. We introduce the k-induction operator Ψ_h (c.f. [11]), defined by $\Psi_h(g) := \min\{\overline{\Phi}_f(g), h\}$ for any fixed function $h:\mathbb{R}\times\mathbb{R}\to\mathbb{R}$. Informally, when applied to a function g, the operator $\Psi_h(g)$ pulls $\overline{\Phi}_f(g)$ down via the pointwise minimum operation with h.

Step 2: Applying k-induction condition. Let $k=2$. We unroll the loop P $(k=2)$ times and examine the $(k=2)$-induction condition to upper-bound the expected value of X_f. The resultant inductive condition from our approach is as follows (here $\leq$ is taken pointwise), which is obtained by applying the operator Ψ_h to a candidate bound function h once (i.e., $k-1$ times):

$$\overline{\Phi}_f(\Psi_h(h)) \leq h \tag{2}$$

We show that under a mild assumption and by using OST, if we have a function h that fulfills this inductive condition, then $\Psi_h(h)$ is an upper bound for the expected value of X_f, for which the *pointwise minimum* in $\Psi_h(h)=\min\{\overline{\Phi}_f(h),h\}$ is the key to derive the piecewise partition of the bound apart from loop unrolling.

Step 3: Simplifying the k-induction condition. Our approach synthesizes a function h w.r.t the condition (2). To the end, we reduce the condition (2) to the form below with four functions h_i $(1\leq i\leq 4)$ combined with a minimum operation:

$$\min\{h_1,h_2,h_3,h_4\}\leq h, \tag{3}$$

where $h_1=[x<0]\cdot y+[x\geq 0]\cdot(0.5\cdot h(x+1,x+y+1)+0.5\cdot h(-1,y))$, $h_2=[x<0]\cdot y+[x\geq 0]\cdot(0.25\cdot h(-1,y+x+1)+0.25\cdot h(x+2,2x+y+3)+0.5\cdot h(-1,y))$, $h_3=[x<0]\cdot y+[x\geq 0]\cdot(0.25\cdot h(-1,y+x+1)+0.25\cdot h(x+2,2x+y+3)+0.5\cdot y)$ and $h_4=[x<0]\cdot y+[x\geq 0]\cdot(0.5\cdot h(x+1,x+y+1)+0.5\cdot y)$. Using our algorithm, we employ a loop unrolling based approach to efficiently derive the simplified constraint (3) and we show that each h_i results from the unfolding of the loop up to depth $k=2$ and corresponds to a loop-free program from the unfolding. See **Stage 2** in Section 5 for the details.

Step 4: Solving the simplified $(k=2)$-induction condition. After *Step 3*, we obtain the constraint in (3) and further synthesize the function h in (3) by assuming a template for h and solving the template w.r.t. the constraint (3). Every synthesized function h leads to a piecewise upper bound $\Psi_h(h)=\min\{\overline{\Phi}_f(h),h\}$ for the expected value of X_f. Since this constraint includes a minimum operation, it is non-convex and non-trivial to solve. Our approach reduces the synthesis problem with a linear template to bilinear programming, and obtains a piecewise linear upper bound $[x<0]\cdot y+[x\geq 0]\cdot(x+y+2)$, which is actually the exact expected value of y. Similarly, our method can also obtain a piecewise linear lower bound $[x<0]\cdot y+[x\geq 0]\cdot(x+y+{}^{13}/_{8})$.

4 Piecewise Bounds via Latticed k-Induction

In this section, we propose a novel combination of OST and latticed k-induction operators to derive bounds for the expected value of X_f. We first introduce expectation functions over which we construct concrete k-induction operators, then define potential functions, and finally show the soundness of potential functions to derive expectation bounds via OST. Throughout this section, we fix a probabilistic while loop $P=\textbf{while}(\varphi)\{C\}$ in the form of (1) and a return function f.

4.1 Expectation Functions

Definition 4.1 (Expectation Functions). An *expectation function* is a function $h : S \to \mathbb{R}$ that assigns to each program state a real value. The partial order $\preceq$ over expectation functions is defined in the pointwise fashion, i.e., $h_1 \preceq h_2 \iff \forall s \in S, h_1(s) \leq h_2(s)$. We denote the set of expectation functions by $\mathcal{E}$ and the lattice by $(\mathcal{E}, \preceq)$, for which the meet operation $\sqcap$ in the lattice is given by $h_1 \sqcap h_2 := \min\{h_1, h_2\}$,where min is the pointwise minimum on functions, i.e., $\forall s \in S, \min\{h_1, h_2\}(s) = \min\{h_1(s), h_2(s)\}$, and the join operation $\sqcup$ is given by $h_1 \sqcup h_2 := \max\{h_1, h_2\}$, where max is the pointwise maximum.

Informally, an expectation function h is that for each program state $s \in S$, the value $h(s)$ bounds the expected value of X_f after the execution of the while loop P when the loop starts with the program state s. Although one observes that the partially ordered set $(\mathcal{E}, \preceq)$ with the meet and join operations defined above is a lattice, we do not use lattice properties in our approach.

To instantiate the k-induction operators for expectation functions, we construct the monotone operator for the lattice $(\mathcal{E}, \preceq)$. To this end, we first define the notion of pre-expectation as follows, wherein $[\varphi]$ denotes the Iverson-bracket of φ. Notice that the random assignment command $x :\approx \mu$ (where μ is a discrete distribution of finite support) can be written in an iterative style of $\{C_1\}\ [p]\ \{C_2\}$, so that we define pre-expectation without random assignment commands.

Definition 4.2 (Pre-expectation [15, 54]). Given an expectation function $h : S \to \mathbb{R}$. We define its *pre-expectation* over a loop-free program Q, $pre_Q(h) : S \to \mathbb{R}$, recursively on the structure of Q:

- $pre_Q(h) := h$, if $Q \equiv \mathsf{skip}$.
- $pre_Q(h) := h[x/e]$, if $Q \equiv x := e$, where $h[x/e]$ denotes $h[x/e](s) = h(s[x/e])$ with $s[x/e](x) = e(s)$ and $s[x/e](y) = s(y)$ for all $y \in \mathsf{Vars}\backslash\{x\}$.
- $pre_Q(h) := pre_{Q_1}(pre_{Q_2}(h))$, if $Q \equiv Q_1; Q_2$.
- $pre_Q(h) := p \cdot pre_{Q_1}(h) + (1-p) \cdot pre_{Q_2}(h)$, if $Q \equiv \{Q_1\}\ [p]\ \{Q_2\}$.
- $pre_Q(h) := [\phi] \cdot pre_{Q_1}(h) + [\neg\phi] \cdot pre_{Q_2}(h)$, if $Q \equiv \mathsf{if}\ (\phi)\ \{Q_1\}\ \mathsf{else}\ \{Q_2\}$.

The intuition of pre-expectation is that given an expectation function h, the pre-expectation pre_Q computes the expected value $pre_Q(h)$ of h after the execution of the command Q. With pre-expectation, we then define the monotone operator to be the characteristic function $\overline{\Phi}_f$ of the probabilistic loop P with respect to the return function f as follows.

For the rest of this section, we fix an initial state s^* and override the set S of program states with $Reach(s^*)$ in Definition 4.1 so that we consider expectation functions restricted to $Reach(s^*)$.

Definition 4.3 (Characteristic Function [15, 33]). The *characteristic function* $\overline{\Phi}_f : \mathcal{E} \to \mathcal{E}$ is defined by $\overline{\Phi}_f(h) := [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(h)$. The monotone operator for the lattice $(\mathcal{E}, \preceq)$ is defined as $\overline{\Phi}_f$.

Informally, the characteristic function $\overline{\Phi}_f$ outputs f if the loop guard φ is violated and the loop terminates in the next step, and the pre-expectation of h w.r.t. the loop body C otherwise. It is straightforward to verify the monotonicity of $\overline{\Phi}_f$. In the following, we omit the subscript f in $\overline{\Phi}_f$ if it is clear from the context. Given the monotone operator, we establish the concrete k-induction operators as follows.

Definition 4.4 (k-Induction Operators for $(\mathcal{E}, \preceq)$). Given an expectation function h, the *upper (resp. lower) k-induction operator* $\overline{\Psi}_h : \mathcal{E} \to \mathcal{E}$ (resp. $\overline{\Psi}'_h : \mathcal{E} \to \mathcal{E}$) is defined by $\overline{\Psi}_h(g) = \min\{\overline{\Phi}_f(g), h\}$ (resp. $\overline{\Psi}'_h(g) = \max\{\overline{\Phi}_f(g), h\}$) for arbitrary expectation function $g \in \mathcal{E}$.

Note that k does not explicitly appear within the operators; rather, it denotes the number of times these operators are iteratively applied.

4.2 Potential Functions

We define potential functions as expectation functions satisfying the k-induction conditions. These potential functions serve as candidate bounds to be synthesized.

Definition 4.5 (Potential Functions). Let k be a positive integer. A *k-upper* (resp. *k-lower*) potential function is an expectation function h that satisfies the *upper* (resp. *lower*) k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ (resp. $\overline{\Phi}_f((\overline{\Psi}'_h)^{k-1}(h)) \succeq h$), respectively.

Remark 2. *Note that both the* base case *and the* induction case *are encapsulated in the operator $\overline{\Phi}_f$ encoding the k-induction condition. Specifically, by the definition of $\overline{\Phi}_f$, the k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ is equivalent to $[\neg\varphi]\cdot f + [\varphi]\cdot \mathrm{pre}_C(\overline{\Psi}_h^{k-1}(h)) \preceq h$. This, in turn, requires that:*

(1) $f(s) \preceq h(s)$ when the program state s satisfies $s \models \neg\varphi$, corresponding to the base case; and

(2) $\mathrm{pre}_C(\overline{\Psi}_h^{k-1}(h))(s) \preceq h(s)$ when $s \models \varphi$, corresponding to the induction case. □

We apply Optional Stopping Theorem (OST) to address our soundness results. We find that the classical OST [24, 56] cannot handle our problem due to the requirement of bounded step-wise difference (see [58, Appendix B.1] in our extended version), while the OST variant proposed in [55] can handle our problem.

Theorem 4.6 (Extended OST [55]). *Let $\{X_n\}_{n=0}^{\infty}$ be a supermartingale adapted to a filtration $\mathcal{F} = \{\mathcal{F}_n\}_{n=0}^{\infty}$ and τ be a stopping time w.r.t the filtration $\mathcal{F}$. Suppose there exist positive real numbers b_1, b_2, c_1, c_2, c_3 such that $c_2 > c_3$ and*

(a) For all sufficiently large natural numbers n, it holds that $\mathbb{P}(\tau > n) \le c_1 \cdot e^{-c_2 \cdot n}$.

(b) For every natural number $n \ge 0$, it holds almost-surely that $|X_{n+1} - X_n| \le b_1 \cdot n^{b_2} \cdot e^{c_3 \cdot n}$.

Then we have that $\mathbb{E}(|X_\tau|) < \infty$ and $\mathbb{E}(X_\tau) \le \mathbb{E}(X_0)$.

Under certain side conditions that guarantee the validity of the extended OST, the potential functions provide upper and lower bounds on the expected value of X_f. Before presenting this result, we introduce some concepts that capture the magnitude of updates to program variables between two consecutive steps.

Definition 4.7 (Affine Programs). A probabilistic program is affine if all conditions and assignments within the program are **affine** functions of the program variables.

Definition 4.8 (Termination Time). The *termination time* T of the loop P is the random variable that for any path of the loop, measures the number of total loop iterations in the path.

Definition 4.9 (Uniform Amplifier). Suppose that the loop P is affine, and for each program variable x, let x_n denote the random variable representing the value of x at the n-th iteration of the loop. A *uniform amplifier* c is a constant $c > 0$ such that, for all $n \ge 0$, $|x_{n+1}| \le c \cdot |x_n| + a$ holds for some fixed constant a.

Definition 4.10 (Bounded Update). The loop P has the *bounded-update* property if there exists a real constant $a > 0$ such that for each program variable x, $|x_{n+1} - x_n| \le a$ for every $n \ge 0$ (see Definition 4.9 for the meaning of x_n).

Remark 3. *Note that any program satisfying the bounded update property also admits a uniform amplifier with $c = 0$.* □

We now present the soundness theorem of k-upper (resp. lower) potential functions. We distinguish between *affine programs* and *polynomial programs*, as each requires different side conditions for potential functions to serve as upper or lower bounds. Notably, the side conditions for affine programs are weaker than those for polynomial programs.

Theorem 4.11. *Suppose the loop P is affine. Let k be a positive integer and h be a polynomial potential polynomial in the program variables with degree d. If there exist real numbers $c_1 > 0$ and $c_2 > c_3 > 0$ such that*

(P1) there exists a uniform amplifier c satisfying $c \le e^{c_3/d}$, and
(P2) the termination time T of P has the concentration property*, i.e., $\mathbb{P}(T > n) \le c_1 \cdot e^{-c_2 \cdot n}$ holds for sufficiently large $n \in \mathbb{N}$.*

hold, then for any initial program state s^*, we have:

- $\mathbb{E}_{s^*}(X_f) \le \overline{\Psi}_h^{k-1}(h)(s^*) \le h(s^*)$ *holds for any k-upper potential polynomial h.*
- $\mathbb{E}_{s^*}(X_f) \ge (\overline{\Psi}'_h)^{k-1}(h)(s^*) \ge h(s^*)$ *holds for any k-lower potential polynomial h.*

Proof Sketch. (See [58, Appendix B.2] in our extended version for the full proof) Let s_n be the random variable of the program state at the n-th iteration with $s_0 = s^*$, and let $H = \overline{\Psi}_h^{k-1}(h)$. A *key point* is that since H is piecewise polynomial (by the definition of $\overline{\Psi}_h$) and condition (P1) holds, condition (b) in Theorem 4.6 holds for process $\{X_n\}_{n\in\mathbb{N}} = \{H(s_n)\}_{n\in\mathbb{N}}$. Combining with the fact that h is a k-upper potential function, one can further deduce $\{X_n\}_{n\in\mathbb{N}} = \{H(s_n)\}_{n\in\mathbb{N}}$ is a supermartingale. Note that X_0 is the initial value of the process X_0. and X_T represents the value of the process X_n at loop termination. By applying Theorem 4.6, we have $\mathbb{E}_{s^*}(X_T) \le \mathbb{E}_{s^*}(X_0)$ (T is a stopping time), thus $\mathbb{E}_{s^*}(X_f) \le \mathbb{E}_{s^*}(X_0) = H(s^*)$. The lower case is derived similarly. □

The side condition (P1) for affine programs requires that the loop P possesses a uniform amplifier constant. In contrast, for polynomial programs, a stronger property is needed: the program must satisfy the bounded update property, which imposes stricter constraints than (P1).

Theorem 4.12. *Let k be a positive integer. Suppose there exist real numbers $c_1 > 0$ and $c_2 > 0$ such that condition (P1') loop P has the bounded update property; and condition (P2) in Theorem 4.11 holds, then for any initial program state s^*, we have*

- $\mathbb{E}_{s^*}(X_f) \le \overline{\Psi}_h^{k-1}(h)(s^*) \le h(s^*)$ *holds for any k-upper potential polynomial h.*
- $\mathbb{E}_{s^*}(X_f) \ge (\overline{\Psi}'_h)^{k-1}(h)(s^*) \ge h(s^*)$ *holds for any k-lower potential polynomial h.*

Remark 4. *See [58, Appendix B.3] in our extended version for the proof of Theorem 4.12. The concentration condition (P2), which ensures exponentially decreasing nontermination probabilities as stated in Theorems 4.11 and 4.12, guarantees that loop P terminates almost surely. This condition has been extensively studied in the literature (see, e.g., [16, 17, 25]).* □

According to Theorems 4.11 and 4.12, synthesizing upper and lower bounds reduces to finding a potential function h that satisfies the conditions outlined in these theorems. However, solving the k-upper and k-lower potential function conditions is challenging due to the intricate combination of minimum and indicator functions involved. In the following sections, we introduce algorithmic approaches to systematically synthesize these upper and lower bounds.

5 Algorithms for Bound Synthesis

In this section, we first present algorithms for synthesizing upper and lower bounds for single-loop programs. We then demonstrate how our approach naturally extends to handle programs containing nested or sequential loops.

5.1 Algorithms for Probabilistic Single Loops

In this subsection, we present algorithms for synthesizing k-upper and k-lower potential functions that satisfy the conditions specified in Theorem 4.11 and Theorem 4.12, leading to piecewise bounds

on the expected value of X_f. Below, we consider a fixed probabilistic loop P of the form (1) along with a return function f. Due to the space limit, we only illustrate the synthesis procedure for upper bounds. The case for lower bounds is nearly analogous, obtained by replacing minimum with maximum and substituting $\preceq$ by $\succeq$. The pseudocode for our algorithm is presented in Algorithm 1. Our approach consists of the following major steps:

Stage 1: Prerequisites Checking and External Inputs. Our algorithm first verifies the side conditions (P1) and (P2) (respectively, (P1') and (P2')) for affine (respectively, polynomial) programs, as specified by Theorems 4.11 and 4.12. The algorithm also accepts the hyperparameter k and a program invariant as input parameters.

Prerequisites checking. When P is affine, condition (P1) is verified by syntactically inspecting the loop body to identify a positive constant c_3, ensuring that each program variable is amplified by at most $e^{c_3/d}$, up to an additive constant, within a single loop iteration, where d denotes the degree of the polynomial template potential function h (c.f. Stage 2). Condition (P2) is guaranteed either by synthesizing a difference-bounded ranking supermartingale (dbRSM) that demonstrates the exponentially decreasing concentration property [16, 17], or by syntactically analyzing probabilistic branching within the loop to extract a suitable constant c_2 satisfying $c_2 > c_3 > 0$. For polynomial programs, condition (P1')—the bounded update property—is checked via an SMT solver (e.g., Z3 [21]), while condition (P2) is ensured analogously to the affine case.

External inputs. Our algorithm requires the following hyperparameters as input: (1) *Induction parameter k:* We specify a positive real number k as the parameter for k-induction, along with the initial program state s^*. (2) *Program invariant:* We assume the existence of an invariant I at the entry point of the loop, which over-approximates the set of reachable program states $Reach(s^*)$. That is, for every $s \in Reach(s^*)$, we have $s \models I$. The state space is thus restricted to program states satisfying I, and the relation $\preceq$ is interpreted over I, i.e., $h_1 \preceq h_2 \iff \forall s \models I,\ h_1(s) \leq h_2(s)$. The rational of this restriction follows from the over-approximation property of I. Invariants can be obtained using external invariant generators, such as [47].

Example 5.1. We still take the example in Section 3 as a running example, which is a discretized version of the Growing Walk in [12]:

$$\texttt{while}\,(\,0 \leq x\,)\,\{\{x := x + 1; y := y + x\}\ [0.5]\ \{x := -1\}\}$$

In this example, our goal is to analyze the expected value of y upon program termination. We check the prerequisites and specify the external inputs as follows: (1) *Prerequisite Verification*: We find that $c = 1$ serves as a uniform amplifier, satisfying $c \leq e^{c_3/d}$ with $c_3 = \ln 1.5$ and $d = 1$. The concentration condition (P2) is also met with $c_2 = \ln 2$. (2) *External Inputs*: We set $k = 2$, and choose the invariant $I = \{x \mid -1 \leq x\}$ with initial state $s^* = (x, y) = (1, 1)$. □

Stage 2: Templates and Constraints. After verifying the prerequisites and identifying the external inputs as described in **Stage 1**, our algorithm predefines a d-degree polynomial template h as the candidate k-upper potential function for the loop P. This template consists of a linear combination of all monomials in the program variables of degree at most d, where each monomial is multiplied by an unknown coefficient.

Next, we apply the k-induction conditions in Definition 4.5, resulting in the constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$. The presence of min and indicator operators within this constraint complicates direct simplification. To address this, we reformulate the constraint into the form $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i is free of the minimum operator. Although a brute-force arithmetic expansion can achieve this transformation (see [58, Appendix C.1] in our extended version for details), our algorithm employs a more efficient unfolding strategy, which we outline below.

The unfolding process for constraint simplification: We symbolically unroll the probabilistic loop from the initial state up to k iterations, exploring all possible unfolding strategies. Here, "symbolic" means that program variables in each program state retain their original variable names and represent undetermined values. An *unfolding strategy* operates at each symbolic program state encountered during the unfolding process (excluding the initial state), and iteratively selects among actions (i), (ii), and (iii): Action (i) corresponds to unfolding the loop once more. Action (ii) represents the scenario where we actively choose to stop unrolling before reaching the total of k unfoldings (the k-induction parameter). In contrast, action (iii) occurs when unrolling is forcibly stopped because the number of unfoldings has reached k; beyond this point, no further unrolling is allowed by our method. Each unfolding strategy, determined by the choices made at each unfolding step, yields a loop-free program. Let $C_1, \ldots, C_m$ denote all loop-free programs generated by applying the above decision process across all possible unfolding strategies. For each loop-free program C_d, we compute the *pre-expectation* $pre_{C_d}(h)$ of h with respect to C_d (see Definition 4.2), allowing us to equivalently rewrite the constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ as:

$$\min\{h_1, h_2, \ldots, h_m\} \preceq h, \tag{4}$$

where each h_i is given by $pre_{C_d}(h)$ for some C_d. According to the computation of pre-expectation (Definition 4.2), each h_i can be represented as $h_i = \sum_r [B_{ir}] \cdot e_{ir}$, where B_{ir} is a predicate independent of the template's unknown coefficients, and e_{ir} is a monolithic polynomial in the program variables, potentially containing unknown coefficients. Moreover, the B_{ir}'s are pairwise logically disjoint.

The following proposition formally establishes the relationship between the unfolding process and the k-induction condition. The proof is provided in our extended version [58, Appendix C.2].

Proposition 5.2. *The upper k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ is equivalent to constraint $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i equals $pre_{C_d}(h)$ for some unique $C_d \in \{C_1, \ldots, C_m\}$ from the unfolding process above.*

By Proposition 5.2, the k-induction constraint can be simplified by computing the pre-expectations of all programs $\{C_1, \ldots, C_m\}$ generated by all possible unfolding strategy within k loop iterations. Since these programs are structurally similar, we can efficiently compute $pre_{C_d}(h)$ for all $C_d \in \{C_1, \ldots, C_m\}$ simultaneously by traversing the k-unfolding of the program loop once. This approach reduces runtime by eliminating excessive and repeated computations.

Remark 5. *The unfolding process also provides an explanation for how the partitioning of the input domain is determined in the final piecewise bound $\overline{\Psi}_h^{k-1}(h)$. Specifically, the partition of the input domain is primarily governed by the number of iterations required for the state to potentially violate the loop guard, mirroring the k-fold unrolling of the loop in k-induction. This partition is further refined by the application of the* min *operator in the k-induction construction, resulting in a tighter and more precise bound.* □

Illustrative Example of the Unfolding Process. We demonstrate our unfolding process via a simple but illustrative example as follows:

$$P := \texttt{while}\,(\,\varphi(x)\,)\,\{\{x := a_1 x + b_1\}\ [p]\ \{x := a_2 x + b_2\}\} \tag{5}$$

where x is a real-valued program variable, $a_i, b_i (i = 1, 2)$ are real constants, $p \in [0, 1]$ and $\varphi(x)$ is a guard condition. Let f be the return function, and let $\overline{\Phi}_f$ be the operator defined as

$$\overline{\Phi}_f(h)(x) := [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1 x + b_1) + (1 - p) \cdot h(a_2 x + b_2))$$

for any function $h : \mathbb{R} \to \mathbb{R}$ (with $S = \mathbb{R}$), where $[\varphi]$ denotes the Iverson bracket for the predicate φ. In this example, we consider the 2-induction operator $\overline{\Psi}_h$ for a fixed function $h : \mathbb{R} \to \mathbb{R}$, as defined

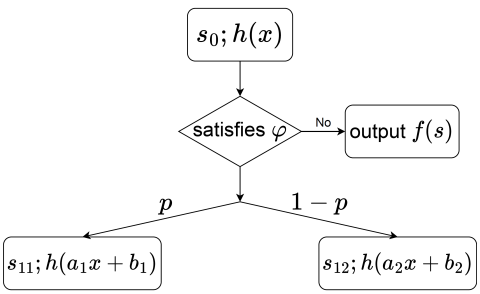
in [11]. Specifically, $\overline{\Psi}_h(g)$ is given by $\overline{\Psi}_h(g) := \min\{\overline{\Phi}_f(g), h\}$, and the corresponding 2-upper induction condition is :

$$\overline{\Phi}_f(\overline{\Psi}_h(h)) \preceq h\,. \tag{6}$$

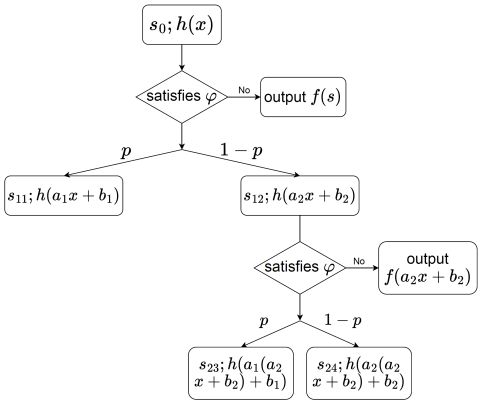
According to Proposition 5.2, we simplify this constraint by transforming (6) into the following form, which expresses the minimum over four functions h_i ($1 \le i \le 4$):

$$\min\{h_1, h_2, h_3, h_4\} \preceq h\,,$$

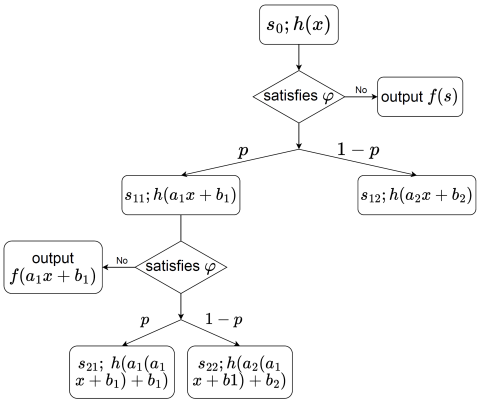
where each h_i corresponds to a loop-free program C_i generated during the unfolding process up to depth $k = 2$. All such unfolded programs are summarized in Fig. 2.



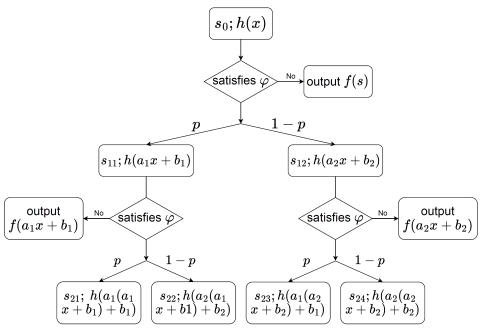
(a) **Case 1:** Program C_1



(b) **Case 2:** Program C_2



(c) **Case 3:** Program C_3



(d) **Case 4:** Program C_4

Fig. 2. Loop-free programs generated by ($k = 2$)-induction

We illustrate the unfolding process as follows. Starting from an initial value x, if $\varphi(x)$ is not satisfied, the loop terminates immediately and outputs $f(x)$. If $\varphi(x)$ holds, we proceed to unfold the loop, resulting in four distinct cases. Due to space constraints, we describe only the first case in detail here; the remaining three cases are depicted in Fig. 2, with further explanations provided in our extended version [58, Appendix C.3]. In Case 1, the loop executes once and transitions to two possible states, $a_1x + b_1$ and $a_2x + b_2$, then it terminates. This corresponds to a single unrolling of

the loop and terminating the unfolding at both resulting symbolic states, yielding the loop-free program C_1 as shown in Fig. 2a. The associated expression is $h_1 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1x + b_1) + (1-p) \cdot h(a_2x + b_2))$, which represents the expected value of $h(x)$ after executing program C_1. Cases 2, 3, and 4 are derived analogously by unrolling the loop up to two iterations.

Example 5.3. Returning to the running example in Example 5.1, we establish a 1-degree, i.e., linear template $h = a \cdot x + b \cdot y + c$, where a, b, c are unknown coefficients. We apply 2-induction condition to synthesize a piecewise linear upper bound. Starting from a symbolic initial program state $s^* = (x, y)$, we unroll the loop once and arrive at two new symbolic program states $(x+1, x+y+1)$ and $(-1, y)$. Over each new state, we take the decision separately and the unfolding strategy produces four loop-free programs. The $pre_{C_d}(h)$ w.r.t. these four programs are as follows:

$$\begin{aligned}
h_1 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot h(-1, y)) \\
h_2 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot h(-1, y)) \\
h_3 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot y) \\
h_4 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot y)
\end{aligned} \tag{7}$$

Thus, we have the simplified constraint $\forall(x, y) \models I, \min\{h_1, h_2, h_3, h_4\} \preceq h$. □

Branch reduction. During the unfolding process used to simplify the latticed k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$, the number of resulting functions h_i in (4) grows rapidly with the number of probabilistic choices in the loop body. This combinatorial growth occurs because, when computing the pre-expectation for probabilistic branches, the sum of two minimum expressions results in a new minimum taken over the Cartesian product of the original function sets. To address this issue, we introduce a heuristic that selects only a small subset of "representative" functions from the complete set of h_i in (4). Importantly, this approach does not compromise soundness (see Theorems 4.11 and 4.12), as the minimum over any subset is always at least as the minimum over the full set.

Taking the case of $k = 2$ as an example, by definition of operator $\overline{\Psi}_h$, we have

$$\begin{aligned}
\overline{\Phi}_f(\overline{\Psi}_h(h)) &= \overline{\Phi}_f(\min\{\{\overline{\Phi}_f(h), h\}\} \\
&= [\neg\varphi] \cdot f + [\varphi] \cdot \sum_{i=1}^{n} p_i \cdot \min\{\{\overline{\Phi}_f(h(u_i(s))), h(u_i(s))\}
\end{aligned}$$

where each p_i denotes a probabilistic choice in the characteristic function $\overline{\Phi}_f$, and u_i represents the corresponding state update function under that choice. Instead of enumerating all possible 2^n combinations in choosing either $\overline{\Phi}_f(h(u_i(s)))$ or $h(u_i(s))$ for each p_i (to expand into the minimum form (4)), one could consider combinations that have at most one $\overline{\Phi}_f(h(u_i(s)))$ and at most one $h(u_i(s))$, so that only a linear number of combinations are considered while retaining soundness. For the case of $k > 2$, a possible way for relaxation is to recursively consider combinations that have at most one $\overline{\Phi}_f(\overline{\Psi}_h^{k-2}(h(u_i(s))))$ and at most one $h(u_i(s))$.

Stage 3: Transforming to Canonical Form. At this stage, our algorithm transforms the constraint of the form (4) from **Stage 2** into the following canonical form:

$$[B_1] \implies \min\{e_{11}, \ldots, e_{m1}\} \leq h, \ \ldots, \ [B_l] \implies \min\{e_{1l}, \ldots, e_{ml}\} \leq h \tag{8}$$

where h is the predefined polynomial template. Each $B_j (j \in \{1, ..., l\})$ is a conjunction of predicates over the program variables that does not involve the template's unknown coefficients, and each e_{ij} is a polynomial expression in these unknown coefficients. The transformation begins by rewriting the inequality (4) as

$$\min\{\textstyle\sum_r [B_{1r}] \cdot e_{1r}, \ \ldots, \ \sum_r [B_{mr}] \cdot e_{mr}\} \preceq h \tag{9}$$

where, as described previously, each h_i is expressed as $h_i = \sum_r [B_{ir}] \cdot e_{ir}$. Next, for each conjunction $B = \bigwedge_{i=1}^{m} B_{ir_i}$ – with each B_{ir_i} taken from the summation $\sum_r [B_{ir}] \cdot e_{ir}$ – we obtain the constraint $\Psi_B = [B] \implies \min_{i=1}^{m} e_{ir_i} \leq h$. The transformed system of inequalities (8) is thus precisely the set of all such Ψ_B constraints. Infeasible constraints (i.e., those with unsatisfiable B) are removed, whenever possible, using an SMT solver such as Z3 [21].

Example 5.4. Continuing from Example 5.3, we convert (7) into its canonical form by partitioning the state space S into two regions: $[x < 0]$ and $[x \geq 0]$, as indicated in (7). Applying **Stage 3** and eliminating unsatisfiable predicates yields the following canonical form:

$$[x < 0] \implies \min\{y\} \leq h$$

$$[x \geq 0] \implies \min \begin{Bmatrix} 0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot h(-1, y) \\ 0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot h(-1, y) \\ 0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot y \\ 0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot y \end{Bmatrix} \leq h \tag{10}$$

□

Stage 4: Solving Constraints. Below, we describe our approach for solving the canonical constraints given in (8). It is important to note that the presence of the *minimum* operator in this canonical form makes the constraint *non-convex*. To address this, we develop distinct algorithms for the linear and polynomial cases. In the linear case, where the program is affine (i.e., all conditions and assignments are linear), we employ a linear template for the k-upper potential function h. In the polynomial case, where the program may be non-affine, we utilize a polynomial template.

Solving constraints (linear case). In our algorithm for the linear case, we require that the return function be piecewise linear and that the invariant be affine in the program variables. We first eliminate the minimum operator in (8) by considering its negation. This allows us to transform the constraint into a set of bilinear constraints using Motzkin's Transposition Theorem, which can then be solved with off-the-shelf bilinear programming solvers such as *Gurobi*.

Below, we present a variant of Motzkin's Transposition Theorem [42], which will be utilized in the subsequent analysis. The proof is provided in our extended version [58, Appendix C.4].

Theorem 5.5 (Motzkin's Transposition Theorem [42]). *Let $S = (A_1 \cdot \mathbf{x} + \mathbf{b_1} \leq 0)$ and $T = (A_2 \cdot \mathbf{x} + \mathbf{b_2} < 0)$ be systems of linear inequalities, where $A_1 = (\alpha_{i,j}) \in \mathbb{R}^{m \times n}$ and $A_2 = (\alpha_{m+i,j}) \in \mathbb{R}^{k \times n}$ are real coefficient matrices, $\mathbf{b_1} = (\beta_1, ...\beta_m)^\top$ and $\mathbf{b_2} = (\beta_{m+1}, ...\beta_{m+k})^\top$ are real vectors, and $\mathbf{x} = (x_1, ...x_n)^\top$. If S is satisfiable, then $S \wedge T$ is unsatisfiable if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, ..., \lambda_{m+k}$, with at least one λ_i for $i \in \{m+1, ..., m+k\}$ being nonzero, such that:*

$$\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)} = 0, \ \ldots, \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)} = 0, \ (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 = 0. \tag{11}$$

Remark 6. *Note that, since $\lambda_i \geq 0$ for $0 \leq i \leq m+k$, the requirement that at least one λ_i for $i \in \{m+1, \ldots, m+k\}$ is nonzero can be equivalently encoded as the linear constraint $\sum_{i=m+1}^{m+k} \lambda_i > 0$.*

In what follows, we demonstrate how to apply Theorem 5.5 to solve the canonical constraints (8). We begin by conjunct the affine invariant I with the antecedent predicates in (8) and eliminating any constraints with unsatisfiable antecedents, resulting in

$$[I \wedge B_j] \implies \min\{e_{1j}, e_{2j}, ..., e_{mj}\} \leq h \quad \text{for } j \in \{1, 2, \ldots, l\}, \tag{12}$$

where we assume that each $I \wedge B_j$ is satisfiable. For each j, we have

$$\begin{aligned}
&\big([I \wedge B_j] \implies \min\{e_{1j}, e_{2j}, ..., e_{mj}\} \le h\big) \text{ holds} \\
\iff\ &\big([I \wedge B_j] \wedge \big(\wedge_{i=1}^{m}(e_{ij} > h)\big)\big) \text{ is not satisfiable} && \text{[Apply Thm 5.5]} \\
\iff\ &\text{exists nonnegative real vector } \boldsymbol{\lambda_j} = (\lambda_{0,j}, \ldots, \lambda_{m_j+k_j,j}), \\
&\text{s.t. } (\lambda_{m_j+1,j}, \ldots, \lambda_{m_j+k_j,j}) \neq \mathbf{0}, \text{ and eq. (11) holds.}
\end{aligned}$$

The second equivalence follows from the Motzkin's Transposition Theorem by setting $S = I \wedge B_j$ and $T = \big(\wedge_{i=1}^{m}(e_{ij} > h)\big)$ for each $j \in \{1, 2, \ldots, l\}$. Note that (11) constitutes a bilinear constraint problem, as its nonlinearity arises solely from the products of unknown template coefficients and the variables $\boldsymbol{\lambda_j}$. Our approach aggregates all such bilinear constraints and utilizes off-the-shelf bilinear solvers to obtain concrete solutions for the template h.

Example 5.6. Continuing from Example 5.4, recall that we choose $x \ge -1$ as the invariant. For the constraint (10), substituting $h(x, y)$ with the template $ax + by + c$ and considering its negation as previously illustrated, we obtain the following inequalities:

$$-x \le 0 \qquad 0.5(a-b)x - 0.5b < 0 \qquad 0.75(a-b)x - (b - 0.25a) < 0$$
$$0.75(a-b)x + 0.5(b-1)y + (0.5c - 0.25a - b) < 0 \qquad 0.5(a-b)x + 0.5(b-1)y + 0.5(c-a-b) < 0.$$

Then by Theorem 5.5, the constraint (10) is equivalent to solving the following set of bilinear constraints involving the unknown coefficients a, b, and c.

$$\begin{aligned}
&\exists \lambda_0 \ge 0, \lambda_1 \ge 0, \cdots, \lambda_5 \ge 0 \quad \textbf{s.t.} \quad (\lambda_2 \neq 0 \vee \lambda_3 \neq 0 \vee \lambda_4 \neq 0 \vee \lambda_5 \neq 0) \wedge \\
&0 = (-1) \cdot \lambda_1 + 0.5(a-b) \cdot \lambda_2 + 0.75(a-b) \cdot \lambda_3 + 0.75(a-b) \cdot \lambda_4 + 0.5(a-b) \cdot \lambda_5 \wedge \\
&0 = 0.5(b-1) \cdot \lambda_4 + 0.5(b-1) \cdot \lambda_5 \wedge \\
&0 = -0.5b \cdot \lambda_2 - (b - 0.25a) \cdot \lambda_3 + (0.5c - 0.25a - b) \cdot \lambda_4 + 0.5(c-a-b) \cdot \lambda_5 - \lambda_0. \quad \square
\end{aligned}$$

Our algorithm utilizes bilinear solvers to address the derived bilinear constraints. Since these constraints define only a feasible region, we heuristically select an objective function to guide the solver toward solutions that yield tighter upper bounds. Specifically, we minimize $h(s^*)$, where s^* is a designated initial program state of interest. Once the template coefficients for h are determined (yielding a candidate h^*), we reconstruct the piecewise linear upper bound by applying the upper k-induction operator $\overline{\Psi}_{h^*}$ iteratively $k-1$ times, resulting in $\overline{\Psi}_{h^*}^{k-1}(h^*)$. We claim that our linear bound algorithm is complete in the sense that the reduction to bilinear programming preserves the original k-induction condition.

Example 5.7. Continuing with Example 5.6, we use the objective function $h = ax + by + c$ with the initial state $s^* = (x, y) = (1, 1)$. Solving the optimization yields the candidate $h^*(x, y) = x + y + 2$. We then reconstruct the piecewise upper bound by applying $\overline{\Psi}_{h^*}$ once, resulting in the upper bound $[x < 0] \cdot y + [x \ge 0] \cdot (x + y + 2)$.

Solving constraints (polynomial case). In our algorithm for the polynomial case, we assume that the return function is piecewise polynomial and that the invariant is a polynomial predicate over the program variables. We design a sound approach that relaxes the k-induction constraint and reduces the relaxed formulation to a semidefinite programming (SDP) problem using Putinar's Positivstellensatz [45]. This relaxation guarantees that the synthesized upper bound h satisfies the original k-induction condition (see Definition 4.5). The algorithm is described as follows.

First, for each constraint in the canonical form (8), namely $[B_j] \implies \min\{e_{1j}, \ldots, e_{mj}\} \le h$ for $j \in \{1, \ldots, l\}$, we relax the constraint by replacing the minimum operator with a convex combination of the terms $\{e_{ij}\}_{i=1}^{m}$. This results in the following relaxed form:

$$[B_j] \implies \textstyle\sum_{i=1}^{m} w_i \cdot e_{ij} \le h, \ , j \in \{1, \ldots, l\} \tag{13}$$

where each weight $w_i \geq 0$ and the set of weights satisfies $\sum_{i=1}^{m} w_i = 1$. Various forms of weight combinations $\{w_i\}_{i=1}^{m}$ can be employed, such as uniform weights (where each $w_i = 1/m$) or randomly generated weights normalized to sum to one. This relaxation is sound: any function h and set $\{eij\}_{i=1}^{m}$ that satisfy the relaxed constraint (13) will also satisfy the original canonical form (8). This follows from the fact that $\sum_{i=1}^{m} w_i \cdot e_{ij} \leq h \implies \min_{i\in\{1,\ldots,m\}} \{e_{ij}\} \leq h$.

Next, we conjunct the invariant I with each constraint in (13), resulting in the following form:

$$\bigwedge_{j\in\{1,\ldots,l\}} [I \wedge B_j] \implies \sum_{i=1}^{m} w_i \cdot e_{ij} \leq h, \tag{14}$$

We then apply Putinar's Positivstellensatz [45], following previous work [16, 55], to generate constraints on the unknown coefficients, which are solved using off-the-shelf SDP solvers (see [58, Appendix C.4] in our extended version for details). As these constraints define only a feasible region, we employ a heuristic objective function to guide the solver towards tighter upper bounds. Specifically, we minimize $\sum_i h(s_i^*)$, where s_i^* are selected initial program states of interest. After obtaining the optimal solution h^* from the SDP solver, we reconstruct the piecewise polynomial upper bound $\overline{\Psi}_{h^*}^{k-1}(h^*)$ by iteratively applying the upper k-induction operator $\overline{\Psi}_{h^*}$ to h^* $k-1$ times.

Algorithm 1: Synthesizing Bounds

Input : Probabilistic loop P in the form of (1) and a piecewise return function f
Output: Piecewise bounds for the expected value of X_f upon termination of P

Prerequisites Checking and External Inputs:
(a) Prerequisites Checking: Verify the prerequisites in Theorem 4.11 (Theorem 4.12).
(b) External Inputs: Generate an invariant I, select parameter k and specify initial program state s^*.
Templates and Constraints:
(a) Predefining a (monolithic) polynomial template h.
(b) Unfolding the loop within k times and calculate $pre_{C_d}(h)$ for all $C_d \in \{C_1, \ldots, C_m\}$ (generated by our unfolding process) to obtain the constraint $\min\{h_1, h_2, \ldots, h_m\} \preceq h$.
Transforming to Canonical Form:
Transform the constraints (4) into the form of (8) through an iterative approach and obtain l canonical constraints;
Constraints Solving:
if the loop P is linear and the template h is linear **then**
  $Cons \leftarrow \emptyset$; ▷ Linear Case
  for $j \leftarrow 1$ **to** l **do**
    Extract the coefficients of the variables from canonical-formed constraints;
    Construct bilinear constraints K_j with auxiliary variables $\boldsymbol{\lambda}_j$;
    $Cons \leftarrow Cons \cup K_j$;
  end
  Call bilinear solver to solve $Cons$ and obtain the piecewise bound with the solution h^*
else
  (a) Soundly relax the original canonical constraints (8) into (14). ▷ Polynomial Case
  (b) Call SDP solver to solve and obtain the piecewise bound with h^*.
end

Correctness. Our algorithms are guaranteed to produce correct bounds by Theorems 4.11 and 4.12. The *Prerequisites Checking* stage ensures that all prerequisites in Theorem 4.11 and Theorem 4.12 are met, and the function h is determined according to the k-induction conditions (see Definition 4.5). Additionally, the invariants we use over-approximate the set of reachable program states, thereby

preserving the soundness of our approach. Specifically, our linear bound algorithm is both sound and complete in the sense that the reduction to bilinear programming exactly preserves the original k-induction condition. In the polynomial case, our algorithm employs a sound relaxation, which likewise guarantees the correctness of the synthesized bounds.

5.2 Extensions: Handling Probabilistic Programs with Multiple Loops

Below, we describe the extension of our approach to probabilistic programs with multiple loops, including both sequential compositions of probabilistic loops and nested loops. For brevity, we focus on the synthesis of upper bounds; the synthesis of lower bounds is entirely analogous.

Sequential Composition. For a sequential composition $P = P_1; \ldots; P_n$ of loops $P_1, \ldots, P_n$ with return function f, our method analyzes each loop in reverse order. To illustrate the approach, we focus on the case $P = P_1; P_2$. Given a k-induction parameter k, the procedure for synthesizing upper bounds proceeds as follows: (i) Begin by computing a piecewise upper bound h_2 for the expected value of f after the execution of P_2. (ii) Then, treat h_2 as the return function for P_1 and compute its piecewise upper bound, resulting in a final bound h_1 for the entire composition. This backward compositional reasoning can be systematically extended to compositions with more than two loops.

Nested Loops. To address nested loops, we incorporate our approach with the methods proposed in [25, 26], applying k-induction exclusively to the innermost loop and 1-induction to the outer loops. Since the innermost loop can be unfolded independently of the outer loops, we are able to derive tight piecewise bounds for the inner loop via k-induction and subsequently propagate these bounds to the outer loops. For clarity, we focus on the case where the program R contains a single inner loop and has the following structure:

$$R = \texttt{while}(\,\psi\,)\{P\} \text{ with } P = \texttt{while}(\,\varphi\,)\{Q\} \text{ and } Q \text{ loop-free.}$$

Our objective is to analyze the expected value of X_f upon termination of the loop. Let Φ_f^{out} denote the characteristic function (see Definition 4.3) with respect to the outer loop and return function f, and let Φ_g^{in} denote the characteristic function for the inner loop P and return function g. While Φ_g^{in} can be computed explicitly, Φ_f^{out} typically cannot. We therefore apply 1-induction to the outer loop and k-induction to the inner loop, as summarized below:

- Define templates h_{out} and h_{in} at the entry of the outer and inner loop respectively.
- For the outer loop, the 1-induction rule yields the constraint $\Phi_f^{out}(h_{out}) \preceq h_{out}$. Since $\Phi_f^{out}(h_{out})$ cannot generally be computed explicitly, we upper-approximate the expected value of h_{out} after executing the inner loop P by h_{in}, i.e., $\Phi_f^{out}(h_{out}) \preceq [\neg\psi]\cdot f + [\psi]\cdot h_{in}$, and the original constraint $\Phi_f^{out}(h_{out}) \preceq h_{out}$ can be strengthened into $[\neg\psi]\cdot f + [\psi]\cdot h_{in} \preceq h_{out}$.
- For the inner loop, we apply the k-induction condition (see Definition 4.4) to ensure that h_{in} upper-approximates the expected value of h_{out} after executing the inner loop. This leads to the constraint $\Phi^{in}_{h_{out}}((\Psi^{in}_{h_{in}})^{k-1}(h_{in})) \preceq h_{in}$, where $\Psi^{in}_{h_{in}}(g) = \min\{\Phi^{in}_{h_{out}}(g), h_{in}\}$ is the upper k-induction operator for the inner loop P (see Definition 2.1).
- Collect the resulting constraints and apply our synthesis algorithm as described in Section 5.

Through this process, we obtain h_{out} as a piecewise upper bound for the expected value of X_f with respect to the return function f upon termination of the entire while loop R.

6 Experimental Results

We implement our algorithms in Python 3.9.12 and Julia 1.9.4. We use Gurobi in Python for bilinear programming and Mosek in Julia for semi-definite programming. All experiments are conducted on a Windows 10 (64-bit) machine equipped with an Intel(R) Core(TM) i7-9750H CPU at 2.60GHz

and 16GB of RAM. We evaluate our algorithms for synthesizing piecewise linear and polynomial upper bounds, as detailed in Section 6.1 and Section 6.2. Results for lower bound synthesis, which exhibit similar performance and comparative advantages, are provided in [58, Appendix D.2, D.3] in our extended version due to space limitations.

Evaluation Goals. Our experiments are designed to address the following research questions:

RQ1. How effective is our approach in generating piecewise bounds?
RQ2. How does our approach compare to the most closely related methods?
RQ3. How do our piecewise bounds compare to monolithic polynomial bounds?

Experimental Settings. We address the evaluation goals for our piecewise linear and polynomial algorithms separately. The experiments are conducted under the following settings:

Invariants. We employ invariants to over-approximate the set of reachable states, which is standard in various existing results [15, 16, 27]. Note that invariants do not provide information about the piecewise partitioning of the bounds to be computed. In our experiment, we minimize their impact by deliberately choosing trivial interval-bound invariants that can be directly derived as the union of loop guard and its post image under the increment/decrement operations within the loop body.

Prerequisites Checking. Our experiments cover both linear and polynomial probabilistic programs (see [58, Appendix E] in our extended version). For linear programs with monolithic linear return functions, we use a linear template and apply our linear algorithm. For more general cases involving polynomial programs with piecewise polynomial return functions, we employ a higher-degree polynomial template and apply our polynomial algorithm. In our piecewise linear experiments, we ensure that the prerequisites (P1) and (P2) in Theorem 4.11 are satisfied as follows. For (P1), we verify syntactically that the uniform amplifier c can typically be set to 1 across most benchmarks, ensuring that (P1) holds for any positive c_2. For the remaining benchmarks, we take the maximum coefficient of the program variables in the loop body as c. For example, in the St-Petersburg benchmark, we set the uniform amplifier c to 2, choosing $c_3 = \ln 2$ (since $e^{c_3} = 2$) and $c_2 = \ln 4$ to meet the required conditions. For (P2), in benchmarks where each loop iteration terminates with probability p and continues with probability $1 - p$, we can syntactically extract p and verify that the concentration property holds, exhibiting exponential decay at a rate of $e^{\ln(1-p)}$. For the remaining benchmarks, we construct difference-bounded ranking supermartingales (dbRSMs) to ensure the concentration property. Such dbRSMs can be synthesized automatically using methods described in [16, 17]. In our piecewise polynomial experiments, we ensure that the prerequisites (P1') and (P2) in Theorem 4.12 are satisfied as follows. For (P1'), we verify the bounded-update property on each polynomial benchmark using an SMT solver [21]. For (P2), we apply the same approach as in the linear case to establish the concentration property for polynomial programs.

Bound Optimization. Recall that in our algorithms described on pages 15 and 16, we optimize the synthesized upper bounds by minimizing their values over the initial states of interest, which serve as the objective function. In the piecewise linear experiments, we typically set the default initial state s^* by assigning the value 1 to all program variables across most benchmarks. For specific cases, such as Fair Coin, we assign initial values $x = 0$ and $y = 0$ — since $(x, y) = (0, 0)$ is the only state from which the loop can be entered — and set the variable i to its default value of 1. In the piecewise polynomial experiments, for path probability estimation benchmarks selected from [13, 28, 46, 55], we adopt the default initial state s^* used in previous work to ensure consistency. For the remaining benchmarks, we first define an interval-bound region, with real-valued variables ranging over $[0, 10]$ and Boolean variables over $[0, 1]$. We then select 10 initial states comprising the boundary points of the region, the midpoints of each boundary, the center point, and uniformly distributed integer points within the region.

Weights Selection. For the polynomial experiments, recall that our algorithm requires a predefined set of weight combinations (see Eq. (14)). We employ uniformly distributed weights (i.e., each weight is $\frac{1}{m}$) and additionally generate 10 sets of randomly selected weights, each normalized to sum to one. Independent computations are performed for each of these 11 weight combinations. From the resulting solutions, we select the function with the minimum objective value as the synthesized upper bound h^*. The total execution time is reported as the cumulative runtime for the 11 independent runs with different weight settings.

Numerical Repair. To address the inherent numerical issues associated with numerical solvers, we apply a post-processing step to repair the computed results. In the linear experiments, we approximate the output floating-point coefficients with rational numbers using continued fractions (see [32], [58, Appendix D.1] in our extended version), and validate these approximations by checking the constraints in (8). This numerical repair is applied to all benchmarks except Expected Time. For this benchmark, since suitable rational approximations could not be found, we truncate the floating-point results to a precision of 10^{-4} and verify their validity against the same constraints. In the polynomial experiments, we similarly truncate all floating-point results to 10^{-4} precision, then substitute the results into the constraints in (14) to check feasibility. Of the 20 benchmarks evaluated, the results for 16 passed our validation procedure, while the remainder remain unknown.

6.1 Piecewise Linear Bound Synthesis

Benchmark Selection. We choose upper-bound benchmarks from existing works [5, 10–12, 20, 25, 26, 28] that fall into our scope and have the following adaptions. First, for those that do not have linear return functions, we add simple linear return functions. Second, for those whose upper bound that can be handled directly by 1-induction (except for several classical examples: k-Geo, RevBin, Fair Coin), we adapt them by reasonable perturbations (such as changing the assignment statement, changing the probability parameters, reducing the continuous distribution to discrete distribution, etc) so that they require $(k > 1)$-induction. Third, for those whose upper bound that cannot be handled by k-induction with small $k = 1, 2, 3$, we adapt them by reasonable perturbations as above so that they can be handled by $(k > 1)$-induction, while still cannot be handled by 1-induction.

In detail, we consider 7 original examples and 6 adapted examples from the literature. The examples Geo, k-Geo and Equal-Prob-Grid are taken from [10, 11], for which we replace the assertion probability with a linear return function *goal* in Equal-Prob-Grid. We consider the benchmark Zero-Conf-Variant adapted from [10, 25]. We revise the assignments and probabilistic parameters in the original program, and add a linear return function *curprobe*. The benchmark St-Petersburg variant is taken from [25] where we replace the probability parameter $\frac{1}{2}$ with $\frac{3}{4}$ since the original program does not satisfy the prerequisites in Theorem 4.11. From [5, 20, 26], we consider the benchmarks Coin, Mart, RevBin and Fair Coin, and revise the assignments, guards on the original benchmarks Bin series so that we obtain a more complex version Bin-Ran. The remaining three examples, Expected Time, Growing Walk and its variant, are all adapted from [12, 28] by reducing the continuous distributions to discrete distributions. For affine programs, as k increases, the bottleneck emerges in the unfolding process of Stage 2. In our experiments with piecewise linear bounds, the parameter k is chosen as the largest value such that the runtime of Stage 2 in Algorithm 1 remains below 600 seconds.

Answering RQ1. We present the experimental results on these 13 benchmarks in Table 1. As bilinear solving is an iterative search for optimal solutions, we set the maximum searching time for Gurobi to 100s. On most benchmarks, we find that a monolithic linear bound with 1-induction does not exist but obtain a piecewise linear upper bound via $(k > 1)$-induction in a few minutes. Our

Table 1. Experimental Results for **RQ1** and **RQ2**, Linear Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "T(s)" (of our approach) stands for the execution time of our approach (in seconds), including the parsing from the program input, transforming the k-induction constraint into the bilinear problems, bilinear solving time and verification time. "Conventional Approach ($k = 1$)" stands for the monolithic linear upper bound synthesized via 1-induction, "k" stands for the k-induction we apply, "Solution" stands for the linear candidate solved by Gurobi, and "Piecewise Linear Upper Bound" stands for our piecewise results. "Result" stands for the synthesized results by other tools and "T(s)" (of their approaches) stands for the execution time of their tools.

Benchmark	f	Conventional Approach ($k=1$)	Our Approach				cegispro2		exist	
			k	Solution	Piecewise Linear Upper Bound	T(s)	Result	T(s)	Result	T(s)
Geo	x	✗	3	$x+1$	$[c>0]\cdot x+[c\le 0]\cdot(x+1)$	1.92	$[c>0]\cdot x+$ $[c\le 0]\cdot(x+1)$	0.05	$x+[c=0]$	17.29
k-Geo	y	$-k+N+$ $x+y+1$	3	$-k+N$ $+x+y+1$	$[k>N]\cdot y+[k\le N-1]\cdot$ $(-k+N+x+y+1)+$ $[N-1<k\le N]\cdot$ $(-0.5k+0.5N+x+y+1)$	132.76	$[k>N]\cdot y+[k\le N]$ $\cdot(-k+N+x+y+1)$	0.38	$y+[k\le n]\cdot$ $(x-k+n+1)$	76.74
Bin-ran	y	✗	2	$0.9x-21i$ $+y+233$	$[i>10]\cdot y+$ $[\frac{90}{11}<i\le 10]\cdot(0.9x-21i+y+233)$ $+[i\le\frac{90}{11}]\cdot(0.9x-18.8i+y+215)$	106.29	inconsistent results	-	inner error	-
Coin	i	✗	2	$i+\frac{8}{3}$	$[x\neq y]\cdot i+$ $[x=y]\cdot(i+\frac{8}{3})$	104.13	not terminate	-	fail	-
Mart	i	✗	3	$i+2$	$[x\le 0]\cdot i+[x>0]\cdot(i+2)$	19.29	violation of non-negativity	-	$i+[x>0]*2$	37.23
Growing Walk	y	✗	3	$x+y+2$	$[x<0]\cdot y+[x\ge 0]\cdot(x+y+2)$	4.03	violation of non-negativity	-	$y+[x\ge 0]\cdot$ $(x+2)$	21.98
Growing Walk -variant	y	$x+y+1$	3	$x+y+1$	$[x<0]\cdot y+$ $[0\le x<1]\cdot(0.5x+y+0.25)$ $+[x\ge 1]\cdot(x+y)$	125.19	violation of non-negativity	-	not terminate	-
Expected Time	t	✗	3	$4.4280x+t$ $+6.2461$	$[x<0]\cdot t+[0\le x<1]\cdot(t+1)+$ $[1\le x<3.258]\cdot(3.9852x+t+7.39)$ $+[3.258\le x<3.3772]\cdot$ $(4.4280x+t+6.2461)+$ $[3.3772\le x]\cdot(3.5867x+t+9.0874)$	109.35	violation of non-negativity	-	not terminate	-
Zero-Conf -variant	cur	✗	3	$cur+140$	$[est>0]\cdot cur+$ $[start=0\wedge est\le 0]\cdot(cur+140)$ $+[start\ge 1\wedge est\le 0]\cdot(cur+42)$	180.42	violation of non-negativity	-	$cur+[est=0]$ $\cdot(-49\cdot start^2-$ $49\cdot start+141)$	392.19
Equal-Prob-Grid	$goal$	✗	2	$goal+1.5$	$[a>10\vee b>10\vee goal\neq 0]\cdot goal$ $+[a\le 10\wedge b\le 10\wedge goal=0]\cdot 1.5$	142.68	$[a>10\vee b>10\vee$ $goal\neq 0]\cdot goal+$ $[a\le 10\wedge b\le 10$ $\wedge goal=0]\cdot 1.5$	0.11	fail	-
RevBin	z	$2x+z$	3	$2x+z$	$[x<1]\cdot z+[1\le x<2]\cdot(z+x+1)$ $+[x\ge 2]\cdot(z+2x)$	70.30	$[x<1]\cdot z+$ $[x\ge 1]\cdot(z+2x)$	0.22	$z+[x>0]\cdot 2x$	151.26
Fair Coin	i	$i-2y+2$	3	$i+\frac{4}{3}$	$[x>0\vee y>0]\cdot i+$ $[x\le 0\wedge y\le 0]\cdot(i+\frac{4}{3})$	129.34	$[x>0\vee y>0]\cdot i+$ $[x\le 0\wedge y\le 0]\cdot$ $(i+\frac{4}{3})$	0.06	$[x+y=0]\cdot$ $\frac{4}{3}+i$	17.95
St-Petersburg variant	y	✗	3	$\frac{3}{2}y$	$[x>0]\cdot y+[x\le 0]\cdot\frac{3}{2}y$	1.53	$[x>0]\cdot y+$ $[x\le 0]\cdot\frac{3}{2}y$	0.04	$y+[x=0]\cdot$ $0.5y$	13.39

approach derives the exact bound, i.e., the tightest upper bound, on the benchmarks Geo, Coin, k-Geo, Mart, Growing Walk, Equal-Prob-Grid, RevBin, Fair Coin, St-Petersburg variant. The exactness of these bounds is established by comparison with the exact invariants synthesized in [5] (see **RQ2**) and with the piecewise lower bounds presented in our extended version [58, Appendix D.2]. We also show that on a significant number of benchmarks (e.g., k-Geo, Bin-Ran, Growing Walk-variant, Expected Time, etc), the piecewise bounds we synthesize are non-trivial (i.e., the program state space S is partitioned into more than $[\varphi]$ and $[\neg\varphi]$).

Answering RQ2. We answer **RQ2** by comparing our approach with the most related approaches [5, 10]. We present our comparison results in Table 1. The main difference between cegispro2 [10] and our approach is that cegispro2 requires an upper bound to be verified as an additional program input and it will only return a super-invariant (i.e., a possibly piecewise upper-bound) that is sufficient to *verify* (i.e., smaller than) the input upper bound, while we intend to synthesize a tight piecewise upper bound directly. The benchmarks Geo, k-Geo are the common benchmarks in these two works and the direct comparisons are as follows: For the benchmark Geo, the piecewise

upper bounds of the two methods are the same. For k-Geo, their piecewise result is consistent with our result over $\mathbb{Z}_{\geq 0}$. While in the scope of real numbers, our piecewise upper bound is tighter than theirs. To have a richer comparison with cegispro2, we give cegispro2 an advantage by feeding our benchmarks (including the above two benchmarks) in Table 1 to cegispro2 paired with the piecewise upper bounds synthesized by our approach. We find cegispro2 cannot adequately handle piecewise inputs. Additionally, it reports violation of non-negativity on 5 of our benchmarks (see Table 1). By feeding one segment from the piecewise bounds synthesized via our approaches for the remaining 8 benchmarks, we find on 6 benchmarks, cegispro2 produce the consistent results with our inputs on $\mathbb{Z}_{\geq 0}$, while some of them (e.g., k-Geo) are incorrect over $\mathbb{R}$. On Bin-Ran, the results they produce are impossible to compare since it produces sophisticated and different results when we feed different segments from our piecewise upper bound. On Coin, the execution using their tool does not terminate, which prevents the output of a result.

The work [5] considers the probabilistic invariant synthesis via data-driven approach. Note that the synthesis of upper bounds (i.e., super-invariants) is not considered in their work, and the only relevant work in [5] with our upper bound synthesis is the exact invariant synthesis. For a further comparison, We apply their tool exist on our benchmarks to try to generate exact invariants. On the benchmarks Geo, k-Geo, Mart, Growing Walk, RevBin, Fair Coin, St-Petersburg variant, exist can generate an exact invariant for each benchmark and we show that on these benchmarks, the piecewise upper bounds we synthesize are equal to their exact invariants so that the upper bounds we synthesize are actually the exact expected value of X_f . On the benchmark Zero-Conf-Variant, they spend about 400s while we obtain a respectable piecewise linear bound in around 180s. For the remaining benchmarks, their tool fails or the computation seems to be stuck.

In conclusion, our approach can handle many benchmarks that these two works [5, 10] cannot handle. When feeding our benchmarks with the bounds synthesized through our approach to cegispro2 and exist, they fail on about 40% of our benchmarks. Over most of the benchmarks that their and our approaches can handle, our bounds are comparable with theirs.

Answering RQ3. In addition **RQ2**, we compare our piecewise linear upper bounds with monolithic polynomial bounds via 1-induction in Table 2. Following [16, 55], we implement the polynomial synthesis with Putinar's Positivstellensatz [45] (see [58, Appendix C.5] in our extended version). For a fair comparison, we generate the polynomial bounds with the same invariant and optimal objective function for each benchmark. All the numerical results in the polynomial bounds are cut to 10^{-4} precision. We compare two results by uniformly taking the grid points in the invariant and evaluate two results, and we compute the percentage of the points that our piecewise upper bound are larger (i.e., no better) than monolithic polynomial, which is shown in the last column "PCT" in Table 2. For each benchmark, we provide difference plots that classify all grid points into three disjoint regions according to the magnitude of difference: red points correspond to cases where our piecewise linear upper bounds are notably smaller (diff $> 10^{-3}$), blue points correspond to cases where the monolithic upper bounds are notably smaller, and gray points represent regions where the two bounds are nearly identical (diff $\leq 10^{-3}$). We display part of the comparison in Fig. 3, see [58, Appendix D.5] in our extended version for other figures. We observe that on our benchmarks except Expected Time, our piecewise linear bounds are significantly tighter than monolithic polynomial bounds. In addition, we quantify the difference between the two upper bounds by subtracting the piecewise upper bound from the monolithic one and taking the unbiased average of the resulting values, which is reported in the last column "Diff" of Table 2. We observe that, except for Expected Time, our piecewise bounds are generally tighter than the monolithic ones, with especially notable improvements on benchmarks such as Mart and Zero-Conf-Variant. We conjecture that the

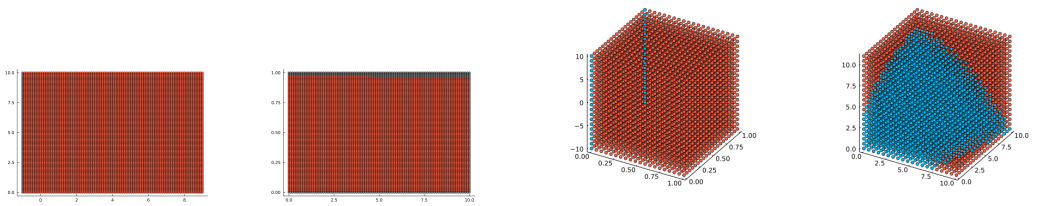
Table 2. Experimental Results for **RQ3**, Linear Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "k" stands for the k-induction condition we apply in this comparison, "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, and "d" stands for the degree of polynomial template we use. "PCT" stands for the percentage of the points that our piecewise upper bound are larger (i.e., no better) than monolithic polynomial, and "Diff" stands for the unbiased average difference between our piecewise polynomial bound and the monolithic polynomial. A positive value indicates how much tighter our piecewise bounds are on average.

Benchmark	f	Our Approach		Monolithic Polynomial via 1-Induction		PCT	Diff
		k	Piecewise Linear Upper Bound	d	Monolithic Polynomial Upper Bound		
Geo	x	3	$[c > 0] \cdot x + [c \leq 0] \cdot (x + 1)$	3	$1.0000 - 1.9996 * c + 1.0000 * x + 0.9996 * c^2 - 0.0002 * x * c + 0.0002 * x * c^2$	0.0%	0.3249
k-Geo	y	3	$[k > N] \cdot y + [k \leq N - 1] \cdot (-k + N + x + y + 1) + [N - 1 < k \leq N] \cdot (-0.5k + 0.5N + x + y + 1)$	2	$269.8049 - 52.761 * N - 1.0000 * k + 1.0000 * y + 1.0000 * x + 2.688 * N^2$	10.0%	77.813
Bin-ran	y	2	$[i > 10] \cdot y + [\frac{90}{11} < i \leq 10] \cdot (0.9x - 21i + y + 233) + [i \leq \frac{90}{11}] \cdot (0.9x - 18.8i + y + 215)$	3	$54.2875 + 26.0308 * i - 38.3708 * y - 20.4776 * x - 1.683 * i^2 - 0.2 * y * i - 0.035 * y^2 - 0.1881 * x * i - 0.4591 * x * y - 1.8397 * x^2 - 0.0129 * i^3 + 0.3751 * y * i^2 + 0.0141 * y^2 * i - 0.0062 * y^3 + 0.5049 * x * i^2 - 0.0123x * y * i - 0.0126 * x * y^2 + 0.9318x^2 * i - 0.0562 * x^2 * y + 1.0057 * x^3$	40.29%	233.49
Coin	i	3	$[x \neq y] \cdot i + [x = y] \cdot (i + \frac{8}{3})$	2	$2.6667 + 1.0000 * i - 0.6381 * y + 4.2840 * x - 2.0286 * y^2 - 2.0067 * x * y + 0.3893 * x^2$	0.0%	3.2915
Mart	i	3	$[x \leq 0] \cdot i + [x > 0] \cdot (i + 2)$	2	$630.7295 + 0.9941 * i + 199099.682 * x + 44.2303 * x^2$	0.0%	$9.9 * 10^5$
Growing Walk	y	3	$[x < 0] \cdot y + [x \geq 0] \cdot (x + y + 2)$	3	$2.5000 + 1.0000 * y + 1.900 * x - 0.5000 * x^2 + 0.1000 * x^3$	0.0%	8.5964
Growing Walk variant	y	3	$[x < 0] \cdot y + [0 \leq x < 1] \cdot (0.5x + y + 0.25) + [x \geq 1] \cdot (x + y)$	3	$1.0000 * y - 0.2380 * x + 0.1041 * y^2 - 0.0686 * x * y + 0.0951 * x^2 + 0.03558 * x * y^2 + 0.0686 * x^2 * y + 0.1430 * x^3$	5.52%	52.315
Expected Time	t	3	$[x < 0] \cdot t + [0 \leq x < 1] \cdot (t + 1) + [1 \leq x < 3.258] \cdot (3.9852x + t + 7.39) + [3.258 \leq x < 3.3772] \cdot (4.4280x + t + 6.2461) + [3.3772 \leq x] \cdot (3.5867x + t + 9.0874)$	3	$3.1203 + 0.9622 * t + 2.8278 * x + 0.0015 * t^2 - 0.01558 * x * t - 0.1397 * x^2 - 0.0003 * x * t^2 - 0.0002 * x^2 * t + 0.0025 * x^3$	82.0%	-11.669
Zero-Conf -variant	cur	3	$[est > 0] \cdot cur + [start == 0 \wedge est \leq 0] \cdot (cur + 140) + [start \geq 1 \wedge est \leq 0] \cdot (cur + 42)$	2	$109.8660 - 0.1357 * cur + 293795.0410 * start + 209178.7117est + 0.0019cur^2 + 0.7202start * cur - 293865.0570 * start^2 + 1.0313 * est * cur + 274251.8886 * est * start - 209283.0750 * est^2$	0.5 %	$2.0 * 10^5$
Equal-Prob-Grid	$goal$	2	$[a > 10 \vee b > 10 \vee goal \neq 0] \cdot goal + [a \leq 10 \wedge b \leq 10 \wedge goal = 0] \cdot 1.5$	2	$1.6661 + 5.7396 * goal - 9.4857 * 10^{-5} * b + 1.5707 * 10^{-5} * a + 0.6003goal^2 - 0.6740b * goal + 1.5975^10 - 5 * b^2 + 2.2074 * 10^{-5} * a * goal$	0.0%	3.757
RevBin	z	3	$[x < 1] \cdot z + [1 \leq x < 2] \cdot (z + x + 1) + [x \geq 2] \cdot (z + 2x)$	2	$z + 0.7098 * x + 1.2902 * x^2$	0.0%	36.909
Fair Coin	i	3	$[x > 0 \vee y > 0] \cdot i + [x \leq 0 \wedge y \leq 0] \cdot (i + \frac{4}{3})$	2	$1.3333 + 1.0000 * i - 0.4141 * y - 0.4141 * x + 1.1743 * i^2 - 2.3486 * y * i + 0.2551 * y^2 - 2.3486 * x * i + 3.6820 * x * y + 0.2551 * x^2$	0.0%	23.677
St-Petersburg variant	y	3	$[x > 0] \cdot y + [x \leq 0] \cdot \frac{3}{2}y$	3	$0.0018 + 1.5006 * y + 808.8832 * x - 0.0112 * y^2 + 7.7505 * x * y - 191.2143 * x^2 + 0.0113 * x * y^2 - 8.251 * x^2 * y - 617.6696 * x^3$	0.0%	192.25

relatively poor performance of our piecewise linear algorithm in Expected Time is due to the fact that the true expected value is closer to a piecewise polynomial function.

6.2 Piecewise Polynomial Bound Synthesis

Benchmark Selection. We select all remaining benchmarks from [5, 10] that are not used in the previous linear experiments, as well as path probability estimation benchmarks from [13, 28, 46, 55], including all unbounded loop benchmarks from [46] in particular. For the former 7 benchmarks from [5], we instantiate the probability parameters with commonly used values (such as 0.5). Note that among them, the benchmarks GeoAr, Bin0, Bin2, Sum0, Duel cannot be handled by our piecewise linear algorithm with k-induction when $k = 1, 2, 3$, even though both the program and the return function are linear. For the benchmarks from [10], the benchmarks chain, brp exhibit



(a) Growing Walk (b) Geo (c) Zero-Conf-variant (d) Bin-Ran

Fig. 3. Difference Plots of the Comparison in Piecewise Linear Case

Red points indicate where our piecewise upper bounds are smaller than the monolithic ones by more than 10^{-3}; blue points indicate where the monolithic bounds are smaller by more than 10^{-3}; gray points denote cases with negligible differences ($\leq 10^{-3}$).

numerical pathologies due to extremely large constants, which can cause numerical instability and render our algorithms ineffective. To address this issue, we scale down these pathological values to more moderate magnitudes—for instance, replacing 1000000000000 with 100 in the chain benchmark and 8000000000 with 800 in the brp benchmark—so that our numerical algorithm can operate reliably. For the benchmarks from [13, 28, 46, 55], since 5 of 9 benchmarks contain continuous distributions originally, we make simple adaptions on these benchmarks by replacing each continuous distribution (e.g. uniform distribution over $[0, 1]$) with a uniform discrete choice of the same range (e.g. 0 with probability 0.5 and 1 also with 0.5), resulting in 5 adapted benchmarks. The benchmark inv-Pend in [46] does not pass our checking of prerequisite (P2). Therefore we make minor modifications to the coefficients in this benchmark so that we can synthesize a dbRSM to satisfy (P2), thereby obtaining the benchmark inv-Pend variant. For piecewise polynomial bound synthesis, larger values of k significantly increase the computational cost. Therefore, we apply 2-induction on these 24 benchmarks.

Answering RQ1. Our algorithm successfully handles all of the aforementioned benchmarks except for four. The failures in these cases are attributed to excessive branching introduced by our algorithm based on Proposition 5.2 (see **Stage 2** in Section 5), and branch reduction techniques (see Page 13) have not yet been incorporated into our implementation. Nevertheless, our current implementation is capable of addressing a wide range of complex benchmarks. For example, the benchmark cav5 comprises 35 lines of code (see [58, Appendix E] in our extended version), the benchmark inv-Pend variant benchmark features 4 variables with complex polynomial updates, posing significant challenges for analysis. We leave further optimization for future work. We present the experimental results for the synthesis of piecewise polynomial upper bounds on the remaining 20 benchmarks in Table 3. Our approach successfully derives piecewise polynomial upper bounds for 16 out of 20 benchmarks within seconds. Of the remaining four, two benchmarks (fig-6 and fig-7) are solved within tens of seconds, while only inv-Pend variant and cav-5 require more than five minutes to compute a result. Our algorithms obtain the exact bound (i.e., the tightest upper bound) on the benchmarks Bin0, Bin2, DepRV, Prinsys, Sum0. The exactness of these results is verified by comparison with the exact invariants synthesized in [5] (see **RQ2**) and with our corresponding lower bounds in our extended version [58, Appendix D.3] .

Answering RQ2. We answer **RQ2** by comparing our approach with the relevant work exist [5] in Table 3, whose illustration is the same to Table 1. It is worth noting that cegispro2 only supports linear bounds and does not accept nonlinear expressions as additional program input. Therefore, we exclude it from our comparison. Note that the only relevant aspect of [5] with respect to

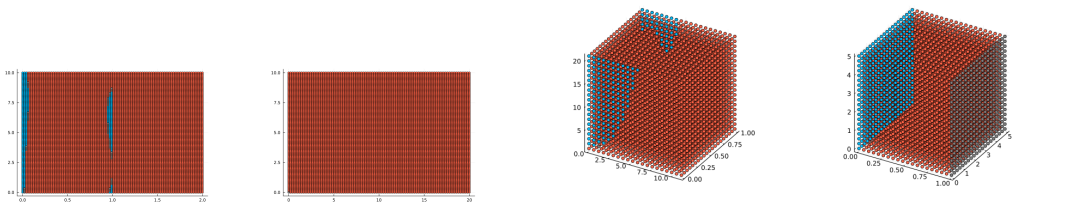
Table 3. Experimental Results for **RQ1** and **RQ2**, Polynomial Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "T(s)" stands for the execution time of our approach (in seconds), including the parsing procedure from the program input, relaxing the k-induction constraint into the SDP problems, the SDP solving time and verification time. "d" stands for the degree of polynomial template we use and "Solution h^*" is the candidate polynomial solved directly by the solver. "Piecewise Polynomial upper Bound" stands for the piecewise bound we synthesize. "Exact" stands for the exact expected result from EXIST.

Benchmark	f	Our Approach				EXIST	
		d	Solution h^*	T(s)	Piecewise Polynomial Upper Bound	Exact	T(s)
GeoAr	x	2	$0.0001*x^2-0.0004*x*y+0.0005*x*z$ $+0.0011*y^2+0.0079*z^2+0.9998*x$ $+1.5398*y-0.0085*z+5.0078$	3.92	$\min\{[z>0]\cdot(0.0001x^2-0.0003*x*y+$ $0.0003*x*z+0.0010y^2+0.0003*y*z$ $+0.0040z^2+0.9995x+2.0416y-$ $0.004z+7.0485)+[z\le 0]\cdot x, h^*\}$	inner error	-
Bin0	x	2	$x+0.5*y*n$	6.05	$x+[n>0]\cdot 0.5*y*n$	$x+[n>0]\cdot$ $0.5*y*n$	79.04
Bin2	x	2	$0.25*n+x+0.25*n^2+0.5*y*n$	5.81	$x+[n>0]\cdot(0.25*n+x$ $+0.25*n^2+0.5*y*n)$	$x+[n>0]\cdot$ $(0.25*n$ $+x+0.25*n^2$ $+0.5*y*n)$	250.60
DepRV	$x*y$	2	$-0.25*n+0.25*n^2+0.5*y*n$ $+0.5*x*n+x*y$	5.91	$[n>0]\cdot(-0.25*n+0.25*n^2+$ $0.5*y*n+0.5*x*n+x*y)$ $+[n\le 0]\cdot x*y$	inner error	-
Prinsys	$[x==1]$	2	$0.5+0.5*x$	2.35	$[x==1]*1+[x==0]*0.5$	$[x==1]*1+$ $[x==0]*0.5$	3.02
Sum0	x	2	$0.25*i^2+0.25*i+x$	2.33	$[i>0]*(0.25*i^2+0.25*i)+x$	$x+[i>0]*$ $(0.25i+0.25i^2)$	105.01
Duel	t	2	$-20.267*x^2-0.4198*x*t-2.5502*t^2$ $+20.6657*x+3.5505*t+0.0013$	6.9	$\min\{[t>0\wedge x\ge 1]\cdot(-10.1335x^2-2.5502t^2$ $+0.2099*x*t+10.1230*x+2.5502*t$ $+0.5015)+[t\le 0\wedge x\ge 1]\cdot(-5.0668*x^2$ $+0.1050*x*t-2.5502*t^2+5.0615*x$ $+3.0504*t+0.2514)+[x<1]\cdot t, h^*\}$	fail	-
brp	$[failed=10]$	2	$38912.3699*failed^2+0.7329*sent^2+$ $3.2173*failed*sent+1486.258*failed$ $-573.6644*sent-2459.9909$	10.12	$\min\{[failed<10\wedge sent<800]\cdot(0.7329sent^2$ $+0.0322*failed*sent+389.1237*failed^2$ $+793.1100*failed-572.1811*sent$ $-2623.2068)+[failed=10], h^*\}$	not terminate	-
chain	$[y=1]$	2	$-0.006*x*y+0.4841*y^2-$ $0.0021*x+0.4477*y+0.1007$	4.79	$\min\{[y=0\wedge x<100]\cdot(-0.0059*x*y$ $+0.4793*y^2-0.0022*x+0.4373*y$ $+0.1079)+[y=1], h^*\}$	fail	-
grid small	$[a<10\wedge$ $b\ge 10]$	3	$0.0018*a*b^2-0.0003*a^3-0.0008*a^2*b$ $-0.0011*b^3+0.0117*a^2-0.0154*a*b+$ $0.0136*b^2-0.097*a+0.0239*b+0.5355$	6.71	$\min\{[a<10\wedge b<10]\cdot(-0.0003*a^3-$ $0.0011*b^3-0.0008*a^2*b+0.0018*a*b^2$ $+0.0109*a^2\cdots+0.0277*b$ $+0.5109)+[a<10\wedge b\ge 10], h^*\}$	Not support	-
grid big	$[a<1000\wedge$ $b\ge 1000]$	2	$0.0159*a^2-0.0319*a*b+0.0159*b^2$ $+0.2715*a-0.3086*b-0.437$	7.74	$\min\{[a<1000\wedge b<1000]\cdot(0.0159*a^2$ $-0.0319*a*b+0.0159*b^2+0.2714*a$ $-0.3087*b-0.4397)+$ $[a<1000\wedge b\ge 1000], h^*\}$	Not support	-
cav-2	$[h>1+t]$	3	0.0	3.78	$[h>t+1]$	fail	-
cav-4	$[x\le 10]$	2	1.0	2.75	1.0	inner error	-
fig-6	$[y\le 5]$	4	$0.0011*x^3*y-0.0001*x^4-0.0001*y^4$ $+0.0008*x*y^3-0.001*x^2*y^2+\cdots$ $+0.5712*x-0.281*y+0.6009$	109.03	$\min\{[x\le 4]\cdot(-0.0001*x^4+0.0011*x^3*y$ $-0.001*x^2*y^2+0.0008*x*y^3-0.0001*y^4$ $+0.0023*x^3\cdots-0.0094*y^2+0.5530*x-$ $0.2782*y+0.6027)+[x>4\wedge y\le 5], h^*\}$	inner error	-
fig-7	$[x\le 1000]$	2	$0.0005*y^2-0.0008*y*i$ $+0.0002*i^2-0.0001*x+$ $0.001*y-0.0005*i+1.0003$	24.32	$\min\{[y\le 0]\cdot(0.0002*i^2-0.0002*x-$ $0.0005*i+1.0004)+[y>0\wedge x\le 1000], h^*\}$	inner error	-
inv-Pend variant	$[pA\le 1]$	3	$0.0058*pAD^2*pA+0.0023*pAD^2*cV-$ $0.1313*pAD^2*cP-0.6278*pAD*pA^2-$ $0.2352pAD*pA*cV\cdots+5.9637*cV*cP$ $+60.4194*cP^2+7.1495*cV+1.0$	412.20	$\min\{[cp>0.5\vee cp<-0.5\vee pA>0.1\vee$ $pA<-0.1]\cdot(0.0058pAD^2pA-0.0011pAD^2cV$ $-0.1313pAD^2*cP+\cdots+0.0689*cP+0.3238)$ $+[-0.5\le cp\le 0.5\wedge -0.1\le pA\le 0.1], h^*\}$	inner error	-
CAV-7	$[x\le 30]$	3	$-0.0001*i^3+0.0002*i^2*x$ $+0.0011*i^2-0.0012*i*x-$ $0.0009*i-0.0001*x+0.9993$	5.26	$\min\{[i<5]\cdot(0.0001*i^2*x+0.0005*i^2-$ $0.0006*i*x+0.0004*i-0.0011*x$ $+0.9983)+[i<5\wedge x\le 30], h^*\}$	inner error	-
cav-5	$[i\le 10]$	3	$-0.0001*i*money^2-0.0004*i^2-$ $0.0006*i*money+0.1029*money^2$ $+0.0037*i+1.0$	892.6	$\min\{[money\ge 10]\cdot(-0.0001*i*money^2-$ $0.0004*i^2-0.0004*i*money+0.0015*i$ $+0.1028*money^2-0.2118*money+$ $3.1283)+[money<10\wedge i\le 10], h^*\}$	inner error	-
Add	$[x>5]$	3	$0.9402-0.1887*y+0.0833*x-0.0751*y^2$ $+0.0556*x*y-0.0107*x^2-0.0281*y^3$ $+0.023*x*y^2-0.0052*x^2*y+0.0005*x^3$	3.63	$\min\{[y\le 1]\cdot(0.9403-0.1953*y+0.0831*x$ $-0.0736*y^2+0.0565*x*y-0.0105*x^2$ $-0.0281*y^3+0.023*x*y^2-0.0052*x^2*y$ $+0.0005*x^3)+[y>1\wedge x>5], h^*\}$	inner error	-
Growing Walk variant2	y	2	$0.0622*x^2-1.2722*x*r+6.5027*r^2$ $+0.6396*x+y-6.5379*r+1.6433$	5.33	$\min\{[r\le 0]\cdot(0.0622*x^2+0.6279*x$ $+y+1.6914)+[r>0]\cdot y, h^*\}$	inner error	-

upper bound synthesis (i.e., super-invariants) is their method for exact invariant synthesis. For comparison, we apply their tool EXIST to our benchmarks in an attempt to generate exact invariants. On the benchmarks BIN0, BIN2, PRINSYS, and SUM0, we show that the piecewise polynomial upper bounds we synthesize are actually the exact expected value of X_f, i.e., the tightest upper bounds, by comparing them with the exact invariants synthesized by EXIST. Among these benchmarks, EXIST spends about 80s on BIN0, about 250s on BIN2, and about 100s on SUM0, while we spend only several seconds to obtain the specific results. Thus, our algorithm is much more efficient. For the benchmarks DUEL, CHAIN, and CAV2, the tool EXIST is able to identify candidates for exact invariants but fails to verify them, and thus does not produce exact invariants. Additionally, EXIST does not support the benchmarks GRID SMALL and GRID BIG. For the remaining benchmarks, EXIST fails to generate results due to internal errors. Moreover, for the benchmark DEPRV, we demonstrate that the piecewise polynomial upper bound synthesized by our approach is exact, as it coincides with the corresponding lower bound (see [58, Appendix D.3] in our extended version). Thus, our method yields the tightest upper bound for 5 out of the 20 benchmarks in this table. In summary, our approach successfully handles more benchmarks than [5], and for those benchmarks that both methods can process, our approach is more efficient and produces comparable bounds.

Answering RQ3. In addition to the comparisons in **RQ2**, we further evaluate our piecewise polynomial upper bounds (obtained via k-induction) against monolithic polynomial bounds of higher degree synthesized using simple induction (i.e., 1-induction). The synthesis of these monolithic polynomial bounds is implemented using Putinar's Positivstellensatz [45] (see [58, Appendix C.5] in our extended version for details). For a fair comparison, we use the same invariant and optimal objective function for each benchmark. We also verify the validity of the monolithic polynomial bounds (see *Numerical Repair*). In our experimental evaluation, we observe that for most benchmarks, when the degree of the polynomial template exceeds 5, numerical performance deteriorates and the synthesized monolithic bounds fail our validation process. Therefore, in this experiment, we restrict the degree of monolithic polynomial bounds to at most 5.

We present the comparison results in Table 4, whose illustration is the same to Table 2. To compare the two synthesized bounds, we uniformly sample grid points from a region of interest (typically a subset of the invariant) and evaluate both results at these points. We compute the percentage of points where our piecewise polynomial upper bound is larger (i.e., no better) than the (higher degree) monolithic polynomial, which is shown in the column "PCT" in Table 4. For each benchmark, we provide difference plots that classify all grid points into three disjoint regions according to the magnitude of difference: red points correspond to cases where piecewise linear upper bounds are notably smaller (diff $> 10^{-3}$), blue points correspond to cases where monolithic upper bounds are notably smaller, and gray points represent regions where the two bounds are nearly identical (diff $\leq 10^{-3}$). We display part of the comparison in Fig. 4, see [58, Appendix D.6] in our extended version for other figures. We show that on all the benchmarks except GRID SMALL, GRID BIG, FIG-6, our piecewise polynomial bounds are *significantly* tighter and simpler than monolithic polynomial bounds. In addition, we quantify the difference between two upper bounds by subtracting the piecewise upper bound from the monolithic one and taking the unbiased average of the resulting values, which is reported in the column "Diff" of Table 4. We observe that, on the benchmarks BIN0, DEPRV, PRINSYS, SUM0, our piecewise polynomial result is consistent with the monolithic one, which makes both the "PCT" and "Diff" values nearly zero. On the benchmarks BIN2, GRID SMALL, CAV-2, FIG-6, INV-PEND VARIANT, ADD, our piecewise upper bounds are close to the monolithic upper bounds on average. On the remaining benchmarks, our piecewise bounds are generally tighter than the monolithic ones, with especially notable improvements on benchmarks such as CHAIN and GRID BIG. Although our running time is a bit longer than that of monolithic polynomial experiments, our



(a) Add (b) cav-5 (c) fig-7 (d) Growing Walk variant2

Fig. 4. Difference Plots of the Comparison in Piecewise Polynomial Case

Red points indicate where our piecewise upper bounds are smaller than the monolithic ones by more than 10^{-3}; blue points indicate where the monolithic bounds are smaller by more than 10^{-3}; gray points denote cases with negligible differences ($\leq 10^{-3}$).

approach allows to synthesize lower-degree polynomials while achieving better precision against higher-degree polynomials. This advantage is critical as the synthesis of higher-degree polynomials suffers from a large amount of numerical errors as stated previously.

7 Related Works & Conclusion

In this work, we propose a novel approach to synthesize piecewise probabilistic bounds for probabilistic programs. Further improvements include optimization on the branch reduction and the constraint solving of latticed k-induction constraints with minimum. Below we compare our approach with most related approaches.

Compared with previous approaches (e.g. [15, 16, 18]) that mostly focus on synthesizing monolithic bounds over probabilistic programs, our approach targets piecewise bounds, and hence is orthogonal. The work [11] proposes latticed k-induction. We claim that their work differs significantly from ours. They do not synthesize bounds and only verify whether a given bound is an upper bound or not. The work [10] synthesize piecewise linear bounds to verify the input upper bound via counterexample-guided inductive synthesis (CEGIS), while we do not need this additional input bound and we solve the bounds by bilinear and semidefinite programming rather than CEGIS. For the verification of lower bounds, their work applies a proof rule in [27, 31] derived from the original OST, while our approach applies extended OST. The work [5] synthesizes (piecewise) exact invariants and sub-invariants (to verify the input lower bound) via data-driven learning. Their work additionally requires a list of features composed of numerical expressions, while our approach captures the piecewise feature via k-induction automatically and without such additional inputs. The works [13, 55, 59] focus on deriving bounds for the posterior distribution in Bayesian probabilistic programs, whereas our work aims at deriving piecewise bounds for the expected output of the probabilistic programs.

Other approaches [3, 7, 8, 35] focus on moment-based invariants generation and high-order moments derivation for probabilistic programs. These works can even handle the probabilistic program with non-polynomial expressions and continuous distributions, but they only consider the probabilistic while loop in a rather restricted form: **while** `true` $\{C\}$. The work [41] enlarges the theoretical foundation through the assumption that all variables appearing in if-conditions (loop guards) are finitely valued , and [43] further provides an algorithm about computing the strongest polynomial moment invariants for this kind of loops, but their works still cannot handle most of our benchmarks. Our approach can handle all the polynomial forms of loop guards and if-conditions. In a similar vein, the works [38, 52] bound higher central moments for running time and other monotonically increasing quantities, but are limited to programs with constant size increments.

Table 4. Experimental Results for **RQ3**, Polynomial Case (Upper Bounds). "f" stands for the return function considered in the benchmark. "Piecewise Polynomial Upper Bound" stands for the results synthesized by our algorithm. "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, and "T(s)" stands for the total execution time. "PCT" stands for the percentage of the points that our piecewise polynomial upper bound are larger (i.e., no better) than (higher degree) monolithic polynomial, and "Diff" stands for the unbiased average difference between our piecewise polynomial bound and the monolithic polynomial. A positive value indicates how much tighter our piecewise bounds are on average.

Benchmark	f	Our Approach			Monolithic Polynomial via 1-induction			PCT	Diff
		d	T(s)	Piecewise Polynomial Upper Bound	d	T(s)	Monolithic Polynomial Upper Bound		
GeoAr	x	2	3.92	$\min\{[z>0]\cdot(0.0001x^2-0.0003*x*y+0.0003*x*z+0.0010y^2+0.0003*y*z+0.0040z^2+0.9995x+2.0416y-0.004z+7.0485)+[z\le 0]\cdot x, h^*\}$	3	0.84	$-0.0001x^3+0.0001*x^2*y-0.0011*x^2*z-0.0004*x*y^2-0.0112*x*y*z+0.164*x*z^2+0.0012*y^3+\cdots-0.0137*y^2+2.7194*y*z+0.9993*x+0.0417*y+89867.2768*z+0.078$	5.0%	4976.4
Bin0	x	2	6.05	$x+[n>0]\cdot 0.5*y*n$	3	0.65	$0.5*y*n+x$	0.0%	0.019
Bin2	x	2	5.81	$x+[n>0]\cdot(0.25*n+x+0.25*n^2+0.5*y*n)$	3	0.53	$-0.0001*x*y^2-0.0002*x*y*n-0.0001*x*n^2+\cdots+0.2496*n^2+0.9986*x+0.033*y+0.2641*n+0.051$	9.2%	0.014
DepRV	$x*y$	2	5.91	$[n>0]\cdot(-0.25*n+0.25*n^2+0.5*y*n+0.5*x*n+x*y)+[n\le 0]\cdot x*y$	3	0.74	$x*y+0.5*x*n+0.5*y*n+0.25*n^2-0.2499*n+0.0001$	0.0%	0.0005
Prinsys	$[x==1]$	2	2.35	$[x==1]*1+[x==0]*0.5$	3	0.75	$0.2973*x^3+0.2027*x+0.5$	0.0%	0.0
Sum0	x	2	2.33	$[i>0]*(0.25*i^2+0.25*i)+x$	4	0.7	$0.25*i^2+0.25*i+x$	0.0%	0.0
Duel	t	2	6.90	$\min\{[t>0\wedge x\ge 1]\cdot(-10.1335x^2-2.5502t^2+0.2099*x*t+10.1230*x+2.5502*t+0.5015)+[t\le 0\wedge x\ge 1]\cdot(-5.0668*x^2+0.1050*x*t-2.5502*t^2+5.0615*x+3.0504*t+0.2514)+[x<1]\cdot t, h^*\}$	4	0.92	$-175.0474x^4-33.1201x^3t-256.8154*x^2*t^2+74.5673*x*t^3+81.1314*t^4-115.4608*x^3+153.7459*x^2*t-125.7204*x*t^2-104.9856t^3+78.3171*x^2+186.7714*x*t-135.7646*t^2+212.334*x+160.6187*t$	0.02%	119.7
brp	$[failed=10]$	2	10.12	$\min\{[failed<10\wedge sent<800]\cdot(0.7329sent^2+0.0322*failed*sent+389.1237*failed^2+793.1100*failed-572.1811*sent-2623.2068)+[failed=10], h^*\}$	4	1.27	$5.6049failed^4+4.902failed^3sent+3.2666failed^3-0.0035failed^2sent^2-7.0269*failed^2*sent+0.0019*failed*sent^2+2.9608*failed*sent-0.0001sent^3+5.1816*failed^2-0.0288*sent^2+2.4293*failed-7.3179*sent-0.9176$	28.8%	2964.2
chain	$[y=1]$	2	4.79	$\min\{[y=0\wedge x<100]\cdot(-0.0059*x*y+0.4793*y^2-0.0022*x+0.4373*y+0.1079)+[y=1], h^*\}$	3	1.15	$-0.0449*x^3-0.5045*x^2*y+5.611*x*y^2-155242.5616*y^3+5.5921*x^2+43.0661*x*y-668140.0947*y^2-117.5705*x+823721.7882*y+160.1718$	0.85%	$1.2*10^5$
grid small	$[a<10\wedge b\ge 10]$	3	6.71	$\min\{[a<10\wedge b<10]\cdot(-0.0003*a^3-0.0011*b^3-0.0008*a^2*b+0.0018*a*b^2+0.0109*a^2\cdots+0.0277*b+0.5109)+[a<10\wedge b\ge 10], h^*\}$	4	1.16	$0.0001*a^3-0.0003*a^2*b+0.0002*a*b^2-0.0001*b^3-0.0002*a^2+0.0001*a*b+0.0003*b^2-0.0326*a+0.0322*b+0.4628$	43.54%	0.0014
grid big	$[a<1000\wedge b\ge 1000]$	2	7.74	$\min\{[a<1000\wedge b<1000]\cdot(0.0159*a^2-0.0319*a*b+0.0159*b^2+0.2714*a-0.3087*b-0.4397)+[a<1000\wedge b\ge 1000], h^*\}$	3	0.83	$-809.0361-1408.2916*b+1397.5134*a+2.9331*b^2-5.8658*a*b+2.9431*a^2-0.0004*b^3+0.0006*a*b^2+0.0011*a^2*b-0.0012*a^3$	34.49%	$3.8*10^5$
cav-2	$[h>t+1]$	3	3.78	$[h>t+1]$	4	0.75	$0.0008*h^2-0.001*h*t+0.001*t^2-0.0066*h-0.0073*t+0.0885$	0.0%	0.0489
cav-4	$[x\le 10]$	2	2.75	1.0	3	0.62	$0.0007*x*y^2-20.236*y^3-0.0007*x*y+13.2821*y^2+6.9539*y+1.0$	0.0%	364.32
fig-6	$[y\le 5]$	4	109.03	$\min\{[x\le 4]\cdot(-0.0001*x^4+0.0011*x^3*y-0.001*x^2*y^2+0.0008*x*y^3-0.0001*y^4+0.0023*x^3\cdots-0.0094*y^2+0.5530*x-0.2782*y+0.6027)+[x>4\wedge y\le 5], h^*\}$	5	1.12	$-0.0001*x^3-0.0002*x^4*y-0.0003*x^2*y^3+0.0001*x*y^4-0.0002*y^5+0.0011*x^4+0.0037*x^3*y\cdots+0.1432*x*y+0.0064*y^2+0.9708*x-0.6526*y+0.575$	42.73%	0.2507
fig-7	$[x\le 1000]$	2	24.32	$\min\{[y\le 0]\cdot(0.0002*i^2-0.0002*x-0.0005*i+1.0004)+[y>0\wedge x\le 1000], h^*\}$	3	2.65	$0.0003*x^2*i-0.083*x^2*y+48.5638*x*y^2+0.5267*x*y*i-0.018*x*i^2+2600.9691*y^3-36.705*y^2*i-2.646*y*i^2\cdots-3.3923*x+56310.8279*y-0.0114*i+7.2868$	2.58%	11570.1
inv-Pend variant	$[pA\le 1]$	3	412.20	$\min\{[cp>0.5\vee cp<-0.5\vee pA>0.1\vee pA<-0.1]\cdot(0.0058pAD^2pA-0.0011pAD^2cV-0.1313pAD^2*cP+\cdots+0.0689*cP+0.3238)+[-0.5\le cp\le 0.5\wedge -0.1\le pA\le 0.1], h^*\}$	4	7.42	$0.2264*pAD^4+1.1448*pAD^3*pA-0.1026*pAD^3*cV-0.1107*pAD^3*cP+5.2869*pAD^2*pA^2+\cdots+10.6625*cP^2-0.0001*pA+53.8573*cV+1.0$	4.04%	0.0
CAV-7	$[x\le 30]$	3	5.26	$\min\{[i<5]\cdot(0.0001*i^2*x+0.0005*i^2-0.0006*i*x+0.0004*i-0.0011*x+0.9983)+[i<5\wedge x\le 30], h^*\}$	4	1.17	$0.0007*i^4-0.0011*i^3*x+0.0005*i^2*x^2-0.0001*i*x^3-0.0045*i^3+0.0052*i^2*x-0.0012*i*x^2+0.0134*i^2-0.012*i*x+0.002*x^2-0.0135*i+0.0046*x+1.0034$	37.37%	0.0049
cav-5	$[i\le 10]$	3	892.6	$\min\{[money\ge 10]\cdot(-0.0001*i*money^2-0.0004*i^2-0.0004*i*money+0.0015*i+0.1028*money^2-0.2118*money+3.1283)+[money<10\wedge i\le 10], h^*\}$	4	1.27	$0.0001*i^2*money^2+0.0002*i*money^3+0.0001*money^4+0.0184*i^2*money-0.0396*i*money^2-0.0168*money^3+0.0009*i^3-0.0291*i^2+2.8701*i+0.2414*i*money+4.264*money^2+1.0$	0.0%	507.20
Add	$[x>5]$	3	3.63	$\min\{[y\le 1]\cdot(0.9403-0.1953*y+0.0831*x-0.0736*y^2+0.0565*x*y-0.0105*x^2-0.0281*y^3+0.023*x*y^2-0.0052*x^2*y+0.0005*x^3)+[y>1\wedge x>5], h^*\}$	4	0.81	$0.9205+0.6262*y+0.0819*x-1.6833*y^2+0.0397*x*y-0.0119*x^2+0.9726*y^3+0.0598*x*y^2-0.0059*x^2*y+0.0006*x^3-0.2007*y^4-0.0104*x*y^3-0.0009*x^2*y^2+0.0001*x^3*y$	3.84 %	0.253
Growing Walk variant2	y	2	5.33	$\min\{[r\le 0]\cdot(0.0622*x^2+0.6279*x+y+1.6914)+[r>0]\cdot y, h^*\}$	3	1.22	$0.999*x*r^2+0.0008*y*r^2+700.3292*r^3-1.999*x*r-0.0008*y*r-1399.6591*r^2+x+y+698.3298*r+1.0001$	5.0 %	211.91

Data-Availability Statement

The artifact of this paper is available at the link [57].

Acknowledgments

We thank the anonymous reviewers of POPL 2026 for their valuable comments and helpful suggestions. This work has been partially funded by the National Key R&D Program of China under grant No. 2022YFA1005100 and 2022YFA1005101, and the National Natural Science Foundation of China under grant No. 62192732, W2511064, 62172271, 62502475.

References

[1] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2021. Learning Probabilistic Termination Proofs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 3–26. https://doi.org/10.1007/978-3-030-81688-9_1

[2] Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. A pre-expectation calculus for probabilistic sensitivity. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434333

[3] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. 2022. Solving Invariant Generation for Unsolvable Loops. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 19–43. https://doi.org/10.1007/978-3-031-22308-2_3

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

[5] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

[6] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 749–758. https://doi.org/10.1145/2933575.2934554

[7] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 255–276. https://doi.org/10.1007/978-3-030-31784-3_15

[8] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020. Mora - Automatic Generation of Moment-Based Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12078)*, Armin Biere and David Parker (Eds.). Springer, 492–498. https://doi.org/10.1007/978-3-030-45190-5_28

[9] Kevin Batz, Tom Jannik Biskup, Joost-Pieter Katoen, and Tobias Winkler. 2024. Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2792–2820. https://doi.org/10.1145/3632935

[10] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 410–429. https://doi.org/10.1007/978-3-031-30820-8_25

[11] Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 524–549. https://doi.org/10.1007/978-3-030-81688-9_25

[12] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language*

Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022, Ranjit Jhala and Isil Dillig (Eds.). ACM, 536–551. https://doi.org/10.1145/3519939.3523721

[13] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 536–551. https://doi.org/10.1145/3519939.3523721

[14] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 33–52. https://doi.org/10.1145/2509136.2509546

[15] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

[16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

[17] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 327–342. https://doi.org/10.1145/2837614.2837639

[18] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2018. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *ACM Trans. Program. Lang. Syst.* 40, 2 (2018), 7:1–7:45. https://doi.org/10.1145/3174800

[19] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 145–160. https://doi.org/10.1145/3009837.3009873

[20] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[22] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. 2003. Bounded Model Checking and Induction: From Refutation to Verification (Extended Abstract, Category A). In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 14–26. https://doi.org/10.1007/978-3-540-45069-6_2

[23] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software Verification Using k-Induction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[24] J. L. Doob. 1971. What is a Martingale? *The American Mathematical Monthly* 78, 5 (1971), 451–463. https://doi.org/10.1080/00029890.1971.11992788 arXiv:https://doi.org/10.1080/00029890.1971.11992788

[25] Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 696–726. https://doi.org/10.1145/3586051

[26] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26

[27] Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.).

Springer, 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

[28] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[29] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 167–181. https://doi.org/10.1145/2593882.2593900

[30] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2013. Prinsys - On a Quest for Probabilistic Loop Invariants. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8054)*, Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio (Eds.). Springer, 193–208. https://doi.org/10.1007/978-3-642-40196-1_17

[31] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 37:1–37:28. https://doi.org/10.1145/3371105

[32] William B. Jones and W. J. Thron. 1984. Continued Fractions: Analytic Theory and Applications. https://api.semanticscholar.org/CorpusID:118226015

[33] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

[34] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018), 30:1–30:68. https://doi.org/10.1145/3208102

[35] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. 2023. Exact and Approximate Moment Derivation for Probabilistic Loops With Non-Polynomial Assignments. *CoRR* abs/2306.07072 (2023). https://doi.org/10.48550/ARXIV.2306.07072 arXiv:2306.07072

[36] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

[37] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. 2019. Interpolating Strong Induction. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 367–385. https://doi.org/10.1007/978-3-030-25543-5_21

[38] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 135–153. https://doi.org/10.1007/978-3-030-17465-1_8

[39] Jia Lu and Ming Xu. 2022. Bisection Value Iteration. In *29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022*. IEEE, 109–118. https://doi.org/10.1109/APSEC57359.2022.00023

[40] Garth P. McCormick. 1976. Computability of global solutions to factorable nonconvex programs: Part I - Convex underestimating problems. *Math. Program.* 10, 1 (1976), 147–175. https://doi.org/10.1007/BF01580665

[41] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. 2022. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1497–1525. https://doi.org/10.1145/3563341

[42] Theodore Samuel Motzkin. 1936. Beiträge zur Theorie der linearen Ungleichungen. *(No Title)* (1936).

[43] Julian Müllner, Marcel Moosbrugger, and Laura Kovács. 2024. Strong Invariants Are Hard: On the Hardness of Strongest Polynomial Invariants for (Probabilistic) Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 882–910. https://doi.org/10.1145/3632872

[44] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512. https://doi.org/10.1145/3192366.3192394

[45] Mihai Putinar. 1993. Positive Polynomials on Compact Semi-algebraic Sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984. http://www.jstor.org/stable/24897130

[46] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 447–458. https://doi.org/10.1145/2491956.2462179

[47] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[48] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[49] Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proc. ACM Program. Lang.* 3, POPL (2019), 39:1–39:31. https://doi.org/10.1145/3290352

[50] Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 5:1–5:46. https://doi.org/10.1145/3450967

[51] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). arXiv:1809.10756 http://arxiv.org/abs/1809.10756

[52] Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 559–573. https://doi.org/10.1145/3453483.3454062

[53] Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1171–1186. https://doi.org/10.1145/3453483.3454102

[54] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 204–220. https://doi.org/10.1145/3314221.3314581

[55] Peixin Wang, Tengshun Yang, Hongfei Fu, Guanyan Li, and C.-H. Luke Ong. 2024. Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving. *Proc. ACM Program. Lang.* 8, PLDI, Article 202 (jun 2024), 26 pages. https://doi.org/10.1145/3656432

[56] David Williams. 1991. *Probability with Martingales.* Cambridge University Press.

[57] Tengshun Yang, Shenghua Feng, Hongfei Fu, Naijun Zhan, Jingyu Ke, and Shiyang Wu. 2025. Artifact of POPL 560. (10 2025). https://doi.org/10.6084/m9.figshare.30354097.v4

[58] Tengshun Yang, Hongfei Fu, Jingyu Ke, Naijun Zhan, and Shiyang Wu. 2024. Piecewise Linear Expectation Analysis via k-Induction for Probabilistic Programs. *CoRR* abs/2403.17567 (2024).

[59] Fabian Zaiser, Andrzej S. Murawski, and C.-H. Luke Ong. 2025. Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops. *Proc. ACM Program. Lang.* 9, POPL (2025), 1104–1135. https://doi.org/10.1145/3704874

Received 2025-07-10; accepted 2025-11-06