Synthesizing invariants by solving solvable loops[⋆]

Steven de Oliveira[1], Saddek Bensalem[2], Virgile Prevosto[1]

1 : CEA, List 2 : Université Grenoble Alpes

Abstract. Formal program verification faces two problems. The first problem is related to the necessity of having automated solvers that are powerful enough to decide whether a formula holds for a set of proof obligations as large as possible, whereas the second manifests in the need of finding sufficiently strong invariants to obtain correct proof obligations. This paper focuses on the second problem and describes a new method for the automatic generation of loop invariants that handles polynomial and non deterministic assignments. This technique is based on the eigenvector generation for a given linear transformation and on the polynomial optimization problem, which we implemented on top of the open-source tool Pilat.

1 Introduction

Program verification relies on different mathematical foundations to let users prove that a piece of code behaves as intended. The problem is however undecidable for any Turing complete language, partly because of loops. This is one of the reasons why loop analysis is a highly studied topic in the field of verification. Let us take for example linear filters, whose purpose are to apply a linear constraint to input signals. Such programs are heavily used in embedded software for analyzing sensors' data and are thus critical for the correction of the system. Yet, linear filters are difficult to verify because of the non-determinism induced by the unknown input signal and the use of floating-point computations. This lack of precision forbids the direct use of exact mathematical techniques.

Figure 1 presents an example of program inspired by linear filters [21]. We claim that loop invariants are a good way to obtain general information about such a loop. In this particular case, the loop admits the invariant $x^2 + y^2 \leqslant 14.9$, bounding the maximal value of $|x|$ and $|y|$ to 3.9: this is an infinite loop. More generally, if we can infer bounds for the value of the loop variables or for polynomial expressions of these variables, we are then able to perform precise analyses, such as reachability. In this paper, we aim at facing two major problems of numeric invariant generation, namely the generation of polynomial relations between variables and the search of inductive spaces to which variables of a program belong, in the context of simple (i.e. non-nested) loops composed of

⋆ This work has been partially conducted within the VESSEDIA project, that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

polynomial and non deterministic assignments. In particular, linear filters are encompassed in such context. The relations we generate have the advantage to be completely independent from the initial state of the loop, making them fully generic, as opposed to full-program based techniques that start from a specific initial state. This work is an extension of the algorithm PILA [11], which generates polynomial equalities between variables manipulated by a simple deterministic loop. We show in this paper that a refined version of this algorithm can also produce inductive inequality invariants and tackle non-deterministic assignments as well as deterministic ones. Moreover, we add to this analysis an optimization algorithm enabling us to minimize the inductive set described by invariants of non deterministic loops.

```
x = non_det(-1,1);
y = non_det(-1,1);
while(x < 4) do
 N = non_det(-0.1,0.1);
 (x,y) = (0.68 * (x-y) + N,\
    0.68 * (x+y) + N);
done
```

Fig. 1: Example of linear filter

Contributions. The original PILA approach generates inductive invariants as equality relations of the form $P(X) = 0$ with P a polynomial. This paper extends this method (Section 2) to generate new kinds of inductive invariants of the form $|P(X)| \leqslant k$ and $|P(X)| \geqslant k$. It is mostly based on linear algebra and is applicable to C programs manipulating integers and floating point numbers. To simplify the presentation, we describe the method on a simple imperative language (Section 3). The two main results of this extension are the treatment of loops with deterministic (Section 4.1) and non-deterministic assignments (Section 4.2). Finally, we explain how to manage imprecision in floating point computations (Section 4.3). In the latter cases, we reduce the problem of generating invariants to the polynomial optimization problem. An algorithm for solving this problem is given. The proposed method in this paper is correct, fully implemented in Pilat and is currently part of the Frama-C suite [17] as an external open-source plug-in, available at [3]. We show its efficiency by applying it on several examples from related literature in section 6. Due to space constraints, proofs have been omitted. They are available in a separate report [12].

2 Overview

When synthesizing invariants, three ingredients are required :

1. what kind of invariants are computed;
2. what will be their most useful shape;
3. how strong they will be.

In abstract interpretation for example, we first choose the type of invariant that will be computed, i.e. the abstract domain, then a symbolic execution of properties of this domain will shape the initial state into an invariant that we will try to keep as strong as possible by applying appropriate widening and narrowing operators.

Overview of the PILA algorithm. Let us first recall how PILA works on a simple example. Consider the loop of figure 2 for which we want to generate all invariants (polynomials P such that $P(x, y) = 0$) of degree 2. By enhancing the loop expressiveness with new variables representing the value of the monomials of variables used in the loop, namely x_2 for x^2, y_2 for y^2 and xy for $x * y$, it first creates linear variables representing monomials.

Let us take for instance x_2. As the new value of x is $0.68.(x - y)$, the new value of x^2 is $0.68^2.(x^2 - 2.x.y + y^2)$. x_2 can then be expressed as a linear application of x_2, xy and y_2. More generally, any monomial of variables of the loop in figure 2 evolves linearly along the execution of the enhanced loop.

```
x = non_det(-1,1);
y = non_det(-1,1);
while(*) do
  (x,y) = (0.68 * (x-y),\
    0.68 * (x+y));
done
```

Fig. 2: Simple affine loop

Next, PILA starts generating invariants. Instead of starting with an initial state, which is not assumed to be known, it generates relations that are preserved by each step of the loop. Let f be the loop transformation, (here $f(x, y) = (0.68*(x-y), 0.68*(x+y))$. A linear application φ is a semi-invariant if, given any valuation of the variables, it stays constant through one iteration of f. In other words, it must respect the following property:

$$\text{If } \varphi(X) = 0 \text{ then } \varphi(f(X)) = 0$$

In linear algebra, this is strictly equivalent to the following:

$$\text{If } \varphi(X) = 0 \text{ then } f^*(\varphi)(X) = 0$$

where $f^*(\varphi) = \varphi \circ f$ is the dual application of f. If there exists a scalar λ such that $f^*(\varphi) = \lambda.\varphi$ (i.e. φ an eigenvector of f^* associated to the eigenvalue λ) the criterion becomes obviously true, thus φ is a semi-invariant.

In fact, it is shown in [11] that eigenvectors of f^* are exactly the set of such invariants bound to the transformation f. More precisely, when an eigenvector φ is associated to the eigenvalue 1 (i.e. $f^*(\varphi) = \varphi$), it represents an affine invariant of f ($\varphi.X = k$). When the associated eigenvalue is not 1, the PILA algorithm is not always capable of lifting the semi-invariant into a proper invariant. In the example of figure 2, the associated eigenvalue of the only semi-invariant $x^2 + y^2$ is 0.9248. PILAT concludes that $x^2 + y^2 = 0$ is inductive but if it does not respect the initial state, this is not an invariant.

The key idea of this paper is to consider not only equalities, but also inequalities. If the left eigenvector φ is associated to an eigenvalue λ such that

$0 < \lambda \leqslant 1$ then $\lambda.\varphi(X)$ will necessarily be smaller than $\varphi(X)$. Thus for any $k \geqslant 0$, the following proposition holds:

$$\text{If } \varphi(X) \leqslant k \text{ then } f^*(\varphi)(X) \leqslant k$$

$\varphi(X) \leqslant k$ is thus inductive. In our example, the relation $x^2+y^2 \leqslant k$ is inductive, and contrarily to $x^2+y^2 = 0$ it can be made an invariant even if the initial values of x and y are not 0: we just have to choose $k = x_{init}^2 + y_{init}^2$.

Non determinism. The same reasoning can be applied to treat non deterministic values in assignments. By setting the non deterministic values to a random value, e.g. 0, we are left to find inductive inequality relations, which can be easily performed as we just saw. In the deterministic case, generated formulas are inductive because the set of possible values for x and y that respects the formula gets bigger by applying the loop transformation once. Adding the non deterministic noise may lead to non inductive formulas. A solution consists in finding upper and lower bounds for this noise and check if the set obtained in deterministic case stays stable under this new transformation. If this is not the case, we must consider a weaker invariant.

3 Setting

Mathematical background. We work in the real field $\mathbb{R}$. Let $(\mathbb{R}^n, \|.\|)$ the normed vector space of dimension n associated to the usual euclidean norm $\|.\|$. Elements of $\mathbb{R}^n$ are denoted $x = (x_1, ..., x_n)^t$ a column vector. The variables vector of a mapping f is denoted X. $\mathcal{M}_n(\mathbb{R})$ is the set of matrices of size $n * n$ and $\mathbb{R}[X]$ is the set of polynomials with coefficients in $\mathbb{R}$. The complex field $\mathbb{C}$ is the algebraic closure of $\mathbb{R}$. Let $|.|$ be the euclidian norm on $\mathbb{C}$. We use $\langle .,. \rangle$ the linear algebra standard notation, $\langle u, v \rangle = u^t.v$, with . the usual dot product (i.e. the sum of the product of each component of u and v). For a linear mapping $f(X) = A.X$, we define its *dual* $f^*(X) = A^t X$. The kernel of a matrix $A \in \mathcal{M}_n(\mathbb{R})$, denoted $\ker(A)$, is the vectorial space defined as $\ker(A) = \{x \in \mathbb{R}^n, Ax = 0\}$. Every matrix of $\mathcal{M}_n(\mathbb{R})$ admits a finite set of eigenvalues $\lambda \in \mathbb{C}$ and their associated eigenspaces E_λ, defined as $E_\lambda = \ker(A - \lambda Id)$, where Id is the identity matrix and $E_\lambda \neq \{0\}$. Similarly, every matrix A admits *left-eigenspaces*, i.e. eigenspaces of A^t. The limit of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ for $\|X\| \to l$ is defined by the maximal value of $f(X)$ with $\|X\|$ in the neighborhood of $l \in \mathbb{R} \cup \{+\infty\}$ and is denoted $\lim\limits_{\|X\| \to l} f(X)$.

Invariants. A formula requires two canonical properties to be a loop invariant: it must be true at the beginning of the loop (initialization); it must be preserved by a loop step (inductivity). Similarly to [11], we define the inductive relation φ by the following constraint.

Definition 1 *Exact*
$\varphi \in \mathbb{R}^n$ is an exact inductive invariant *for a linear mapping f iff*

$$\forall X \in \mathbb{R}^n, |\langle \varphi, X \rangle| = 0 \Rightarrow |\langle \varphi, f(X) \rangle| = 0 \tag{1}$$

In the present paper, we add to this definition the concept of convergent and divergent inductive relation.

Definition 2 *Convergence*
$\varphi \in \mathbb{R}^n$ *is a* convergent inductive invariant *for a linear mapping f iff*

$$\forall X \in \mathbb{R}^n, \forall k \in \mathbb{R}, \mid \langle \varphi, X \rangle \mid \leqslant k \Rightarrow \mid \langle \varphi, f(X) \rangle \mid \leqslant k \tag{2}$$

Definition 3 *Divergence*
$\varphi \in \mathbb{R}^n$ *is a* divergent inductive invariant *for a linear mapping f iff*

$$\forall X \in \mathbb{R}^n, \forall k \in \mathbb{R} \mid \langle \varphi, X \rangle \mid \geqslant k \Rightarrow \mid \langle \varphi, f(X) \rangle \mid \geqslant k \tag{3}$$

The convergent invariant definition could have been written equivalently $\mid \langle \varphi, X \rangle \mid \leqslant \mid \langle \varphi, f(X) \rangle \mid$. We choose the other notation as the idea of the technique is to find a suitable value of k such that $\mid \langle \varphi, X \rangle \mid \leqslant k$ is an invariant of the loop. A vector φ satisfying the inductive relation is called a *semi-invariant* in contrast with *invariants* that also verify the initialization criterion, denoted $\langle \varphi, X_{init} \rangle$ with X_{init} the variables' initial values. The exact semi-invariants set of a linear mapping f is the union of all eigenspaces of f^* as proven in [11]. Also, we define the solvability of a mapping introduced in [27].

Definition 4 *Let $g \in (\mathbb{R}[X])^m$ be a polynomial mapping. g is solvable if there exists a partition of X into sub-vectors of variables $x = w_1 \uplus ... \uplus w_k$ and we can divide g into different mappings g_{w_j} manipulating variables of w_j such that*

$$g_{w_j}(x) = M_j w_j^t + P_j(w_1, ..., w_{j-1}, N)$$

with P_j a polynomial and N eventual non deterministic parameters.

For example, the mapping $g_N(x, y) = (x+y^2, y+N)$ depending on the parameter N is solvable because we can set $w_1 = \{y\}$ and $w_2 = \{x\}$. $g_y(x, y) = y + P_1(N)$, where $P_1 = N$ and $g_x(x, y) = x + P_2(y)$ where $P_2(y) = y^2$. We also can write $g_N(x, y) = (g_x(x, y), g_y(x, y))$

Remark. As shown in [11], deterministic solvable assignments are linearizable, i.e. they can be replaced by equivalent linear mappings. This allows to consider deterministic linear mappings $X' = A.X$ with X a vector containing both variables and monomials of those variables to represent deterministic solvable assignments.

Programming model. We use a basic programming language whose syntax is given in figure 3. Var is the set of variables used by the program. Variables take their value in $\mathbb{R}$. A program state is then a partial mapping $Var \rightharpoonup \mathbb{R}$. Any given program only uses a finite number n of variables, thus program states can be represented as vectors $X = (x_1, ..., x_n)^t$. Finally, we assume that for all programs, there exists $x_{n+1} = \mathbb{1}$ a constant variable always equal to 1. This allows to represent any affine assignment by a matrix. The expression $non\_det(exp_1, exp_2)$ returns a random value between the valuation of exp_1 and exp_2 when the program reaches this location. Multiple variables assignments occur simultaneously

```
i ::=   skip                                   exp ::=   cst ∈ ℝ
        | i; i                                           | x ∈ Var
        | (x_1, .., x_n) := (exp_1, ..., exp_n)          | exp + exp
        | while * do i done                              | exp * exp
                                                         | non_det(exp, exp)
```

Fig. 3: Code syntax

within a single instruction. We say an assignment $X = exp$ is affine (resp. solvable) when exp is an affine (resp. solvable) combination. Also, we say that an instruction is non-deterministic when it is an assignment in which the right value contains the expression non_det.

4 Convergent and divergent linear applications

4.1 Deterministic assignments

Being an inductive invariant requires for a formula F to be true after an iteration of the loop under the hypothesis that F holds before the iteration. The left eigenspace of a linear transformation (i.e. the eigenspace of the dual transformation) is exactly its set of exact invariants as defined in definition 1.

Convergence. By linear algebra, $|\langle\varphi, X\rangle| \leqslant k \Rightarrow |\langle f^*(\varphi), X\rangle| \leqslant k$ is strictly equivalent to the definition 2 of convergent semi-invariants. The formula $|\langle\varphi, X\rangle| \leqslant k$ represents what we call a *domain described by* φ, i.e. a polynomial relation over the variables of the program. The previous constraint specify that the domain described by φ is stable by f. The loop in figure 2 admits the invariant $x^2 + y^2 \leqslant 2$, a domain described by $\varphi = (0, 0, 0, 1, 0, 1)^t$ in the base $(\mathbb{1}, x, xy, x_2, y, y_2)$ where x_2 represents x^2, xy represents $x * y$ and y_2 represents y^2. We can check with the PILA algorithm that φ is an exact semi-invariant of the loop as it is a left eigenvector of the transformation performed by the loop. As such, it generates a vectorial space of exact semi-invariants $I = \{k.(x^2+y^2) = 0 \mid k \in \mathbb{R}\}$, which is a very poor result as $x^2 + y^2$ is constant only if it starts at 0 (else, $k = 0$ and we don't know anything about $x^2 + y^2$). We focus now on the eigenvalue associated to φ on f^*, which is 0.9248. Thus, we can replace $|\langle f^*(\varphi), X\rangle|$ by $|\lambda|.|\langle\varphi, X\rangle|$, which returns $|\langle\varphi, X\rangle| \leqslant k \Rightarrow |\lambda|.|\langle\varphi, X\rangle| \leqslant k$. As $|\lambda| < 1$, the vector φ satisfies the equation, thus φ is a convergent semi-invariant. Knowing the maximal initial value of $x^2 + y^2$ allows to determine the value of k, which is 2.

More generally, the set of convergent semi-invariants is exactly the set of eigenvectors bound to an eigenvalue λ such that $|\lambda| < 1$. The proof of this assertion requires the following lemma :

Lemma 1 $(\forall k, |\langle \varphi, X\rangle| \leqslant k \Rightarrow |\langle \varphi, f(X)\rangle| \leqslant k) \Rightarrow f^*(\varphi) = \lambda.\varphi$

In other words, convergent invariants are eigenvectors. The goal of the following property is to characterize the associated eigenvalue.

Property 1 φ *is a convergent semi-invariant* $\Leftrightarrow \exists\lambda, |\lambda| \leqslant 1, f^*(\varphi) = \lambda.\varphi$

Proof. If $|\lambda| \leqslant 1$, then φ is a convergent semi-invariant (see introduction of section 4.1). As the exact semi-invariants set of f is the union of the eigenspaces of f^*, we can deduce that this set is a superset of all the relations satisfying the definition 2. Moreover by the lemma 1, we have

$$(|\langle \varphi, X\rangle| \leqslant k \Rightarrow |\langle \varphi, f(X)\rangle| \leqslant k) \Rightarrow (| < \varphi, X > | \leqslant k \Rightarrow |\lambda|.| < \varphi, X > | \leqslant k)$$

For $k = | < \varphi, X > |$ it is true if and only if $|\lambda| \leqslant 1$. □

Divergence. The same reasoning applies to the generation of divergent invariants. For example, an eigenvalue λ such that $|\lambda| > 1$ associated to a semi-invariant φ implies that $|\langle \varphi, X\rangle| \geqslant k$ is an inductive invariant.

Property 2 $\exists\lambda, |\lambda| > 1, f^*(\varphi) = \lambda.\varphi \Rightarrow \varphi$ *is a divergent semi-invariant*

Proof. If there exists λ such that $f^*(\varphi) = \lambda.\varphi$, then we have that being a divergent semi invariant is equivalent to

$$| < \varphi, X > | \ \geqslant \ k \Rightarrow |\lambda|.| < \varphi, X > | \ \geqslant \ k$$

If we also have that $|\lambda| > 1$, then the previous equation is true. □

Note that this time, we only have an implication. For example, the transformation $f(x, \mathbb{1}) = (x + \mathbb{1}, \mathbb{1})$ admits $x \geqslant x_{init}$ as a divergent invariant but the only left eigenvector of f is $(0, 1)$, which correspond to the invariant "$\mathbb{1}$ *is constant*". Moreover, not all invariants of the form $P(X) \leqslant k$ are generated : the loop with the only assignment $x = x - 1$ admits the (non-convergent) invariant $x \leqslant x_{init}$. This invariant does not enter the scope of our setting as $|x| \leqslant x_{init}$ is false for $2x_{init} + 1$ iterations of $x = x - 1$.

4.2 Non-deterministic assignments

Some programs depend on inputs given all along their execution, for example linear filters. More generally, an important part of program analysis consists in studying non-deterministic assignments. As an example let us consider the program in figure 4, a slightly modified version of the program in figure 2. Our previous reasoning is not applicable now because, due to the non-determinism of N, the loop is no longer a linear mapping.

```
while (*) do
  N = non_det(-0.1,0.1);
   (x,y) = (0.68 * (x-y) + N, \
     0.68(x+y) + N);
done
```

Fig. 4: Non deterministic variant of the Figure 2

Idea. Intuitively, we will represent this loop by a matrix parametrized by N. For that purpose we use the concept of abstract mapping introduced in [15].

Definition 5 *An abstract linear mapping $f : \mathbb{R}^q \mapsto \mathcal{M}_n(\mathbb{R})$ is a mapping associating a vector $N \in \mathbb{R}^q$ to a matrix. We call f^* the dual mapping of f (i.e. the mapping such that $f^*(N) = (f(N))^T$). The expression of the parametrized matrix with respect to an abstract linear mapping will be called the* abstract matrix*.*

In our setting, the parameters are the non-deterministic values. For example, the previous loop can be represented by the abstract matrix M_N :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ N & 0.68 & 0 & 0 & -0.68 & 0 \\ N^2 & 1.36N & 0 & 0.462 & 0 & -0.462 \\ N^2 & 1.36N & 0.925 & 0.462 & -1.36N & 0.462 \\ N & 0.68 & 0 & 0 & 0.68 & 0 \\ N^2 & 1.36N & 0.925 & 0.462 & 1.36N & 0.462 \end{pmatrix}$$

Remark. Similarly to deterministic solvable mappings defined in section 3, non deterministic solvable mappings can be linearized to an abstract matrix. By considering non deterministic parameters as constants, the problem is reduced to the linearization of deterministic solvable mappings.

We have shown in section 4.1 that M_0 admits the invariant $e_0 = (0, 0, 0, 1, 0, 1)$ associated to the eigenvalue $\lambda_0 = 0.9248$. By decomposing M_N as the sum of M_0 and $(M_N - M_0)$, we also have $e_0.M_N = e_0.M_0 + e_0.(M_N - M_0) = \lambda_0.e_0 + \delta_0^N$, where $\delta_0^N = e_0.(M_N - M_0) = (2N^2, 2.72N, 0, 0, 0, 0)$. As the eigenvalue λ_0 is smaller than 1, we are looking for relations φ such that $\forall X, |\langle \varphi, X\rangle| \leqslant k \Rightarrow |\langle M_N^T.\varphi, X\rangle| \leqslant k$. We will call e_0 a *candidate invariant* for M_N. For e_0 to be a proper invariant for this transformation, the following property must hold:

$$\forall X, |\langle e_0, X\rangle| \leqslant k \Rightarrow |\lambda_0 \langle e_0, X\rangle + \langle \delta_0^N, X\rangle| \leqslant k \tag{4}$$

Intuitively, multiplying $\langle e_0, X\rangle$ by λ_0 reduces its norm strictly under k. We need to make sure that adding $\langle \delta_0^N, X\rangle$ does not contradict the induction criterion by increasing the result over k. The variables of the program depend on k, as does $\langle \delta_0^N, X\rangle$. If it increases faster than $|\lambda_0 \langle e_0, X\rangle|$ when k is increased, then no value of k will make the candidate invariant inductive. In particular, if $\langle e_0, X\rangle$ is a polynomial P of degree d, we need to be able to give an upper bound of $\langle \delta_0^N, X\rangle$ knowing that $|P(X)| < k$. If the degree of $\langle \delta_0^N, X\rangle$ is strictly smaller than d, then it will grow asymptotically slower than $|P(X)|$, thus for a big enough k the induction criterion is respected.

Property 3

$$\left(\forall X, |\langle e_0, X\rangle| \leqslant k \Rightarrow |\langle \delta_0^N, X\rangle| \leqslant (1 - |\lambda_0|).k\right) \Rightarrow \tag{5}$$

$|\langle e_0, X\rangle| \leqslant k$ *is an invariant of the loop.*

In our example, $\langle \delta_0^N, X \rangle = 2.72 * N * x + 2 * N^2$. The polynomial x is of degree 1 while $< e_0, X >= x^2 + y^2$ is of degree 2. We need to find a k such that

$$-0.0752 * k \leqslant 2.72 * N * x + 2 * N^2 \leqslant 0.0752 * k \tag{6}$$

Optimizing expressions. We will now maximize and minimize $2.72 * N * x + 2 * N^2$, knowing that $x^2 + y^2 \leqslant k$ and $-0.1 \leqslant N \leqslant 0.1$. Solving this problem is very close to solving a constrained polynomial optimization (CPO) problem [7]. CPO techniques provide ways to find values minimizing and maximizing expressions under a set of inequalities constraints. Our main issue is related to the parameter k that must be known in order to use CPO directly. We will not investigate in this article how CPO works in detail, but how we can reduce the problem of finding an optimal k to the CPO problem.

Assuming we have a function *min* computing the minimum, if it exists, of an expression under polynomial constraints, we propose an algorithm that refines the value of k in figure 5. The idea is to find k by dichotomy.

```
Data:
$\lambda$ : float
$Q$ : objective function
$P$ : polynomial constraint
non_det_c : non deterministic constraints
$N$ : int
Result: $k$ such that $\forall X, P(X) \leqslant k \Rightarrow f(X) \leqslant (1 - |\lambda|).k$
low_k = 0;
up_k = MAX_INT;
k = MAX_INT / 2;
i = 0;
while $i<N \wedge up\_k = MAX\_INT$ do
    i = i+1;
    Pk = function (x → P(x) + k);
    min = min(Q,[Pk] ∪ non_det_c);
    max = min(-1*Q,[Pk] ∪ non_det_c);
    if $min > (-1+|\lambda|) * k$ and $max < (1-|\lambda|)*k$ then
        up_k = k;
    else
        low_k = k;
    end
    k = (low_k + up_k) / 2;
end
```

Fig. 5: Dichotomy search of a k satisfying the condition (6)

- If k doesn't satisfy the constraints, we try a bigger one.
- If we find a k satisfying the two conditions, then it is a potential candidate. We can still try to refine it by searching for a smaller k.

We can improve this algorithm by guessing an upper value of k instead of taking an arbitrary maximal value MAX_INT. For our example, we started at $k = 50$ and found that $k = 14.9$ respects all the constraints.

- $x^2 + y^2 \leqslant 14.9 \Rightarrow |x| \leqslant 3.9$
- $|N| \leqslant 0.1$
- $|2.72 * x * N + 2 * N^2| \leqslant 1.08$, and $k * (1 - |\lambda|) = 1.12$.

Convergence. Note however that the existence of a k satisfying (6) is not guaranteed. For example, the set $S = \{(x, y, N) | x^2 + y^2 \leqslant k \wedge -0.1 \leqslant N \leqslant 0.1\}$ is a compact set for any value of k, which means that x, y and N have maximum and minimum values. This implies the existence of a lower and an upper bound for every expression composed with x, y and N, but the value of those expressions may be always higher than k such as for $x^2 + y^2 + 1$ bounded by $k + 1$.

Property 4 *Let P and Q two polynomials and $M > 0 \in \mathbb{R}$.*
If $\lim_{\|X\| \to +\infty} |\frac{Q(X)}{P(X)}| < M$, then there exists $k \in \mathbb{R}$ such that for all $k' \geqslant k$

$$|P(X)| \leqslant k' \Rightarrow |Q(X)| \leqslant M.k'$$

By taking $M = (1 - |\lambda_0|)$, this theorem gives us a sufficient condition to guarantee the convergence of the algorithm in figure 5.

Corollary. *If the objective has a lower degree in the deterministic variables than the candidate invariant, then the algorithm converges. If it has the same degree, then it depends on the main coefficients.*

As we are dealing with two polynomials P and Q, then if P (the candidate invariant) has a higher degree than Q (the objective function) in all its variables, the limit of $\frac{Q(X)}{P(X)}$ will be 0, which is enough to ensure the convergence of the method. If we come back to the objective function for the loop of figure 2, $Q(X) = 2.72.x.N + 2.N^2$ is a polynomial of degree 1 in x and 0 in y, thus $\lim_{\|X\| \to +\infty} |\frac{Q(X,N)}{P(X)}| = 0$ and we can be sure that the optimization will converge.

On the other hand, if we have $X = (x, y)$, $P(X) = x^2 + y^2$ and $Q(X, N) = 10.N(x^2 + y^2 + 1)$, with $|N| \leqslant 0.1$, the optimization procedure may not produce a result by theorem 4 because $\lim_{\|X\| \to +\infty} |\frac{Q(X,N)}{P(X)}| = 10N$ is higher than $1 - |\lambda|$ for $N = 0.1$.

Initial state. The knowledge of the initial state is not one of our hypotheses yet, but the previous theorem provides the necessary information we need to treat the case where the initial state is strictly higher than the minimal k we found. The previous theorem tells us that there exists a k such that for all

$k' \geqslant k$, k' is a solution to the optimization problem. Our optimization algorithm is searching for a value of k for which the set is inductive, though, and this solution may be only local : there may be a $k' > k$ which is not a solution of the optimization procedure. If the value of $P(X_{init})$ is strictly higher than k, there are two possibilities :

- it satisfies the objective (6), optimization is then not necessary as $k = P(X_{init})$ is correct, and we directly have a solution.
- it doesn't satisfy the objective, we have to find a $k > P(X_{init})$ satisfying it.

In both cases, we can enhance the optimization algorithm by first testing the objective (6) with $k = P(X_{init})$. If it does not respect the objective, then starting the dichotomy with $low\_k = P(X_{init})$ will return a solution (guaranteed by the property 4) strictly higher than $P(X_{init})$.

4.3 Rounding error

When dealing with real life programs, performing floating point arithmetic generates rounding error. As for an input signal abstracted by a non deterministic value, we can add to every computation that may lead to a rounding error a non deterministic value whose bounds are determined by the variables types and values.

Addition. Addition over two floating-point values lose some properties like associativity. For example, $(2^{64} - 2^{64}) + 2^{-64}$ will be strictly equal to 2^{-64} but $2^{64} + (-2^{64} + 2^{-64})$ will be equal to 0. To deal with addition, we can consider the highest possible error between a real value and its floating point representation, a.k.a. the machine epsilon. It is completely dependent of the C type used : for *float* (single precision) it corresponds to 2^{-23} ; for *double* (double precision) it is 2^{-52}. More generally, let x and y be two reals, with $\tilde{x}$ and $\tilde{y}$ their respective C representation. The IEEE model [2] says that an operation on floating point numbers must be equivalent to an operation on the reals, and then round the result to one of the nearest[1] floating point number. In this case, the relative error $|(\tilde{x} + \tilde{y}) - (x + y)| = (x + y) * \varepsilon$ where ε is the highest machine epsilon between the machine epsilon of the type of x, y and $(x + y)$. The error is relative to the value of x and y. This is not a problem, as we authorize in our setting non deterministic calls with expressions as argument.

Multiplication. A similar approximation happens during a multiplication of two floating point values. The relative error $|(\tilde{x} * \tilde{y}) - (x * y)| = x * y * \varepsilon$ Thus for every multiplication, we can add a non deterministic value between $-x * y * \varepsilon$ and $x * y * \varepsilon$.

With these considerations, we are able to provide precise bounds for rounding error for every operation performed in the loop.

[1] depending on rounding mode, this may be the floating point value immediately below or above the result.

Remark. Note that we can also deal with value casting. For example, when a cast from a floating point value to a integer is performed, the maximal error is bounded by 1 which can be abstracted in our setting by a non deterministic assignment.

5 Related work

There exist mainly two kinds of polynomial invariants: equality relations between variables, representing precise relations, and inequality relations, providing bounds over the different values of the variables. After the results of Karr in [16,22] on the complete search of affine equality relations between variables of an affine program, Müller-Olm and Seidl [23] have proposed an inter-procedural method for computing polynomial equalities of bounded degree as invariants. For linear programs, *Farkas' lemma* can be used to encode the invariance condition [9] under non linear constraints. Similarly, for polynomial programs, Gröbner bases have been shown to be an efficient way to compute the exact relation set of minimal polynomial loop invariants composed of *solvable assignments* by computing the intersection of polynomial ideals [27,26]. Even if this algorithm is known to be EXP-TIME complete in the degree of the invariant searched, high degree invariants are very rare for common loops and the tool ALIGATOR [18], inspired from this technique for *P-solvable loops*[20,19], is very efficient for low degree loops. Finally, [8] presents a technique that avoids the combination problem by using abstract interpretation to generate *abstract invariants*. This technique is implemented in the tool FASTIND. The main issue is the completion loss: some invariants are missed and a maximal degree must be provided. The direct use of exact mathematical techniques is also not very efficient for the analysis of non-deterministic assignments.

Synthesis of inequality invariants has become a growing field [21,29], for example in linear filters analysis and automatic verification in general as it provides good knowledge of the variables bounds when computing floating point operations. Abstract interpretation [10] with widening operators allows good approximation of loops with the desired format. A recent work [14] mixes abstract interpretation and loop acceleration (i.e. the precise computation of the transitive closure of a loop) to extend the framework and obtain precise upper and lower bounds on variables in the polyhedron domain. Very precise and computing non-trivial relations for complex loops and conditions, it has the drawback to be applicable to a very restricted type of transformations (linear transformation with eigenvalues λ such that $|\lambda| = 0$ or 1). We see this technique as complementary to ours as it generates invariants we do not find (such as $k \leqslant k_{init}$ for loop counters) and conversely. In order to treat non-deterministic loops, [21] refines as precisely as possible the set of reachable states for linear filters, harmonic oscillators and similar loops manipulating floating point numbers using a very specific abstract domain. A specific domain of polynomial inequalities have been imlplemented by [5], allowing conditions in the form $P(X) \leqslant 0$.

Dynamic analysis is also widely used in the detection of invariants. Daikon [13] infers linear *likely invariants*, i.e. candidate invariants, by confronting a given property pattern against a large number of executions. By expanding the pattern to more expressive terms with polynomial and array expressions, [24,25] infers general and disjunctive polynomial and array invariants.

6 Application and results

The plug-in Pilat, written in OCaml as a Frama-C plug-in (compatible with the latest stable release, Aluminium) and originally generating exact relations for deterministic C loops, has been extended with convergent invariant generation and non deterministic loop treatment for simple C loops. It implements our main algorithm of invariant generation in addition to the optimization algorithm of figure 5, and generates invariants as ACSL [6] annotations, making them readily understandable by other Frama-C plugins. The tool is available at [3].

```
int simple_filter(int s0, int s1){
 float r;
 while (1) {
  r = 1.5*s0 - 0.7*s1 + [-0.1,0.1]
  s1 = s0;
  s0 = r;
 }
}
```

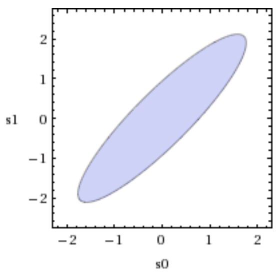

Fig. 6: Generation of one of the smallest polynomial invariant of degree 2 for a linear filter [21,31]

Let us now detail the work performed by Pilat over the example of figure 6 (taken from [21]). First, our tool generates the *shape* of the invariant, i.e. the polynomial P such that $|P(X)| \leqslant k$ is inductive for a certain k of the loop by setting the non deterministic choice to 0. We know by property 1 that such an invariant is an eigenvector of the transformation. By expressing $s_0^2, s_0 * s_1$ and s_1^2 as linear variables, we find the eigenvector $e_0 = (1.42857, -2.14285, 1)$ (in the base $(s_0^2, s_0 s_1, s_1^2)$) associated to the eigenvalue 0.7. Thus, $P_0(s_0, s_1) = 1.42857 * s_0^2 - 2.14285 * s_0 * s_1 + s_1^2 \leqslant k$ is an invariant of the loop *when N is set to 0*. The error made between the deterministic transformation (with $N = 0$) and the non deterministic one (with $N \in [-0.1, 0.1]$) is given by $Q(s_0, s_1, N) = 2 * N * s_1 - 2.142 * N * s_0 - 1.428 * N^2$. Q has a lower degree than P for a fixed N, so we have that $\lim\limits_{\|(s0,s1)\| \to +\infty} \frac{Q(s0,s1,N)}{P(s0,s1)} = 0 < 1 - \lambda$. The optimization

	PILAT Input		Results			Abs. Int. [21]
Program	Var	Degree	# invariants	Generation (in s)	Optimization (in s)	Proof (in s)
Deterministic						
Example 1	2	2	1	0.003	–	1.6
Dampened oscillator	2	2	1	0.007	–	0.036
Harmonic oscillator	2	2	1	0.004	–	0.035
Sympletic oscillator	2	2	1	0.002	–	0.008
[4] filter	2	1	1	0.0035	–	0.0017
Non deterministic						
Simple linear filter	2	2	1	0.0015	1.3	6.5
Example 3	2	2	1	0.003	1.7	4.3
Linear filter	2	2	1	0.0019	1	1
Lead-lag controller	2	1	2	0.002	2.5	6
Gaussian regulator	3	2	1	0.007	2.5	–
Controller	4	2	5	0.066	14	–
Low-pass filter	5	2	2	0.06	7	–

Table 1: Performance results with our implementation Pilat. Tests have been performed on a Dell Precision M4800 with 16GB RAM and 8 cores. The first part represents deterministic loops (thus, no optimization is necessary). The second part of the benchmark are non deterministic loops. Tests with abstract interpretation have been performed with the fixpoint solver described in [21] by attempting to prove goals implied by the invariants our tool synthesizes when they were compatible. Details and benchmark are available in [1]

procedure is now certain to converge, thus we minimize and maximize $Q(X, N)$ with the hypothesis $P(s0, s1) \leqslant k$. By starting the procedure with $k = 50$ (which is usually a good heuristic) and performing 10 iterations the optimization procedure returns $k = 0.87891$, thus $1.42857 * s_0^2 - 2.14285 * s_0 s_1 + s_1^2 \leqslant 0.87891$ is an inductive invariant represented in Figure 6. This is a real invariant, *assuming the initial state satisfies the relation.*

Let us now consider that the initial state of the loop is $(s_0, s_1) = (2, 1)$. Then at the beginning of the loop, $1.42857 * s_0^2 - 2.14285 * s_0 s_1 + s_1^2 = 2.42858 > 0.87891$, which does not respect the invariant. In this case the procedure starts by testing the optimization criterion with $k = 2.14285$. This choice of k is correct. In conclusion, we know that $1.42857 * s_0^2 - 2.14285 * s_0 s_1 + s_1^2 \leqslant 2.42858$ is an invariant of the loop.

More generally, we evaluated our method over the benchmark used in [28] for which we managed to find an invariant for every program containing no conditions. Though this benchmark has been built to test the effectiveness of a specific abstract domain, we managed to find similar results with a more general technique. Our results are given in table 1. As ellipsoids are a suitable representation for those examples, we have choosen 2 as the input degree of almost all our examples. The optimization script is based on Sage [30]. Note that the candidate generation is a lot faster than the optimization technique, mainly because of two reasons : computing *min* is time consuming for a large number of constraints; it is imprecise and its current implementation is incorrect (outputs an under approximation of the answer), we have to approximate its results to get a correct over approximation.

7 Conclusion and future work

Invariant generation for non deterministic linear loop is known to be a difficult problem. We provide to this purpose a surprisingly fast technique generating inductive relations that mostly relies on linear algebra algorithms widely used in many fields of computer science. Also, the optimization procedure for the non determinism treatment returns strong results. These invariants will be used in the scope of Frama-C [17] as a help to static analyzers, weakest precondition calculators and model-checkers.

We are currently facing three major issues that we intend to address in the future. The current optimization algorithm is assumed to have an exact *min* function. However, such function is both time consuming and imprecise. In addition, conditions are treated non deterministically, which reduces the strength of our results and limits the size of our benchmark to simple loops (linear filters with saturation are not included in our setting). Finally, the search of invariants for nested loops is a complex problem on which we are currently focusing.

References

1. Benchmark for the invariant generation. Available at `http://steven-de-oliveira.fr/content/bench/pilat_nd.pdf`.
2. IEEE Standard for Floating-Point Arithmetic. IEEE 754-2008.
3. PILAT. Available at `https://github.com/Stevendeo/Pilat`.
4. A. Adjé, S. Gaubert, and E. Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Logical Methods in Computer Science*, 8(1), 2012.
5. R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, pages 19–34, 2005.
6. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI C Specification Language, 2008.
7. D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.
8. D. Cachera, T. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to gröbner bases. *SCP*, 93, 2014.
9. M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. *Linear Invariant Generation Using Non-linear Constraint Solving*, pages 420–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
11. S. de Oliveira, S. Bensalem, and V. Prevosto. Polynomial invariants by linear algebra. In *ATVA 2016*, pages 479–494. Springer, 2016.

12. S. de Oliveira, S. Bensalem, and V. Prevosto. Synthesizing invariants by solving solvable loops. Technical report, CEA, 2016. available at http://steven-de-oliveira.fr/content/publis/2017_atva.pdf.
13. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.
14. L. Gonnord and P. Schrammel. Abstract acceleration in linear relation analysis. *Science of Computer Programming*, 93:125–153, 2014.
15. B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. *ACM SIGPLAN Notices*, 49(1):529–540, 2014.
16. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
17. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3), 2015.
18. L. Kovács. Aligator: A mathematica package for invariant generation (system description). In *Automated Reasoning*. Springer, 2008.
19. L. Kovács. Reasoning algebraically about P-solvable loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–264. Springer, 2008.
20. L. Kovács. A complete invariant generation approach for P-solvable loops. In *Perspectives of Systems Informatics*. Springer, 2010.
21. A. Miné, J. Breck, and T. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In *European Symposium on Programming Languages and Systems*, pages 560–588. Springer, 2016.
22. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *Automata, Languages and Programming*, pages 1016–1028. Springer, 2004.
23. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *ACM SIGPLAN Notices*, volume 39. ACM, 2004.
24. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 683–693. IEEE, 2012.
25. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*, pages 608–619. ACM, 2014.
26. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.
27. E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4), 2007.
28. P. Roux. *Analyse statique de systèmes de contrôle commande: synthèse d'invariants non linéaires*. PhD thesis, Toulouse, ISAE, 2013.
29. P. Roux, R. Jobredeaux, P.-L. Garoche, and É. Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pages 105–114. ACM, 2012.
30. W. Stein et al. Sage: Open source mathematical software. *7 December 2009*, 2008.
31. Wolfram|Alpha. Polynomial invariant for the simple_filter function, http://www.wolframalpha.com/input/?i=(-2.14285714286*(s1*s0)%2B1.42857142857*(s0*s0))%2B1.*(s1*s1)+%3C%3D++0.87891.