Finding Polynomial Loop Invariants for Probabilistic Programs

Yijun Feng¹, Lijun Zhang², David N. Jansen², Naijun Zhan², and Bican Xia¹

¹ LMAM & School of Mathematical Sciences, Peking University
² State Key Laboratory of Computer Science, Institute of Software, CAS

Abstract. Quantitative loop invariants are an essential element in the verification of probabilistic programs. Recently, multivariate Lagrange interpolation has been applied to synthesizing polynomial invariants. In this paper, we propose an alternative approach. First, we fix a polynomial template as a candidate of a loop invariant. Using Stengle’s Positivstellensatz and a transformation to a sum-of-squares problem, we find sufficient conditions on the coefficients. Then, we solve a semidefinite programming feasibility problem to synthesize the loop invariants. If the semidefinite program is unfeasible, we backtrack after increasing the degree of the template. Our approach is semi-complete in the sense that it will always lead us to a feasible solution if one exists. Experimental results show the efficiency of our approach.

1 Introduction

Probabilistic programs extend standard programs with probabilistic choices and are widely used in protocols, randomized algorithms, stochastic games, etc. In such situations, the program will usually report incorrect results with a certain probability, rendering classical program specification methods [11,18] inadequate. As a result, formal reasoning about the correctness needs to be based on quantitative specifications. Typically, a probabilistic program consists of steps that choose probabilistically between several states, and the specification of a probabilistic program contains constraints on the probability distribution of final states, e.g. through the expected value of a random variable. Therefore the expected value is often considered in correctness verification [22,20,14].

To reason about correctness for probabilistic programs, quantitative annotations are needed. Most importantly, correctness of while loops can be proved by inferring special bounds on expectations, usually called quantitative loop invariants [22]. As in the classical setting, finding such invariants is the bottleneck of proving program correctness. For some restricted classes, such as linear loop invariants, some techniques have been established [21,20,13]. To use them to synthesize polynomial loop invariants, so-called linearization can be used [1], a technique widely applied in linear algebra. It views higher-degree monomials as new variables, establishes their relationship with existing variables, and then exploits linear loop invariant generation techniques. However, the number of monomials grows exponentially when the degree increases. Kapur et al. [28]
introduce solvable mappings, which are a generalization of affine mappings, to avoid non-polynomial effects generated by polynomial programs. Recently, Chen et al. [6] applied multivariate Lagrange interpolation to synthesize polynomial loop invariants directly.

Another important problem for probabilistic programs is the almost-sure termination problem, answering whether the program terminates almost surely. In [13], Fioriti and Hermanns argued that Lyapunov ranking functions, used in deterministic program termination analysis, cannot be extended to probabilistic programs. Then they extended the ranking supermartingale approach [2] to the bounded non-deterministic case, and provided a compositional and sound proof method for the almost-sure termination problem. In [19], Kaminski and Katoen investigated the computational hardness of computing expected outcomes (including lower bounds, upper bounds and exact expected outcomes) and deciding almost-sure termination (including non-universal and universal situation) of probabilistic programs. In [5], Chatterjee et al. have further investigated termination problems for affine probabilistic programs. Recently, they also presented a method [4] to synthesize efficient ranking supermartingales through Putinar’s Positivstellensatz [27] and used it to prove the termination of probabilistic programs. Their method is sound and semi-complete over a large class of programs.

In this paper, we develop a technique exploiting semidefinite programming through another Positivstellensatz to synthesize the quantitative loop invariants. Positivstellensatzes are essential theorems in real algebra to describe the structure of polynomials that are positive (or non-negative) on a semialgebraic set. While our approach shares some similarities with the one in [4], the difference to the termination problem requires a variation of the theorem. In detail, Putinar’s Positivstellensatz deals with the situation when the polynomial is strictly positive on a quadratic module, which is not enough for quantitative loop invariants. In the program correctness problem, equality constraints are taken into consideration as well as inequalities. Therefore in our method, Stengle’s Positivstellensatz [29] dealing with general real semi-algebraic sets is being used.

As previous results [20,15,6], our approach is constraint-based [9]. We fix a polynomial template for the invariants with a fixed degree and generate constraints from the program. The constraints can be transformed into an emptiness problem of a semialgebraic set. By Stengle’s Positivstellensatz [29], it suffices to solve a semidefinite programming feasibility problem, for which efficient solvers exist. From a feasible solution we can then obtain the corresponding coefficients of the template. We verify the correctness of the template. If the solver does not provide a feasible solution or if the coefficients are not correct, we refine the analysis by adding constraints to block the undesired solutions or increasing the degree of the template, which will always lead us to a feasible solution if one exists.

The method is applied to several case studies taken from [6]. The technique usually solves the problem within 1 second, which is about one tenth of the time taken by the tool described in [6]. Our tool supports real variables rather than discrete ones, and supports polynomial probabilistic programs. We illus-
trate these features by analyzing a non-linear perceptron program and a model for airplane delay with continuous distributions. Moreover, we conduct a sequence of trials on parameterized probabilistic programs to show that the main influence factor on the running time of our method is the degree of the invariant template. We compare our results on these examples with the prototype LIP in [7], Prinsys [13] and Probabilistic Program Analyzer PPA [2].

2 Preliminaries

In this section we introduce some notations. We use $X_n$ to denote an $n$-tuple of variables $(X_1, \ldots, X_n)$. For a vector $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, $X_n^\alpha$ denotes the monomial $X_1^{\alpha_1} \cdots X_n^{\alpha_n}$, and $d = \sum_i \alpha_i$ is its degree.

**Definition 1.** A polynomial $f$ in variables $X_1, \ldots, X_n$ is a finite linear combination of monomials: $f = \sum_\alpha c_\alpha X_n^\alpha$ where finitely many $c_\alpha \in \mathbb{R}$ are non-zero.

The degree of a polynomial is the highest degree of its component monomials. Extending the notation, for a sequence of polynomials $F = (f_1, \ldots, f_s)$ and a vector $\alpha = (\alpha_1, \ldots, \alpha_s) \in \mathbb{R}^s$, we let $F_\alpha$ denote $\prod_{i=1}^s f_i^{\alpha_i}$. The polynomial ring with $n$ variables is denoted with $\mathbb{R}[X_n]$, and the set of polynomials of degree at most $d$ is denoted with $\mathbb{R}[X_n]_{\leq d}$. For $f \in \mathbb{R}[X_n]$ and $z_n = (z_1, \ldots, z_n) \in \mathbb{R}^n$, $f(z_n) \in \mathbb{R}$ is the value of $f$ at $z_n$.

A constraint is a quantifier-free formula constructed from (in)equalities of polynomials. It is linear if it contains only linear expressions. A semialgebraic set is a set described by a constraint:

**Definition 2.** A semialgebraic set in $\mathbb{R}^k$ is a finite union of sets of the form $\{x \in \mathbb{R}^k | P(x) = 0 \land \bigwedge_{Q \in Q} Q > 0\}$, where $Q$ is a finite set of polynomials.

A polynomial $p(X_n) \in \mathbb{R}[X_n]$ is a sum of squares (or SOS, for short), if there exist polynomials $f_1(x), \ldots, f_m(x) \in \mathbb{R}[X_n]$ such that $p(X_n) = \sum_{i=1}^m f_i^2(X_n)$.

Appendix A introduces a way to transform SOS problems into a semidefinite programming problem (or SDP, for short), which is a generalization of linear programming problem. We introduce SDP problems briefly in Appendix B.

2.1 Probabilistic Programs

We use a simple probabilistic guarded-command language to construct probabilistic programs with the grammar:

$$P ::= \text{skip} \mid \text{abort} \mid x := E \mid P \mid P[p]P \mid \text{if} \,(G) \{P\} \mid \text{else} \{P\} \mid \text{while} \,(G)\{P\}$$

where $G$ is a Boolean expression and $E$ is a real-valued expression defined by the grammar:

$$E ::= c \mid x_n \mid r \mid \text{constant} \mid \text{variable} \mid \text{random variable}$$

$$E + E \mid E \cdot E \mid \text{arithmetic}$$

$$E < E \mid E \land E \mid \neg E \mid \text{guards}$$
Random variable \( r \) follows a given probability distribution, discrete or continuous. For \( p \in [0, 1] \), the probabilistic choice command \( P_0 [p] P_1 \) executes \( P_0 \) with probability \( p \) and \( P_1 \) with probability \( 1 - p \).

**Example 3.** The following probabilistic program \( P \) describes a simple game:

\[
z := 0; \text{while}(0 < x < y) \ {(x := x + 1; 0.5 \cdot x := x - 1; \ z := z + 1) }.
\]

The program models a game where a player has \( x \) dollars at the beginning and keeps tossing a coin with probability 0.5. The player wins one dollar if he tosses a head and loses one dollar for a tail. The game ends when the player loses all his money, or he wins \( y - x \) dollars for a predetermined \( y \). The variable \( z \) records the number of tosses made by the player during this game.

We assume that the reader is familiar with the basic concepts of probability theory and in particular expectations, see e.g. [12] for details. An expectation is called a post-expectation when it is to be evaluated on the final distribution, and it is called a pre-expectation when it is to be evaluated on the initial distribution. Let \( \text{pre}_E, \text{post}_E \) be expectations and \( \text{prog} \) a probabilistic program. We say that the sentence \( \langle \text{pre}_E \rangle \text{prog} \langle \text{post}_E \rangle \) holds if the expected value of \( \text{post}_E \) after executing \( \text{prog} \) is equal to or greater than the expected value of \( \text{pre}_E \). Note that when \( \text{post}_E \) and \( \text{pre}_E \) are functions, the comparison is executed pointwise.

Classical programs can be viewed as special probabilistic programs in the following sense. For classical precondition \( \text{pre} \) and postcondition \( \text{post} \), let the characteristic function \( \chi_{\text{pre}} \) equal 1 if the precondition is true and 0 otherwise, and define \( \chi_{\text{post}} \) similarly. If one considers a Hoare triple \( \{ \text{pre} \} \text{prog} \{ \text{post} \} \) where \( \text{prog} \) is a classical program, then it holds if and only if \( \langle \chi_{\text{pre}} \rangle \text{prog} \langle \chi_{\text{post}} \rangle \) holds in the probabilistic sense.

### 2.2 Probabilistic Predicate Transformers

Let \( P_0, P_1 \) be probabilistic programs, \( E \) a expression, \( \text{post} \) a postcondition, \( \text{pre} \) a precondition, \( G \) a Boolean expression, and \( p \in (0, 1) \). The probabilistic predicate transformer \( \text{wp} \) can be defined as follows [10]:

\[
\text{wp}(\text{skip}, \text{post}) = \text{post}
\]
\[
\text{wp}(\text{abort}, \text{post}) = 0
\]
\[
\text{wp}(x := E, \text{post}) = \text{post}[x/E_S(E)]
\]
\[
\text{wp}(h_0; h_1, \text{post}) = \text{wp}(h_0, \text{wp}(h_1, \text{post}))
\]
\[
\text{wp}(\text{if}(G) \text{then}(h_0) \text{else}(h_1), \text{post}) = \chi_G \cdot \text{wp}(h_0, \text{post}) + (1 - \chi_G) \cdot \text{wp}(h_1, \text{post})
\]
\[
\text{wp}(h_0 [p] h_1, \text{post}) = p \cdot \text{wp}(h_0, \text{post}) + (1 - p) \cdot \text{wp}(h_1, \text{post})
\]
\[
\text{wp}(\text{while}(G) \{ h_0 \}, \text{post}) = \mu X. (\chi_G \cdot \text{wp}(h_0, X) + (1 - \chi_G) \cdot \text{post})
\]

Here \( \text{post}[x/E_S(E)] \) denotes the formula obtained by replacing free occurrences of \( x \) in \( \text{post} \) by the expectation of expression \( E \) over the state space \( S \). The least fixed point operator \( \mu \) is defined over the domain of expectations [24][22], and it can be shown that \( \langle \text{pre} \rangle P \langle \text{post} \rangle \) holds if and only if \( \text{pre} \leq \text{wp}(P, \text{post}) \). Thus, \( \text{wp}(P, \text{post}) \) is the greatest lower bound of precondition expectation of \( P \) with respect to \( \text{post} \), and we say \( \text{wp}(P, \text{post}) \) is the weakest pre-expectation of \( P \) w. r. t. \( g \).
2.3 Positivstellensatz

Hilbert’s Nullstellensatz is very important in algebra, and its real version, known as Positivstellensatz, is crucial to our method. First, some concepts are needed to introduce the theorem.

- The set \( I \subseteq \mathbb{R}[X_n] \) is an ideal if it satisfies: (i) \( 0 \in I \), (ii) \( I \) is closed under addition, and (iii) If \( a \in I \) and \( b \in \mathbb{R}[X_n] \), then \( a \cdot b \in I \).
- The set \( P \subseteq \mathbb{R}[X_n] \) is a positive cone if it satisfies: (i) If \( a \in \mathbb{R}[X_n] \), then \( a^2 \in P \), and (ii) \( P \) is closed under addition and multiplication.
- The set \( M \subseteq \mathbb{R}[X_n] \) is a multiplicative monoid with 0 if it satisfies: (i) \( 0, 1 \in M \), and (ii) \( M \) is closed under multiplication.

We are interested in finitely generated ideals, positive cones and multiplicative monoids with 0. Let \( \mathcal{F} = \{f_1, \ldots, f_s\} \) be a finite set of polynomials. We recall that

- Any element in the ideal generated by \( \mathcal{F} \) is of the form \( k_1f_1 + k_2f_2 + \cdots + k_sf_s \), where \( k_1, \ldots, k_s \in \mathbb{R}[X_n] \).
- Any element in the positive cone generated by \( \mathcal{F} \) is of the form \( \sum_{\alpha \in \{0,1\}^s} k_\alpha \mathcal{F}^\alpha \) where \( k_\alpha \) is a sum of squares for all \( \alpha \in \{0,1\}^s \).

In the sum, \( \alpha \) denotes an \( s \)-length vector with each element 0 or 1.

- Any nonzero element in the multiplicative monoid with 0 generated by \( \mathcal{F} \) is of the form \( \mathcal{F}^\alpha \), where \( \alpha = (\alpha_1, \ldots, \alpha_s) \in \mathbb{N}^s \).

The Positivstellensatz due to Stengle states that for a system of real polynomial equalities and inequalities, either there exists a solution, or there exists a certain polynomial which guarantees that no solution exists.

**Theorem 4 (Stengle’s Positivstellensatz [29])**. Let \( (f_j)_{j=1}^s, (g_k)_{k=1}^t, (h_l)_{l=1}^w \) be finite families of polynomials in \( \mathbb{R}[X_n] \). Denote by \( P \) the positive cone generated by \( (f_j)_{j=1}^s \), by \( M \) the multiplicative monoid with 0 generated by \( (g_k)_{k=1}^t \), and by \( I \) the ideal generated by \( (h_l)_{l=1}^w \). Then the following are equivalent:

1. The set
   \[
   \left\{ \mathbf{z}_n \in \mathbb{R}^n \mid \begin{array}{l}
   f_j(\mathbf{z}_n) \geq 0, \quad j = 1, \ldots, s \\
   g_k(\mathbf{z}_n) \neq 0, \quad k = 1, \ldots, t \\
   h_l(\mathbf{z}_n) = 0, \quad l = 1, \ldots, w
   \end{array} \right\}
   \]  
\hspace{1cm} (1)

   is empty.

2. There exist \( f \in P, g \in M, h \in I \) such that \( f + g^2 + h = 0 \).
3 Problem formulation

The question that concerns us here is to verify whether the loop sentence
\[
\langle \text{preE} \rangle \text{while}(G) \{ \text{body} \} \langle \text{postE} \rangle
\]
holds or not, when given the pre-expectation \( \text{preE} \), post-expectation \( \text{postE} \), a Boolean expression \( G \), and a loop-free probabilistic program \( \text{body} \). One way to solve this problem is to calculate the weakest pre-expectation \( \text{wp}(\text{while}(G, \{ \text{body} \}), \text{postE}) \) and to check whether it is not smaller than \( \text{preE} \). However, the weakest pre-expectation of while sentence requires a fixed-point computation, which is not trivial. To avoid the fixed point, the problem can be solved through a quantitative loop invariant.

**Theorem 5 ([15]).** Let \( \text{preE} \) be a pre-expectation, \( \text{postE} \) a post-expectation, \( G \) a Boolean expression, and \( \text{body} \) a loop-free probabilistic program. To show
\[
\langle \text{preE} \rangle \text{while}(G) \{ \text{body} \} \langle \text{postE} \rangle,
\]
it suffices to find a loop invariant \( I \) which is an expectation such that

1. (boundary) \( \text{preE} \leq I \) and \( I \cdot (1 - \chi_G) \leq \text{postE} \);
2. (invariant) \( I \cdot \chi_G \leq \text{wp}(\text{body}, I) \);
3. (soundness) the loop terminates from any state in \( G \) with probability 1, and
   (a) the number of iterations is finite, or
   (b) \( I \) is bounded from above by some fixed constant, or
   (c) the expected value of \( I \cdot \chi_G \) tends to zero as the number of iterations tends to infinity.

Since soundness of a loop invariant is not related to pre- and postconditions, and can be verified from its type before any specific invariants are found, so we only focus on the boundary and invariant conditions in Theorem 5. The soundness property is left to be verified manually in case studies.

For pre-expectation \( \text{preE} \) and post-expectation \( \text{postE} \), the boundary and invariant conditions in Theorem 5 provide the following requirements for a loop invariant \( I \):
\[
\begin{align*}
\text{preE} & \leq I \\
I \cdot (1 - \chi_G) & \leq \text{postE} \\
I \cdot \chi_G & \leq \text{wp}(\text{body}, I).
\end{align*}
\]

The inequalities induced by the boundary and invariant conditions contain indicator functions, which we find difficult to analyze if they appear on both sides. So first we rewrite the expectations to a standard form. For a Boolean expression \( P \), we use \( [P] \) to represent its integer value, i.e. \( [P] = 1 \) if \( P \) is true, and \( [P] = 0 \) otherwise. An expectation is in disjoint normal form (DNF) if it is of the form
\[
f = [P_1] \cdot f_1 + \cdots + [P_k] \cdot f_k
\]
where the \( P_i \) are disjoint expressions, which means any two of the expressions cannot be true simultaneously, and the \( f_i \) are polynomials.
Lemma 6 ([20]). Suppose \( f = [P_1] \cdot f_1 + \cdots + [P_k] \cdot f_k \) and \( g = [Q_1] \cdot g_1 + \cdots + [Q_l] \cdot g_l \) are expectations over \( X_n \) in DNF. Then, \( f \leq g \) if and only if (pointwise)

\[
\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{l} P_i \land Q_j \Rightarrow f_i \leq g_j \\
\land \bigwedge_{i=1}^{k} P_i \land \left( \bigwedge_{j=1}^{l} \neg Q_j \right) \Rightarrow f_i \leq 0 \\
\land \bigwedge_{j=1}^{l} \left( \bigwedge_{i=1}^{k} \neg P_i \right) \land Q_j \Rightarrow 0 \leq g_j. \tag{3}
\]

Example 7. Consider the following loop sentence for our running example:

\[
\langle xy - x^2 \rangle z := 0; \text{ while}(0 < x < y) \{ x := x + 1 \; \text{[0.5]} \; x := x - 1; z := z + 1; \} \; \langle z \rangle
\]

For this case, the following must hold for any loop invariant \( I \).

\[
xy - x^2 \leq I \\
I \cdot [x \leq 0 \lor y \leq x] \leq z \\
I \cdot [0 < x < y] \leq 0.5 \cdot I(x + 1, y, z + 1) + 0.5 \cdot I(x - 1, y, z + 1)
\]

By Lemma 6, these requirements can be written as

\[
xy - x^2 \leq I \land \\
x \leq 0 \lor y \leq x \Rightarrow I \leq z \land \\
0 < x < y \Rightarrow 0 \leq z \land \\
0 < x < y \Rightarrow I \leq 0.5 \cdot I(x + 1, y, z + 1) + 0.5 \cdot I(x - 1, y, z + 1) \land \\
x \leq 0 \lor y \leq x \Rightarrow 0 \leq 0.5 \cdot I(x + 1, y, z + 1) + 0.5 \cdot I(x - 1, y, z + 1) \land \\
0 < x < y \\
x \leq 0 \lor y \leq x \Rightarrow 0 \leq 0.5 \cdot I(x + 1, y, z + 1) + 0.5 \cdot I(x - 1, y, z + 1) \land \\
0 < x < y \\
x \leq 0 \lor y \leq x \Rightarrow 0 \leq 0.5 \cdot I(x + 1, y, z + 1) + 0.5 \cdot I(x - 1, y, z + 1) \land \\
0 < x < y
\]

The program in this example originally served as a running example in [6]. There, after transforming the constraints into the form above, Lagrange interpolation is applied to synthesize the coefficients in the template. In our approach, we check the correctness of each conjunct in (4–8) by checking the nonnegativity of the polynomial on the right side over a semialgebraic set related to polynomials on the left side. In this way, we can use the Positivstellensatz to synthesize the coefficients.

4 Constraint Solving by Semidefinite Programming

Our aim is to synthesize coefficients for the fixed invariant template for simple (Subsection 4.1) and nested (Subsection 4.2) programs. Checking the validity of constraints can be transformed into checking the emptiness of a semialgebraic set. Then, we show that the emptiness problem can be turned into sum-of-squares constraints using Stengle’s Positivstellensatz.
Our Approach in a Nutshell. For a given polynomial template as a candidate quantitative loop invariant, it needs to satisfy boundary and invariant conditions. Our goal is to synthesize the coefficients in the template. These conditions describe a semialgebraic set, and the satisfiability of the constraints is equivalent to the non-emptyness of the corresponding semialgebraic set. Applying the Positivstellensatz (see Section 2.3), we will transform the problem to an equivalent semidefinite programming problem (See B) using Lemma 8. Existing efficient solvers can be used to solve the problem. A more efficient yet sufficient way is to transform the problem into a sum-of-squares problem (See Appendix A) using Lemma 9, and then solving by semidefinite programming. After having synthesized the coefficients of the template, we verify whether they are valid. In case of a negative answer, which may happen due to floating-point errors, some refinements can be made by adding further constraints, which is described in Section 4.3. If the problem is still unsolved, we might have to raise the maximum degree of the template and reiterate the procedure.

4.1 Synthesis algorithm for simple loop programs

Now we are ready for the transformation method. Each conjunct obtained in Lemma 6 is of the form $F \Rightarrow G$, where $F$ is a quantifier-free formula constructed from (in)equalities between polynomials in $\mathbb{R}^d[X_n]$, and $G$ is of the form $f \leq g$, $f \leq 0$ or $0 \leq g$, with $f, g \in \mathbb{R}^d[X_n]$. If $F$ contains negations, we use De Morgan’s laws to eliminate them. If there is a disjunction in $F$, we split the constraints into sub-constraints as $\varphi \lor \psi \Rightarrow \chi$ is equivalent to $(\varphi \Rightarrow \chi) \land (\psi \Rightarrow \chi)$. After these simplifications, $F \Rightarrow G$ can be written in the form $\bigwedge_i (f_i \geq 0) \Rightarrow g \geq 0$ where $\geq \in \{\geq, =\}$. Observe that a constraint $\bigwedge_i (f_i \geq 0) \Rightarrow g \geq 0$ is satisfied if and only if the set $\{x \mid f_i(x) \geq 0 \text{ for all } i; -g(x) \geq 0; \text{ and } g(x) \neq 0\}$ is empty. In this way, we transform our constraint into the form required by Theorem 4.

Summarizing, Constraint (2) is satisfied if and only if all semialgebraic sets created using the procedure above are empty. Now we are ready to transform this constraint to an SDP problem.

Lemma 8 ([26,10]). The emptiness of (1) is equivalent to the feasibility of an SDP problem.

See Appendix C.1 for a constructive proof. Although the transformation in Lemma 8 is effective, it is complicated in practice. In the following lemma we present a simpler yet sufficient procedure.

Lemma 9. The following statements hold (with $\geq \in \{\geq, =\}$):

1. $f(X_n) \geq 0 \Rightarrow g(X_n) \geq 0$ holds if $g(X_n) - u \cdot f(X_n)$ is a sum of squares for some $u \in \mathbb{R}_{\geq 0}$.
2. $f(X_n) = 0 \Rightarrow g(X_n) \geq 0$ holds if $g(X_n) - v \cdot f(X_n)$ is a sum of squares for some $v \in \mathbb{R}$.
3. $f_1(X_n) \geq_1 0 \land f_2(X_n) \geq_2 0 \Rightarrow g(X_n) \geq_0 0$ holds if $g(X_n) - r_1 \cdot f_1(X_n) - r_2 \cdot f_2(X_n)$ is a sum of squares for some $r_1, r_2 \in \mathbb{R}$; if $\geq_1$ is $\geq$, it is additionally required that $r_1 \geq 0$. 
4. \( f_1(X_n) \geq 0 \lor f_2(X_n) \geq 0 \Rightarrow g(X_n) \geq 0 \) holds if \( g(X_n) - r_1 \cdot f_1(X_n) \) is a sum of squares and \( g(X_n) - r_2 \cdot f_2(X_n) \) is a sum of squares for some \( r_1, r_2 \in \mathbb{R} \); if \( \geq_1 \) is \( \geq_1 \), it is additionally required that \( r_1 \geq 0 \).

We omit the proof because the lemma is obvious.

**Example 10.** Applying the above procedure, Constraint (5) in Example 7 is split into \((x \leq 0 \Rightarrow I \leq z) \land (y \leq x \Rightarrow I \leq z)\) and then normalized to \((-x \geq 0 \Rightarrow z - I \geq 0) \land (x - y \geq 0 \Rightarrow z - I \geq 0)\). This holds if \( z - I + u_1 x \) is a SOS for some \( u_1 \in \mathbb{R}_{\geq 0} \) and \( z - I + u_2 (y - x) \) is a SOS for some \( u_2 \in \mathbb{R}_{\geq 0} \). The other constraints can be handled in a similar way.

Algorithm 1 Loop Invariant Generation with Refinement

**Input:** sentence \(:=\langle preE \rangle \textbf{while}\langle G \rangle\{body\}\langle postE \rangle\) with program variables \(X_n\)

**Output:** a loop invariant satisfying the boundary and invariant conditions

1: loop
2: \( d := 2 \)
3: Choose a template for \(I \in \mathbb{R}^{\leq d}[X_n]\)
4: Let \(f\) be Constraint (2), i.e. the boundary and invariant conditions from Theorem 5 for sentence
5: while constraints be the SDP problem equivalent to \(f\) according to Lemma 8
6: if \(I\) satisfies the boundary and invariant conditions then
7: Output \(I\) and terminate
8: Refine constraints
9: end if
10: end while
11: \( d := d + 2 \)
12: end loop

We summarize our approach in Algorithm 1. The aim is to synthesize the coefficients of template \(I\). Algorithm 1 is semi-complete in the sense that by raising degree \(d\), it will generate an invariant if there exists one. Its termination is guaranteed in principle by Theorem 4 and the equivalence between SOS and SDP, though due to floating-point errors, the algorithm may fail to find \(I\) in practice.

In practice, Lemma 9 is often used instead of Lemma 8 for efficiency. Step 5 in Algorithm 1 is replaced by: “Let \(constraints\) be the relaxation of \(f\) to an SOS problem according to Lemma 9; this can be translated to an equivalent SDP problem, which is simpler than the direct translation of Lemma 8 using the technique of Subsection A as explained in Appendix B.”
Example 11. We extend Example 7 and Lemma 9. Since one invariant is enough for our problem, we choose Constraints (4), (5), and (7). The initial condition \( z = 0 \) is not included in those constraints, so (4) needs to be refined as \( z = 0 \Rightarrow xy - x^2 \leq I \).

First, we set a template for \( I \). Assume \( I \) as a quadratic polynomial with three variables \( x, y, z \):

\[
I = c_0 + c_1 x + c_2 y + c_3 z + c_{11} x^2 + c_{12} xy + c_{13} xz + c_{22} y^2 + c_{23} yz + c_{33} z^2
\]

where \( c_0, \ldots, c_{33} \in \mathbb{R} \) are coefficients that remains to be determined.

For Constraint (4), we add initial constraint \( z = 0 \), and get the following corresponding constraint:

\[
I - (xy - x^2) + v \cdot z \geq 0 \quad (4')
\]

For (5), the antecedens is a conjunction of two constraints. As in Example 5, (5) is split into two constraints. By adding cross-product term with degree \( d \leq 2 \), (5) can be transformed into

\[
z - I + u_1 \cdot x \geq 0 \quad \text{and} \quad z - I - u_2 \cdot (x - y) \geq 0 \quad (5')
\]

For (7), the constraint \( 0 < x < y \) need to be split into two inequalities \( 0 < x \wedge 0 < y - x \). Then similarly as (5), (7) can be transformed into

\[
0.5 \cdot I(x+1, y, z+1) + 0.5 \cdot I(x-1, y, z+1) - I - u_3 \cdot x - u_4 \cdot (y-x) - u_5 \cdot x(y-x) \geq 0 \quad (7')
\]

In this means the example can be transformed into an SDP problem with constraints (4'), (5'), and (7'), and positive constraints on multipliers \( u_1 \geq 0, \ldots, u_5 \geq 0 \), arbitrary value for value \( v \). Then the resulting SDP problem can be submitted to any SOS solver.

The result using solver SeDuMi [30] is shown below. Here we ignore the amounts smaller than the order of magnitude of \( 10^{-6} \). We get a result that the invariant

\[
I = -7.1097 \cdot 10^{-10} - 3.8818 \cdot 10^{-10}x - 0.4939 \cdot 10^{-10}y + z - x^2 + xy
+ 2.7965 \cdot 10^{-10}xz + 0.97208 \cdot 10^{-10}y^2 + 4.4656 \cdot 10^{-10}yz - 0.28694 \cdot 10^{-10}z^2
\]

\[
\approx z - x^2 + xy,
\]

which can be proven correct by verifying the corresponding constraints.

4.2 Synthesis algorithm for nested loop programs

We are now turning to programs containing nested loops. To simplify our discussion, we assume the program only contains a single inner loop, i.e. it can be written as

\[
P = \text{while}(G)\{\text{body}\}
= \text{while}(G)\{\text{body1; while} (G_{\text{inn}})\{\text{body}_{\text{inn}}\}; \text{ body2; }\}
\]
where \textit{body1} and \textit{body2} are loop-free program fragments. For a given \textit{preE} and \textit{postE}, we need to verify if there exists an invariant \textit{I} that satisfies Constraint (2) for program \textit{P}. We denote the inner loop by \textit{P}_{inn} = \text{while}(G_{inn})\{\textit{body}_{inn}\}.

For such a program, the main difficulty is how to deal with \textit{wp(body, I)} in Constraint (2). We propose a method here that takes the inner and outer iteration into consideration together and uses the verified pre-expectation of the inner loop to relax the constraint.

Assume templates for the polynomial invariants: \textit{I} for the outer loop and \textit{I}_{inn} for the inner loop \textit{P}_{inn}. Since \textit{body2} is loop-free, it is easy for us to obtain \( \tilde{I} = \text{wp(body2, I)} \). We use \( \tilde{I} \) as post-expectation \textit{postE}_{inn} for the inner loop, and guess a pre-expectation for it. In practice, we calculate the pre-expectation of the inner loop for each variable of the program separately.

\textbf{iii} DYJ: How do we calculate it?
\textbf{iii} FYJ: I guess a pre-expectation for each variable and use the algorithm to prove it is a nice guess and substitute it into \( \tilde{I} \) to obtain the pre-expectation \textit{preE}_{inn}. We then try to synthesize the coefficients for \textit{I}_{inn} so that it satisfies Constraint (2) with \textit{preE}_{inn} and \textit{postE}_{inn}. If such an invariant is found, we use \textit{preE}_{inn} for \textit{wp(P}_{inn}, \tilde{I}).

\textbf{iii} FYJ: No, we need to find \textit{I}_{inn} to prove that we can use \textit{preE}_{inn} to relax \textit{wp(P}_{inn}, \tilde{I}), \textit{wp} for the inner loop is the difficult part here.

Then the constraint for loop invariant \textit{I} can be summarized as

\begin{equation}
\begin{align*}
\text{preE} & \leq \text{I} \\
\text{I} \cdot [1 - \chi G] & \leq \text{postE} \\
\text{I} \cdot \chi G & \leq \text{wp(body, I)}. \\
\text{preE}_{inn} & \leq \text{I}_{inn} \\
\text{I}_{inn} \cdot [1 - \chi G_{inn}] & \leq \text{postE}_{inn} \\
\text{I}_{inn} \cdot \chi G_{inn} & \leq \text{wp(body}_{inn}, \text{I}_{inn}).
\end{align*}
\end{equation}

\textbf{iii} DYJ: I turned the last clause around: we have \( \tilde{I} \) and need to find \textit{preE}_{inn}, I think.

\textbf{iii} FYJ: No, it is not yet ok: Still there are two separate equation systems that just happen to be written together. From (9), the relation between \( \text{I} \) and \( \text{I}_{inn} \) cannot be seen.

\textbf{iii} FYJ: The relation is that \textit{postE}_{inn} = \text{wp(body2, I)}, and \textit{wp(body, I)} uses \textit{preE}_{inn} in is calculation for \textit{wp(P}_{inn}, \textit{body2, I}) part where \textit{wp} is the weakest precondition using \textit{preE}_{inn} in wp-calculation instead of \textit{wp}(\tilde{I}, \textit{P}_{inn}).

Then we have the following lemma.

\textbf{Lemma 12.} An invariant that satisfies Constraint (9) also satisfies (2), therefore it is a loop invariant for program \textit{P}.

See Appendix C.2 for the proof.
4.3 Constraint Refinement

In practice, wrong or trivial coefficients may be generated using our method because of floating-point errors. We discuss how to refine the constraints using several methods.

The first situation is often faced when dealing with equalities or inequalities. Due to the inaccuracy of floating-point calculations, it is hard for a software to check equations like $x = 0$. A common trick to avoid this problem is to turn the constraint into $x \geq 0 \land x \leq 0$. As for inequalities, taking $x \neq 0$ as an example, a way to solve the problem is adding a new variable $y$ to transform the constraint into $xy \geq 1$, since $xy \geq 1$ implies $x \neq 0$ for any value of $y$. The new constraints are in the form required by Theorem 4.

Numerical errors may also lead to an unsound invariant: we may get some coefficients with a small magnitude, which often result from floating-point inaccuracies. A common solution for this problem is to ignore those small numbers, usually smaller than $10^{-6}$ in practice. In Example 11, we eliminated the terms with a small order of magnitude, but the resulting invariant can still be incorrect if the remaining coefficients are approximate. Symbolic computation can be used to guarantee soundness. Checking whether the generated invariant satisfies Constraint (2) is a special case of quantifier elimination $\forall x_n \in \mathbb{R}^n, f(x_n) \geq 0$. Such problem can be solved efficiently using an improved Cylindrical Algebraic Decomposition (CAD) algorithm implemented in [17]. If the invariant violates some of the constraints, we can try to strengthen the constraint (e.g., change $x \geq 0$ to $x \geq 0.1$) and repeat our method.

5 Experimental Results

We have implemented a prototype in Python to test our technique. We call the MATLAB toolbox YALMIP [21] with the SeDuMi solver [30] to solve the SDP feasibility problem. The experiments were done on a computer with Intel(R) Core(TM) i7-4710HQ CPU and 16 GiB of RAM. The operating system is Window 7 (32bit). Our prototype and the detailed experimental results can be found at [http://iscasmc.ios.ac.cn/ppsdp](http://iscasmc.ios.ac.cn/ppsdp). For each probabilistic program, we give the description of the while loop with pre- and post-expectations in Table 5 and Appendix D. The annotated pre-expectation serves as an exact estimate of the annotated post-expectation at the entrance of the loop. We apply the method to several different types of examples. A summary of the results is shown in Table 1. The first seven probabilistic programs are taken from paper [6], thus we skip the detailed descriptions of them. We have further constructed three case studies to illustrate continuous distributions, polynomial probabilistic programs and nested loop programs. The details of these examples are included in Appendix D.

As we can see from the table, the running time of our method is within one second. We also provide running time for the tool Probabilistic Program Analyzer in [2] and compare with other prototypes in Section 5.1.
Table 1. The column “Name” shows the name of each experiments. The annotated pre- and post-expectations are shown in the columns “preE” and “postE”. The inferred quantitative loop invariant for each example is given in the column “Invariant”. The column “Time” gives the running time needed of our tool: the first one is the total running time, and the second one is the time used in SeDuMi solver.

| Name       | preE       | postE     | Invariant                                                                 | Time (s) |
|------------|------------|-----------|---------------------------------------------------------------------------|----------|
| ruin       | $xy - x^2$ | $z$       | $z + xy - x^2$                                                            | 0.4/0.3  |
| bin1       | $x + \frac{1}{3} ny$ | $x$       | $x + \frac{1}{3} ny$                                                    | 0.4/0.2  |
| bin2       | $\frac{1}{n^2} - \frac{1}{n} + \frac{3}{n} ny$ | $x$       | $x + \frac{1}{n} n^2 - \frac{1}{n} n + \frac{3}{n} ny$                 | 0.7/0.3  |
| bin3       | $\frac{1}{n^2} - \frac{1}{n} + \frac{3}{n} ny^2$ | $x$       | $x - 0.0057n - 0.0014x^2 + 0.1763xn + 712.909n^2 + 0.0014x^2 n + 0.4114xn^2 + 0.4188ny^2 - 0.0178n^3$ | 0.7/0.3  |
| geo        | $x + 3zy$  | $x$       | $x + 3zy$                                                                | 0.3/0.1  |
| sum        | $\frac{1}{3} n^2 + \frac{1}{n} n$ | $x$       | $x + \frac{1}{3} n^2 + \frac{1}{n} n$                                  | 0.3/0.1  |
| prod       | $\frac{1}{n^2} - \frac{1}{n} n$ | $xy$      | $-\frac{2}{n} n + xy + \frac{1}{2} xn + \frac{1}{2} yn + \frac{1}{2} n^2$ | 0.3/0.1  |
| simple     | $-2b$      | $n$       | $n - 2b$                                                                 | 0.3/0.1  |
| perceptron |            |           |                                                                            | (-)      |
| airplane   | 106.548x   | $h$       | $106.548x - 106.548n + h$                                                | 0.4/0.2  |
| delay      |            |           |                                                                            | 22.8/8.1 |
| airplane   | 282.507x   | $h$       | $282.507(x - n) + h$                                                     | 0.5/0.2  |
| delay2     |            |           |                                                                            | 15.5/9.0 |
| fair coin  | $\frac{1}{4} - \frac{1}{4} x - \frac{1}{y} + \frac{1}{n}$ | $n$ | -                                                                        | (-)      |
| nested     | $20(m - x)$| $k$       | $k + 20(m - x)$                                                          | 0.5/0.3  |
| loop       |            |           |                                                                            | (-)      |

5.1 Evaluation

Other approaches to compute loop invariants in probabilistic programs are the Lagrange Interpolation Prototype (LIP) in [6], the Probabilistic Program Analyzer (PPA) in [2] and PRINSYS in [15]. The tools are executed on the same computer, LIP and PPA under Linux and the other two under Windows. In Table 2, we compare the features supported by the four tools. A more detailed description is given in Appendix E.

We have tested the examples in Table 1 on these four tools. PRINSYS takes the longest time and fails to verify any of non-linear examples presented. LIP fails to verify any examples that include a continuous variable or have a degree larger than 3; additionally it is always about 10 times slower than our tool. (PRINSYS and LIP are able to handle the “fair coin” example.) We now compare with PPA in more detail.

PPA fails to verify examples ruin, bin3 and geo. We observe that it cannot treats constraints of the form $x = y$ or $x \neq y$ (where $x$ and $y$ might be a variable or a constant), or inequalities including logical connectives such as $\land$ or $\lor$, which
Table 2. Comparison of the features supported by 4 tools

|                      | Prinsys | LIP        | PPA       | Our tool   |
|----------------------|---------|------------|-----------|------------|
| Type of Program      | Linear  | Polynomial ($d \leq 3$) | Linear    | Polynomial |
| Type of Invariant    | Linear  | Polynomial | Polynomial | Polynomial |
| Computation Method   | Symbolic | Symbolic  | Numerical | Numerical  |
| Distribution of Variable | Discrete | Discrete   | Continuous | Continuous |

appear in these examples. Also, it cannot verify the simple perceptron, as it is a non-linear program. Furthermore, PPA cannot deal with nested loop programs.

We now consider the parametric linear program in Section D.3. Table 3 gives a comparison of time consumption of the main technical step in our prototype and PPA. The number of constraints grows with the number of variables in our approach, similarly with the running time. On the other side, PPA is less influenced by the number of variables and thus is significantly faster in instances with more variables.

Table 3. Comparison of running time (in seconds) of the parameterized linear example

| Number of variables | n = 15 | n = 20 | n = 25 | n = 30 | n = 35 | n = 40 |
|---------------------|--------|--------|--------|--------|--------|--------|
| Solver time of our tool | 0.41   | 1.30   | 2.44   | 8.30   | 20.96  | 46.62  |
| VolComp time for PPA  | 0.87   | 0.96   | 0.87   | 0.97   | 0.85   | 0.96   |

6 Conclusion

In this paper, we propose a method to synthesize polynomial quantitative invariants for recursive probabilistic programs by semialgebraic programming via a Positivstellensatz. First, a polynomial template is fixed whose coefficients remains to be determined. The loop and its pre- and post-expectation can be transformed into a semialgebraic set, of which the emptiness can be decided by finding a counterexample generated by the Positivstellensatz. Semidefinite programming provides an efficient way to synthesize such a counterexample. The method can be applied on polynomial programs containing continuous or discrete variables and including nested loops. When floating point errors prevent finding a loop invariant polynomial right away, we currently can correct them ad hoc (by deleting terms with very small coefficients and sometimes strengthening the constraints), but we would like to develop a more systematic treatment.

References

1. Barthe, G., Espitau, T., Ferrer Fioriti, L. M., Hsu, J.: Synthesizing probabilistic invariants via Doob’s decomposition. arXiv preprint 1605.02765 (2016)
2. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) Computer aided verification: CAV. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013)

3. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Müller-Olm, M., Seidl, H. (eds.) Static analysis: SAS. LNCS, vol. 8723, pp. 85–100. Springer, Cham (2014)

4. Chatterjee, K., Fu, H., Goharshady, A. K.: Termination analysis of probabilistic programs through Positivstellensatz’s. In: Chaudhuri, S., Farzan, A. (eds.) Computer aided verification: CAV. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016)

5. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: Bodik, R., Majumdar, R. (eds.) POPL’16. pp. 327–342. ACM, New York (2016)

6. Chen, Y.-F., Hong, C.-D., Wang, B.-Y., Zhang, L.: Counterexample-guided polynomial loop invariant generation by Lagrange interpolation. In: Kroening, D., Păsăreanu, C. S. (eds.) Computer aided verification: CAV. LNCS, vol. 9206, pp. 658–674. Springer, Cham (2015)

7. Choi, M.-D., Lam, T. Y., Reznick, B.: Sums of squares of real polynomials. In: Jacob, B., Rosenberg, A. (eds.) K-Theory and Algebraic Geometry: Connections with Quadratic Forms and Division Algebras. Proceedings of Symposia in Pure Mathematics, vol. 58, Part 2, pp. 103–126. American Mathematical Society, Providence, R. I. (1995)

8. Civil Aviation Administration of China (CAAC): Statistical communiqué on the development of aviation in 2015 [in Chinese]. http://www.caac.gov.cn/XXGK/XXGK/TJSJ/201605/P020160531575434538041.pdf (2016)

9. Colón, M. A., Sankaranarayanan, S., Sipma, H. B.: Linear invariant generation using non-linear constraint solving. In: Hunt, Jr., W. A., Somenzi, F. (eds.) Computer aided verification: CAV. LNCS, vol. 2725, pp. 420–432. Springer, Berlin (2003)

10. Dai, L., Xia, B., Zhan, N.: Generating non-linear interpolants by semidefinite programming. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification: CAV. LNCS, vol. 8044, pp. 364–380. Springer, Berlin (2013)

11. Dijkstra, E.: A Discipline of Programming, vol. 4. Prentice-Hall, Englewood Cliffs (1976)

12. Feller, W.: An introduction to probability theory and its applications: volume I, vol. 1. Wiley (1968)

13. Ferrer Fioriti, L. M., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: POPL’15: principles of programming languages. pp. 489–501. ACM, New York (2015)

14. Gordon, A. D., Henzinger, T. A., Nori, A. V., Rajamani, S. K.: Probabilistic programming. In: Dwyer, M. B., Herbsleb, J. (eds.) Future of Software Engineering (FOSE 2014). pp. 167–181. ACM, New York (2014)

15. Gretz, F., Katoen, J.-P., McIver, A.: PRINSYS: on a quest for probabilistic loop invariants. In: Joshi, K., Siegle, M., Stoelinga, M., D’Argenio, P. R. (eds.) Quantitative Evaluation of Systems: QEST. LNCS, vol. 8054, pp. 193–208. Springer, Berlin (2013)

16. Gretz, F., Katoen, J.-P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perf. Eval. 73, 110–132 (2014)

17. Han, J., Jin, Z., Xia, B.: Proving inequalities and solving global optimization problems via simplified cad projection. J. Symb. Comput. 72, 206–230 (2016)

18. Hoare, C. A. R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
19. Kaminski, B. L., Katoen, J.-P.: On the hardness of almost-sure termination. In: Italiano, G. F., Pighizzini, G., Sannella, D. T. (eds.) Mathematical foundations of computer science 2015: MFCS. LNCS, vol. 9234, pp. 307–318. Springer, Heidelberg (2015)

20. Katoen, J.-P., McIver, A. K., Meinicke, L. A., Morgan, C. C.: Linear-invariant generation for probabilistic programs. In: Cousot, R., Martel, M. (eds.) Static analysis: SAS. LNCS, vol. 6337, pp. 390–406. Springer, Berlin (2010)

21. Löfberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB. In: 2004 IEEE International symposium on computer aided control systems design [CACSD]. pp. 284–289. IEEE, Piscataway, NJ (2004)

22. McIver, A., Morgan, C. C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, New York (2005)

23. Meyer, C. D.: Matrix Analysis and Applied Linear Algebra. SIAM, Philadelphia (2000)

24. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Progr. Lang. Syst. 18(3), 325–353 (1996)

25. Olmedo, F., Kaminski, B. L., Katoen, J.-P., Matheja, C.: Reasoning about recursive probabilistic programs. In: Grohe, M., et al. (eds.) Proceedings of the 31st Annual ACM–IEEE Symposium on Logic in Computer Science (LICS 2016). pp. 672–681. ACM, New York (2016)

26. Parrilo, P. A.: Semidefinite programming relaxations for semialgebraic problems. Math. Program., Ser. B 96(2), 293–320 (2003)

27. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana Univ. Math. J. 42(3), 969–984 (1993)

28. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symb. Comput. 42(4), 443–476 (2007)

29. Stengle, G.: A nullstellensatz and a positivstellensatz in semialgebraic geometry. Math. Ann. 207(2), 87–97 (1974)

30. Sturm, J. F.: Using ScDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optimization Methods and Software 11(1–4), 625–653 (1999)

31. Toh, K. C., Todd, M. J., Tütüncü, R. H.: SDPT3, a MATLAB software package for semidefinite programming, version 1.3. Optimization Methods and Software 11(1–4), 545–581 (1999)
The following appendices are added for the convenience of the reviewers. They will become part of a technical report accompanying our publication, once accepted. We appreciate the reviewers’ consideration.—The authors.

A Sum-of-Squares Problems

The set of sum-of-squares polynomials is a proper subset of the nonnegative polynomials with good algebraic properties, allowing efficient calculations. The “Gram matrix” method [7] is a way to decompose a polynomial into a sum of squares using semidefinite programming.

Consider a polynomial
\[ f = \sum_{|\alpha| \leq d} c_\alpha X^n_\alpha \in \mathbb{R}^d[X_n] \]
with degree at most \( d \).

\( f \) is a sum of squares if and only if it has a representation
\[ f = X_n A X^n_T = \sum_{\alpha, \beta \in \Delta} a_{\alpha \beta} X^n_\alpha X^n_\beta, \]
where the \(|\Delta| \times |\Delta|\) matrix \( A = (a_{\alpha \beta})_{\alpha, \beta \in \Delta} \) is positive semidefinite. So checking whether \( f \) is a sum of squares is equivalent to solving the following constraint problem:

\[
\begin{cases}
    c_0 = a_{00}, \\
    c_\gamma = \sum_{\alpha + \beta = \gamma} a_{\alpha \beta} \text{ for } \gamma \neq 0 \\
    \text{and } A = (a_{\alpha \beta})_{\alpha, \beta \in \Delta} \text{ is positive semidefinite}
\end{cases}
\]  

(10)

The above problem can be solved using a semidefinite program; we refer to Appendix B for details. Semidefinite programs can be efficiently solved both in theory and in practice and have seen active research in recent years. Some of the tools being used are SeDuMi [30] and SDPT3 [31].

B Semidefinite Programming

A semidefinite program can be seen as a generalization of a linear program where the constraints are described by a cone of positive semidefinite matrices.

We use \( S^n \) to denote all real \( n \times n \) symmetric matrices. Then for a matrix \( A \in S^n \), \( A \) is positive semidefinite if all eigenvalues of \( A \) are \( \geq 0 \) (one can find more about positive semidefinite matrix in any book about linear algebra such as [23]). For matrices \( A \) and \( B \) in \( S^n \), we write \( A \succeq B \) if and only if \( A - B \) is positive semidefinite.

We use \( \langle \cdot, \cdot \rangle \) to denote the scalar product of two matrices or vectors, i.e. for \( A = (a_{ij}), B = (b_{ij}) \in S^n \) and \( x, y \in \mathbb{R}^n \)

\[
\langle A, B \rangle := \text{Tr}(A^T B) = \sum_{i,j=1}^{n} a_{ij} b_{ij} \\
\langle x, y \rangle := x^T y
\]

A semidefinite program is defined as follows:

\[
\begin{align*}
\text{minimize} & \quad \langle C, X \rangle \\
\text{subject to} & \quad \langle A_i, X \rangle = b_i \text{ for } i = 1, \ldots, k \\
& \quad X \succeq 0
\end{align*}
\]  

(11)
where $X \in S^n$ is the decision variable, $b_i \in \mathbb{R}$ and $C, A_i \in S^n$ are given symmetric matrices.

Sum-of-squares problem (10) can be transformed into an SDP problem as follows. Let $X = (a_{\alpha \beta})_{\alpha, \beta \in \Delta}$, $A_i$ (for $i \in \Delta$) be the symmetric matrix whose $(\alpha, \beta)$ entry is 1 if $\alpha + \beta = i$ and 0 otherwise, and let $b_i = c_i$. In this way, (10) is transformed into the form (11) without objective.

C Proofs of Lemmas

C.1 Proof of Lemma 8

Lemma 8 ([26,10]). The emptiness of (1) is equivalent to the feasibility of an SDP problem.

Proof. We use the notations from Theorem 4 in this proof. The emptiness of (1) is equivalent to the existence of a solution for equation $f + g^2 + h = 0$ with $f \in P$, $g \in M$, $h \in I$ due to Theorem 4.

Note that $f$, $g$ and $h$ can be represented as

$$
- f = \sum_{\alpha \in \{0,1\}^n} u_\alpha F^\alpha, \quad \text{where } u_\alpha \text{ is a sum of squares for all } \alpha \in \{0,1\}^n \text{ and }
\quad F = \{f_1, \ldots, f_s\}, \quad \text{and}
- g = G^\alpha \text{ with } G = \{g_1, \ldots, g_t\}, \text{ and}
- h = v_1 h_1 + v_2 h_2 + \cdots + v_w h_w, \text{ where } v_1, \ldots, v_w \in \mathbb{R}[X_n].
$$

Fix a maximal degree $d$. Then we only consider $F^\alpha$ and $G^\alpha$ with $|\alpha| \leq d$. We set templates with degree $d$ for $u_\alpha$ and $v_0, \ldots, v_w$ and treat their coefficients in $\mathbb{R}$ as parameters. Every polynomial $v_i$ can be presented as the difference of two SOS polynomials: $v_i = \frac{(v_i+1)^2 - (v_i-1)^2}{4}$. Then $f + g^2 + h = \sum_{i=1}^l \delta_i p_i$, where $l$ is some integer, $p_i$ is one of $f_1, g_i, h_i$, or $-h_i$, and $\delta_i$ is a SOS. Then the equation $f + g^2 + h = 0$ can be transformed into a set of equations by merging coefficients of each $X_n^\alpha$ with maximal degree $2d$

One can formulate additional constraints that $\delta_\alpha$ needs to be a SOS and the coefficient matrix $A_\alpha$ with $\delta_\alpha = X_n A_\alpha X_n^T$ to be positive semidefinite. The equation set with constraints can be transformed into an SDP problem of the form in Appendix B.
C.2 Proof of Lemma 12

Proof. If the inner loop terminates, the existence of inner invariant shows the fact that

$$ \text{pre}_\text{inn} = \tilde{\text{wp}}((\text{while}(G_{\text{inn}})\{\text{body}_{\text{inn}}\}; \text{body}2), I) \leq \text{wp}(\text{body}_{\text{inn}}, \text{post}_\text{inn}) = \text{wp}(\text{body}_{\text{inn}}, \text{wp}(\text{body}2, I)) = \text{wp}((\text{while}(G_{\text{inn}}, I)\{\text{body}_{\text{inn}}\}; \text{body}2)) $$

The correctness of inequality due to the monotonicity of $\text{wp}$ [25]. Then for the whole program, we have

$$ I \cdot \chi_G \leq \tilde{\text{wp}}(\text{body}, I) \leq \text{wp}(\text{body}, I) $$

which satisfies [2].

D Experimental Details

D.1 Non-linear Probabilistic Programs

We use non-linear probabilistic program to model perceptron, which is an algorithm for supervised learning of binary classifiers in machine learning. It gives a linear classifier function to decide whether an input belong to one class or another based on a set of given data. Assume there is a data $x = (x_i, y_i)$, $y_i$ is the desired output value of $x_i$, the linear function $f(x)$ maps the data to a single binary value

$$ f(x_i) = \begin{cases} 1 & \text{if } w \cdot x_i + b > 0 \\ 0 & \text{otherwise} \end{cases} $$

When the outcome does not match $d_i$, the random perceptron updates its classifier by $w \leftarrow w + x_i y_i \ [\eta] w$ and $b \leftarrow b + y_i \ [\eta] b$ where $\eta$ is a learning rate.

When the input of the perceptron is one data $(x, y)$, the algorithm to generate a simple perceptron can be describes as:

```
real x, y;
real w, b;
int n := 0;
while(y (w \cdot x + b) \leq 0){
    w := w + y \cdot x, b := b + y \ [\eta] \text{skip}; n := n + 1
}
```

To simplify to calculation, we let $(x, y) = (1, 1), \eta = 0.25, w = 0$ and $b < 0$ in our trial, the expected times before the function can correctly classify the input is $E(n) = -2b$, as the method shows.
D.2 Probabilistic Program with Real Variables

The data in this example comes from [8], which is an official report by Civil Aviation Administration of China to report statistical data for all aspects of civil aviation in China in 2015. An airplane takes 2 hours and 15 minutes from Beijing Capital International Airport (PEK) to Shanghai Pudong International Airport (PVG). The average delay of an airliner is 21 minutes and can be approximated as a normal distribution, and the rate of normal flight is 68.3%. Assume an airliner takes this flights \( x \)-times in a month, the expected total time can be modeled as

\[
\begin{align*}
  h &:= 0; \\
n &:= 0; \\
\text{while}(n \leq x)\{ \\
  &\quad h := h + 135 + \text{NormDist}(21, \sigma) \times 0.683 \text{ skip}; n := n + 1 \\
}\end{align*}
\]

where \( \text{NormDist}(\mu, \sigma) \) is a normal distribution with average of \( \mu \) and standard deviation of \( \sigma \). Since the sum of normal distributions is also a normal distribution, we can calculate that \( E(h) = 106.548x \), which can be proved by our prototype, with the synthesized invariant loop \( 106.548x - 106.548n + h \).

Further, we consider a slightly more involved version. A flight starting from Beijing to Hongkong takes 220 minutes with an average delay of 40 minutes. An alternative route starts from Beijing, takes a shift in Shanghai, then ends in HongKong. The first flight takes 135 minutes, with an average delay of 21 minutes. The second flight takes 135 minutes, with an average delay of 40 minutes. If a passenger takes \( x \)-times flights for this routine, the total expected time for him can be modeled as:

\[
\begin{align*}
  h &:= 0; \\
n &:= 0; \\
\text{while}(n \leq x)\{ \\
  &\quad h := h + 220 + \text{NormDist}(40, \sigma) \times 0.683 \text{ skip}; h := h + 270 + \text{NormDist}(21, \sigma) + \text{NormDist}(40, \sigma); \\
  &\quad n := n + 1 \\
}\end{align*}
\]

We can see \( E(h) = 282.507x \), which can be verified by our method, with the synthesized loop invariant \( 282.507(x - n) + h \).

D.3 Parametric Example

In this section we consider some parametric examples such that we can observe how our approach scales with the number of variables and the degree of the templates.
**Parametric Linear Program** We first consider a linear program. The paradigm of program here goes as

```plaintext
h := 0;
while(t > 0){
    h := h + x_1 + \cdots + x_n [0.5] h := h + x_1 + \cdots + x_n + \text{UnifDist}(0, 2n)
    t := t - 1
}
```

with \( n \) an integer used as a parameter. The post-expectation of the program is \( h \), and the related pre-expectation is \((\frac{3}{2} + x_1 + \cdots + x_n) t\).

The degree of invariant template is fixed to be 2, the synthesized invariant of this parameterized programs are of the form of \( h + (\frac{3}{2} + x_1 + \cdots + x_n) t \). In Table 4, we observe that the number of coefficients remained to be determined by SDP tool is quadratic to the number of variables. Moreover, fitting method shows time is approximately cubic to the number of variables.

**Table 4.** Running time (in seconds) with degree 2 for the parametric linear program. The line for "Number of parameters" counts the numbers of coefficients in invariant template remaining to be determined. The line for "solvertime" describes the time SDP solver takes to solve the constraint solving problem.

| Number of variables | \( n = 15 \) | \( n = 20 \) | \( n = 25 \) | \( n = 30 \) | \( n = 35 \) | \( n = 40 \) |
|---------------------|--------------|--------------|--------------|--------------|--------------|--------------|
| Number of parameters| 136          | 231          | 351          | 496          | 741          | 946          |
| Solvertime          | 0.41         | 1.30         | 2.44         | 8.30         | 20.56        | 46.62        |
| Total time          | 1.11         | 2.26         | 4.14         | 12.58        | 22.53        | 78.40        |

**Quadratic Program** We consider the quadratic program obtained from the above program by replacing \( x_1 \) by \( x_1^2 \). We need the degree = 4 for the template. In Table 5, we observe that the number of coefficients is quadratic to the number of variables. Runtime grows rapidly when variables are being added. And if we observe the case when the amount of coefficient is similar in degree 2 and 4 with 30 or 8 variables separately. The solver time is 8.30s for the former, and 12.66s for the latter. So the increase in time is not only due to the number of coefficients in invariant template. One reason is that the constraint matrix in SDP problem becomes much more coarse, which makes the solver much more difficult to solve the problem.

**Polynomial Program** Our next trial is to consider parameterized version of Example 7

\( \langle xy^n - x^2 \rangle z := 0; \text{while}(0 < x < y^n)\{x := x + 1 [0.5] x := x - 1; z := z + 1; \} \langle x \rangle \)
Table 5. Running time (in seconds) for the parametric polynomial program with degree 4

| Number of variables | $n = 5$ | $n = 6$ | $n = 7$ | $n = 8$ | $n = 9$ | $n = 10$ |
|---------------------|---------|---------|---------|---------|---------|----------|
| Number of parameters| 126     | 210     | 330     | 495     | 715     | 1001     |
| Solvetime           | 0.96    | 1.59    | 4.29    | 12.66   | 29.42   | 96.24    |
| Total time          | 1.43    | 4.57    | 5.27    | 14.98   | 36.04   | 107.30   |

with $n$ as a parameter. The relevant invariant template has a total degree $2n$. Table 6 shows the result.

Table 6. Running time (in seconds) for parameterized version of Example 7

| Total degree of program | $n = 6$ | $n = 7$ | $n = 8$ | $n = 9$ | $n = 10$ | $n = 11$ |
|-------------------------|---------|---------|---------|---------|---------|----------|
| Number of parameters    | 455     | 680     | 969     | 1330    | 1771    | 2300     |
| Solvetime               | 19.83   | 64.63   | 182.69  | 470.43  | 1099.2  | 2223.5   |
| Total time              | 23.19   | 75.02   | 213.90  | 498.52  | 1153.1  | 2431.6   |

Similarly as Table 5, the result shows a rapid increment of time as degree gets larger and it grows faster than that of Table 4, which indicates that degree is the major influence on executing time of our method.

As a conclusion, the main influence on runtime for our method is the total degree of invariant template. The amount of variables also has an influence.

D.4 Nested Loop program

We present an example originally come from [5] for analysis of almost sure termination. Here we try to generate an loop invariant for the program.

```c
real x, y;
int k = 0;
while(x ≤ m){
    y = 0;
    while(y ≤ n){
        y = y + UnifDist(−0.1, 0.2)
    }
    x = x + UnifDist(−0.1, 0.2);
    k = k + 1
}
```

Let $postE = k$, for $preE = 20(m − x)$, we can synthesize an invariant $I = k + 20(m − x)$ as the method shows.
Detailed Evaluation for the Tools

We give a detailed explanation for Table 2 here.

- **Prinsys** [15], LIP (Lagrange Interpolation Prototype) [6] and our method take a probabilistic program with a template, then derive constraints for the invariant. Template in **Prinsys** is linear consisting assertions for the program and undecided parameters as input. LIP and our prototype could synthesize a polynomial invariant. However as the degree growing up, LIP could hardly generate a valid result.

- **Prinsys** supports linear programs, LIP also supports polynomial programs in theory, but does not terminate within half an hour timeout. Our prototype applies SDP solver, and can handle polynomial programs.

- **PPA** (Probabilistic Program Analyzer) [2] takes the program and property to verify as input and derives the expectation invariant by obtaining a pre-fixed point over a polyhedral cone of expressions. PPA does not return an explicit invariant in any form, instead it returns the expectation for the property to be valid. The method is applied on linear programs. As shown below, PPA scales very well for linear programs, but it does not supports polynomial programs.

- **PPA** and our prototype are numeric, thus in PPA equality in loop condition can be treated only by changing them into inequalities, while in our prototype, equality with constant number in right side might fail due to numerical error. Thus, both our prototype and PPA fail to verify the coin example (see Table 1). The failure might due to the inaccuracy in numerical calculation, where integer is turned into float after multiplication with a float, which may lead to a failure for testing equality. In this way, the requirements in Theorem 5 are obeyed.

On the contrary, **Prinsys** and LIP are symbolic using exploit SMT-solvers and need to perform quantifier elimination, and can handle programs with constraints using equations more flexibly. Both tools succeed in verifying the coin example.

- **Prinsys** and LIP can handle only discrete distributions. On the other side, PPA and our prototype can handle continuous distributions.

- For program verification on Hoare triple, our method is semi-complete in the way that if the Hoare triple holds, the method can theoretically synthesize an invariant. **Prinsys** needs to encode pre- and post-expectation into its template, which requires manually construction and is often tricky. LIP does not have an assurance in completeness, and PPA cannot support program verification because it requires all variables to be assigned a value.