Polynomial invariants by linear algebra

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

[1] CEA, LIST, Software Reliability and Security Lab
`{steven.deoliveira,virgile.prevosto}@cea.fr`
[2] Université Grenoble Alpes
`saddek.bensalem@imag.fr`

Abstract. We present in this paper a new technique for generating polynomial invariants, divided in two independent parts : a procedure that reduces polynomial assignments composed loops analysis to linear loops under certain hypotheses and a procedure for generating inductive invariants for linear loops. Both of these techniques have a polynomial complexity for a bounded number of variables and we guarantee the completeness of the technique for a bounded degree which we successfully implemented for C programs verification.

1   Introduction

When dealing with computer programming, anyone should be aware of the underlying behavior of the whole code, especially when it comes to life-critical projects composed of million of lines of code [10]. Manual code review cannot scale to the size of actual embedded programs. Testing allows to detect many vulnerabilities but it is never enough to certify their total absence. Indeed, the cost of generating and executing sufficient test cases to meet the most stringent coverage criteria [4] that are expected for critical software becomes quickly prohibitive as the size of the code under test grows. Alternatively, formal methods techniques based on abstraction allow us to prove the absence of error.

However, since a program can, at least in theory, have an infinite number of different behaviors, the verification problem is undecidable and these techniques either lose precision (emitting false alarms) and/or require manual input. One of the main issue of such approach is the analysis of loops, considered as a major research problem since the 70s [2]. Program verification based on Floyd-Hoare's inductive assertion [9] and CEGAR-like techniques [7] for model-checking uses loop invariants in order to reduce the problem to an acyclic graph analysis [3] instead of unrolling or accelerating loops [11]. Thus, a lot or research nowadays is focused on the automatic inference of loop invariants [16,21].

We present in this paper a new technique for generating polynomial invariants, divided in two independent parts : a *linearization* procedure that reduces the analysis of solvable loops, defined in [21], to the analysis of linear loops ; an *inductive invariant generation* procedure for linear loops. Those two techniques are totally independent from each other, we aim to present in this article their composition in order to find polynomial invariants for polynomial loops. We also

add an extension of this composition allowing to treat loops with complex behaviors that induces the presence of complex numbers in our calculation. The linearization algorithm has been inspired by a compiler optimisation technique called *operator strength reduction* [8]. Our invariant generation is completely independent from the initial state of the loop studied and outputs parametrized invariants, which is very effective on programs using a loop multiple times and loops for which we have no knowledge of the initial state. In addition to being complete for a certain class of polynomial relations, the invariant generation technique has the advantage to be faster than the already existing one for such loops as it relies on polynomial complexity linear algebra algorithms.

Furthermore, a tool implementing this method has been developed in the Frama-C framework for C programs verification [14] as a new plug-in called PILAT (standing for **P**olynomial **I**nvariants by **L**inear **A**lgebra **T**ool). We then compared our performances with ALIGATOR [16] and FASTIND [5], two invariant generators working on similar kinds of loops. First experiments over a representative benchmark exposed great improvements in term of computation time.

Outline. The rest of this paper is structured as follows. Section 2 introduces the theoretical concepts used all along the article and the kind of programs we focus on. Section 3 presents the application of our technique on a simple example. Section 4.1 presents the linearization step for simplifying the loop, reducing the problem to the study of affine loops. Section 4.2 presents our contribution for generating all polynomial invariants of affine loops. Section 4.3 extends the method with the treatment of invariants containing non-rational expressions. Finally, section 5 compares PILAT to ALIGATOR and FASTIND. .

State of the art. Several methods have been proposed to generate invariants for kinds of loops that are similar to the ones we address in this paper. In particular, the weakest precondition calculus of polynomial properties in [19] is based on the computation of the affine transformation kernel done by the program. This method is based on the computation of the kernel of the affine transformation described by the program. More than requiring the whole program to be affine, this method relies on the fact that once in the program there exists a non-invertible assignment, otherwise the kernel is empty. This assumption is valuable in practice, as a constant initialization is non- invertible, so the results may appear at the end of a whole-program analysis and highly depend on the initial state of the program. On the other hand, our method can generate parametrized invariants, computable without any knowledge of the initial state of a loop, making it more amenable to modular verification.

From a constant propagation technique in [18] to a complete invariant generation in [21], Gröbner bases have proven to be an effective way to handle polynomial invariant generation. Such approaches have been successfully implemented in the tool ALIGATOR [16]. This tool generates all polynomial invariants of any degree from a succession of p-solvable polynomial mappings in very few

steps. It relies on the iterative computation of Gröbner bases of some polynomial ideals, which is a complicated problem proven to be EXPSPACE-complete [17].

Attempts to get rid of Gröbner bases as in [5] using abstract interpretation with a constant-based instead of a iterative-based technique accelerates the computation of invariants by generating abstract loop invariants. However, this technique is incomplete and misses some invariants. The method we propose here is complete for a particular set of loops defined in [21] in the sense that it finds all polynomial relations P of a given degree verifying $P(X) = 0$ at every step of the loop, and has a polynomial complexity in the degree of the invariants sought for a given number of variables.

2 Preliminaries

Mathematical background. Given a field $\mathbb{K}$, $\mathbb{K}^n$ is the vector space of dimension n composed by vectors with n coefficients in $\mathbb{K}$. Given a family of vector $\Phi \subset \mathbb{K}^n$, $Vect(\Phi)$ is the vector space generated by Φ. Elements of $\mathbb{K}^n$ are denoted $x = (x_1, ..., x_n)^t$ a column vector. $\mathcal{M}_n(\mathbb{K})$ is the set of matrices of size $n * n$ and $\mathbb{K}[X]$ is the set of polynomials using variables with coefficients in $\mathbb{K}$. We note $\overline{\mathbb{K}}$ the algebraic closure of $\mathbb{K}$, $\overline{\mathbb{K}} = \{x | \exists P \in \mathbb{K}[X], P(x) = 0\}$. We will use $\langle .,. \rangle$ the linear algebra standard notation, $\langle x, y \rangle = x \cdot y^t$, with $\cdot$ the standard dot product. The kernel of a matrix $A \in \mathcal{M}_n(\mathbb{K})$, denoted $\ker(A)$, is the vector space defined as $\ker(A) = \{x | x \in \mathbb{K}^n, A.x = 0\}$. Every matrix of $\mathcal{M}_n(\mathbb{K})$ admits a finite set of eigenvalues $\lambda \in \overline{\mathbb{K}}$ and their associated eigenspaces E_λ, defined as $E_\lambda = \ker(A - \lambda Id)$, where Id is the identity matrix and $E_\lambda \neq \{0\}$. Let E be a $\mathbb{K}$ vector space, $F \subset E$ a sub vector space of E and x an element of F. A vector y is *orthogonal* to x if $\langle x, y \rangle = 0$. We denote $F^\perp$ the set of vectors orthogonal to every element of F.

Programming model. We use a basic programming language whose syntax is given in figure 1. Var is a set of variables that can be used by a program, and which is supposed to have a total order. Variables take value in a field $\mathbb{K}$. A program state is then a partial mapping $Var \rightharpoonup \mathbb{K}$. Any given program only uses a finite number n of variables. Thus, program states can be represented as a vector $X = (x_1, ..., x_n)^t$. In addition, we will note $X' = (x'_1, ..., x'_n)^t$ the program state after an assignment. Finally, we assume that for all programs, there exists $x_{n+1} = x'_{n+1} = \mathbb{1}$ a constant variable always equal to 1.

The i OR i instruction refers to a non-deterministic condition.

Each i will be refered to as one of the *bodies* of the loop.

Multiple variables assignments occur simultaneously within a single instruction. We say that an instruction is affine when it is an assignment for which the right values are affine. If not, we divide instructions in two categories with respect to the following definition, from [21]

Definition 1 *Let $g \in \mathbb{Q}[X]^m$ be a polynomial mapping. g is solvable if there exists a partition of X into subvectors of variables $x = w_1 \uplus ... \uplus w_k$ such that*

$$
\begin{array}{llll}
i ::= & i\,;i & exp ::= & cst \in \mathbb{K} \\
& |\quad (x_1,..,x_n):=(exp_1,...,exp_n) & & |\ x \in Var \\
& |\quad i \textbf{ OR } i & & |\ exp + exp \\
& |\quad \textbf{while } (*) \textbf{ do } i \textbf{ done} & & |\ exp * exp
\end{array}
$$

Fig. 1: Code syntax

$\forall j,\ 1 \leqslant j \leqslant k$ *we have*

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

with $(M_i)_{1 \leqslant i \leqslant k}$ *a matrix family and* $(P_i)_{1 \leqslant i \leqslant k}$ *a family of polynomial mapping.*

An instruction is solvable if the associated assignment is a solvable polynomial mapping. Otherwise, it is unsolvable. Our technique focuses on loops containing only solvable instructions, thus it is not possible to generate invariants for nested loops. It is however possible to find an invariant for a loop containing no inner loop even if it is itself inside a loop, that's why we allow the construction.

3 Overview of our approach

Steps of the generation. In order to explain our method we will take the following running example, for which we want to compute all invariants of degree 3:

```
while  (*)  do
    ( x , y )  :=  ( x  +  y*y ,  y  +  1)
done
```

Our method is based on two distinct parts :

1. reduction of the polynomial loop to a linear loop;
2. linear invariant generation from the linearized loop.

We want to find a linear mapping f that *simulates* the behavior of the polynomial mapping $P(x, y) = (x + y^2, y + 1)$. To achieve this, we will express the value of every monomial of degree 2 or more using brand new variables. Here, the problem comes from the y^2 monomial. In [19], it is described how to consider the evolution of higher degree monomials as affine applications of lower or equal degree monomials when the variables involved in those monomials evolve affinely. We extend this method to express monomials transformations of the loop by affine transformations, reducing the problem to a simpler loop analysis. For example here, $y' = y + 1$ is an affine assignment, so there exists an affine representation of $y_2 = y^2$, which is $y_2' = y_2 + 2.y + 1$. Assuming the initial y_2

is correct, we are sure to express the value of y^2 with the variable y_2. Also, if we want to find invariants of degree 3, we will need to express all monomials of degree 3, i.e. xy and y_3 the same way. (monomials containing x^i with $i \geqslant 2$ are irrelevant as their expression require the expression of degree 4 monomials). Applying this method to P gives us the linear mapping $f(x, y, y_2, xy, y_3, \mathbb{1}) = (x + y_2, y + \mathbb{1}, y_2 + 2.y + \mathbb{1}, xy + x + y_2 + y_3, y_3 + 3.y_2 + 3.y + \mathbb{1}, \mathbb{1})$, with $\mathbb{1}$ the constant variable mentioned in the previous section.

Now comes the second part of the algorithm, the invariant generation. Informally, an invariant for a loop is a formula that

1. is valid at the beginning of the loop ;
2. stays valid after every loop step.

We are interested in finding *semi-invariants* complying only with the second criterion such that they can be expressed as a linear equation over X, containing the assignment's original variables and the new ones generated by the linearization procedure. In this setting, a formula satisfying the second criterion is then a vector of coefficients φ such that

$$\langle \varphi, X \rangle = 0 \Rightarrow \langle \varphi, f(X) \rangle = 0 \tag{1}$$

By linear algebra, the following is always true

$$\langle \varphi, f(X) \rangle = \langle f^*(\varphi), X \rangle \tag{2}$$

where f^* is the dual of f. If φ happens to be an eigenvector of f^* (i.e. there exists λ such that $f^*(\varphi) = \lambda\varphi$), the equation (1) becomes

$$\langle \varphi, X \rangle = 0 \Rightarrow \langle f^*(\varphi), X \rangle = 0 \text{ by (2)}$$
$$\langle \varphi, X \rangle = 0 \Rightarrow \langle \lambda.\varphi, X \rangle = 0$$
$$\langle \varphi, X \rangle = 0 \Rightarrow \lambda. \langle \varphi, X \rangle = 0$$

which is always true. We just need to *transpose* the matrix representing f to compute f^*. It returns $f^*(x, y, y_2, y_3, \mathbb{1}) = (x, y + y_2 + y_3, x + y_2 + 3.y_3, y_3, y + y_2 + y_3 + \mathbb{1}, y + y_2 + y_3 + \mathbb{1})$. f^* only admits the eigenvalue 1. The eigenspace of f^* associated to 1 is generated by two independants vectors, $e_1 = (-6, 1, -3, 2, 0)^t$ and $e_2 = (0, 0, 0, 0, 1)^t$. Eventually, we get the formula $F_{k1, k2} = (k_1.(-6.x + y - 3.y_2 + 2.y_3) + k_2.\mathbb{1} = 0)$ as invariant, with $k_1, k_2 \in \mathbb{Q}$. By writing $k = -\frac{k_2}{k_1}$ and replacing $\mathbb{1}$ with 1, we can rewrite it with only one parameter, $F_k = (-6.x + y - 3.y_2 + 2y_3 = k)$. In this case, information on the initial state of the loop allows to fix the value of the parameter k. For example if the loop starts with $(x = 0, y = 0)$, then $-6.x + y - 3.y^2 + 2.y^3 = 0$, and F_0 is an invariant. The next section will show how the work done on our example can be generalized on any (solvable) loop. In particular, section 4.1 will deal with the linearization of polynomial assignments. Then we will see in section 4.2 that the eigenspace of the application actually represents all the possible invariants of f and that we can always reduce them to find a formula with only one parameter.

Extension of the basic method. The application's eigenvector may not always be rational. For example, applying the previous technique on a mapping such as $f(x, y) = (y, 2.x)$ will give us invariants with coefficients involving $\sqrt{2}$. Dealing with irrational and/or complex values raises some issues in our current implementation setting. Therefore, we propose in section 4.3 a solution to stick with rational numbers. Eventually, we treat the case when a condition occur in loops in section 4.4.

4 Automated generation of loop invariants

4.1 Strength reduction of polynomial loops

Lowerization. Let P be a program containing a single loop with a single solvable assignment $X := g(X)$. In order to reduce the invariant generation problem for solvable polynomial loops to the one for affine loops, we need to find a linear mapping f that perfectly matches g. As shown in figure 2, the first loop L1 is

```
L1 :

while (*) do
  (x,y,z) := (x + 1, y + 2, z + x*y)
done

L2 :

xy = x*y
while (*) do
  (x,y,xy,z) := (x + 1, y + 2, xy + 2x + y + 2, z + xy)
done
```

Fig. 2: Polynomial and affine loop having the same behavior

polynomial but there exists a similar affine loop, namely L2, computing the same vector of values plus and thanks to an extra variable xy.

Definition 2 *Let g be a polynomial mapping of degree d using m variables. g is linearizable if there exists a linear mapping f such that $X' = g(X) \Rightarrow (X', P(X')) = f(X, P(X))$, where $P : \mathbb{Q}^m \to \mathbb{Q}^n$ is a polynomial of degree d.*

By considering polynomials as entries of the application, we are able to consider the evolution of the polynomial value instead of recomputing it for every loop step. This is the case in the previous example, where the computation of xy as $x * y$ is made once at the beginning of the loop. Afterwards, its evolution depends linearly of itself, x and y. Similarly, if we want to consider y^n for some $n \geq 2$, we would just need to express the evolution of y^n by a linear combination

of itself and *lower degree monomials*, which could themselves be expressed as linear combinations of lower degree monomials, until we reach an affine application. We call this process the polynomial mappings *lowerization* or *linearization*.

Remark. This example and our running example have the good property to be linearizable. However, this property is not true for all polynomials loops. Consider for example the mapping $f(x) = x^2$. Trying to express x^2 as a linear variable will force us to consider the monomials x^4, x^8 and so on. Thus, we need to restrain our study to mappings that *do not polynomially transform a variable itself*. This class of polynomials corresponds to solvable polynomial mappings, defined in Definition 1.

Property 1 *For every solvable polynomial mapping g, g is linearizable.*

For example, let $g(x, y) = (x + y^2, y + 1)$. g is linearized by $f(x, y, y_2) = (x + y_2, y + 1, y_2 + 2y + 1)$. Indeed with $(x', y') = g(x, y)$, we have $(x', y', y'^2) = f(x, y, y^2)$

Linearization Algorithm. The algorithm is divided in two parts : the solvability verification of the mapping and, if successful, the linearization process. The solvability verification consists in finding an appropriate partitioning of the variables that respects the solvable constraint. It is nothing more than checking that a variable v cannot be in a polynomial (i.e. non linear) assignment of another variable that itself depend on v. This check can be reduced to verifying the acyclicity of a graph, which can be computed e.g. by Tarjan's [22] or Johnson's [12] algorithms.

The linearization process then consists in considering all monomials as new variables, then finding their linear evolution by replacing each of their variables by the transformation made by the initial application. This may create new monomials, for which we similarly create new variables until all necessary monomials have been mapped to a variable. Since we tested the solvability of the loop, the variable creation process will eventually stop. Indeed, if this was not the case, this would mean that a variable x transitively depends on x^d with $d > 1$.

Elevation. We saw how to transform a polynomial application into a linear mapping by adding extra variables representing the successive products and powers of every variable. This information can be useful in order to generate invariants but in fact, most of the time, this is not enough. In our running example of section 2, $g(x, y) = (x + y^2, y + 1)$, the degree of the mapping is 2 but there exists no invariant of degree 2 for this loop. In order to deal with higher-degree invariants, we need not just to linearize g, we also have to add more variables to our study. As we can represent monomials of variables of a solvable mapping as linear applications, we can extend the method to generate higher degree monomials such as y^3 for example : we *elevate* g to a higher degree. The process of elevation is described in [19] as a way to express polynomial relations on a linear program.

Property 2 *Every solvable polynomial mapping g using n variables is lineariz-able by a linear mapping f using at most $\binom{n+d}{d}$ new variables, where d is the degree of P, the polynomial linearizing g as in definition 2.*

Note. The complexity of the transformation is *polynomial* for d or n fixed. The lowerization algorithm can be used as shown above by adding variables computing the high degree monomials we want to linearize. Moreover, $\binom{n+d}{d}$ is an upper bound and in practice, we usually need much less variables. For instance, in our running example, we don't need to consider $x.y^2$. Indeed, if we tried to linearize this monomial, we would end up with $x.y^2 = x.y^2 + x.y + x + y^4 + 2y^3 + y^2$, a polynomial of degree 4. Detecting that a monomial m is relevant or not can be done by computing the degree of its transformation. For example, the assignment of x is a degree 2 polynomial, so x^2 associated transformation will be of degree 4. Here, there is actually only two interesting monomials of degree 3, which are xy and y^3. Though those variables will be useless for the linearized mapping, they are still easily computable: $y_3' = y_3 + 3.y_2 + 3.y + 1$ and $xy = xy + x + y_2 + y_3$. This limits the necessary variables to only 6 $(x, y, y_2, y_3, xy, \mathbb{1})$ instead of $\binom{5}{2} = 10$. This upper bound in only reached for affine transformations when searching for polynomial invariants, as all possible monomials need to be treated.

4.2 Invariant generation

The transformation described previously doesn't linearize a whole program, but only a loop. Polynomial assignments must be performed before the loop starts to initialize the new monomials. The method we present only focuses on the loop behavior itself, allowing any kind of operation outside of the loop.

Eigenspace. Loop invariants are logical formulas satisfied at every step of a loop. We can characterize them with two criteria : they have to hold at the beginning of the loop (initialization criterion) and if they hold at one step, then they hold at the next step (heredity criterion). Our technique is based on the discovery of linear combinations of variables that are equal to 0 and satisfying the heredity criterion. For example, the loop of section 3 admits the formula $-6.x + y - 3.y_2 + 2y_3 = k$ as a good invariant candidate. Indeed, if we set k in accordance with the values of the variables at the beginning of the loop, then this formula will be true for any step of the loop. We call such formulas *semi-invariants*.

Definition 3 *Let $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$ and $f : \mathbb{K}^n \mapsto \mathbb{K}^n$ two linear mappings. φ is a semi-invariant for f iff $\forall X$, $\varphi(X) = 0 \Rightarrow \varphi(f(X)) = 0$.*

Definition 4 *Let $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$, $f : \mathbb{K}^n \mapsto \mathbb{K}^n$ and $X \in \mathbb{K}^n$. φ is an invariant for f with initial state X iff $\varphi(X) = 0$ and φ is a semi-invariant for f.*

The key point of our technique relies on the fact that if there exists λ, $f^*(\varphi) = \lambda\varphi$, then we know that φ is a semi-invariant. Indeed, we can rewrite definition 3

by $\langle \varphi, x \rangle = 0 \Rightarrow \langle \varphi, f(x) \rangle = 0$. By linear algebra, we have $\langle \varphi, f(x) \rangle = \langle f^*(\varphi), x \rangle$, with f^* the dual of f. If $\exists \lambda, f^*(\varphi) = \lambda \varphi$, then we can deduce that $\langle \varphi, x \rangle = 0 \Rightarrow \lambda \langle \varphi, x \rangle = 0$. This formula is always true, thus we know that φ is a semi-invariant. Such φ are commonly called *eigenvectors* of f^*. We will not adress the problem of computing the eigenvectors of an application as this problem have been widely studied (in [20] for example).

Recall our running example $g(x, y) = (x + y^2, y + 1)$, linearized by the application $f(x, y, y_2, xy, y_3, \mathbb{1}) = (x + y_2, y + \mathbb{1}, y_2 + 2y + \mathbb{1}, xy + x + y_2 + y_3, y_3 + 3y_2 + 3y + \mathbb{1}, \mathbb{1})$. f^* admits $e_1 = (-6, 1, -3, 0, 2, 0)^t$ and $e_2 = (0, 0, 0, 0, 0, 1)^t$ as eigenvectors associated to the eigenvalue $\lambda = 1$. It means that if $\langle k_1.e_1 + k_2 e_2, x \rangle = 0$, then

$$\langle k_1.e_1 + k_2 e_2, f(X) \rangle = \langle f^*(k_1.e_1 + k_2 e_2), X \rangle$$
$$= \langle \lambda(k_1.e_1 + k_2 e_2), X \rangle$$
$$= 0$$

In other words, $\langle k_1.e_1 + k_2 e_2, X \rangle = 0$ is a semi-invariant. Then, by expanding it, we can find that $-6.x + y - 3.y_2 + 2y_3 = k$, with $k = -\frac{k_2}{k_1}$ is a semi-invariant. In terms of the original variables, we have thus $-6.x + y - 3.y^2 + 2y^3 = k$.

Being an eigenvector of f^* does not just guarantee a formula to be a semi-invariant of a loop transformed by f. This is also a necessary condition.

Theorem 1 $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$ *is a semi-invariant if and only if* $\exists \lambda \in \mathbb{K}, \exists \varphi \in E_\lambda$, *where* $E_\lambda = ker(f^* - \lambda Id)$.

It is now clear that the set of invariants is exactly the union of all eigenspaces of f^*, i.e. a vector space union (which is not a vectorial space itself). An element φ of E_λ of basis $\{e_1, ... e_n\}$ is a linear combination of $e_1, ..., e_n$:

$$\varphi = \sum_{k=1}^{n} k_i e_i$$

The parameters k_i can be chosen with respect to the initial state of the loop.

Expression of eigenvectors as invariants.   More than a syntactic sugar, the variable $\mathbb{1}$ brings interesting properties over the kind of invariants we generate for an application f. The vector $e_{\mathbb{1}}$ such that $\langle e_{\mathbb{1}}, X \rangle = \mathbb{1}$ is always an eigenvector associated to the eigenvalue 1. Indeed, by definition $f(\mathbb{1}) = \mathbb{1}$, hence $f^*(e_{\mathbb{1}}) = e_{\mathbb{1}}$. For example, let's take the mapping $f(x, y, xy, \mathbb{1}) = (2x, \frac{1}{2}y + 1, xy + 2x, \mathbb{1})$. This mapping admits 3 eigenvalues : 2, $\frac{1}{2}$ and 1. There exists two eigenvectors for the eigenvalue 1 : $(-2, 0, 1, 0)$ and $(0, 0, 0, 1) = e_{\mathbb{1}}$. We have then the semi-invariant $k_1.(-2x + xy) + k_2 = 0$, or $-2x + xy = \frac{-k_2}{k_1}$. This implies that the two parameters k_1 and k_2 can be reduced to only one paramter $k = \frac{-k_2}{k_1}$, which simplifies a lot the equation by providing a way to compute the parameter at the initial state if we know it. For our example, $\frac{-k_2}{k_1}$ would be $-2x_{init} + x_{init}.y_{init}$, where x_{init} and y_{init} are the initial values of x and y. More generally, each eigenvector associated

to 1 gives us an invariant φ that can be rewritten as $\varphi(X) = k$, where k is inferred from the initial value of the loop variables.

We can generalize this observation to eigenvectors associated to any eigenvalue. To illustrate this category, let us take as example $f(x, y, z) = (2x, 2y, 2z)$. Eigenvectors associated to 2 are $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$ and $e_3 = (0, 0, 1)$, thus $k_1 x + k_2 y + k_3 z = 0$ is a semi invariant, for any k_1, k_2 and k_3 satisfying the formula for the initial condition of the loop. However, if we try to set e.g. $k_1 = k_2 = 1$, using $x + y + kz = 0$ as semi invariant, we won't be able to find a proper invariant when y_{init} or $x_{init} \neq 0$ and $z_{init} = 0$. Thus, in order to keep the genericity of our formulas, we cannot afford to simplify the invariant as easily as we can do for invariants associated to the eigenvalue 1. Namely for every e_i, we have to test whether $\langle e_i, X_{init} \rangle = 0$. For each e_i for which this is the case, $\langle e_i, X \rangle = 0$ is itself an invariant if $\langle e_i, X_{init} \rangle = 0$. However, if there exists an i such that $\langle e_i, X_{init} \rangle \neq 0$, then we can simplify the problem. For example, we assume that $z_{init} \neq 0$. Then $k_1 x_{init} + k_2 y_{init} + k_3 z_{init} = 0 \Leftrightarrow \frac{k_1 x_{init} + k_2 y_{init}}{z_{init}} = -k_3$. We know then that $k_1 x + k_2 y = \frac{k_1 x_{init} + k_2 y_{init}}{z_{init}} z$ is a semi-invariant. By writing $g(k_1, k_2) = \frac{k_1 x_{init} + k_2 y_{init}}{z_{init}}$, we have

$$\begin{cases} x = g(1, 0)z \\ y = g(0, 1)z \end{cases}$$

As g is a linear application, these two invariants implies that $\forall k_1, k_2, k_1 x + k_2 y = g(k_1, k_2)z$ is a semi-invariant.

Property 3 *Let $\mathcal{F}$ a semi-invariant expressed as $\mathcal{F} = \sum\limits_{i=0}^{n} k_i e_i$.*

If $\langle e_0, X_{init} \rangle \neq 0$, then we have that

$$\bigwedge_{i=1}^{n} (\langle e_i, X \rangle = -\frac{\langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle} \langle e_0, X \rangle) \text{ is an invariant} \Leftrightarrow \langle \mathcal{F}, X_{init} \rangle = 0$$

We are now able to use pairs of eigenvectors to express invariants by knowing the initial condition.

Algorithm. As we are restricting our study to solvable loops, that we know can be replaced without loss of generality by linear loops, we assume the input of this algorithm is a family of linear mappings. We can easily compose them via their matrix representation. We end up with a new matrix A. Computing the dual of A is computing the matrix A^T. Then, eigenvectors of A^T can be computed by many algorithms in the linear algebra literature [20]. As the eigenvalue problem is known to be polynomial, our invariant generation algorithm is also polynomial.

4.3   Extension of the method

Let $A \in \mathcal{M}_n(\mathbb{Q})$. In the general case, A admits irrational and complex eigenvalues and eigenvectors, which end up generating irrational or complex invariants. We cannot accept such representation for a further analysis of the input program because of the future use of these invariants, by SMT solvers for example

which hardly deal with non-rational numbers. For example, let us take the function $f(x,y) = (y, 2x)$. This mapping admits two eigenvalues : $\lambda_x = \sqrt{2}$ and $\lambda_y = -\sqrt{2}$. In this example, the previous method would output the invariants $k.(x + \sqrt{2}y) = 0$ and $k'.(x - \sqrt{2}y) = 0$. With x and y integers or rationals, this would be possible iff $k = k' = 0$. However, by considering the variable xy the invariant generation procedure outputs the invariant $k.(xy) = 0$, which is possible if x or y equals 0. This raises the issue of finding a product of variables that will give us a rational invariant. We aim to treat the problem at its source : the algebraic character of the matrix eigenvalues. A value x is algebraic in $\mathbb{Q}$ if there exists a polynomial P in $\mathbb{Q}[X]$ such that $P(x) = 0$. Assuming we have a geometric relation between the complex eigenvalues λ_i (i.e. a product q of eigenvalues that is rational), we will build a monomial m as a product of variables x_i associated to λ_i such that the presence of this monomial induces the presence of a rational eigenvalue, namely q. Moreover, a rational eigenvalue of a matrix is always associated to a rational eigenvector. Indeed, the kernel of a rational matrix is always a $\mathbb{Q}$-vectorial space. If $\lambda \in \mathbb{Q}$ is an eigenvalue of A, then $A - \lambda.Id$ is a rational matrix and its kernel is not empty.

Definition 5 *Let $A \in \mathcal{M}_n(\mathbb{Q})$. We denote $\Psi_d(A)$ the elevation matrix such that $\forall X = (x_1, ..., x_n) \in \mathbb{Q}^n, \Psi_d(A).p(X) = p(A.X)$, with $p \in (\mathbb{Q}[X]^k)$ a polynomial associating X to all possible monomials of degree d or lower.*

For example, if we have $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ as a transformation for $X = (x, y)$, and $x \prec y$, we have as transformation for the variables (x^2, xy, y^2, x, y) the matrix

$$\Psi_2(A) = \begin{pmatrix} a^2 & 2ab & b^2 & 0 & 0 \\ ac & ad + bc & bd & 0 & 0 \\ c^2 & 2cd & d^2 & 0 & 0 \\ 0 & 0 & 0 & a & b \\ 0 & 0 & 0 & c & d \end{pmatrix}$$

Property 4 *Let $A \in \mathcal{M}_d(\mathbb{Q}), \Lambda(M)$ the eigenvalue set of a matrix M and d an integer. Then for any product p of d or less elements of $\Lambda(A), p \in \Lambda(\Psi_d(A))$.*

We can generalize this property for more variables. After working with two variables, we get a new matrix with new variables that we can combine similarly, and so on. Thanks to this property, if we have a multiplicative relation between eigenvalues we are able to create *home-made* variables in the elevated application whose presence implies the presence of rational eigenvalues.

Though we could brute-force the search of rational products of irrational eigenvalues in order to find all possibilities of variable products that have rational eigenvalues, we could search for algebraic relations, i.e. multiplicative relations between algebraic values. This subject is treated in [13] and we will not focus on it. However, we can guarantee that there exists at least one monomial having a rational eigenvalue. Indeed, it is known that the product of all eigenvalues of a rational matrix is equal to its determinant. As the determinant of a rational

matrix is always rational, we know that the product of all variables infers the presence of the determinant of the matrix as eigenvalue of the elevated matrix. Coming back to the previous example, we have the algebraic relation $\lambda_x.\lambda_y = -2$. If we consider the evolution of xy, we have $(xy') = 2xy$. Note that the eigenvalue associated to xy is 2 and not -2. Indeed, we know that $A = P^{-1}JP$, with

$$P = \begin{pmatrix} 1 & -1 \\ \sqrt{2} & -\sqrt{2} \end{pmatrix}$$

and J an upper-triangular matrix, which means the eigenvalues of A are on the diagonal of J. xy in the base of J would be $(x + \sqrt{2}y)(x - \sqrt{2}y) = x^2 - 2y^2$, and we have well $\lambda_x^2 - 2\lambda_y^2 = -2$.

Finally, by knowing that $\lambda_x^2 = 2$, $\lambda_y^2 = 2$ and $\lambda_x\lambda_y = -2$, we will consider the variables x^2, y^2 and xy in our analysis of f. We can deduce new semi-invariants from these variables : $k_1(xy) + k_2(2x^2 + y^2) = 0$ with the eigenvectors associated to 2 and $k.(y^2 - 2x^2) = 0$ with the eigenvector associated to -2.

4.4 Multiple loops

In this short section, we present our method to treat non-deterministic loops, i.e. loops with non-deterministic conditions. At the beginning of each iteration, the loop can choose randomly between all its bodies. This representation is equivalent to the definition in section 2.

Definition 6 *Let $F = \{A_i\}_{1 \leqslant i \leqslant n}$ a family of matrices and $Inv(F)$ the set of invariants of a loop whose different bodies can be encoded by elements of F.*

$$Inv(F) = \{\varphi | \forall X, \varphi.X = 0 \Rightarrow \bigwedge_{i=1}^{n} \varphi.A_i.X = 0\}$$

Property 5 *Let $F = \{A_i\}_{1 \leqslant i \leqslant n}$ a family of matrices.*

$$Inv(F) = \bigcap_{i=1}^{n} Inv(A_i)$$

As the set of invariants of a single-body loop are a vectorial spaces union, its intersection with another set of invariants is also a vector space union. Although we do not consider the condition used by the program to choose the correct body, we still can discover useful invariants. Let us consider the following example, taken from [21], that computes the product of x and y in variable z :

```
while (∗) do
    (x, y, z) := (2x, (y−1)/2, x + z)
    OR
    (x, y, z) := (2x, y/2, z)
done
```

We have to deal with two applications : $f_1(x, y, z) = (2x, (y - 1)/2, x + z)$ and $f_2(x, y, z) = (2x, y/2, z)$. The elevation to the degree 2 of f_1 and f_2 returns applications having both 10 eigenvectors. For simplicity, we focus on invariants associated to the eigenvalue 1.

f_1^* has 4 eigenvectors $\{e_i\}_{i\in[1,4]}$ associated to 1 such that

- $\langle e_1, X \rangle = -x + xy$
- $\langle e_2, X \rangle = x + z$
- $\langle e_3, X \rangle = xz + x^2 + z^2$
- $\langle e_4, X \rangle = \mathbb{1}$

f_2^* also has 4 eigenvectors $\{e_i'\}_{i\in[1,4]}$ associated to 1 such that

- $\langle e_1', X \rangle = xy$
- $\langle e_2', X \rangle = z$
- $\langle e_3', X \rangle = z^2$
- $\langle e_4', X \rangle = \mathbb{1}$

First, we notice that $e_4 = e_4'$. Then, we can see that $\langle e_1 + e_2, X \rangle = xy + z = \langle e_1' + e_2', X \rangle$. Thus, $e_1 + e_2 = e_1' + e_2'$. Eventually, we find that $e_1 + e_2 + k.e_4 \in (Vect(\{e_i\}_{i\in[1,4]}) \cap Vect(\{e_i'\}_{i\in[1,4]}))$. That's why $(\langle e_1 + e_2 + k.e_4, X \rangle = 0)$ is a semi-invariant for both f_1 and f_2, hence for the whole loop. Replacing $\langle k.e_4, X \rangle$ by $k = -k'$ and $\langle e_1 + e_2, X \rangle$ by $xy + z$ gives us $xy + z = k'$.

Algorithm. The intersection of two vector spaces corresponds to the vectors that both vector spaces have in common. It means that such elements can be expressed by elements of the base of each vector space. Let B_1 and B_2 the bases of the two vector spaces. If $e \in \text{Vect}\{B_1\}$ and $e \in \text{Vect}\{B_2\}$, then $e \in \ker\{(B_1 B_2)\}$. To compute the intersection of a vector space union, we just have to compute the kernels of each combination of vector space in the union.

5 Implementation and experimentation

In order to test our method, we implemented an invariant generator as a plugin of Frama-C [14], a framework for the verification of C programs written in OCaml. Tests have been made on a Dell Precision M4800 with 16GB RAM and 8 cores. Time does not include parsing time of the code, but only the invariant computation from the Frama-C representation of the program to the formulas. Moreover, our tool doesn't implement the extension of our method and may output irrational invariants or fail on complex eigenvalues. Benchmark is available at [6]. The second column of the table 1 represents the number of variables used in the program. The third column represents the invariant degree used for PILAT and FASTIND. The last three columns are the computation time of the tools in *ms*. O.O.T. represents an aborted ten minutes computation and – indicates that no invariant is found.

All the tested functions are examples for which the presence of a polynomial invariant is compulsory for their verification. The choice of high degree for some functions is motivated by our will to show the efficiency of our tool to find high degree invariants as choosing a higher degree induces computing a bigger set of relations. In the other cases, degree is choosen for its usefulness.

For example in figure 3 we were interested in finding the invariant $x + qy = k$ for eucli_div. That's why we set the degree to 2. Let X be the vector of variables $(x, y, q, xq, xy, qy, y_2, x_2, q_2, \mathbb{1})$. The matrix A representing the loop in figure 3 has only one eigenvalue : 1. There exist 4 eigenvectors $\{e_i\}_{i\in[1;4]}$ associated to 1

Table 1: Performance results with our implementation Pilat

Program			Time (in ms)		
Name	Var	Degree	Aligator [15]	Fastind [5]	Pilat
divbin	5	2	80	6	2.5
hard	6	2	89	13	2
mannadiv	5	2	27	6	2
sqrt	4	2	33	5	1.5
djikstra	5	2	279	31	4
euclidex2	8	2	1759	10	6
lcm2	6	2	175	6	3
prodbin	5	2	100	6	2.5
prod4	6	2	13900	–	8
fermat2	5	2	30	9	2
knuth	9	3	O.O.T.	347	192
eucli_div	3	2	13	6	2
cohencu	5	2	90	5	2
read_writ	6	2	82	–	12
illinois	4	2	O.O.T.	–	8
mesi	4	2	620	–	4
moesi	5	2	O.O.T.	–	8
petter_4	2	10	19000	37	3
petter_5	2	10	O.O.T.	37	2
petter_6	2	10	O.O.T.	37	2

Input : degree = 2

```
int eucli_div(int x, int y){
    int q = 0;
    while (x > y) {
        x = x-y;
        q ++;
    }
    return q;
}
```

Frama-C output :

```
int eucli_div(int x, int y){
    int q = 0;
    int k = x + y*q;
    // invariant x + y*q = k;
    while (x > y) {
        x = x-y;
        q ++;
    }
    return q;
}
```

Fig. 3: Euclidean division C loop and generation of its associated invariants.

in A, so $\left\langle \sum\limits_{i=1}^{4} k_i e_i, X \right\rangle = 0$ is a semi-invariant. One of these eigenvectors, let's say e_1, correspond to the constant variable, i.e. $e_1.X = \mathbb{1} = 1$, thus we have $\left\langle \sum\limits_{i=2}^{4} k_i e_i, X \right\rangle = -k_1$ as invariant. In our case, $\langle e_2, X \rangle = y$, $\langle e_3, X \rangle = x + yq$ and $\langle e_4, X \rangle = y_2$. We can remove $(y = k)$ and $(y_2 = k)$ that are evident because y does not change inside the loop. The remaining invariant is $x + yq = k$.

6 Conclusion and future work

We presented a simple and effective method to generate non-trivial invariants. One of its great advantages is to only rely on linear algebra theory, and generate modular invariants. Still our method has some issues that we are currently investigating. First, it is incomplete for integers : invariants we generate are only correct for rationals. Perhaps surprisingly, this issue does not come from the invariant generation, but from the linearization procedure which badly takes into account the division. For example in C, the operation $x' = \frac{x}{2}$ with x uneven returns $\frac{x-1}{2}$. This behavior is not taken into account by the elevation, which can freely multiply this x by a variable y with $y' = 2y$. This returns the assignment $xy' = xy$ which is false if x is odd. Next, we do not treat interleaving loops as we cannot yet compose invariants with our generation technique. The tool has been successfully implemented as an independent tool of Frama-C.

Our next step is to use those invariants with the Frama-C tools Value (a static value analyser) and WP (a weakest precondition calculus API) to apply a CEGAR-loop on counter-examples generated by CaFE, a temporal logic model checker based on [1]. Also, we want the next version of the tool to handle irrational eigenvalues as decribed in section 4.3.

References

1. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS 2004*, pages 467–481, 2004.
2. S. K. Basu and J. Misra. Proving loop programs. *IEEE Trans. Software Eng.*, 1(1):76–86, 1975.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 300–309, 2007.
4. B. Botella, M. Delahaye, S. H. T. Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *4th International Workshop on Automation of Software Test, AST 2009*, pages 70–78, 2009.
5. D. Cachera, T. P. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. *Sci. Comput. Program.*, 93:89–109, 2014.
6. E. Carbonell. Polynomial invariant generation. http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html.

7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 2000*, pages 154–169, 2000.

8. K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001.

9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

10. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

11. H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *ATVA 2012*, pages 187–202, 2012.

12. D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1), 1975.

13. M. Kauers and B. Zimmermann. Computing the algebraic relations of C-finite sequences and multisequences. *J. Symb. Comput.*, 43(11):787–803, 2008.

14. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3), 2015.

15. L. Kovács. Aligator: A mathematica package for invariant generation (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008*, pages 275–282, 2008.

16. L. Kovács. A complete invariant generation approach for P-solvable loops. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009*, pages 242–256, 2009.

17. E. Mayr. *Membership in polynomial ideals over Q is exponential space complete.* Springer, 1989.

18. M. Müller-Olm and H. Seidl. Polynomial constants are decidable. In *Static Analysis, 9th International Symposium, SAS 2002*, pages 4–19, 2002.

19. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL 2004*, pages 330–341, 2004.

20. V. Y. Pan and Z. Q. Chen. The complexity of the matrix eigenproblem. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 507–516, 1999.

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

22. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2), 1972.

7 Appendix

Linearization theorem. The following justifies the linearization theorem of section

Property 1 *For every solvable polynomial mapping g, g is linearizable.*

Proof. Let $g \in \mathbb{Q}[x]^m$ be a solvable polynomial mapping. There exists a partition of variables $x = w_1 \cup ... \cup w_k$ such that

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

We proceed by induction on the size k of the partition on the variables of g. We can state that :

- If $k = 1$, $x = w_1$, then $g_{w_1}(x) = M_1 w_1^T + P_1$, where P_1 is a constant. Then it is clear that g_{w_1} is an affine transformation.
- Assume we can compute a linear application f from g such that $(g(x), P(g(x))) = f(x, P(x))$ if there exists a partition of k sets of variable satisfying the solvable hypothesis. Let h be a solvable polynomial mapping for which there exists a partition of x into $k + 1$ subvectors of variables $x = w_1 \uplus ... \uplus w_{k+1}, w_i \cap w_j = \emptyset$ if $i \neq j$. By induction hypothesis, we can linearize h_{w_i} for $1 \leqslant i \leqslant k$. Now, the key point is to find a way to linearize

$$h_{w_{k+1}}(x) = M_{k+1} w_{k+1}^T + P_{k+1}(w_1, ..., w_k)$$

First, let's note that no variable of w_{k+1} have been used in any other h_i. Let $v = \prod_{i=0}^{n} v_i^{\lambda_i}$ a product of variables in $(w_1 \cup ... \cup w_k)$. It can appear in P as v^d, where d in an integer. We know, by induction hypothesis, that the evolution of v_i following the h transformation can be expressed as a linear application f with the help of extra variables.

Lemma 1 *Let $g : \mathbb{K}^n \mapsto \mathbb{K}^n$ a linear application. There exists a linear application $f : \mathbb{K}^m \mapsto \mathbb{K}^m$ with $m \geqslant n$ such that for all $k \in \mathbb{N}$ and $P \in (\mathbb{K}[X])^{m-n}$, $X' = g^k(X) \Rightarrow (X', P(X')) = f^k(X, P(X))$*

Proof. First, let's prove it for $n = 2$ and $P(x, y) = (x.y, x^2, y^2)$. Let $g(x, y) = (ax + b_x y, a_y x + b_y y)$. Then, $(x.y)' = a_x.a_y x^2 + (a_x.b_y + a_y.b_x)x.y + b_x.b_y.y^2$, which is a linear combination of x^2, y^2 and xy. Similarly, $(x^2)'$ and $(y^2)'$ can be expressed as linear combinations of x^2, y^2 and $x.y$.
Next, if the degree of the polynomial is 1, one just need to take $f = g$. To generalize this proof for all degrees polynomials, let's assume that for $d \in N$, the lemma is true with P of degree d. To express a polynomial of degree $d + 1$, one need to express all monomials m_{d+1} of degree $d + 1$, which is the product of a variable x and a monomial m_d of degree d. By hypothesis, the monomial of degree d is expressible in g as a linear combination of lower or equal degree monomials. Considering m_d as a variable of g, the variable for which we seek a linear representation is the product of the two variables x and m_d, a case which has been treated before. □

We can then use Lemma 1 to linearize all monomials of elements of $w_1, ..., w_k$ occuring in P. In the end, we can write h as a linear application over the initial variables and the auxiliary variables introduced by the linearization. $\square$

Complexity. The following justifies the complexity theorem of elevation in section 4.1

Property 2 *Every solvable polynomial mapping g using n variables is lineariz-able by a linear mapping f using at most $\binom{n+d}{d}$ new variables, where d is the degree of P, the polynomial linearizing g as in definition 2.*

Proof. This property comes from [19]. $\binom{n+d}{d}$ is the total number of variables needed to express all monomials products of n variables of degree lower or equal to d.

Invariants set. The following justifies the completude of the invariant generation procedure in section 4.2

Theorem 1 $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$ *is a semi-invariant if and only if $\exists \lambda \in \mathbb{K}, \exists \varphi \in E_\lambda$, where $E_\lambda = ker(f^* - \lambda Id)$.*

Proof. This comes from a well known linear algebra result :

Lemma 2 *Let $\mathbb{K}$ be a field, E be a $\mathbb{K}$*vectorial space, F a sub-$\mathbb{K}$ vectorial space of E and $f : E \to E$ a linear application.*
$f(F) \subset F \Leftrightarrow f^*(F^\perp) \subset F^\perp$

Proof. By linear algebra, we have $< f(x), x' > = < x, f^*(x') >$. Let $x \in F, x' \in F^\perp$. If $f(F) \subset F$, then $< f(x), x' > = 0$, so $< x, f^*(x') > = 0$.
As we have $f^*(F^\perp) \subset F^\perp$, we conclude by seeing $(F^\perp)^\perp = F$ and $(f^*)^* = f$. $\square$

Let $Vect(X)$ the vectorial space generated by X. Let φ a semi-invariant, so $(\varphi.x = 0 \Rightarrow \varphi(f(x)) = 0)$. This means that $Vect(\varphi)^\perp$ is stable by f, so by property 2, $(Vect(\varphi)^\perp)^\perp = Vect(\varphi)$ is stable by f^*. As $\varphi \in Vect(\varphi)$, we have $f^*(\varphi) = k * \varphi$. $\square$

Constant deplacement. The following justifies the operation of preserving the equivalency of two invariants by moving constants in section 4.2.

Property 3 *Let $\mathcal{F}$ a semi-invariant expressed as $\mathcal{F} = \sum\limits_{i=0}^{n} k_i e_i$.*

If $\langle e_0, X_{init} \rangle \neq 0$, then we have that

$$\bigwedge_{i=1}^{n} (\langle e_i, X \rangle = -\frac{\langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle} \langle e_0, X \rangle) \text{ is an invariant} \Leftrightarrow \langle \mathcal{F}, X_{init} \rangle = 0$$

Proof.

$$\langle \mathcal{F}, X_{init} \rangle = 0 \Leftrightarrow \left\langle \sum_{i=1}^{n} k_i e_i, X_{init} \right\rangle = -k_0 \langle e_0, X_{init} \rangle$$

$$\Leftrightarrow \sum_{i=1}^{n} k_i \langle e_i, X_{init} \rangle = -k_0 \langle e_0, X_{init} \rangle$$

$$\Leftrightarrow \frac{\sum_{i=1}^{n} k_i \langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle} = -k_0$$

Let $g(c_1, ..., c_n) = -\frac{\sum_{i=1}^{n} c_i . \langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle}$. We have $g(u_i) = -\frac{\langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle}$ We note that g is linear, thus :

$$\langle \mathcal{F}, X_{init} \rangle = 0 \Leftrightarrow g(k_1, ..., k_n) \langle e_0, X_{init} \rangle = \sum_{i=1}^{n} k_i \langle e_i, X_{init} \rangle$$

$$\Rightarrow \bigwedge_{i=1}^{n} (\langle e_i, X \rangle = -\frac{\langle e_i, X_{init} \rangle}{\langle e_0, X_{init} \rangle} \langle e_0, X \rangle)$$

by setting $k_i = 1$ and $k_j = 0$ for all $j \neq i$.

Now to prove that this transformation does not make us lose precision, we will construct $\mathcal{F}$ with the n equations.

If $\bigwedge_{i=1}^{n} (\langle e_i, X \rangle = g(u_i) \langle e_0, X \rangle)$, then as g is a linear application we have that

$$\sum_{i=1}^{n} k_i \langle e_i, X \rangle = g(k_1, ..., k_n) \langle e_0, X \rangle$$

We conclude by setting k_0 to $-g(k_1, ..., k_n)$

$\square$

Extension of the method. The following justifies the theorema of construction of rational eigenvalue in section .

Property 4 *Let $A \in \mathcal{M}_d(\mathbb{Q}), \Lambda(M)$ the eigenvalue set of a matrix M and d an integer. Then for any product p of d or less elements of $\Lambda(A), p \in \Lambda(\Psi_d(A))$.*

Proof. First of all, we will prove some simple properties about Ψ_d.

Lemma 3

1. $\Psi_k(A.B) = \Psi_k(A).\Psi_k(B)$
2. $\Psi_k(A^{-1}) = \Psi_k(A)^{-1}$

Proof. 1. $\Psi_k(A).\Psi_k(B)p(X) = \Psi_k(A).p(B.X) = p(A.B.X) = \Psi_k(A.B)p(X)$
2. $\Psi_k(A^{-1}).\Psi_k(A).p(X) = p(A.A^{-1}X) = p(X)$ so $\Psi_k(A^{-1}).\Psi_k(A) = Id$.
$\square$

To deal with this, let's consider J the Jordan normal form of A. As we are working with $\mathbb{C}$, which is an algebraically closed field, A is similar to J (ie. $\exists P.A = P^{-1}JP$), with

$$
J = \begin{pmatrix} J_1 & 0 & ... & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & ... & 0 & J_k \end{pmatrix}, \text{ and } J_k = \begin{pmatrix} \lambda_k & 1 & ... & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & ... & 0 & \lambda_k \end{pmatrix}
$$

As $A = P^{-1}JP$, $\Psi_k(A) = \Psi_k(P)^{-1}.\Psi_k(J)\Psi_k(P)$, so $\Psi_k(A)$ is similar to $\Psi_k(J)$. This means that they have the same set of eigenvalues. The transformation of a variable x by J can be either of the form $\lambda.x$, either $\lambda.x + y$, with λ an eigenvalue of J and y another variable. Another interesting information from the jordan normal form is that this variable y does not depend on x. To understand the proof, let's see what happens for $d = 2$ and $n = 2$.

If λ and λ' are eigenvalues of A, then $J = \begin{pmatrix} \lambda & b \\ 0 & \lambda' \end{pmatrix}$ with $b = 0$ or 1 . So

$$
\Psi_2(J) = \begin{pmatrix} \lambda^2 & 2\lambda\lambda' & \lambda'^2 & 0 & 0 \\ 0 & \lambda\lambda' & b\lambda' & 0 & 0 \\ 0 & 0 & \lambda'^2 & 0 & 0 \\ 0 & 0 & 0 & \lambda & b \\ 0 & 0 & 0 & 0 & \lambda' \end{pmatrix}.
$$

$\Psi_2(J)$ is upper triangular, so its eigenvalues are on the diagonal : $\lambda, \lambda', \lambda^2, \lambda\lambda'$ and λ'^2

The proof for any number of variables and any degree is similar. We just need to analyse the elevation of the Jordan normal form of A.

Definition 7 *Let f a linear application. We define a dependency order $\prec_f$ on Var a total order such that for all $x \in Var$, $f(X)$ restricted to x depends only on a linear combination of variables V for which $\forall y \in V, y \preceq_f x$. We also say that f respects $\prec_f$.*

The idea behind this is that an upper-triangular matrix J induces such an order : the last element x_n only depend on himself, the previous element x_{n-1} depends on himself and x_n, so $x_n \prec_A x_{n-1}$, etc.. We note that for any triangular matrix, the diagonal coefficients are the eigenvalues of the said matrix and of all similar matrix.

In other words, if for the ith variable of an application, the line of its matrix is composed of $i - 1$ zeros then λ at the ith position, then its eigenvalue is λ.

We define $\prec_J$ as an order that J respects. By choosing a right order for monomials of p, we will show that $\Psi_k(J)$ is upper-triangular. We will take the graded lexicographic order $\prec_J^g$ with respect to $\prec_J$, i.e. the order such that :

- if $x \prec_J y$ two variables, then $x \prec_J^g y$;
- if $m_1 \prec_J^g m_2$, then for any monomial m_3, $m_1.m_3 \prec_J^g m_2.m_3$.

If $x \prec_J^g y$ then by definition $x \prec_J y$. Moreover, $m_1 \prec_J^g m_2 \Leftrightarrow (m_1$ does not appear in the expression of m_2), then let m_3 any monomial. We can clearly see that $m_1.m_3$ does not appear in the expression of $m_2.m_3$. Thus :

Lemma 4 *Let J an upper triangular matrix, $\prec_J$ be a dependency order respected bu J. Then for any $k \in \mathbb{N}$, $\prec_J^g$ can be a dependency order for $\Psi_k(J)$.*

By our construction of the linearization process of 4.1, we can state that for all $x, y, z \in Var$, if $x \prec_J y \preceq_J z$, then $y^i z^j \prec_J^g x^{i+j}$ is impossible for all i, j as x does not depend on y or z, but the contrary is false. Thus :

Lemma 5 *Let f a linear application, $\prec_f$ a dependency order on f, $x, y \in Var$. Let $Mon_d(x, y)$ the set of variables representing monomials of degree lower or equal to d depending on x and y. Then $z \prec_f x \prec_f y \Rightarrow v \in Mon_d(x, y), z^d \preceq_f^g v$.*

In other words, in a triangular matrix M representing x, y, z and its monomials, if x and y are over z and M respects $\prec_M^g$, then $x^i.y^j$ will always be over z^k, for every i, j, k with $k \leqslant i + j$.

Let x, y, z, t variables such that $x' = \lambda_x.x + j_z z$, and $y' = \lambda_y.y + j_t t$ where j_z and $j_t = 0$ or 1. Then the matrix $\Psi_k(J)$ will set $x^i.y^j$ to $(\lambda_x.x + j_z z)^i (\lambda_y.y + j_t t)^j$.

Eventually, when one develop $x^i.y^j$, there is :

- 0 for *monomial variables* of strictly higher degree ;
- 0 for *monomial variables* containing t such that $t \prec x \prec y$ by lemma 5 ;
- 0 for monomials $x^{i'}.y^{j'}$ with $i' + j' = d$ and $i' > i$;
- a coefficient $\lambda_x^i.\lambda_y^j$ for the variable $x^i.y^j$, which will be on the diagonal of $\Psi_k(J)$.

The third point is true because if $x \prec_J y$, then $x^i y^j \prec_J^g x^{i-1} y^{j+1}$ and $\prec_J^g$ is a dependency order for $\Psi_k(J)$ 4.

$\Psi_k(J)$ will be upper-triangular itself by respecting $\prec_J^g$.

$\square$

Multiple loop. The following justifies the multiple loop proposition of section 4.3

Property 5 *Let $F = \{A_i\}_{1 \leqslant i \leqslant n}$ a family of matrices.*

$$Inv(F) = \bigcap_{i=1}^{n} Inv(A_i)$$

Proof.

Lemma 6

Let $F = \{A_i\}_{1 \leqslant i \leqslant m}, G = \{B_i\}_{1 \leqslant i \leqslant n}$ two matrix families. Then $Inv(F \cup G) = Inv(F) \cap Inv(G)$

Proof. As we have $((p \Rightarrow q) \wedge (p \Rightarrow r)) \Leftrightarrow (p \Rightarrow (q \wedge r))$,

$$Inv(F) \cap Inv(G) = \{\varphi | \forall X, \begin{matrix} (\varphi.X = 0 \Rightarrow \bigwedge_{i=1}^{m} \varphi.A_i.X = 0) \\ \wedge (\varphi.X = 0 \Rightarrow \bigwedge_{i=1}^{n} \varphi.B_i.X = 0) \end{matrix} \}$$

$$= \{\varphi | \forall X, \varphi.X = 0 \Rightarrow (\bigwedge_{i=1}^{m} \varphi.A_i.X = 0 \wedge \bigwedge_{i=1}^{n} \varphi.B_i.X = 0)\}$$

$$= Inv(F \cap G)$$

□

By recurrence over the size n of a family F, if n equals 1 it is true. If it is true for a certain n, then $Inv(F \cup \{A\}) = Inv(F) \cap Inv(\{A\})$ by the previous lemma.

□