
# Learning Probabilistic Termination Proofs

Alessandro Abate(✉), Mirco Giacobbe(✉), and Diptarko Roy(✉)

University of Oxford, Oxford, UK
{alessandro.abate,mirco.giacobbe, diptarko.roy}@cs.ox.ac.uk


**Abstract.** We present the first machine learning approach to the termination analysis of probabilistic programs. Ranking supermartingales (RSMs) prove that probabilistic programs halt, in expectation, within a finite number of steps. While previously RSMs were directly synthesised from source code, our method learns them from sampled execution traces. We introduce the *neural ranking supermartingale*: we let a neural network fit an RSM over execution traces and then we verify it over the source code using satisfiability modulo theories (SMT); if the latter step produces a counterexample, we generate from it new sample traces and repeat learning in a counterexample-guided inductive synthesis loop, until the SMT solver confirms the validity of the RSM. The result is thus a sound witness of probabilistic termination. Our learning strategy is agnostic to the source code and its verification counterpart supports the widest range of probabilistic single-loop programs that any existing tool can handle to date. We demonstrate the efficacy of our method over a range of benchmarks that include linear and polynomial programs with discrete, continuous, state-dependent, multi-variate, hierarchical distributions, and distributions with undefined moments.

## 1 Introduction

Probabilistic programs are programs whose execution is affected by random variables [17,19,23,29,36]. Randomness in programs may emerge from numerous sources, such as uncertain external inputs, hardware random number generators, or the (probabilistic) abstraction of pseudo-random generators, and is intrinsic in quantum programs [34]. Notable exemplars are randomised algorithms, cryptographic protocols, simulations of stochastic processes, and Bayesian inference [7,33]. Verification questions for probabilistic programs require reasoning about the probabilistic nature of their executions in order to appropriately characterise properties of interest. For instance, consider the following question, corresponding to the program in Fig. 1: will an ambitious marble collector eventually gather any arbitrarily large amounts of red and blue marbles? Intuitively, the question has an affirmative answer regardless of the initially established target amounts, since there is always a chance of collecting a marble of either color. Notice that, if the probabilistic choice is replaced with non-determinism, as often happens in software verification, an adversary may exclusively draw one color of marble


A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 3–26, 2021.
https://doi.org/10.1007/978-3-030-81688-9_1

and make the program run forever. The question that matches the original intuition is whether the expected number of steps to termination is finite; this is the *positive almost-sure termination* (PAST) question [8,10,13,19,27].

```
1  while (red > 0 || blue > 0) do
2    p ∼ Bernoulli(.01);
3    if p == 1 then
4      red = red - 1
5    else
6      blue = blue - 1
7    fi
8  od
```

**Fig. 1.** The ambitious marble collector (the variables `red` and `blue` are initialised non-deterministically).

Probabilistic termination analysis is typically mechanised through the automated synthesis of *ranking supermartingales* (RSMs), which are functions of the program variables whose value (i) decreases in expectation by a discrete amount across every loop iteration and (ii) is always bounded from below; an RSM formally witnesses that a program is PAST [10,13]. Early techniques for discovering RSMs reduced the synthesis problem from the source code of the program into constraint solving [10]. These methods have lent themselves to various generalisations, including polynomial programs, programs with non-determinism, lexicographic and modular termination arguments, and persistence properties [2,14–16,20,25]. Recently, for special classes of probabilistic programs or term rewriting systems, novel automated proof techniques that leverage computer algebra systems and satisfiability modulo theories (SMT) have been introduced [5,6,38,39,41]. All the above methods are sound and, under specific assumptions, complete; they represent the state of the art for the class of programs they have been designed for. However, their assumptions are often too restrictive for the analysis of many simple programs. In particular, to the best of our knowledge, none can identify an RSM for the program in Fig. 1. For this simple program, it is easy to argue that the expected output of the *neural network* depicted in Fig. 2 decreases after every iteration of the loop and that it is always non-negative (see Ex. 1). As such, this neural network is an appropriate RSM for the program.

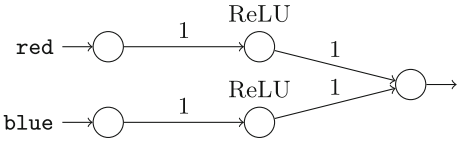


**Fig. 2.** A neural ranking supermartingale for the program in Fig. 1.

We present a novel method for discovering RSMs using machine learning together with SMT solving. We introduce the *neural ranking supermartingale* (NRSM) model, which lets a neural network mimic a supermartingale over sampled execution traces from a program. We train an NRSM using standard optimisation algorithms over a loss function that makes the neural network decrease—in average—across sampled iterations. We phrase the certification problem into that of computing a counterexample for the NRSM. To do so, we encode the neural network together with the expected value of the program variables; then, we use an SMT solver for verifying that the expected output of the network decreases along every execution. If the solver falsifies the NRSM, then it provides a counterexample that we use to guide a resampling of the execution traces; with this new data we retrain the neural network and repeat verification in a *counterexample-guided inductive synthesis* (CEGIS) fashion, until the SMT solver determines that no counterexample exists [4,44]. In the latter case, the solver has certified the generated NRSM; our method thus produces a *sound* PAST proof or runs indefinitely. Our procedure does not return for programs that are not PAST and may, in general, not return for some PAST instances. However, we experimentally demonstrate that, in practice, our method succeeds over a broad range of PAST benchmarks within a few CEGIS iterations. Previously, machine learning has been applied to the termination analysis of deterministic programs and to the stability analysis of dynamical systems [1,12,21,24,28,30–32,42,43,45]; our method is the first machine learning approach for probabilistic termination analysis.

Our approach builds upon two key observations. First, the average of expressions along execution traces statistically approximates their true expected value. Thanks to this, we obtain a machine learning model for *guessing* RSM candidates that only requires execution traces and is thus agnostic to the source code. Second, solving the problem of *checking* an RSM is simpler than solving the entire termination analysis problem. Reasoning about source code is entirely delegated to the checking phase which, as such, supports programs that are out of reach to the available probabilistic termination analysers.

We experimentally demonstrate that our method is effective over many programs with linear and polynomial expressions, with both discrete and continuous distributions. This includes joint distributions, state-dependent distributions, distributions whose parameters are in turn random (hierarchical models), and distributions with undefined moments (e.g., the Cauchy distribution). We compare our method with a tool based on Farkas' lemma and with the tools Amber and Absynth [2,39,41]; whilst our software prototype is slower than these alternatives, it covers the widest range of benchmark single-loop programs.

Summarising, our contribution is fivefold. First, we present the first machine learning method for the termination analysis of probabilistic programs. Second, we introduce a loss function for training neural networks to behave as ranking supermartingales over execution traces. Third, we show an approach to verify the validity of ranking supermartingales using SMT solving, which applies to a wide variety of single-loop probabilistic programs. Fourth, we experimentally

demonstrate over multiple baselines and newly-defined benchmarks the practical efficacy of our method. Fifth, we built a software prototype for evaluating our method.

$$
\begin{array}{rll}
x \in \text{Vars} & & \text{(variables)} \\
N \in \mathbb{R} & & \text{(numerals)} \\
\text{op}_2 & ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \ldots & \text{(binary operators)} \\
E & ::= x \mid N \mid E \ \text{op}_2 \ E \mid \texttt{-}E & \text{(arithmetic expressions)} \\
D & ::= \texttt{Bernoulli(}\ E\ \texttt{)} \mid \texttt{Gaussian(}\ E\texttt{,}\ E\ \texttt{)} \mid \ldots & \text{(probability distributions)} \\
B & ::= B \ \text{op}_2 \ B \mid \texttt{!}\ B \mid E \ \text{op}_2 \ E \mid \texttt{true} \mid \texttt{false} & \text{(Boolean expressions)} \\
C & ::= \texttt{skip} & \text{(commands)} \\
 & \mid x\ \texttt{=}\ E & \text{(deterministic assignment)} \\
 & \mid x \sim D & \text{(probabilistic assignment)} \\
 & \mid C\ \texttt{;}\ C & \text{(sequential composition)} \\
 & \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi} & \text{(conditional composition)}
\end{array}
$$

**Fig. 3.** Syntax of loop-free probabilistic programs.

## 2 Termination Analysis of Probabilistic Programs

We treat the termination analysis of single-loop probabilistic programs. We consider an imperative language that includes C-like arithmetic and Boolean expressions, and sequential and conditional composition of commands [13,17,19,23].

*Syntax.* A grammar for this language is shown in Fig. 3. We analyse single-loop programs of the form

$$
\begin{array}{l}
\texttt{while}\ G\ \texttt{do} \\
\quad U \\
\texttt{od}
\end{array}
$$

where the loop guard $G$ is a Boolean expression and the update statement $U$ is a command. Variables are real-valued and can be either assigned to arithmetic expressions using the usual = operator, or sampled from probability distributions using the $\sim$ operator. Probability distributions, which can be either discrete or continuous, take not only parameters that are constant, and thus known at compile time, but also parameters that depend on other variables, and thus determined only at run time. In other words, distributions may depend on the current state of the program, which is a random variable. Also, they may depend on other random variables; as such, distributions may be multi-variate, resulting from models with coupled and hierarchically-structured variables.

*Semantics.* The operational semantics of a probabilistic program induces a probability space over runs, together with a stochastic process [13]. A state of the process is an element of $\mathbb{R}^n$ with $n = |\text{Vars}|$, that is, a valuation of the variables in the program. The space of outcomes $\Omega_{\mathsf{run}}$ of a program is the set of runs. A run is a possibly infinite sequence of variable valuations (taken at the beginning of every loop iteration). This comes with a $\sigma$-algebra $\mathcal{F}$ of measurable subsets of $\Omega_{\mathsf{run}}$. Initial states are chosen non-deterministically and, thereafter, the process is purely probabilistic. Every initial state $x_0 \in \mathbb{R}^n$ determines a unique probability measure $\mathbb{P}^{(x_0)} \colon \mathcal{F} \to [0,1]$, namely a probability measure conditional on the state $x_0$. The associated stochastic process is $X^{(x_0)} = \{X_t^{(x_0)}\}_{t \in \mathbb{N}}$, where $X_t^{(x_0)}$ is a random vector representing the state at the $t$-th step, initialised as $X_0^{(x_0)} = x_0$. Given an initial condition $x_0$ and a solution process $X^{(x_0)}$, the associated termination time is a random variable $T^{(x_0)}$ denoting the length of an execution, which takes values in $\mathbb{N} \cup \{\infty\}$.

*Positive Almost-Sure Termination.* Runs are probabilistic and thus also the notion of termination requires a quantitative semantics. The termination question is generalised to the notions of *almost-sure* and *positive almost-sure* termination. Almost-sure termination (AST) indicates whether the joint probability of all runs that do not terminate is zero; positive almost-sure termination (PAST), which is stronger, indicates whether the expected number of steps to termination is finite. Formally, a probabilistic program terminates positively almost-surely if $\mathbb{E}[T^{(x_0)}] < \infty$ for all $x_0 \in \mathbb{R}^n$. Notably, this implies that the program also terminates almost-surely, that is, $\mathbb{P}[T^{(x_0)} < \infty] = 1$ for all $x_0 \in \mathbb{R}^n$. We provide conditions ensuring that probabilistic programs are PAST and, consequently, that they are AST. Notice that the converse may not be true, that is, there exist programs that are AST but not PAST. Our method addresses the PAST question only, by building upon the theory of ranking supermartingales [10].

*Ranking Supermartingales.* A scalar stochastic process $\{M_t\}$ is an RSM if, for some $\epsilon > 0$ and lower bound $K \in \mathbb{R}$,

$$\mathbb{E}\left[M_{t+1} \mid M_t = m_t, \ldots, M_0 = m_0\right] \leq m_t - \epsilon \tag{1}$$

and $M_t \geq K$ for all $t \geq 0$. In other words, this a process whose values are bounded from below and whose expected value decreases by a discrete amount at each step of the program. We prove that a program is PAST by mapping $X^{(x_0)}$ into an RSM. Our goal is finding a function $\eta \colon \mathbb{R}^n \to \mathbb{R}$ such that, for every initial condition $x_0$, it satisfies the following two properties:

(i) $\mathbb{E}[\eta(X_{t+1}^{(x_0)}) \mid X_t^{(x_0)} = x] \leq \eta(x) - \epsilon$ for all $x \in I$ and
(ii) $\eta(x) \geq K$ for all $x \in I$,

where $I \subseteq \mathbb{R}^n$ is some sufficiently strong loop invariant that can be the loop guard or, possibly, a stronger condition. Function $\eta$ maps the entire stochastic process into an RSM. For this reason, we call $\eta$ an RSM for the program.

**Input**: Single-loop probabilistic program $(G, U)$,
Initial state $x_0 \in \mathbb{R}^n$
**Output**: Transition samples $S \subset \mathbb{R}^n \times \mathcal{P}(\mathbb{R}^n)$

$S \leftarrow \emptyset$;
$P' \leftarrow \{x_0\}$;
**for** $i \leftarrow 1$ **to** $k$ **do** // $k$ = path length
  $P \leftarrow P'$;
  $P' \leftarrow \emptyset$;
  $p \leftarrow$ pick arbitrary element from $P$;
  **if** `eval(`$G$`,`$p$`)` $= \mathit{True}$ **then**
    **for** $j \leftarrow 1$ **to** $m$ **do** // $m$ = branching factor
      $P' \leftarrow P' \cup \{$`exec(`$U$`,`$p$`)`$\}$
    $S \leftarrow S \cup \{(p, P')\}$;
**return** $S$

**Algorithm 1:** Interpreter

*Example 1.* Consider the ambitious marble collector problem from Fig. 1. An RSM for this program is a function $\eta$ mapping variables `red` and `blue` to $\mathbb{R}$. Rephrasing condition (i) over this program, $\eta$ is required to satisfy

$$0.01 \cdot \eta(\texttt{red} - 1, \texttt{blue}) + 0.99 \cdot \eta(\texttt{red}, \texttt{blue} - 1) \leq \eta(\texttt{red}, \texttt{blue}) - \epsilon, \quad (2)$$

for all $\texttt{red}, \texttt{blue} \in \mathbb{Z}$ that satisfy $\texttt{red} > 0 \vee \texttt{blue} > 0$, that is, the loop guard. So, for example, function $\eta(\texttt{red}, \texttt{blue}) = \texttt{red} + \texttt{blue}$ satisfies this condition; however, it may take any negative value over the arguments `red` and `blue` such that $\texttt{red} > 0 \vee \texttt{blue} > 0$, thus violating condition (ii). By contrast, the neural network in Fig. 2 succeeds at satisfying both conditions. In fact, the network realises function $\eta(\texttt{red}, \texttt{blue}) = \max\{\texttt{red}, 0\} + \max\{\texttt{blue}, 0\}$, which satisfies Eq. (2) and is bounded from below by zero. □

## 3 Training Neural Ranking Supermartingales

Our framework synthesises RSMs by learning from program execution traces. We define a loss function, that measures the number of sampled program transitions that do not satisfy the RSM conditions. Applying gradient-descent optimisation to the loss function guides the parameters to values at which the candidate's value decreases, on average, across sampled program transitions. Since the learner does not require the underlying program (only execution traces), the learner is agnostic to the structure of program expressions, and the cost of evaluating the loss function does not scale with the size of the program.

A dataset of sampled transitions is produced using an instrumented program interpreter (Algorithm 1). At a program state $p$, the interpreter runs the loop body $m$ times to sample successor states $P'$, where $m$ is a branching factor hyper-parameter, before resuming execution from an arbitrarily chosen successor. The dataset $S$ consists of the union of pairs $(p, P')$ generated by the interpreter.

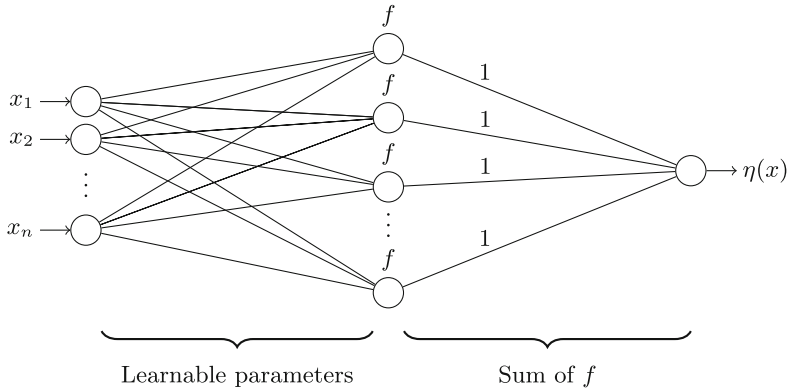


**Fig. 4.** Neural ranking supermartingale architecture.

The loss function is used to optimise the parameters of an NRSM, whose architecture is shown in Fig. 4. This is a neural network with $n$ inputs, one output neuron, and one hidden layer. The hidden layer has $h$ neurons, each of which applies an activation function $f$ to a weighted sum of its inputs. In our experiments, the activation function $f$ is either $f(x) = x^2$ or $f(x) = \mathrm{ReLU}(x)$, where $\mathrm{ReLU}(x) = \max\{x, 0\}$.

Therefore, we employ either of the two following functional templates, defined over the learnable parameters $w_{i,j}$ and $b_i$:

– Sum of ReLU (SOR):

$$\eta(x_1, \ldots, x_n) = \sum_{i=1}^{h} \mathrm{ReLU}\left(\sum_{j=1}^{n} w_{i,j} x_j + b_i\right); \tag{3}$$

– Sum of Squares (SOS):

$$\eta(x_1, \ldots, x_n) = \sum_{i=1}^{h} \left(\sum_{j=1}^{n} w_{i,j} x_j + b_i\right)^2. \tag{4}$$

These choices of activation mean that our NRSMs are restricted to non-negative outputs, and therefore satisfy condition (ii) by construction. The learner therefore needs to find parameters that satisfy condition (i), which requires $\eta$ to decrease in expectation by at least some positive constant $\epsilon > 0$.

The role of the loss function is to allow the learner parameters to be optimised such that the NRSM decreases, on average, across sampled transitions. That is, the loss function evaluates the number of sampled transitions for which the NRSM does not satisfy the RSM condition (i), and the lower its value, the more the neural network behaves like an RSM.

Concretely, the loss associated with a state $p$ and its successors $P'$ is:

$$L(p, P') = \text{softplus}\left(\mathbb{E}_{p' \sim P'}[\eta(p')] - \eta\left(p\right) + \epsilon\right), \tag{5}$$

where $\text{softplus}(x) = \ln(1 + e^x)$, and $\mathbb{E}_{p' \sim P'}[\eta(p')]$ is the average of $\eta$ over the sampled successor states $p'$ from $P'$.

We then train an NRSM by solving the following optimisation problem:

$$\min \frac{1}{|S|} \sum_{(p,P') \in S} L(p, P'), \tag{6}$$

which aims to minimise the average loss over all sampled transitions in the dataset $S$, over the trainable weights $w_{1,1}, \ldots, w_{h,n} \in \mathbb{R}$ and biases $b_1, \ldots, b_h \in \mathbb{R}$. This objective is non-convex and non-linear, and we resort to gradient-based optimisation (see Sect. 6).

The softplus in Eq. (5) forces the parameters to satisfy condition (i) uniformly across all sampled transitions in the dataset, rather than decreasing by a large amount in expectation over some transitions at the expense of failing to decrease sufficiently quickly for others. Furthermore, for NRSMs of SOR form we replace the ReLU activation function by softplus, to help gradient descent converge faster. Softplus approximates the ReLU function, and has the same asymptotic behaviour, but results in an NRSM that is differentiable w.r.t. the network parameters at all inputs, unlike ReLU [22, p.193]. However, since softplus is a transcendental function, we revert back to using a simpler ReLU activation when verifying an SOR candidate.

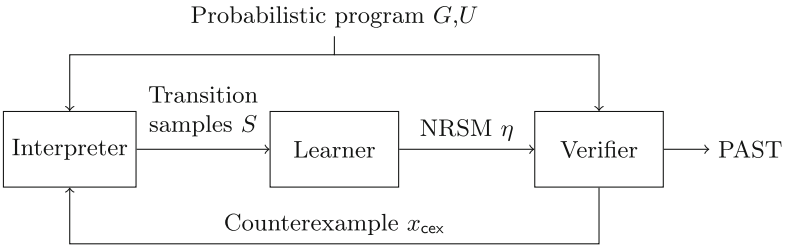


**Fig. 5.** CEGIS architecture for the adversarial training of NRSM.

A CEGIS loop integrates the learner and verifier (Fig. 5). The dataset $S$ sampled by the interpreter is used to train an NRSM candidate $\eta$ according to Eq. (6). The verifier checks whether $\eta$ satisfies condition (i), concluding either that the program is PAST, or producing a counterexample program state $x_{\mathsf{cex}}$ for which $\eta$ does not satisfy (i). The interpreter generates new traces, starting at $x_{\mathsf{cex}}$, forcing it to explore parts of the state space over which the NRSM fails to decrease sufficiently in expectation.

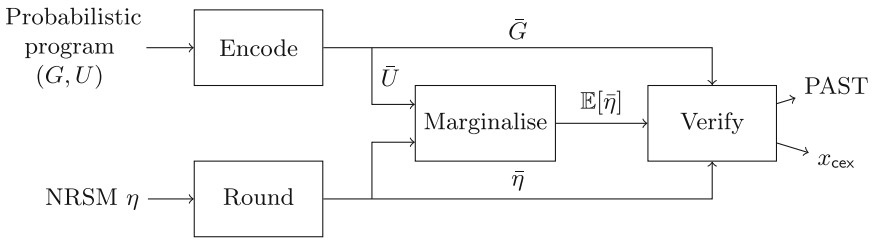


**Fig. 6.** Verifier architecture.

## 4 Verifying Ranking Supermartingales by SMT Solving

To verify an NRSM we must check that it decreases in expectation by at least some constant (condition (i)). Condition (ii) is satisfied by construction because the network's output is non-negative for every input, leaving only condition (i) to verify. The architecture of the verifier is depicted in Fig. 6. First, a program $(G, U)$ is translated into an equivalent logical formulation denoted by $\bar{G}$ and $\bar{U}$ ('Encode' block), which are used to construct a closed-form term $\mathbb{E}[\bar{\eta}]$ for the NRSM's expected value at the end of the loop body ('Marginalise' block). Secondly, given an NRSM $\eta$, its parameters are rounded and encoded as a logical term $\bar{\eta}$ ('Round' block). Then, the satisfiability of the following formula is decided using SMT solving:

$$\bar{G}(x_1 \dots x_n) \wedge \mathbb{E}[\bar{\eta}](x_1 \dots x_n) > \bar{\eta}(x_1 \dots x_n) - \epsilon. \tag{7}$$

This is the dual satisfiability problem for the validity problem associated with condition (i) on page 5. If Eq. (7) is unsatisfiable, then $\bar{\eta}$ is a valid RSM and we conclude the program is PAST. Otherwise, the solver yields a counterexample state $x_{\mathsf{cex}} \in \mathbb{R}^n$.

The rounding strategy ('Round' block) provides multiple candidates to the verifier by adding i.i.d. noise to parameters and rounding them to various precisions. Setting parameters that are numerically very small to zero is useful since learning that a parameter should be exactly zero could require an unbounded number of samples; rounding provides a pragmatic way of making this work in practice. If none of the generated candidates are valid NRSMs, all counterexamples are passed back to the interpreter which generates more transition samples for the learner (Fig. 5).

$$\begin{aligned}
&x \in \text{Vars} && \text{(variables)}\\
&N \in \mathbb{R} && \text{(numerals)}\\
&\tau ::= x \mid N \mid \tau + \tau \mid \tau - \tau \mid \dots && \text{(terms)}\\
&\phi ::= \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \tau \leq \tau \mid \tau = \tau \mid \dots && \text{(formulae)}
\end{aligned}$$

**Fig. 7.** Quantifier-free first-order logic formulae.

Notice that, if a program's guard predicate is not strong enough to allow a valid RSM to be verified as such, the CEGIS loop will run indefinitely. In general, stronger supporting loop invariants may need to be provided.

### 4.1 From Programs to Symbolic Store Trees

We now introduce a translation from a loop-free probabilistic program to a *symbolic store tree* (Fig. 8), a datastructure representing the distribution over program states at the end of a loop iteration as a function of the variable valuation at its start. Marginalising out the probabilistic choices made in the loop yields the NRSM expectation $\mathbb{E}[\bar{\eta}]$.

$$\begin{aligned}
&\pi ::= \tau \mid \texttt{Bernoulli}(\tau) \mid \texttt{Gaussian}(\tau, \tau) \mid \ldots && \text{(probabilistic terms)}\\
&\Sigma = \{x_1 \mapsto \pi_1, \ldots, x_n \mapsto \pi_n\} && \text{(symbolic store)}\\
&\sigma ::= \texttt{node}(\phi, \sigma, \sigma) \mid \Sigma && \text{(symbolic store tree)}
\end{aligned}$$

**Fig. 8.** Symbolic store tree.

This requires a form of symbolic execution. We represent program states symbolically using *symbolic stores*, denoted $\Sigma$ (Fig. 8), which map program variables to *probabilistic terms*. A probabilistic term $\pi$ can be either a first-order logic term (Fig. 7) representing an arithmetic expression, or a placeholder for a probability distribution whose parameters are terms (allowing them to be functions of the program state). Finally, *symbolic store trees* $\sigma$ (Fig. 8) represent the set of control-flow paths through the loop body, arising from if-statements; it is a binary tree with symbolic stores at the leaves, and internal nodes labelled by logical formulae over program variables.

$$\begin{aligned}
&\text{enc}(\Sigma, x) = \Sigma(x)\\
&\text{enc}(\Sigma, -O) = -\text{enc}(\Sigma, O) \qquad \text{enc}(\Sigma, \texttt{!}\ O) = \neg\text{enc}(\Sigma, O)\\
&\text{enc}(\Sigma, O_1\ \text{op}_2\ O_2) = \text{enc}(\Sigma, O_1)\ \boxed{\text{op}_2}\ \text{enc}(\Sigma, O_1)\\
&\\
&\text{enc}(\Sigma, \texttt{skip}) = \Sigma\\
&\text{enc}(\Sigma, x\ \texttt{=}\ E) = \Sigma[x' \mapsto \text{enc}(\Sigma, E)]\\
&\text{enc}(\Sigma, C_1\ \texttt{;}\ C_2) = \text{enc}(\text{enc}(\Sigma, C_1), C_2)\\
&\text{enc}(\Sigma, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}) = \texttt{node}(\text{enc}(\Sigma, B), \text{enc}(\Sigma, C_1), \text{enc}(\Sigma, C_2))\\
&\\
&\text{enc}(\texttt{node}(\phi, \sigma_1, \sigma_2), C) = \texttt{node}\,(\phi, \text{enc}(\sigma_1, C), \text{enc}(\sigma_2, C))\\
&\\
&\text{enc}(\Sigma, x \sim \texttt{Bernoulli}(E)) = \Sigma[x' \mapsto \nu, \nu \mapsto \texttt{Bernoulli}(\text{enc}(\Sigma, E))]\\
&\text{enc}(\Sigma, x \sim \texttt{Gaussian}(E_1, E_2)) = \Sigma[x' \mapsto \nu, \nu \mapsto \texttt{Gaussian}(\text{enc}(\Sigma, E_1), \text{enc}(\Sigma, E_2))]\\
&\quad\vdots \qquad\qquad \text{where every } \sim \text{ command creates a fresh } \nu \text{ variable.}
\end{aligned}$$

**Fig. 9.** Translation from a loop-free command to a symbolic store tree.

Figure 9 defines a translation from an initial symbolic store tree and command to a new symbolic store tree characterising the distribution over states after executing the command. At the top level, we provide the command $G$ (the loop body) and the initial symbolic store $\{x'_1 \mapsto x_1, \ldots, x'_n \mapsto x_n\}$, where primed variables represent the variable valuation at the end of the iteration, whereas unprimed variables represent the variable valuation at the beginning of the loop.

The first four cases of Fig. 9 define the translation of arithmetic expressions (to terms) and Boolean expressions (to formulae), by replacing program syntax with the corresponding logical operators.

The next four cases define the translation of commands. `skip` leaves the symbolic store unchanged. For deterministic assignments, the right hand side of the assignment is translated in the current symbolic store and bound to the variable. Sequential composition involves translating the first command, and translating the second command in the resulting store tree. A conditional statement creates a new `node` in the symbolic store tree that selects between the two recursively-translated branches, based on the formula derived from the guard predicate. These rules assume the store tree to be a leaf-level symbolic store, because the next rule handles the case where the initial symbolic store tree is a `node`. Finally, if the command is a probabilistic assignment, we translate the parameters to terms, and bind the resulting probabilistic term to a freshly generated symbol. This allows variables to be overwritten by multiple probabilistic sampling operations in the body of the loop. The mapping of variables to distributions in leaf-level stores defines the probability density over particular probabilistic choices.

*Example 2.* Figure 10 is the store tree produced for the ambitious marble collector program (Fig. 1). Each leaf-level store in the program's store tree corresponds to a particular control-flow path through the loop body. The interpretation of a symbolic store tree is that if we fix the outcomes of the probabilistic sampling operations performed by the loop body, then the state of the variables at the end of the iteration is determined by the predicates labelling the internal nodes.

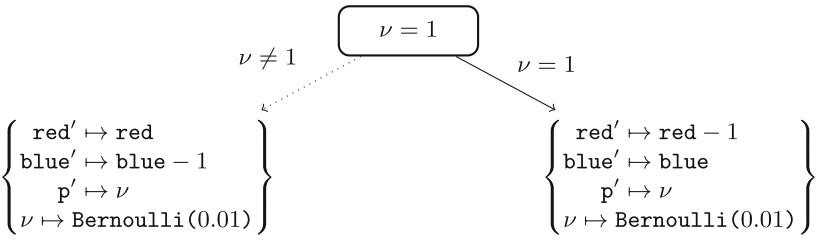


**Fig. 10.** A store tree for the program in Fig. 1.

### 4.2 Marginalisation

To construct the closed-form logical term representing the NRSM's expected value at the end of an iteration, the probabilistic choices in the symbolic store tree must be marginalised out. If the program is limited to discrete random variables with finite support, we automatically marginalise the random choices by enumeration (for both SOR- and SOS-form NRSMs), as illustrated by Ex. 3.

*Example 3.* The ambitious marble collector program of Fig. 1, yields the symbolic store tree of Fig. 10. Suppose we want to marginalise the NRSM:

$$\eta(\texttt{red}, \texttt{blue}) = \mathrm{ReLU}(w_{1,1} \cdot \texttt{red} + w_{1,2} \cdot \texttt{blue} + b_1) \\ + \mathrm{ReLU}(w_{2,1} \cdot \texttt{red} + w_{2,2} \cdot \texttt{blue} + b_2) \quad (8)$$

with respect to this symbolic store tree. We first apply the encoding of the NRSM to each leaf-level symbolic store of Fig. 10, and enumerate the possible choices for the probabilistic choices (which in this example is limited to $\nu \in \{0, 1\}$), using the bindings of $\nu$ to distributions in leaf-level stores to compute the probability mass of each choice. After resolving the predicates for each choice of $\nu$, this yields:

$$0.01 \cdot \eta(\texttt{red} - 1, \texttt{blue}) + 0.99 \cdot \eta(\texttt{red}, \texttt{blue} - 1). \quad (9)$$

The term (9) is then provided as the value of the NRSM's expectation to the verifier. □

If the program samples from continuous distributions, we marginalise SOS-form NRSMs (but not SOR-form NRSMs) by substituting symbolic moments for a set of supported built-in distributions, including `Gaussian`, `MultivariateGaussian`, and `Exponential`, though could include any distribution whose closed-form symbolic moments are available. Example 4 provides an example. This strategy is general enough to support a wide variety of programs, including those of Sect. 5. If a sampling distribution lacks symbolic moments, the cumulative distribution function can also be utilised, which is illustrated in the `slicedcauchy` case study (Fig. 15).

*Example 4.* Consider an NRSM $\eta(\mathtt{x}) = (w\mathtt{x} + b)^2$ and a symbolic store tree $\texttt{node}(p = 1, \sigma_1, \sigma_2)$ where $\sigma_1 = \{x \mapsto x + v, v \mapsto \texttt{Exp}(\lambda), p \mapsto \texttt{Bernoulli}(3/4)\}$ and $\sigma_2 = \{x \mapsto x - v, v \mapsto \texttt{Exp}(\lambda), p \mapsto \texttt{Bernoulli}(3/4)\}$. `Exp(λ)` denotes the exponential distribution with parameter $\lambda$, with pdf denoted $p_{\texttt{Exp}(\lambda)}(v)$. We apply $\eta$ to each leaf-level symbolic store, and marginalise the probabilistic choices. We marginalise $p$ first by enumerating over its possible values, and then marginalise $v$. There are no dependencies between the distributions in this example, so the order in which they are marginalised does not matter.

$$\int_0^\infty \left(\frac{3}{4}\eta(x + v) + \frac{1}{4}\eta(x - v)\right) p_{\texttt{Exp}(\lambda)}(v)\mathrm{d}v. \quad (10)$$

The result of marginalisation is a closed-form expression for Eq. (10). Note that since

$$\eta(x+v) = w^2v^2 + 2(wx+b)wv + (wx+b)^2 \tag{11}$$

and $\int_0^\infty v^n p_{\texttt{Exp}(\lambda)}(v)\mathrm{d}v = \frac{n!}{\lambda^n}$, we use linearity of integration to perform the following simplification, by substituting expressions for the moments of `v` in terms of the parameter $\lambda$:

$$\int_0^\infty \eta(x+v)p_{\texttt{Exp}(\lambda)}(v)\mathrm{d}v = \frac{2w^2}{\lambda^2} + \frac{2(wx+b)w}{\lambda} + (wx+b)^2, \tag{12}$$

which is used to reduce Eq. (10) to a closed form. This is the method used to perform marginalisation for several case studies, including `crwalk`, `gaussrw` and `expdistrw`. □

Notably, our verifier requires the expected value of the RSM to be computed (or soundly approximated) in closed form. We automate marginalisation for discrete distributions of finite support, but require manual intervention for continuous distributions. Nevertheless, our learning component is automated in both cases. Characterising the space of programs with continuous distributions that admit fully automated verification of an RSM is an open question.

## 5 Case Studies

Existing tools for synthesising RSMs reduce the problem to constraint-solving [2,10,11,14], which can limit the generality of the synthesis framework. For instance, methods that convert the RSM constraints into a linear program using Farkas' lemma can only handle programs with affine arithmetic, and can only synthesise linear/affine (lexicographic) RSMs [2,10]. A second restriction of existing approaches is that they typically require the moments of distributions to be compile-time constants. This rules out programs whose distributions are determined at runtime, such as hierarchical and state-dependent distributions. Since the loss function of Eq. (6) only requires execution traces, our learner is agnostic to the structure of program expressions, imposing minimal restrictions on the kinds of expressions that can occur, or the kinds of distributions that can be sampled from. This allows us to learn RSMs for a wider class of programs compared to existing tools, as we will illustrate in this section using a number of case studies.

### 5.1 Non-linear Program Expressions and NRSMs

Many simple programs do not admit linear or polynomial RSMs, such as Fig. 1. Since the program cannot be encoded as a prob-solvable loop (due to the disjunctive guard predicate which cannot be replaced by a polynomial inequality),

it cannot be handled by another recent tool, Amber [39]. However, this program admits the following piecewise-linear NRSM:

$$\mathrm{ReLU}(0 \cdot \texttt{red} + 1 \cdot \texttt{blue} + 11) + \mathrm{ReLU}(1 \cdot \texttt{red} + 0 \cdot \texttt{blue} + 11), \tag{13}$$

whose parameters are learnt by our method, within the first CEGIS iteration.

```
1  while (i <= 10 && s > 0) do
2      r ~ DiscreteUniform({-2, 2});
3      s = r + s * i;
4      p ~ Bernoulli(3/4);
5      if (p == 1) then
6          i = i + 1
7      else
8          i = i - 1
9      fi
10 od
```

**Fig. 11.** Probabilistic factorial (`probfact`).

Similarly, we learn the piecewise-linear NRSM:

$$\mathrm{ReLU}(-1 \cdot \mathtt{i} + 0 \cdot \mathtt{s} + 12) + \mathrm{ReLU}(0 \cdot \mathtt{i} + 0 \cdot \mathtt{s} + 9) \tag{14}$$

for the program in Fig. 11, which contains a bilinear assignment (cf. multiplication of `s` and `i` on line 3), so this program is not supported by [2]. The conjunction in the guard means it is not supported by Amber, either.

```
1  while (x < 10) do
2      rho ~ ContinuousUniform(-0.5, 1);
3      covM = [[1, rho], [rho, 1]];
4      w1, w2 ~ MultivariateGaussian([0, 0], covM);
5      x = x + power((w1 + w2), 2) - 2
6  od
```

**Fig. 12.** Random walk with correlated variables (`crwalk`).

### 5.2 Multivariate and Hierarchical Distributions

Figure 12 is a random walk that samples from a multivariate Gaussian distribution, with zero mean, unit variances, and correlation sampled uniformly in the range $\left[-\frac{1}{2}, 1\right]$. The `MultivariateGaussian` of line 4 is an instance of a hierarchical distribution, having parameters that are random variables. This program also contains a non-linear (polynomial) expression that updates the value of `x`. For `crwalk` we learn an SOS-form NRSM:

$$(0.1 \cdot \mathtt{x} - 47.2)^2, \tag{15}$$

proving this program is PAST. To verify this, the NRSM expectation is computed via the symbolic moments of the multivariate Gaussian distribution, given its covariance matrix (line 3), and then marginalising w.r.t. `rho` (again, using the moments of the uniform distribution over $\left[-\frac{1}{2}, 1\right]$). Unfortunately, it is challenging to translate many simple programs containing hierarchical distributions into ones that can be handled by existing tools. For instance, although it is possible to simulate sampling from a bivariate Gaussian of arbitrary correlation by sampling from independent standard Gaussian distributions, this would involve computing a non-polynomial function of the correlation. Similarly, for the program in Fig. 14 (further discussed below), if a variable is exponentially distributed, $X \sim \texttt{Exponential}(1)$, then $\frac{X}{\lambda} \sim \texttt{Exponential}(\lambda)$, providing a way of simulating an exponential distribution with arbitrary parameter $\lambda$. However, this again requires a non-polynomial program expression (i.e. the reciprocal of $\lambda$) when $\lambda$ is part of the program state and not a constant, and therefore out of scope for methods that restrict program expressions to being linear/polynomial.

### 5.3 State-Dependent Distributions and Non-Linear Expectations

```
1  while (x < 0 && y < 0) do
2      s1 ~ Gaussian(0, 1/4);
3      vx = min(2, max(0.1, vx + s1));
4      s2 ~ Gaussian(0, 1/4);
5      vy = min(2, max(0.1, vx + s2));
6      s3 ~ Gaussian(0, 1/4);
7      rho = min(1, max(-1, rho + s3));
8      mean = [sqrt(1+power(x, 2)),sqrt(1+power(y, 2))];
9      cov = rho * sqrt(vx * vy);
10     covM = [[vx cov], [cov vy]];
11     w1, w2 ~ MultivariateGaussian(mean, covM);
12     x = x + w1;
13     y = y + w2
14 od
```

**Fig. 13.** Gaussian random walk with time-varying and coupled noise (`gaussrw`).

Once we allow hierarchical distributions, it is natural to consider *state-dependent* distributions, i.e. distributions whose parameters depend on the program state rather than being sampled from other distributions. As an example, consider the program in Fig. 13 (a 2-dimensional Gaussian random walk with state-dependent moments). This is unsupported by existing tools because the mean of the Gaussian is a non-polynomial function of the program state. However, after defining the function $\sqrt{1+\texttt{x}^2}$ by means of the following *polynomial* logical inequalities:

$$\texttt{mu\_x}^2 = 1 + \texttt{x}^2 \tag{16}$$

$$\texttt{mu\_x} \geq 1 \tag{17}$$

(similarly for `mu_y`), we express the expected value of an SOS-form NRSM in terms of symbolic moments `mu_x`, etc. Since these moments are state-dependent, we cannot marginalise them out as in the hierarchical case. Instead we perform non-deterministic abstraction, providing inequalities $\frac{1}{10} \leq \texttt{vx}, \texttt{vy} \leq 2$ and $-1 \leq \texttt{rho} \leq 1$ as further verifier assumptions.

```
1  while (x < 10) do
2      s ~ Gaussian(0, 1);
3      lambda = min(10, max(1, lambda + s);
4      step ~ Exponential(lambda);
5      p ~ Bernoulli(3/4);
6      if (p == 1) then
7          x = x + step
8      else
9          x = x - step
10     fi
11 od
```

**Fig. 14.** State-dependent exponential random walk (`expdistrw`).

Even if program expressions are linear, the presence of state-dependent distributions can result in a non-linear verification problem, if the moments are themselves non-linear functions of the program variables. For instance, the program in Fig. 14 represents a 1-dimensional random walk, with steps sampled from an exponential distribution. Since the $n^{\text{th}}$ moment of `Exponential(`$\lambda$`)` is $\frac{n!}{\lambda^n}$, the expectation of an SOS-form NRSM is non-polynomial but still expressible in the theory of non-linear real arithmetic (see Ex. 4). For `expdistrw` we learn

$$(0.1 \cdot \texttt{x} - 3.3)^2, \tag{18}$$

whereas for `gaussrw` in Fig. 13 we learn

$$(0 \cdot \texttt{x} - 1 \cdot \texttt{y} + 11)^2 + (0 \cdot \texttt{x} + 0 \cdot \texttt{y} + 8)^2. \tag{19}$$

We translate the program in Fig. 14 for Amber by replacing the update for $\lambda$ by instead sampling it uniformly from $[1, 10]$. Amber correctly identifies the program is AST, and that $(10 - \texttt{x})$ is a supermartingale expression (note, not an RSM), though does not report that the program is PAST (answering "maybe").

### 5.4 Undefined Moments

The ability to evaluate the cumulative distribution function (CDF) of a sampled distribution could be useful in marginalisation, even if the moments of the sampled distribution are undefined or not known analytically to infinite precision. An example is Fig. 15: the program samples from the standard Cauchy distribution, for which all moments are undefined. Since the sampled value is *only*

used to determine which branch of a conditional is taken, the RSM expectation is well defined, and can be expressed in terms of the standard Cauchy CDF. Namely, the if-branch is taken with probability $q = 1 - \left(\frac{1}{\pi}\arctan(10) + \frac{1}{2}\right)$. This equation is not expressible using polynomials; so we perform a sound approximation by introducing a new variable that is quantified over a small interval surrounding a finite precision approximation to $q$. This allows us to learn and verify the SOR-form NRSM:

$$\mathrm{ReLU}(1.2 \cdot \mathtt{x} + 9.1). \tag{20}$$

```
1  while (x > 0) do
2      p ∼ StandardCauchy();
3      if (p > 10) then
4          x = x + 2
5      else
6          x = x - 1
7      fi
8  od
```

**Fig. 15.** Sliced Cauchy distribution (`slicedcauchy`).

For our experimental evaluation (Sect. 6) we create a modified version of each of the six case studies described in this section, as follows:

- program `marbles3` is a generalisation of `marbles` to three marble types, instead of two;
- `probfact2` uses 5/8 as the Bernoulli parameter, rather than 3/4;
- `crwalk2` samples `rho` from a Beta(1, 3) distribution, instead of a uniform distribution over $\left[-\frac{1}{2}, 1\right]$;
- `expdistrw2` samples from an exponential distribution, where parameter `lambda` is replaced by `lambda*lambda`;
- `gaussrw2` uses $[3 + 1/(1 - \mathtt{x}), 3 + 1/(1 - \mathtt{y})]^T$ for its mean vector, instead of $[\sqrt{1 + \mathtt{x}^2}, \sqrt{1 + \mathtt{y}^2}]^T$; and
- `slicedcauchy2` has a loop guard of $\mathtt{x} < 10$, instead of $\mathtt{x} > 0$, and swaps the two branches of the conditional.

### 5.5 Rare Transitions

A limitation of relying on a sampled transition dataset to learn NRSM parameters is we rely on the average $\mathbb{E}_{p' \sim P'}[\eta(p')]$ in Eq. (5) being accurate (see Sect. 3). This assumption is challenged by programs that have certain control-flow paths of very low probability, which are unlikely to be sampled by the interpreter. For example, in the context of the ambitious marble collector (Fig. 1), Fig. 16 shows that when the probability of obtaining a red marble decreases below $2^{-7}$, our success rate drops. This is because a lower probability makes the corresponding control-flow path rarer in the dataset, to the point where the expected value of the NRSM cannot be estimated accurately.

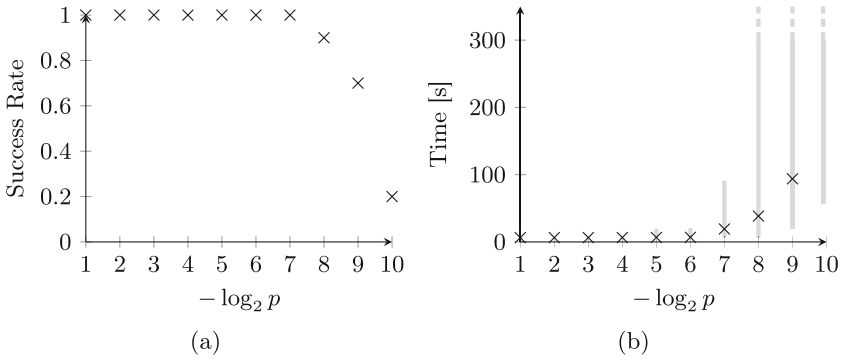


**Fig. 16.** Success rate and execution times for the ambitious marble collector program (Fig. 1), where $p$ is the probability of taking the if-branch. Success rate refers to the fraction of 10 executions that succeeded in finding an NRSM before a timeout of 300 s. Execution times show the median time with the error bar ranging between the minimum and maximum times of the 10 executions.

## 6 Experimental Results

We built a prototype implementation of our framework (in Python) and present experimental results for benchmarks adapted from previous work, as well as our own case studies (from Sect. 5). The case studies illustrate programs for which our framework synthesises an RSM, yet existing tools cannot prove to be PAST.

The learner is implemented with Jax [9]. To train NRSMs, we use AdaGrad [18] for gradient-based optimisation, with a learning rate of $10^{-2}$. Parameters are initialised by sampling from Gaussian distributions: weight parameters are sampled from a zero-mean Gaussian, whereas the bias parameters are sampled either from a Gaussian with mean 10 (for SOR candidates) or mean 0 (for SOS candidates). We verify the NRSMs using the SMT solver Z3 [26,40]. The outcomes are obtained on the following platform: macOS Catalina version 10.15.4, 8 GB RAM, Intel Core i5 CPU 2.4 GHz QuadCore, 64-bit.

As mentioned in Sect. 4, the verifier checks a candidate NRSM over states satisfying the loop predicate, which characterises the set of reachable states. For our experiments, we manually provide the NRSM expectation, and augment the guard predicate with additional invariants where necessary. We generate outcomes using two different rounding strategies (Sect. 4): an "aggressive" rounding strategy which generated between 80 and 120 candidates per CEGIS iteration, and a "weaker" rounding strategy producing between 15 to 25 candidates per CEGIS iteration. The outcomes in Table 1 used the aggressive rounding strategy.

**Table 1.** Experimental results over existing (top section) and newly added benchmarks (bottom section); (c) indicates the benchmark uses continuous distributions, (d) indicates it only uses discrete distributions. All reported times are in seconds, oot indicates time-out after 300 s, n/a indicates the tool terminated without definite answer, and—indicates the benchmark is unsupported. Our method is run 10 times with different seeds; the overall success rate is reported. Runtimes of interpretation, training, verification phases, and # of CEGIS iterations refer to the run with median total runtime.

| Program | Amber [39] | Farkas' lemma [2] | Absynth [41] | Succ. rate | Inter. | Train. | Verif. | #iter | NRSM |
|---|---|---|---|---|---|---|---|---|---|
| `Hare & Tortoise` (d) | 0.04 | ≈0 | 0.09 | 10/10 | 0.61 | 3.86 | 0.70 | 0 | SOR |
| `exmini/terminate` (d) | — | 0.02 | oot | 10/10 | 1.75 | 29.35 | 7.67 | 2 | SOR |
| `aaron2` (d) | 0.03 | 0.02 | 0.02 | 10/10 | 0.04 | 2.27 | 0.01 | 0 | SOR |
| `catmouse` (c) | 0.03 | 0.02 | — | 9/10 | 0.39 | 12.41 | 3.68 | 1 | SOS |
| `counterex1c` (d) | — | 0.02 | 0.22 | 8/10 | 1.00 | 6.71 | 0.02 | 0 | SOR |
| `easy1` (d) | 0.12 | 0.01 | 0.05 | 10/10 | 1.12 | 5.55 | 1.27 | 0 | SOR |
| `easy2` (c) | 0.04 | 0.02 | — | 10/10 | 1.55 | 6.79 | 0.18 | 0 | SOS |
| `ndecr` (d) | 0.04 | 0.02 | 0.03 | 10/10 | 1.18 | 5.63 | 0.02 | 0 | SOR |
| `random1d` (c) | 0.05 | 0.02 | — | 10/10 | 1.14 | 4.86 | 0.79 | 0 | SOS |
| `rsd` (d) | error | 0.01 | oot | 10/10 | 1.14 | 6.18 | 2.04 | 0 | SOR |
| `speedFails1` (d) | 0.07 | 0.01 | 0.04 | 10/10 | 0.45 | 4.09 | 0.67 | 0 | SOR |
| `speedPldi2` (d) | — | 0.02 | 0.40 | 9/10 | 1.36 | 7.85 | 0.02 | 0 | SOR |
| `speedPldi3` (d) | — | 0.02 | 0.36 | 8/10 | 2.58 | 30.70 | 2.12 | 1 | SOR |
| `speedPldi4` (d) | — | 0.02 | 0.17 | 10/10 | 0.68 | 5.07 | 0.04 | 0 | SOR |
| `speedSingleSingle` (c) | 0.03 | 0.02 | — | 10/10 | 0.39 | 2.85 | 0.51 | 0 | SOS |
| `speedSingleSingle2` (d) | — | 0.02 | 0.15 | 10/10 | 0.83 | 7.30 | 0.04 | 0 | SOR |
| `wcet0` (d) | — | 0.02 | 0.10 | 10/10 | 1.45 | 5.64 | 0.09 | 0 | SOR |
| `wcet1` (d) | — | 0.02 | 0.10 | 10/10 | 0.85 | 4.31 | 0.09 | 0 | SOR |
| `probfact` (d) | — | — | n/a | 10/10 | 0.49 | 6.12 | 0.16 | 0 | SOR |
| `probfact2` (d) | — | — | n/a | 10/10 | 0.45 | 5.89 | 0.23 | 0 | SOR |
| `marbles` (d) | — | — | n/a | 10/10 | 0.84 | 10.83 | 0.91 | 0 | SOR |
| `marbles3` (d) | — | — | n/a | 10/10 | 0.40 | 70.14 | 7.87 | 2 | SOR |
| `crwalk` (c) | — | — | — | 10/10 | 0.53 | 3.06 | 1.56 | 1 | SOS |
| `crwalk2` (c) | — | — | — | 10/10 | 1.32 | 3.11 | 0.75 | 1 | SOS |
| `expdistrw` (c) | n/a | — | — | 10/10 | 0.05 | 1.53 | 0.01 | 0 | SOS |
| `expdistrw2` (c) | n/a | — | — | 10/10 | 4.92 | 3.15 | 1.03 | 1 | SOS |
| `gaussrw` (c) | — | — | — | 10/10 | 10.30 | 3.45 | 0.75 | 0 | SOS |
| `gaussrw2` (c) | — | — | — | 9/10 | 15.46 | 4.91 | 5.33 | 0 | SOS |
| `slicedcauchy` (c) | — | — | — | 10/10 | 0.02 | 3.31 | 0.01 | 0 | SOR |
| `slicedcauchy2` (c) | — | — | — | 10/10 | 0.01 | 2.16 | 0.03 | 0 | SOR |

*Benchmarks from Previous Work.* We run our prototype on single-loop programs from the WTC benchmark suite [3], augmented with probabilistic branching and assignments [2]. These correspond to the programs in the first section of Table 1. We perturb assignment statements by adding noise sampled from a discrete uniform distribution of support $\{-2, 2\}$, or a continuous uniform distribution on

the interval $[-2, 2]$. The *while* loops are also made probabilistic; with probability $1/2$ the loop is executed, and with the remaining probability a `skip` command is executed.

We compare our framework against three existing tools. The first is Amber [39]: where possible, we translate instances from the WTC suite into the language of Amber, but this is not possible for some programs where the loop predicate is a logical conjunction or disjunction of predicates (indicated by dashes in Table 1). Second, we compare against a tool for synthesising affine lexicographic RSMs (referred to as Farkas' lemma) for affine programs (i.e. containing only linear expressions), based on reduction to linear programming via Farkas' lemma [2]. This is applicable to probabilistic programs with nested-loops, unlike our method. However, since it is limited to affine programs and affine lexicographic RSMs, it is not able to analyse all the programs we consider (again, indicated by dashes in Table 1). The third tool is Absynth [41], for which we are able to encode all programs that were limited to discrete random variables.

The experimental results (Table 1) show that for all the WTC benchmarks our approach has a success rate of at least $8/10$, and is able to synthesise an RSM within 2 iterations (for the seed that results in median total execution time). For 15 of the 18 WTC benchmarks no full CEGIS iterations are required. As expected our approach, particularly the learning component, is much slower than all three tools. However, our framework has broader applicability, as illustrated with the next set of experiments.

*Newly Defined Case Studies.* The examples in the second section of Table 1 (from Sect. 5) are not proven PAST by any of the three tools. Our approach is able to do so with a success rate of at least $9/10$, under the "aggressive" rounding strategy. Of the new examples, `marbles3` (Sect. 5) requires the longest time, since we use an NRSM with $h = 3$ ReLU nodes (see Sect. 3), and six of the nine parameters must be brought sufficiently close to zero to learn a valid RSM. For `gaussrw`/`gaussrw2`, we find it necessary to set an SMT solver time limit within the CEGIS loop (of 200 ms for `gaussrw`, and 5 s for `gaussrw2`), such that candidates taking longer than this to verify are skipped. The fact that these examples are harder to verify is unsurprising, given that they give rise to non-polynomial decision problems, containing equationally defined rational expressions. In comparing the two rounding strategies, we find that using the "aggressive" strategy tends to result in fewer CEGIS iterations, reducing the learner time, while increasing the verifier time: this is to be expected, since a larger number of candidates needs to be checked in each CEGIS iteration.

## 7 Conclusion

We have presented the first machine learning method for the termination analysis of probabilistic programs. We have introduced a loss function for training neural networks so that they behave as RSMs over sampled execution traces; our training phase is agnostic to the program and thus easily portable to different

programming languages. Reasoning about the program code is entirely delegated to our checking phase which, by SMT solving over a symbolic encoding of program and neural network, verifies whether the neural network is a sound RSM. Upon a positive answer, we have formally certified that the program is PAST; upon a negative answer, we obtain a counterexample that we use to resample traces and repeat training in a CEGIS loop. Our procedure runs indefinitely for programs that are not PAST, as these necessarily lack a ranking supermartingale, and may run indefinitely for some PAST programs. Nevertheless, we have experimentally demonstrated over several PAST benchmarks that our method is effective in practice and covers a broad range of programs w.r.t. existing tools.

Our method naturally generalises to deeper networks, but whether these are necessary in practice remains an open question; notably, neural networks with one hidden layer were sufficient to solve our examples. We have exclusively tackled the PAST question, and techniques for almost-sure (but not necessarily PAST) termination and non-termination exist [16,37,39]. Our results pose the basis for future research in machine learning (and CEGIS) for the formal verification of probabilistic programs. Different verification questions will require different learning models. Our approach lends itself to extensions toward probabilistic safety, exploiting supermartingale inequalities, and towards the non-termination question, using repulsing supermartingales [16]. Adapting our method to termination analysis with infinite expected time is also a matter for future investigation [37]. Moreover, we have exclusively considered purely probabilistic single-loop programs: generalisations to programs with non-determinism, arbitrary control-flow, and concurrency are material for future work [15,20,35].

**Acknowledgments.** This work was in part supported by a partnership between Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS) and Innovate UK under project HICLASS (113213), by the Engineering and Physical Sciences Research Council (EPSRC) Doctoral Training Partnership, by the Department of Computer Science Scholarship, University of Oxford, and by the DeepMind Computer Science Scholarship.

## References

1. Abate, A., Ahmed, D., Giacobbe, M., Peruffo, A.: Formal synthesis of Lyapunov neural networks. IEEE Control. Syst. Lett. **5**(3), 773–778 (2021)
2. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2(POPL)**, 34:1–34:32 (2018)
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) Static Analysis, pp. 117–133. Springer, Berlin, Heidelberg (2010)
4. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–8. IEEE (2013)
5. Avanzini, M., Dal Lago, U., Yamada, A.: On probabilistic term rewriting. Sci. Comput. Program. **185**, 102338 (2020)

6. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. Proc. ACM Program. Lang. **4(OOPSLA)**, 172:1–172:30 (2020)
7. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: How long, O Bayesian network, will I sample thee? In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 186–213. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_7
8. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
9. Bradbury, J., et al.: JAX: composable transformations of Python+NumPy programs (2018). http://github.com/google/jax
10. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
11. Chakarov, A., Voronin, Y.-L., Sankaranarayanan, S.: Deductive proofs of almost sure persistence and recurrence properties. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 260–279. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_15
12. Chang, Y., Roohi, N., Gao, S.: Neural Lyapunov control. In: NeurIPS, pp. 3240–3249 (2019)
13. Chattenjee, K., Fu, H., Novotný, P.: Termination analysis of probabilistic programs with martingales. In: Barthe, G., Katoen, J.P., Silva, A. (eds.) Foundations of Probabilistic Programming, p. 221–258. Cambridge University Press (2020)
14. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
15. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. ACM Trans. Program. Lang. Syst. **40**(2), 7:1–7:45 (2018)
16. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL, pp. 145–160. ACM (2017)
17. Dahlqvist, F., Silva, A.: Semantics of probabilistic programming: a gentle introduction. In: Barthe, G., Katoen, J.P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 1–42. Cambridge University Press (2020)
18. Duchi, J.C., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. In: COLT, pp. 257–269. Omnipress (2010)
19. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: POPL, pp. 489–501. ACM (2015)
20. Fu, H., Chatterjee, K.: Termination of nondeterministic probabilistic programs. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 468–490. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_22
21. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. CoRR abs/2102.03824 (2021)
22. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT Press (2016)
23. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
24. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53

25. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proc. ACM Program. Lang. **3(OOPSLA)**, 129:1–129:29 (2019)
26. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
27. Kaminski, B.L., Katoen, J.-P., Matheja, C.: On the hardness of analyzing probabilistic programs. Acta Informatica **56**(3), 255–285 (2018). https://doi.org/10.1007/s00236-018-0321-1
28. Kapinski, J., Deshmukh, J.V., Sankaranarayanan, S., Aréchiga, N.: Simulation-guided Lyapunov analysis for hybrid dynamical systems. In: HSCC, pp. 133–142. ACM (2014)
29. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
30. Kura, S., Unno, H., Hasuo, I.: Decision tree learning in CEGIS-based termination analysis. In: CAV (2021)
31. Le, T.C., Antonopoulos, T., Fathololumi, P., Koskinen, E., Nguyen, T.: DynamiTe: Dynamic termination and non-termination proofs. Proc. ACM Program. Lang. **4(OOPSLA)**, 189:1–189:30 (2020)
32. Lee, W., Wang, B.-Y., Yi, K.: Termination analysis with algorithmic learning. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 88–104. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_12
33. Lee, W., Yu, H., Rival, X., Yang, H.: Towards verified stochastic variational inference for probabilistic programs. Proc. ACM Program. Lang. **4(POPL)**, 16:1–16:33 (2020)
34. Li, Y., Ying, M.: Algorithmic analysis of termination problems for quantum programs. Proc. ACM Program. Lang. **2(POPL)**, 35:1–35:29 (2018)
35. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 112–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_7
36. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, Berlin (2005)
37. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2(POPL)**, 33:1–33:28 (2018)
38. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
39. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP 2021. LNCS, vol. 12648, pp. 491–518. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_18
40. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: PLDI, pp. 496–512. ACM (2018)
42. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE, pp. 246–256. ACM (2013)
43. Richards, S.M., Berkenkamp, F., Krause, A.: The Lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems. In: CoRL. Proceedings of Machine Learning Research, vol. 87, pp. 466–476. PMLR (2018)

44. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
45. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 54–70. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_4