Counterexample- and Simulation-Guided Floating-Point Loop Invariant Synthesis

Anastasiia Izycheva[1], Eva Darulova[2], and Helmut Seidl[1]

[1] Fakultät für Informatik, TU München, izycheva@in.tum.de,seidl@in.tum.de
[2] MPI-SWS, eva@mpi-sws.org

Abstract. We present an automated procedure for synthesizing sound inductive invariants for floating-point numerical loops. Our procedure generates invariants of the form of a convex polynomial inequality that tightly bounds the values of loop variables. Such invariants are a prerequisite for reasoning about the safety and roundoff errors of floating-point programs. Unlike previous approaches that rely on policy iteration, linear algebra or semi-definite programming, we propose a heuristic procedure based on simulation and counterexample-guided refinement. We observe that this combination is remarkably effective and general and can handle both linear and nonlinear loop bodies, nondeterministic values as well as conditional statements. Our evaluation shows that our approach can efficiently synthesize loop invariants for existing benchmarks from literature, but that it is also able to find invariants for nonlinear loops that today's tools cannot handle.

Keywords: Invariant Synthesis, Floating-Point Arithmetic, CEGIS, Simulation

1 Introduction

Finding and proving inductive loop invariants is one of the fundamental tasks in program verification, allowing to prove a property for all program executions even in the presence of unbounded loops. *Proving* (or disproving) that a given loop invariant is inductive is generally an easier task and can in many cases be straight-forwardly automated using off-the-shelf SMT solvers [31,6]. *Finding* a loop invariant, however, is in general difficult to do manually, and automating this process remains an important challenge [7,14,16,19].

In this paper, we focus on numerical loops over floating-point variables, which are found across domains such as embedded control systems and scientific computing. Reasoning about floating-point arithmetic is additionally complex due to its unintuitive nature: every arithmetic operation introduces a roundoff error w.r.t. the ideal real-valued execution and overflow or invalid operations introduce the special values $\pm$infinity and Not-a-Number. For floating-point computations, it is thus of utmost importance to bound the values of variables as accurately as possible. This information is required for proving safety, absence of floating-point special values, and bounds of roundoff errors [13].

Some previous techniques for loop invariant synthesis for numerical programs require a target property to be given [40,38,29,19,43]; in most cases this is a set of unsafe states that should be proven to be unreachable. However, for floating-point loops where the goal is to compute as tight invariants as possible, specifying unsafe states essentially amounts to finding the invariant itself.

Abstract interpretation-based techniques [9] do not require a target property. Nonetheless, existing efficient linear abstract domains that rely on widening are often not strong enough to find non-trivial inductive invariants, i.e. where the bounds are not $\pm$infinity. As our evaluation shows, even conjunctions of linear inequalities as provided by convex polyhedra [10,4] are often insufficient.

We thus require nonlinear loop invariants expressed as polynomial inequalities to handle many numerical loops. However, existing techniques each have limitations, as they require templates to be given by the user [1]; are limited to linear loops only [36]; do not always produce invariants that satisfy the precondition [33]; or require a target range in order to produce tight invariants [29].

Here, we propose a rather pragmatic approach. We use concrete executions and polynomial approximation in order to obtain candidate invariants which, when combined with counterexample-guided refinement, allows us to synthesize invariants for floating-point loops. Our approach does not require a target bound, efficiently produces tight polynomial inequality invariants, handles linear as well as nonlinear loops, and soundly takes into account floating-point roundoff errors. Our algorithm thus generates exactly the invariants we are looking for. This generality, naturally, comes at a certain cost. While previous approaches provide certain completeness guarantees [36,33,29] at the expense of the above listed limitations, our algorithm effectively trades completeness for a wider applicability. Nevertheless, we empirically observe that our proposed algorithm, despite being based on a heuristic search, is remarkably effective.

Our algorithm performs a form of iterative counterexample-guided invariant generation: it proposes a candidate invariant, checks it using an off-the-shelf SMT solver, and if the solver returns a counterexample, uses it to adjust the next candidate invariant. We cannot query the SMT solver for the polynomial coefficients directly, as such a query would require quantifiers, which neither the real-valued nor the floating-point SMT theories support well. Instead, candidate invariants are generated based on simulation, i.e. concrete executions of the loop, and polynomial approximation which guesses the shape of the invariant based on the convex hull of seen program values. Concrete executions (instead of abstractions) starting from a given precondition allow our algorithm to accurately capture the behavior of linear as well as nonlinear loop bodies, and thus to generate tight invariants. Our algorithm abstracts the floating-point semantics of the loop body by a sound roundoff error bound, adds it as nondeterministic noise, and then uses a real-valued decision procedure to verify the proposed candidate invariants. This approach is more efficient than using the currently limited floating-point decision procedures, but at the same time accurate, as the error bound is computed based on a concrete candidate invariant, and thus adds only as much noise as is necessary.

```
x ∈ [0.0, 0.1]
y ∈ [0.0, 0.1]

while (true) {
  x := x + 0.01 * (-2*x - 3*y + x*x)
  y := y + 0.01 * (x + y)
}
```

(a) Example benchmark

$$-0.03x - 0.1y + 0.44x^2 + xy + 0.86y^2 \leq 0.02$$
$$x \in [-0.5, 0.3]$$
$$y \in [-0.2, 0.4]$$

(b) Generated invariant

Fig. 1: Running example

Our approach is motivated by the fact that the numerical invariants we are looking for are robust to some noise [36], and the loops of interest do not have a single inductive invariant, but they admit *many similar* invariants. The robustness to noise is important; few developers program with the exact floating-point semantics in mind, rather they treat it as a noisy version of real arithmetic. If a loop was not robust to noise, even small changes in roundoff errors e.g. due to non-associativity, would lead to large changes in the overall loop behavior. This robustness allows us to use a heuristic search to find *one of these* invariants.

We implement our algorithm as a Python library in a tool called Pine. Pine can fully automatically handle non-nested loops with linear and nonlinear assignments, nondeterministic noise and conditional statements. In this paper, we focus on convex invariants that consist of a single polynomial inequality of degree two, and note that an extension to more complex bounded invariant shapes (non-convex shapes, disjunctive invariants, higher degrees) requires mainly engineering work to find a suitable way to fit the polynomial(s) from the convex hull. We evaluate Pine on a number of existing benchmarks from literature and show that it computes invariants that are on average 12.4x smaller than those computed by existing tools. Furthermore, we show that Pine computes invariants for nonlinear loops, which are out of reach for state-of-the-art tools.

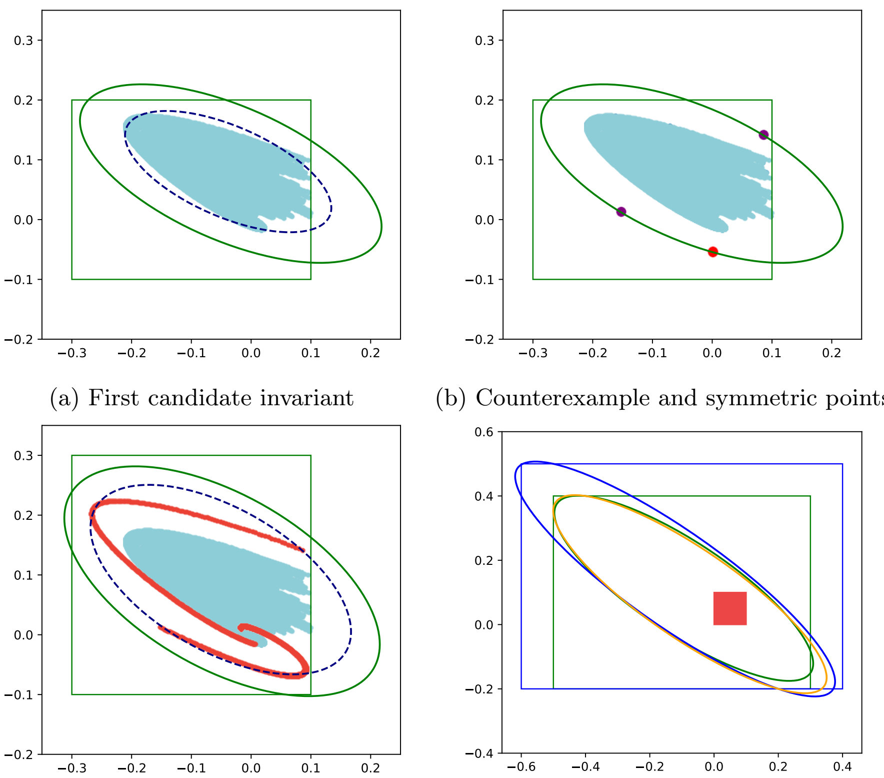
Contributions To summarize, our paper makes the following contributions:

- a novel algorithm for polynomial inequality invariant synthesis for linear and nonlinear floating-point loops,
- an open-source prototype implementation Pine[3], and
- a detailed evaluation on existing and new benchmarks.

2 Overview

Before explaining our invariant synthesis algorithm in detail, we illustrate it at a high-level on an example. Figure 1a shows our example loop that simulates a dynamical system together with the precondition on the loop variables.

[3] https://github.com/izycheva/pine



(a) First candidate invariant

(b) Counterexample and symmetric points

(c) Second candidate invariant

(d) Multiple possible invariants

Fig. 2: Nonlinear benchmark candidate invariants

PINE starts by simulating the loop to collect a set of concrete points that an inductive invariant definitely has to include. For this, PINE samples $m = 100$ random values from the input ranges $x \in [0, 0.1]$ and $y \in [0, 0.1]$, and executes the loop $n = 1000$ times for each point. Sampled points are shown in light blue in Figure 2a-2c. Since we are looking for a convex invariant, PINE next computes the convex hull of the sampled points. This reduces the number of points to consider and gives us an initial estimate of the shape of the invariant.

We consider invariants that include variable ranges and a shape enclosing all values expressed as an ellipsoid, i.e. a second degree polynomial inequality. We obtain the polynomial coefficients by computing the minimum volume ellipsoid enclosing the convex hull, and the variable bounds from the minimum and maximum values seen in the sampled points:

$$-0.0009x - 0.004y + 0.0103x^2 + 0.021xy + 0.0298y^2 \leq 5.4 \cdot 10^{-5} \wedge$$

$$x \in [-0.2098, 0.0976], y \in [-0.0159, 0.1723]$$

The computed ellipsoid is depicted in Figure 2a by blue dashed ellipse. We observe that this candidate invariant is noisy. To remove (a part of) this noise, we scale and round the (normalized) polynomial coefficients and the range bounds (the latter is rounded outwards). Obtained ellipsoid and ranges form the first candidate invariant (marked green in Figure 2a):

$$-0.03x - 0.13y + 0.35x^2 + 0.7xy + y^2 \leq 0.01 \wedge x \in [-0.3, 0.1], y \in [-0.1, 0.2]$$

Pine uses an off-the-shelf SMT solver (Z3) to check whether this candidate invariant is inductive. For our candidate invariant the check fails and the solver returns a counterexample $C_1 : (x = 0.0, y = -0.0542)$ (red dot in Figure 2b). By counterexample, we mean a point that itself satisfies the candidate invariant, but after one loop iteration results in a point for which the invariant no longer holds. In our example C_1 satisfies the candidate invariant, but after one iteration we obtain $C_1' : (x = 0.001626, y = -0.054742)$ that violates the invariant.

Pine uses this counterexample to refine the candidate invariant. However, instead of recomputing the convex hull and ellipsoid shape immediately, we generate additional counterexamples in order to not bias the shape in a single direction that is (randomly) determined by the solver's counterexample. In particular, Pine computes counterexamples that are symmetric to C_1 along the symmetry axes of the ellipsoid and satisfy the candidate invariant. Figure 2b shows the counterexamples generated for our running example (purple dots).

Pine then uses another round of simulation, starting from the set of counterexamples, to obtain a new set of points that need to be included in an invariant (by transitivity, if a counterexample point is included after one loop iteration, then the points after additional iterations also have to be included). The new set of points is then used to generate the next candidate invariant. Figure 2c shows simulated points in red, and the new candidate invariant in green. Note that Figure 2c contains three simulation traces - one for each counterexample, and the traces originated from the bottom left counterexample and C_1 coincide.

Pine repeats this iterative process until either an invariant is found, or a maximum number of refinement iterations is reached. For our example, Pine finds an inductive invariant (shown green in Figure 2d) after 6 iterations.

The invariant so found holds for a real-valued loop, i.e. when the loop body is evaluated under real arithmetic. The last step of Pine's algorithm is to verify that the invariant also holds under a floating-point loop semantics. To do this, Pine uses an off-the-shelf analysis tool to get the worst-case roundoff error bound for each expression in the loop body. The errors are then added as nondeterministic noise terms to the loop, and the invariant is re-checked by the SMT solver. For our running example, this check succeeds, and the following invariant is confirmed:

$$-0.03x - 0.1y + 0.44x^2 + xy + 0.86y^2 \leq 0.02 \ \wedge x \in [-0.5, 0.3], y \in [-0.2, 0.4]$$

Figure 2d shows several invariants generated by Pine for our example, for different parameters of its algorithm. Note that these invariants are similar, but differ slightly in shape and volume. The range component of the invariant is shown by the green and blue boxes; the red box denotes the input ranges.

3 Problem Definition

The input to our algorithm is a loop body together with a precondition. We consider simple non-nested loops given by the following grammar:

```
L ::= while(true){ B }
B ::= if (G) S else S | S
S ::= $\epsilon$ | $x_i$ := p($x_1$, ..., $x_n$) + $u_j$; S
G ::= * | p $\leq$ 0
```

In each iteration, the loop updates a set of variables $\mathsf{x}_i \in \mathcal{X}$. The right-hand-side of each assignment consists of polynomial expressions p in the loop variables together with an (optional) nondeterministic noise term u_j, which is bounded in magnitude and that denotes any additional noise, e.g. input error from sensor values. The loop body can include a top-level conditional statement, which can also be used to express the loop exit condition. The conditions of the if-statement can either be nondeterministic choice or a polynomial inequality. We note that adding support for more complex conditions as well as nested and chained if-statements would only affect the way we parse the loop and encode it in the SMT query and is not a fundamental limitation of our algorithm.

The precondition specifies the initial ranges for all variables x_i, as well as bounds on the nondeterministic noise variables: $\mathsf{x}_i \in [a_i, b_i]$, $\mathsf{u}_j \in [c_j, d_j]$. The loop and noise variables take values in the set $\mathbb{F}$ of floating-point values. Then the semantics of a loop body b is given by $[\![\mathsf{b}]\!] :: (\mathcal{X} \to \mathbb{F}) \to 2^{(\mathcal{X}\to\mathbb{F})}$, which is defined by

$$
\begin{aligned}
[\![\epsilon]\!]\,\rho &= \{\rho\} \\
[\![\mathsf{x}_i := p + \mathsf{u}_j; s]\!]\,\rho &= \bigcup\{[\![s]\!](\rho \oplus \{\mathsf{x}_i \mapsto p(\rho) + u\}) \mid u \in [c_j, d_j]\} \\
[\![\mathtt{if}\,(*)\, s_1\,\mathbf{else}\, s_2]\!]\,\rho &= [\![s_1]\!]\,\rho \cup [\![s_2]\!]\,\rho \\
[\![\mathtt{if}\,(p \leq 0)\, s_1\,\mathbf{else}\, s_2]\!]\,\rho &= \{\rho_1 \in [\![s_1]\!]\,\rho \mid p(\rho) \leq 0\} \cup \\
&\quad\ \{\rho_2 \in [\![s_2]\!]\,\rho \mid p(\rho) > 0\}
\end{aligned}
$$

Here, $p(\rho)$ denotes the value of the polynomial p for the variable assignment ρ under the floating-point arithmetic semantics specified by the IEEE 754 standard [22]. The set of initial program states is given by

$$\mathsf{Init} = \{\rho : \mathcal{X} \to \mathbb{R} \mid \forall x_i \in \mathcal{X}.\, \rho(x_i) \in [a_i, b_i]\}$$

Our goal is to find an inductive invariant $\mathcal{I}$ such that

$$\mathsf{Init} \subseteq \mathcal{I} \quad \wedge \quad \forall \rho \in \mathcal{I}.\, [\![b]\!]\,\rho \subseteq \mathcal{I} \tag{1}$$

i.e., $\mathcal{I}$ subsumes the initial states and is preserved by each iteration of the loop. We consider convex invariants given by a polynomial inequality together with ranges for variables:

$$\mathcal{I} = \{\rho \mid \mathcal{P}(\rho) \leq 0, \rho(\mathsf{x}_i) \in \mathcal{R}_i = [l_i, h_i]\}$$

The goal is thus to find the coefficients of the polynomial $\mathcal{P}$ and the lower and upper bounds (l_i, h_i) for the variables of the loop. In this paper, we consider polynomials $\mathcal{P}$ of degree two, although our algorithm generalizes to higher degrees. We observe that second degree polynomials are already sufficient for a large class of loops.

Additionally, we are interested in finding as small an invariant as possible, where we measure size by the volume enclosed by an invariant. We note that the ellipsoid (the polynomial inequality), is not only needed to prove the inductiveness of many invariants, but it can also enable more accurate verification based on our inductive invariants, for instance by techniques relying on SMT solving. For this reason, we do not only measure the volume as the size of the box described by $\mathcal{R}$, but rather as the intersection between the box and the ellipsoid shape, which can be substantially smaller.

4 Algorithm

Figure 3 shows a high-level view of our invariant synthesis algorithm. The input to the algorithm is a loop together with a precondition on the loop variables, and the output is a polynomial $\mathcal{P}$ and a set of ranges $\mathcal{R}$, a range $\mathcal{R}_i$ for each program variable x_i, that define the synthesized invariant:

$$\mathcal{P}(x_1, ..., x_n) \leq 0 \land x_1 \in \mathcal{R}_1 \land \ldots \land x_n \in \mathcal{R}_n \tag{2}$$

The key component of our algorithm is the invariant synthesis, which infers the shape of the bounding polynomial and the variable ranges (lines 1-21). The algorithm first synthesizes an invariant assuming a real-valued semantics for the loop body (`withRoundoff == False`).

The synthesis starts by simulating the loop on a number of random inputs from the precondition, keeping track of all the seen points, i.e. tuples $(x_1, ..., x_n)$. From the obtained points, the algorithm next guesses the shape of a candidate invariant, i.e. a polynomial $\mathcal{P}$ and a set of ranges $\mathcal{R}$ (line 5-7). We check this candidate invariant using an off-the-shelf SMT solver (line 12). If the candidate is not an invariant or is not inductive, the solver returns a counterexample. The algorithm generalizes from the counterexample (line 16-20) and uses the newly obtained points to refine the candidate invariant. We repeat the process until either an invariant is found, or we reach a maximum number of iterations (empirically, all benchmarks required less than 100 iterations).

After the real-valued invariant is generated, the algorithm checks whether it also holds for the floating-point implementation of the loop (line 29). Should this not be the case, invariant synthesis is repeated taking floating-point roundoff errors into account in every refinement iteration. Since roundoff errors are usually relatively small, this recomputation is seldom necessary, so that Pine first runs real-valued invariant synthesis for performance reasons.

```
def get_real_invariant(loop, init, withRoundoff):
  pts = simulate(loop, random.sample(init, $m$), $n$)
  // update pts iteratively
  for i in range(0, max_iters):
    pts = convexHull(pts)
    ranges = round(min(pts), max(pts), $prec_{range}$)
    coefficients = getShape(pts, $prec_{poly}$)
    inv = (coefficients, ranges)

    if withRoundoff:
      loop = addRoundoff(loop, ranges)
    cex = checkInvariant(loop, inv)
    if cex is None:
      return inv
    else:
      addCex = getAdditionalCex(loop, inv, cex, $cex\_num$, $d$)
      symPts = getSymmetricPts(cex, inv)
      nearbyPts = getNearbyPts(cex, $d$, inv)
      pts = pts ∪ cex ∪ addCex ∪ symPts ∪ nearbyPts
      pts = simulate(loop, pts, $k$)
  return None

def get_fp_invariant(loop, init):
  inv = get_real_invariant(loop, init, withRoundoff=False)
  if inv is None:
    return None
  else:
    loopFP = addRoundoff(loop, inv.ranges)
    cex = checkInvariant(loopFP, inv)
    if cex is None:
      return inv
    else:
      return get_real_invariant(loop, init, withRoundoff=True)
```

Fig. 3: High-level invariant synthesis algorithm (parameters are in cursive)

4.1 Simulation

The synthesis starts by simulating the loop execution. For this, Pine samples m values from the variables' input ranges Init uniformly at random, and concretely executes the loop n times for every sample. As a result, we obtain $m \times n$ points, i.e. combinations of variable values, that appear in the concrete semantics of the loop and thus have to be included in an invariant. The sampled points provide a starting point for the invariant search.

4.2 Candidate Invariant Conjecture

The invariant we are looking for has two parts: variable ranges $\mathcal{R}$ and a polynomial shape $\mathcal{P}(x)$ enclosing all variable values. To obtain $\mathcal{R}$ and $\mathcal{P}(x)$, Pine first reduces the number of samples by computing the convex hull of the sampled points. We consider invariant shapes that are convex, therefore the values inside the shape can be safely discarded. Extending our algorithm to non-convex shapes is a matter of finding an appropriate way to reduce the number of samples.

The minimum and maximum values of each loop variable x_i in the convex hull vertices determine the range $\mathcal{R}_i$.

Pine infers the shape $\mathcal{P}(x)$ enclosing the convex hull vertices using two optimization methods: minimum volume enclosing ellipsoid (MVEE), and least squares curve fitting. The minimum volume enclosing ellipsoid method computes a bounding ellipsoid such that all points are inside the shape. Pine utilizes a library that computes MVEE by solving the following optimization problem:

$$\begin{gathered} \text{minimize} \log(\det(E)) \\ s.t. (x_i - c)^T E \ (x_i - c) \leq 1 \end{gathered}$$

where x_i are the individual points, c is a vector containing the center of the ellipsoid and E contains the information about the ellipsoid shape [30].

While MVEE computes the desired shape, the library that we use supports only two dimensions, and it is furthermore possible that it diverges. To support higher-dimensional loops, or when MVEE fails, we resort to using least squares. With the method of least squares, we find coefficients such that the sum of the squares of the errors w.r.t. to the given points is minimized. For a degree 2 polynomial in variables x and y, Pine transforms the points into the matrix A with entries having the values of $[1, x, y, x^2, xy]$, and a vector b which consists of the values of y^2. By solving the system of equations $Az = b$ for z, we obtain the coefficients of the polynomial. By setting $b = y^2$, we set the last coefficient to 1 in order to avoid the trivial (zero) solution. Least squares computes a tight fit, but will, in general, not include all of the points *inside* the polynomial shape, so that we additionally have to enlarge the 'radius' such that it includes all points. While we do not explore this further in this work, we note that the above sketched least-squares approach also generalizes to fit polynomials of higher degree than 2, using suitable constraints to ensure convex shapes [27].

In this paper, we only consider convex shapes described by a single polynomial inequality (and ranges). However, with a suitable fitting method it is possible to include more complex shapes. For instance, for disjunctive invariants one can first perform clustering, and then fit the polynomials using MVEE or the least squares method.

4.3 Reducing the Noise

Both methods used to infer a shape are approximate, i.e. they find a polynomial that is close to the actual shape up to a tolerance bound. Furthermore, they fit a set of points that is incomplete in that it only captures a (random) subset of all of the possible concrete executions. This makes the inferred polynomial shapes inherently noisy and unlikely to be an invariant. We reduce the noise by first normalizing and then rounding the polynomial coefficients to a predefined precision $prec_{poly}$, i.e. to a given (relatively small) number of digits after the decimal point. This effectively discards coefficients (rounds to zero) whose magnitude is significantly smaller than the largest coefficient found. For the remaining coefficients, it removes the—likely noisy—least significant digits.

Similarly, the lower and upper bounds of the computed ranges $\mathcal{R}$ capture only the values seen in simulation and are thus likely to be under-approximating the true ranges. We round the lower and upper bounds outwards to a predefined precision $prec_{range}$, thus including additional values.

The precisions (number of decimal digits) chosen for rounding the polynomial coefficients and the ranges should be high enough to not lead to too large over-approximations, but nonetheless small enough to discard most of the noise. We have empirically observed that the polynomial coefficients should be more precise than the range bounds by one digit, and that $prec_{poly} = 2$ and $prec_{range} = 1$ seems to be a good default choice.

4.4 Checking a Candidate Invariant

The obtained polynomial and variables ranges form a candidate invariant, which we check for inductiveness using an off-the-shelf SMT solver by encoding the (standard) constraint $(\mathsf{Init} \rightarrow I(x)) \wedge (I(x) \wedge L \rightarrow I(x'))$, where $I(x) = \mathcal{P}(x) \leq 0 \bigwedge_i x_i \in \mathcal{R}_i$, L is the loop body relating the variables x before the execution of the loop body to the variables x' after.

We translate conditional statements using the SMT command `ite`. Non-deterministic terms receive fresh values from the user-defined range at every loop iteration. Since the ranges do not change we add constraints on the ranges of non-deterministic terms only to I and Init. We encode the above constraint in SMT-LIB using the real-valued theory [24]. The SMT solver evaluates the query and returns a counterexample if it exists. If no counterexample is returned, a candidate invariant is confirmed to be inductive and returned.

4.5 Generalizing from Counterexamples

The counterexample returned by the SMT solver is added to the existing set of points that the invariant has to cover. However, this additional point is arbitrary, depending on the internal heuristics of the solver. In order to speed up invariant synthesis, and to avoid biasing the search in a single direction and thus skewing the invariant shape, we generate additional points that also have to be covered by the next invariant candidate. We consider three different generalizations: additional counterexamples, symmetric points and nearby points.

PINE obtains additional counterexamples from the solver by extending the SMT query such that the initial counterexample is blocked and the new counterexample has to be a minimum distance d away from it. PINE will iteratively generate up to *cex_num* additional counterexamples, as long as the solver returns them within a (small) timeout (*cex_num* is a parameter of the algorithm).

Our second generalization strategy leverages the fact that the candidate invariant is an ellipsoid and thus has several axes of symmetry. Pine computes points that are symmetric to the counterexample with respect to all axes of symmetry of the ellipsoid, and adds them as additional points if they satisfy I or Init (i.e. they are also valid counterexamples).

Nearby points are the points that are at a distance d to the counterexample. Pine computes these points in all directions, i.e. $x_i \pm d$, and adds them to the set of points, if they are valid counterexamples. The rationale behind this generalization is that points in the vicinity of a counterexample are often also likely counterexamples. Adding the nearby points allows us to explore an entire area, instead of just a single point.

Pine then performs a second simulation of the loop starting from the newly added set of counterexamples for k iterations. All obtained points are added to the original sampled values and we proceed to synthesize the next candidate invariant.

4.6 Floating-Point Invariant

We encode the SMT queries to check the inductiveness of our candidate invariants using the real-valued theory. We note that it is in principle possible to encode the queries using the floating-point theory, and thus to encode the semantics of the loop body, including roundoff errors, exactly. However, despite the recent advances in floating-point decision procedures [8], we have observed that their performance is still prohibitively slow for our purpose (CVC4's state-of-the-art floating-point procedure [8] was several orders of magnitude slower than Z3's real-valued procedure [24]).

We thus use a real-valued SMT encoding and soundly over-approximate the roundoff errors in the loop body. We compute a worst-case roundoff error bound rnd for each expression in the loop body using an off-the-shelf roundoff analysis tool. Static analyses for bounding roundoff errors [12,42] assume the following abstraction of floating-point arithmetic operations: $(x \circ_{fl} y) = (x \circ y)(1 + e) + d \quad |e| \leq \epsilon, |d| \leq \eta$, where $\circ \in \{+, -, *, /\}$ and $\circ_{fl}$ is the floating-point counter-part. The so-called machine epsilon ϵ bounds the relative error for arithmetic operations on normal numbers and η bounds the absolute error on the so-called subnormal numbers (very small numbers close to zero that have a special representation). The static analyses use interval abstract domains to bound the ranges of all intermediate arithmetic expressions, and from those compute the new roundoff errors committed by each arithmetic operation, as well as their propagation through the rest of the program. These techniques compute roundoff error bounds for loop-free code, which is sufficient for our purpose, since we only need to verify that $I(x) \wedge L \rightarrow I(x')$, i.e. the executions of the (loop-free) loop body remain within the bounds given by I.

The computed roundoff error bound is added to the expression as a nondeterministic noise term bounded by $[-rnd, rnd]$. Note that unlike in existing work [36] that derives one general error bound for all programs assuming a large enough number of arithmetic operations, our roundoff error is computed on-demand for each particular candidate invariant. The magnitude of roundoff errors depends on ranges of inputs, and so by computing the roundoff error only for the invariant's ranges, we are able to add only as little noise as is necessary.

Our algorithm first finds a real-valued invariant and then verifies whether it also holds under a floating-point loop semantics. If not, we restart the invariant synthesis and take roundoff errors into account for each candidate invariant, recomputing a new tight roundoff error in each iteration of our algorithm (line 11). We do not include roundoff errors in the first run of the synthesis for better performance, since in practice, we rarely need to recompute the invariant.

Except for the roundoff error analysis, our algorithm is agnostic to the finite precision used for the implementation of the loop. By choosing to compute roundoff errors w.r.t. different precisions, it thus supports in particular both single and double floating-point precision, but also fixed-point arithmetic of different bit lengths [12], which is particular relevant for embedded platforms that do not have a floating-point unit.

4.7 Implementation

We have implemented the algorithm from Figure 3 in the tool PINE as a Python library in roughly 1600 lines of code, relying on the following main libraries and tools: the Qhull library for computing the convex hull[4], a library for computing the minimum volume ellipsoid[5], the least-squares function from scipy (`scipy.linalg.lstsq`), the Python API for the Z3 SMT solver version 4.8.7, and the Daisy tool [12] for computing roundoff errors. Simulations of the loop are performed in 64-bit floating-point arithmetic.

5 Experimental Evaluation

We evaluate PINE on a set of benchmarks from scientific computing and control theory domains. We aim to answer the following research questions:

RQ1: How does PINE compare with state-of-the-art tools?
RQ2: How quickly does PINE generate invariants?
RQ3: How sensitive is PINE's algorithm to parameter changes?

5.1 State-of-the-Art Techniques

We compare the invariants synthesized by PINE to those generated by two state-of-the-art tools: Pilat [33] and SMT-AI [36]. These two tools are the only ones that compute polynomial inequality invariants for floating-point loops without requiring a target condition to be given.

[4] `www.qhull.org`
[5] `https://github.com/minillinim/ellipsoid`

Pilat reduces the generation of invariants of a loop body f to computing the eigenvector ϕ of f that is associated to the eigenvalue 1, i.e. $f(\phi) = \phi$ and ϕ is thus an invariant. Pilat can, in principle, handle nonlinear loops by introducing a new variable for each nonlinear term and thus effectively linearizing it. This transformation is similar to how we use least-squares to fit a polynomial (Section 4.2). Pilat handles floating-point roundoff errors by (manually) including nondeterministic noise for each floating-point operation that captures the roundoff error: $(x \circ y) \cdot \delta$, where $\circ \in \{+, -, \times, /\}$ and $|\delta| \leq \epsilon$ is bounded by the machine epsilon. For simplicity, we ignore errors due to subnormal numbers.

SMT-AI [36] and Adje et al. [1] implement policy iteration using the ellipsoid abstract domain. The approach by Adje et al. requires the ellipsoid template to be provided, while SMT-AI generates templates automatically. For our comparison we therefore consider the more general approach of SMT-AI. SMT-AI generates the ellipsoid templates from Lyapunov functions [3], which are functions known from control theory for proving that equilibrium points of dynamical systems are stable. These functions prove that a loop is bounded and thus the shape effectively serves as an invariant. It is known that for linear loops one can generate the polynomial shapes automatically using semi-definite programming. Since such an automated method does not exist for nonlinear functions, SMT-AI is limited to linear loops. Semi-definite programming can compute different polynomial shapes, and SMT-AI selects shapes to be tight using a binary search. SMT-AI first computes a real-valued invariant, like Pine, and then verifies that it also satisfies a floating-point loop. Unlike Pine, SMT-AI derives one generic roundoff error bound for all (reasonably-sized) loops, and does not recompute the invariant if the floating-point verification fails. We were unfortunately not able to install SMT-AI, so that we perform our comparison on the benchmarks used by SMT-AI, comparing to the (detailed) results reported in the paper [35,36].

Interproc [18] is a static analyzer based on abstract interpretation. It infers numerical invariants using boxes, octagons, linear congruences and convex polyhedra. A user can choose between two libraries that implement these domains: APRON [23] and Parma Polyhedra Library [4]. We tried Interproc on our set of benchmarks, and on 2 benchmarks it produced some bounds for a subset of the program variables. However, the invariants were not convex, and we could not compute their volume. We therefore exclude Interproc from the comparison.

Another potential competitor is an approach by Mine et al.[29] that combines interval and octagon abstract domains with constraint solving. The invariants discovered are effectively ellipsoids, i.e. second-degree polynomial inequalities. However, their approach fundamentally requires target bounds. Since the goal of Pine is to find such tight bounds, and not only prove that they are inductive, we do not compare with Mine et al.[29].

5.2 Experimental Setup

Our set of benchmarks contains both linear and nonlinear loops. Each benchmark consists of a loop body which iterates an infinite number of times. The linear benchmarks *filter_ goubault*, *filter_ mine**, *arrow_ hurwicz*, *harmonic*, *symplectic* are taken from related work [1,29] and implement linear filters and oscillators. Benchmarks *ex** are taken from the evaluation of SMT-AI [35] and comprise linear controllers, found for instance in embedded systems.

We additionally include the nonlinear benchmark *pendulum**, that simulates a simple pendulum and *rotation**, which repeatedly rotates a 2D vector by an (small) angle that is nondeterministically picked in each iteration. Both benchmarks use the sine function, which we approximate using a Taylor approximation. The *nonlin_ example** are nonlinear dynamical systems collected from textbook examples on Lyapunov functions.

Three of our benchmarks contain operations on nondeterministic noise terms. Most benchmarks are 2-dimensional, except for *ex4**, which has 3 variables, *ex2** and *ex5**, which have 4 variables, and *ex6** that has 5 variables.

We run our evaluation on a MacBook Pro with an 3.1 GHz Intel Core i5 CPU, 16 GB RAM, and macOS Catalina 10.15.3.

5.3 Comparison with State-of-the-Art

Each tool generates an invariant with an elliptic shape, and Pine and SMT-AI provide additionally ranges for variables. We compare the inductive invariants generated by each tool based on their volume. The volume of an invariant is given by the set of points satisfying $\mathcal{P}(x) \leq 0 \wedge \bigwedge_i x_i \in \mathcal{R}_i$, where the variable ranges may intersect with the ellipsoid. We compute this intersection (approximately) using a Monte-Carlo simulation with $3 \cdot 10^6$ samples, by comparing how many samples are within the invariant to how many are inside the variable ranges (for the latter we know the volume exactly). Our volume estimates are accurate to two decimal digits.

We run Pine with a default set of parameters, that we determined empirically (see Section 5.5). In order to compare with other tools that only support single floating-point precision, Pine computes roundoff errors (and invariants) for 32-bit floating-point precision.

Columns 2-4 in Table 1 show the volumes of the invariants generated by SMT-AI, Pilat, and Pine. '-' denotes the cases where a tool did not generate an invariant. Benchmarks for which we did not have data for SMT-AI are marked as 'undef'. 'PF' denotes cases where an invariant was generated, but it did not satisfy the given precondition. 'TO' marks cases when a tool took longer than 20 minutes to generate an invariant. Here, smaller volume is better, the best volumes are marked bold.

Due to the inherent randomness in its algorithm, we run Pine 4 times and compute the average volume and running time across the runs. The last column shows variations in volume with respect to the average (i.e. (max - min)/average).

We observe that Pine produces the tightest invariants on 17/24 (70%) of the linear benchmarks. Additionally, Pine generates invariants for all nonlinear benchmarks in our set, whereas Pilat was not able to generate invariants for any of them. Pine produces invariants that are in the best case on average 20x tighter than the ones by SMT-AI, and 2.7x tighter than the ones by Pilat (compared on the 6 benchmarks, for which it was able to generate an invariant). In the

	Benchmark	SMT-AI	Pilat	PINE	PINE avg time, s	Volume variation
Non-linear	pendulum-approx	undef	-	**12.92**	21.09	30.03%
	rot.nondet-small	undef	-	**5.97**	30.13	16.25%
	rot.nondet-large	undef	-	**6.67**	33.78	10.87%
	nonlin-ex1	undef	-	**0.23**	14.43	18.51%
	nonlin-ex2	undef	-	**0.56**	7.32	5.23%
	nonlin-ex3	undef	-	**7.07**	12.45	3.35%
Linear	arrow-hurwitz	undef	-	**4.40**	4.75	7.00%
	harmonic	undef	18.41	**3.52**	10.81	9.70%
	symplectic	undef	PF	**2.32**	7.71	12.11%
	filter-goubault	undef	PF	**1.84**	4.94	1.31%
	filter-mine1	undef	PF	**6.32**	7.18	1.58%
	filter-mine2	undef	1.16	**0.49**	4.48	71.92%
	filter-mine2-nondet	undef	4.92	**4.45**	10.70	66.38%
	pendulum-small	undef	12.53	**9.10**	7.11	7.51%
	ex1- filter	**475.06**	498.37	-	43.61	-
	ex1-reset-filter	**475.98**	-	-	45.95	-
	ex2-2order	17.37	**1.07**	4.92	7.45	46.73%
	ex2-reset-2order	17.36	-	**3.08**	6.28	6.65%
	ex3-leadlag	-	-	-	46.68	-
	ex3-reset-leadlag	-	-	-	44.56	-
	ex4-gaussian	0.61	-	**0.22**	16.93	46.16%
	ex4-reset-gaussian	17.05	-	**1.45**	23.10	137.47%
	ex5-coupled-mass	5,538.47	TO	**100.61**	8.63	9.48%
	ex5-reset-coupled-mass	5,538.34	-	**81.02**	8.44	27.54%
	ex6-butterworth	65.25	-	**25.43**	16.34	272.89%
	ex6-reset-butterworth	700.06	-	**10.30**	219.34	0.00%
	ex7-dampened	**12.17**	-	18.68	19.96	15.71%
	ex7-reset-dampened	**12.17**	-	-	39.70	-
	ex8-harmonic	5.75	-	**2.32**	6.99	9.77%
	ex8-reset-harmonic	5.75	-	**2.85**	7.15	28.08%
	ex5+6	**6,927.12**	TO	TO	TO	-

Table 1: Volumes of invariants generated by PINE, Pilat and SMT-AI, PINE's average running time and variation in invariant volumes across 4 runs

worst case (observed over our 4 runs), the factors decrease to 13.8x and 1.8x respectively. Only for the benchmarks *ex6-butterworth* and *filter-mine2-nondet*, the worst-case volumes computed by PINE become 1.9x and 1.6x larger than the ones computed by SMT-AI and Pilat, respectively, and are thus still of the same order of magnitude.

5.4 Efficiency

PINE generates invariants in on average 25, and at most 220 seconds; the largest running time is also the benchmark with the largest number of variables. PINE

m	n	cex_num	d	k	symPts	nearbyPts	volume
100	1000	0	0.5	500	✓		2.283
100	1000	0	0.5	100		✓	2.297
100	10000	5	0.25	100	✓		2.311
100	1000	2	0.25	100		✓	2.314
100	10000	1	0.5	500		✓	2.335

Table 2: Top-5 minimum volume configurations

was able to confirm the real-valued invariant also for the floating-point semantics for all but two *rotation** benchmarks, for which it had to recompute the invariants two out of four times. We consider the running times to be acceptably low such that it is feasible to re-run PINE several times for an input loop, in order to obtain a smaller invariant, if needed.

5.5 Parameter Sensitivity

We now evaluate the influence of different parameter settings on the performance of our proposed algorithm in terms of its ability to find tight inductive invariants. For this, we explored the parameter space of our algorithm on 13 of our benchmarks that include (non-)linear infinite loops without branching. We evaluate the different combinations of varying the following parameters:

- whether or not symmetric points are used
- whether or not nearby points are used
- number of random inputs and loop iterations for initial simulation (algorithm parameter m-n): 100-1k, 1k-1k, 100-10k
- number of loop iterations for counterexamples simulation (k): 0, 100, 500
- number of additional counterexamples (cex_num): 0, 1, 2, 5 (when $cex\_num =$ 0, no additional counterexample is generated)
- distance to nearby points (in % of the range) (d): 10%, 25%, 50%
- three different precisions for rounding: ($prec_{poly} = 1, prec_{range} = 0$), ($prec_{poly} = 2, prec_{range} = 1$), ($prec_{poly} = 3, prec_{range} = 2$), where $prec_{poly}, prec_{range}$ give the number of decimal digits for the polynomial coefficients and the variable ranges, respectively.

In total, we obtain 1296 configurations. We run PINE with each of them once.

Default Configuration 185 parameter configurations were successful on all of the 13 benchmarks. From these, we select the configuration that generates invariants with the smallest average volume across the benchmarks as our default configuration: $prec_{poly} = 2, prec_{range} = 1, m = 100, n = 1000, k = 500$. To generalize from counterexamples the default configuration uses only symmetric points.

Table 2 shows the 5 best configurations, according to average volume (we normalized the volume across benchmarks). We note that the differences between volumes for successful configurations are small, so that we could have chosen any of these configurations as the default.

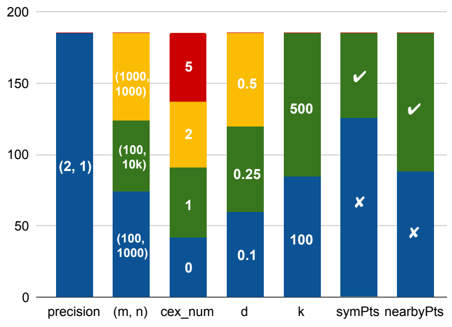


Fig. 4: Proportion of parameters appearing in successful configurations

Benchmark	min	avg	max
pendulum-approx	9.88	12.03	20.21
rot.nondet-small	4.76	5.78	8.07
rot.nondet-large	6.27	6.92	11.43
nonlin-ex1	0.06	0.20	0.83
nonlin-ex2	0.55	0.56	0.59
nonlin-ex3	6.84	7.04	7.27
harmonic	3.28	3.84	4.46
symplectic	2.12	2.21	2.68
filter-goubault	1.82	1.83	1.85
filter-mine1	6.29	6.40	8.72
filter-mine2	0.28	1.03	3.83
filter-mine2-nondet	1.74	2.44	11.37
pendulum-small	8.61	10.14	24.55

Fig. 5: Volumes of invariants with successful configurations

Successful Configurations We study the 185 successful configurations to see which parameter values appear the most frequently, and thus seem most successful in finding invariants. Figure 4 shows the distribution of the different parameters in the set of successful configurations. For instance, the precisions $(prec_{poly} = 1, prec_{range} = 0)$ and $(prec_{poly} = 3, prec_{range} = 2)$ do not appear at all in the successful configurations, i.e. only $(prec_{poly} = 2, prec_{range} = 1)$ was able to find invariants for all benchmarks. On the other hand, nearby points are included in the generalization in roughly half of the configurations.

From Figure 4, we conclude that simulating the loop starting from counterexamples (line 21 in Figure 3) is crucial in finding an invariant - none of the configurations without this additional simulation worked on all benchmarks. On the other hand, whether this simulation runs 100 or 500 loop iterations seems to make less of a difference.

For the remaining parameters, we do not observe a strong significance; they are roughly equally distributed among the successful configurations. From this we conclude that our algorithm is not sensitive to particular parameter settings, and will find invariants successfully for many different parameter configurations.

The choice of parameters does, however, influence the size of the invariants generated, at least for certain benchmarks. Figure 5 shows the minimum, maximum and average volumes for each benchmark across successful configurations. While for some benchmarks, the variation is small, for others the best configuration produces invariants that are half the size from the worst one.

Across the 1296 configurations, we observe that if a real-valued invariant is found, it is also confirmed in 89% of cases, and thus has to be re-computed in only 11% of cases. The only outlier that needs recomputation more often is *rotation-nondet-large*, which rotates a vector by a larger angle, and therefore is understandably more sensitive to enlarging the coordinates with some noise.

Last but not least, we used Pine's default configuration to generate invariants for fixed-point precision with uniform 16 bit length for all our 30 benchmarks (including *ex**). The smaller bit length results in larger roundoff errors, so that Pine had to recompute an invariant for 5 additional benchmarks (i.e. where the real-valued invariant was not confirmed), but was able to find an inductive invariant for as many loops as with floating-point implementation.

6 Related Work

Many tools and libraries [23] infer invariants over program variables using abstract interpretation. The abstract domains range from efficient and imprecise intervals [11], over octagons [28], to more expensive and expressive polyhedra [41,4]. For programs with elliptic invariants most linear abstract domains are insufficient to express an invariant [36].

Ellipsoid domains have been defined that work for specific types of programs, e.g. digital filters [17] and programs where variables grow linearly with respect to the enclosing loop counters [34]. Performing abstract interpretation using policy iterations instead of widening allows the use of the ellipsoid abstract domain more generally [1,20]. This approach requires templates of the ellipsoids to be given, however. Recent works [36,33] are able to discover ellipsoid inductive loop invariants without the need for templates, but being based on semidefinite programming and linear algebra, respectively, are fundamentally limited to linear loops only. Alternatively, Bagnara et.al. [5] have explored an abstract domain that approximates polynomial inequalities by convex polyhedra and leverages the operations, including widening, of polyhedra. Sankaranarayanan et.al. [37] show how to generate polynomial equality invariants by reducing the problem to a constraint satisfaction problem.

Our algorithm builds on several ideas that have been explored in loop invariant synthesis previously, including the use of concrete executions to derive polynomial templates and counter-example based refinement. Floating-point loops and in particular the uncertainties introduced due to roundoff errors pose unique challenges that existing techniques cannot handle, as we discuss next.

Several works have explored the use of machine-learning in teacher-learner frameworks [19,43]: the learner guesses a candidate invariant from a set of examples, and the teacher checks whether the invariant is inductive. If it is not, the teacher provides feedback to the learner in form of additional (counter)examples. These approaches rely on a target property to be given (to provide negative examples) and are thus not immediately applicable to synthesizing floating-point inequality invariants. The framework C2I [38] employs a learner-teacher framework, but where the learner uses a randomized search to generate candidate invariants. While surprisingly effective, the approach is, however, limited to a a fixed search, e.g. linear inequalities with a finite set of given constants as coefficients. Sharma et.al. [40] present a learning based algorithm to generate invariants that are arbitrary boolean combinations of polynomial inequalities, but require a set of good and bad states and thus an assertion to be given.

The tool InvGen [21] generates integer linear invariants from linear templates, using concrete program executions to derive constraints on the template parameters. The tool NumInv [32] and the Guess-And-Check algorithm [39] generate polynomial equality invariants using a similar approach. For integer programs and in particular equality constraints, this approach is exact. In our setting with floating-point programs and inequalities such constraints cannot be solved exactly and thus require a different, approximate, approach. NumInv and Guess-And-Check furthermore employ counterexamples returned by the solver for refinement of the invariant. These counterexamples are program inputs, however, due to the complexity of the floating-point or real-arithmetic decision procedures, this technique does not scale to our target numerical programs. We are thus restricted to counterexamples to the invariant property.

Abductive inference in the tool Hola [14] and enumerative synthesis in FreqHorn [15] are two further techniques that have been used to generate invariants for numerical programs, but are unfortunately not applicable to generate the invariants we are looking for. Hola relies on quantifier elimination which solvers do not support (well) for floating-points and reals; FreqHorn generates the invariant grammar from the program's source code, but for our invariants the terms do not appear in the program itself.

Allamigeon et. al. [2] extend ellipsoidal analyses to generate disjunctive and non-convex invariants for switched linear systems. We do not consider disjunctive invariants in this work and leave their exploration to future work.

Recurrence-based techniques [25,26] generate loop invariants that exactly capture the behavior of a numerical integer loop. While these techniques work for arbitrary conditional branches, imperative code and nested loops, they generate invariants of a different form, i.e. in general not polynomial inequalities and are thus orthogonal to our approach.

7 Conclusion

We propose a novel algorithm for synthesizing polynomial inequality invariants for floating-point loops. For this, we show how to extend the well-know technique of counterexample-guided invariant synthesis to handle the uncertainties arising from finite-precision arithmetic. The key insight to make our iterative refinement work is that a single counterexample is not sufficient and the algorithm has to explore the space of counterexamples more evenly in order to successfully generalize. While the resulting algorithm is heuristic in nature, it proved to be remarkably effective on existing benchmarks as well as on handling benchmarks out of reach of existing tools.

Acknowledgements We would like to thank Sebastian Bruggisser for helping to debug our ellipsoids.

References

1. Adjé, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. Logical Methods in Computer Science **8**(1) (2012)

2. Allamigeon, X., Gaubert, S., Goubault, E., Putot, S., Stott, N.: A fast method to compute disjunctive quadratic invariants of numerical programs. ACM Trans. Embedded Comput. Syst. **16**(5s), 166:1–166:19 (2017)
3. Astrom, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press (2008)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1), 3 – 21 (2008)
5. Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of basic semi-algebraic invariants using convex polyhedra. In: Static Analysis (2005)
6. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification (CAV) (2011)
7. Bradley, A.R.: Sat-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011)
8. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2019)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth ACM Symposium on Principles of Programming Languages (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '78 (1978)
11. Cousot, P., Radhia, C.: Static determination of dynamic properties of programs. In: ISOP (1976)
12. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018)
13. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Trans. Program. Lang. Syst. **39**(2), 8:1–8:28 (2017)
14. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Object Oriented Programming Systems Languages & Applications (OOPSLA) (2013)
15. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018)
16. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: FMCAD (Formal Methods in Computer Aided Design) (2017)
17. Feret, J.: Static analysis of digital filters. In: Programming Languages and Systems (ESOP) (2004)
18. Gal Lalire, M. Argoud, B.J.: A web interface to the interproc analyzer, `http://pop-art.inrialpes.fr/interproc/interprocwebf.cgi`
19. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Computer Aided Verification (CAV) (2014)
20. Gawlitza, T.M., Seidl, H.: Numerical invariants through convex relaxation and max-strategy iteration. Formal Methods Syst. Des. **44**(2), 101–148 (2014)
21. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: Computer Aided Verification (CAV) (2009)
22. IEEE, C.S.: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)

23. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification (CAV) (2009)
24. Jovanovic, D., de Moura, L.M.: Solving non-linear arithmetic. In: Automated Reasoning - 6th International Joint Conference, IJCAR (2012)
25. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-linear reasoning for invariant synthesis. Proc. ACM Program. Lang. **2**(POPL), 54:1–54:33 (2018)
26. Kovács, L.: Reasoning algebraically about p-solvable loops. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008)
27. Magnani, A., Lall, S., Boyd, S.: Tractable fitting with convex polynomials via sum-of-squares. In: Proceedings of the 44th IEEE Conference on Decision and Control (2005)
28. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)
29. Miné, A., Breck, J., Reps, T.W.: An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In: Programming Languages and Systems (ESOP) (2016)
30. Moshtagh, N.: Minimum Volume Enclosing Ellipsoid (2020 (retrieved May 21, 2020)), `https://www.mathworks.com/matlabcentral/fileexchange/9542-minimum-volume-enclosing-ellipsoid`
31. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008)
32. Nguyen, T., Antonopoulos, T., Ruef, A., Hicks, M.: Counterexample-guided approach to finding numerical invariants. In: Foundations of Software Engineering (ESEC/FSE) (2017)
33. de Oliveira, S., Bensalem, S., Prevosto, V.: Synthesizing invariants by solving solvable loops. In: Automated Technology for Verification and Analysis (ATVA) (2017)
34. Oulamara, M., Venet, A.J.: Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. In: Computer Aided Verification (CAV) (2015)
35. Roux, P., Garoche, P.: Integrating policy iterations in abstract interpreters. In: Automated Technology for Verification and Analysis (ATVA) (2013)
36. Roux, P., Garoche, P.: Practical policy iterations - A practical use of policy iterations for static analysis: the quadratic case. Formal Methods Syst. Des. **46**(2), 163–196 (2015)
37. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: Principles of Programming Languages. POPL (2004)
38. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Computer Aided Verification (CAV) (2014)
39. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Programming Languages and Systems (ESOP) (2013)
40. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Static Analysis Symposium (SAS) (2013)
41. Singh, G., Püschel, M., Vechev, M.: A practical construction for decomposing numerical abstract domains. Proc. ACM Program. Lang. **2**(POPL) (2017)
42. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: Formal Methods (FM) (2015)
43. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: Programming Language Design and Implementation (PLDI) (2018)