Acceleration of the Abstract Fixpoint Computation in Numerical Program Analysis

Olivier Bouissou

CEA LIST, Laboratory for the Modeling and Analysis of Interacting Systems
91191 Gif-sur-Yvette, France

Yassamine Seladji

CEA LIST, Laboratory for the Modeling and Analysis of Interacting Systems
91191 Gif-sur-Yvette, France

Alexandre Chapoutot

ENSTA Paristech - UEI
32, boulevard Victor 75739 Paris cedex 15, France

Abstract

Static analysis by abstract interpretation aims at automatically proving properties of computer programs, by computing invariants that over-approximate the program behaviors. These invariants are defined as the least fixpoint of a system of semantic equations and are most often computed using Kleene iteration. This computation may not terminate so specific solutions were proposed to deal with this issue. Most of the proposed methods sacrifice the precision of the solution to guarantee the termination of the computation in a finite number of iterations. In this article, we define a new method which allows to obtain a precise fixpoint in a short time. The main idea is to use numerical methods designed for accelerating the convergence of numerical sequences. These methods were primarily designed to transform a convergent, real valued sequence into another sequence that converges faster to the same limit. In this article, we show how they can be integrated into Kleene iteration in order to improve the fixpoint computation in the abstract interpretation framework. An interesting feature of our method is that it remains very close to Kleene iteration and thus can be easily implemented in existing static analyzers. We describe a general framework and its application to two numerical abstract domains: the interval domain and the octagon domain. Experimental results show that the number of iterations and the time needed to compute the fixpoint undergone a significant reduction compared to Kleene iteration, while its precision is preserved.

Key words: Abstract numerical domains, acceleration of convergence, numerical sequence transformation, widening operator with thresholds.

1. Introduction

In the field of static analysis of numerical programs, abstract interpretation [11, 13] is widely used to compute over-approximations of the set of behaviors of programs. This set is usually defined as the least fixpoint of a monotone map on an abstract domain given by the (abstract) semantics of the program. Using Tarski's theorem [39], this fixpoint is computed as the limit of the iterates of an abstract function starting from the least element. These iterates build a sequence of abstract elements that (order theoretically) converges towards the least fixpoint. Since this sequence may converge slowly (or only after infinitely many steps), the theory of abstract interpretation introduces the concept of *widening* [13].

Briefly, a widening operator ∇ is a two-argument function which tries to predict the limit of the iterates based on the relative position of two consecutive iterates. For example, if these two iterates increased the widening operator replaces them per $+\infty$ ($-\infty$ for the decreasing cases). The comparison's criterion between two iterates depend on the abstract domain used for the analysis. A widening operator often makes large over-approximation because it must make the sequence of iterates converge in a finite time. Over-approximation may be reduced afterward using a *narrowing* operator but the precision of the final approximation still strongly depends on the precision of operator ∇. Various techniques have been proposed to improve it. *Delayed* widening, this method applies ∇ operator after n iteration steps only (where n is a user-defined integer). *Widening with thresholds* [2] is an other technique, where a user defines staticly a set of thresholds. These thresholds are successively used like a candidate of a fixpoint value. However, they suffer from their lack of automatizing: thresholds must be chosen *a priori* and are defined by the user. Some methods try to automatically discover thresholds from the program syntax [25, 37]: whenever an inequality (e.g. the condition of a loop) is found, a threshold (or a landmark in [37]) is added, its value depending on the constants appearing in the inequality. So the thresholds are based on a *syntactic* criterion; in our work we define thresholds automatically using the *dynamics* of the program variables. To do so, we propose the use of *numerical analysis methods* which allow to exploit the numerical properties of the program variables.

In particular, we show that it is possible to use *sequence transformation techniques* in order to automatically and efficiently derive an approximation to the limit of Kleene iterates. This approximation may not be safe (i.e. may not contain the actual limit), but we show how to use it in the context of abstract interpretation. Sequence transformation techniques (also known as convergence acceleration methods) are widely studied in the field of numerical analysis [6]. They transform a converging sequence $(x_n)_{n\in\mathbb{N}}$ of real numbers into a new sequence $(y_n)_{n\in\mathbb{N}}$ which converges faster to the same limit. In some cases (depending on the method), the acceleration is such that $(y_n)_{n\in\mathbb{N}}$ is ultimately constant. Some recent work [8] applied these techniques in the case of sequences of *vectors* of real numbers: vector sequence transformations introduce *relations* between elements of the vector and perform better than scalar ones.

Our main contribution is to show that we can use the sequence transformation methods in order to improve the fixpoint computation in static analysis, especially to analyse numerical programs with floating-point variables. This is particularly interesting because

Email addresses: `olivier.bouissou@cea.fr` (Olivier Bouissou), `yassamine.seladji@cea.fr` (Yassamine Seladji), `alexandre.chapoutot@ensta-paristech.fr` (Alexandre Chapoutot).

this kind of prorams often slowly reach their fixpoint, as all the bits of the representation of floating-point variables must stabilize before the fixpoint is reached. Our idea is to define *dynamic* thresholds for widening that are very close to the actual fixpoint. This increased precision is obtained because sequence transformations use the information given by the values of all the iterates (not only the values of the two latest iterates like for the operator ∇) and *quantitative information* (i.e. relative to the distance between elements) to predict the limit. They thus exploit more information than the widening and make a better prediction. In this article, we present the extension of the work begun in [4]; besides the domain of intervals, we apply it to the relational domain of octagons. We believe that this work may be used with any abstract domain, especially the ones with a *pre-defined shape* (e.g. templates [35]). Let us remark that our techniques are well-suited for accelerating the invariant generation of numerical programs with floating-point variables and that we do not address the case of integer variables as in [17, 28].

This article is organized as follows. In Section 2, we explain on a simple example using the interval abstract domain how numerical acceleration methods may be used to speed-up the abstract fixpoint computation. In Section 3, we recall the theoretical basis of this work. We present our main theoretical contribution and its applications with the interval domain and the octagon domain in Section 4 and Section 5. Section 6 presents some experimental results on various floating-point programs that show the interest of our approach, while Sections 7 and 8 discuss related works and perspectives.

Notations. In the rest of this article, (x_n) will denote a sequence of real numbers (i.e. $(x_n) \in \mathbb{R}^{\mathbb{N}}$), while $(\vec{x}_n)$ denotes a sequence of vector of real numbers (i.e. $(\vec{x}_n) \in \left(\mathbb{R}^p\right)^{\mathbb{N}}$ for some $p \in \mathbb{N}$). The symbol $\vec{\mathcal{X}}_n$ will be used to represent *abstract iterates*, i.e. $\vec{\mathcal{X}}_n \in A$ for some abstract lattice A. In the interval abstract domain, $\vec{\mathcal{X}}_n$ is represented by the interval $[\underline{x}_n, \overline{x}_n]$ such that $-\infty \le \underline{x}_n \le \overline{x}_n \le +\infty$.

2. An introductive example

In this section, we explain, using a simple example, how sequence acceleration techniques can be used in the context of static analysis, applying it to the interval abstract domain. In short, our method works as follows:

- Let $(\vec{\mathcal{X}}_n)$ be a sequence of intervals computed by the Kleene iteration and that is chosen to be widened (see [5] for details on how to choose the widening points).
- From $(\vec{\mathcal{X}}_n)$ we extract a vector sequence $(\vec{x}_n)$: at stage k, $\vec{x}_k$ is a vector that contains the infimum and supremum of each variable of the program. As Kleene iteration converges towards the least fixpoint of the abstract transfer function, the sequence $(\vec{x}_n)$ converges towards a limit $\vec{x}$ which is the vector containing the infimum and the supremum of this fixpoint.
- We then compute an accelerated sequence $(\vec{y}_n)$ that converges towards $\vec{x}$ faster than $(\vec{x}_n)$. Once this sequence has reached its limit (or is sufficiently close to it), we use $\vec{x}$ as a threshold for a widening on $(\vec{\mathcal{X}}_n)$ and thus obtain, in a few steps, a post-fixpoint (a proof is given in Section 4).

In the rest of this section, we detail these steps.

```
 1  void main() {
 2    float x1,x2,x3,xn1,xn2,xn3,u1,u2,u3;
 3  /* 1 <= x1 <= 2, 1 <= x2 <= 4, 1 <= x3 <= 20 */
 4  /* 1 <= u1 <= 6, 1 <= u2 <= 3, 1 <= u3 <= 2  */
 5    while(1) {
 6      xn1 = −0.4375 * x1+ 0.0625 * x2 + 0.2652 * x3 + 0.1 * u1;
 7      xn2 = 0.0625 * x1 + 0.4375 * x2 + 0.2652 * x3 + 0.1 * u2;
 8      xn3 = −0.2652 * x1 + 0.2652 * x2 + 0.375 * x3 + 0.1 * u3;
 9      x1=xn1; x2=xn2; x3=xn3;
10    }
11  }
```

Fig. 1. `contraction.c`, a simple linear program.

The program. We consider a program which iterates the function $F(X) = A \cdot X + B \cdot U$ where A, B and U are constant matrices and X is the vector of variables (see Figure 1). Similar programs are very common for industrial use: Newton or Gauss-Seidl iterations for solving a linear equation are special cases. Here, this program actually serves to estimate (using Euler's method) the evolution of a linear dynamical system, i.e. the solution of a differential equation of the form $\frac{dX}{dt} = A'.X(t) + U'(t)$ where $A' = 10(A - I)$ and $\forall t,\ U'(t) = U$. Initially, we have:
$\texttt{x1} \in [1,2], \texttt{x2} \in [1,4], \texttt{x3} \in [1,20], \texttt{u1} \in [1,6], \texttt{u2} \in [1,4]$ and $\texttt{u3} \in [1,2]$.

Using the interval abstract domain, the analysis of this program converges in 127 iterations (without using a widening operator) and we obtain for Line 5 the following invariant:
$\texttt{x1} \in [-5.1975, 8.8733]$, $\texttt{x2} \in [-2.6244, 11.1263]$ and $\texttt{x3} \in [-4.7187, 20]$.

Extracting the sequence. From this program, we can define a vector sequence of dimension 6, $\vec{x}_n = \left(\underline{x}_n^1, \overline{x}_n^1, \underline{x}_n^2, \overline{x}_n^2, \underline{x}_n^3, \overline{x}_n^3\right)$, which represents the evolution of the supremum and the infimum of each variable `x1`, `x2` and `x3` at Line 5. Note that we are not interested in the formal definition of these sequences, but only in their numerical values that are dynamically extracted from Kleene iterates. Each sequence $(\overline{x}_n^i)$ (respectively $(\underline{x}_n^i)$) is increasing (respectively decreasing) and the sequence $(\vec{x}_n)$ converges towards a vector $\vec{x}$ containing the infima and the suprema of the fixpoint. For example, the evolutions of the sequences $(\underline{x}_n^1)$ and $(\overline{x}_n^1)$ are given in the table of Figure 2 (lines labelled by $\overline{\texttt{x1}}$ and $\underline{\texttt{x1}}$ respectively). We see that the convergence is very slow because the last digits are very slow to converge: the first digit converges after 10 iterations, the first 3 after 50 iterations and all of them after 127 iterations.

Accelerating the sequence. We then used the *vector ε-algorithm* [8] to build a new sequence that converges faster towards $\vec{x}$. This method works as follows (a more formal definition will be given in Section 3.2): it computes a series of sequences $(\vec{\varepsilon}_n^{\,k})$ for $k = 1, 2, \ldots$ such that each sequence $(\vec{\varepsilon}_n^{\,k})$ for k *even* converges towards $\vec{x}$ and the *diagonal* $(\vec{d}_n) = (\vec{\varepsilon}_0^{\,2n})$ also converges towards $\vec{x}$. This diagonal sequence is the result of the ε-algorithm and it is called the *accelerated sequence.* It converges faster than the original sequence: it is very close to the limit after only 6 elements (which require 11 elements of the original sequence, as it will be explained later). For example, the evolutions of the accelerated sequences associated with $(\underline{x}_n^1)$ and $(\overline{x}_n^1)$ are given in the table of Figure 2

Iterate	1	2	3	4	5	10	20	50	127
$\overline{\texttt{x1}}$	2.00	5.71	6.57	7.49	7.87	8.67	8.86	8.8728	8.8733
$\underline{\texttt{x1}}$	1.00	-0.44	-2.29	-3.03	-3.69	-4.86	-5.17	-5.1971	-5.1975
$\overline{y_1}$	2.00	4.94	8.03	8.83	8.87	8.8733	-	-	-
$\underline{y_1}$	1.00	-3.89	-4.25	-5.18	-5.19	-5.1975	-	-	-

Fig. 2. Sequence extracted from the program of Figure 1 for variable `x1` and its accelerated version (a cell with a '-' means the value is the same as in the previous cell).

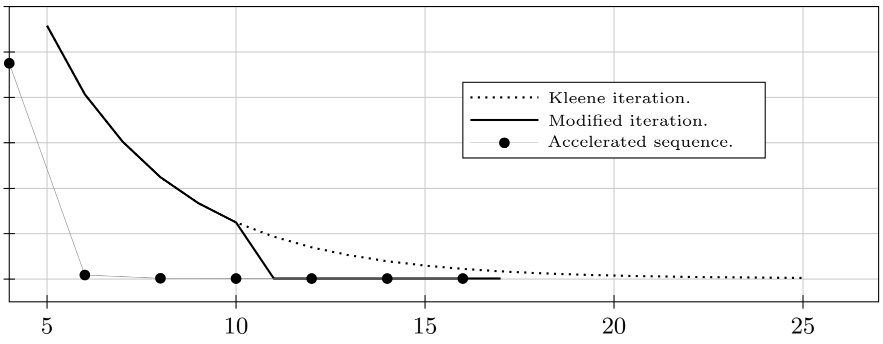


Fig. 3. Infimum value of `x1`. We only display the iterates 5 to 25. At the 11th iteration, the accelerated value is used as a threshold for widening, and the iteration stops after 17 steps.

(lines labelled with $\overline{y_1}$ and $\underline{y_1}$ respectively). We see that these sequences are much faster to converge than the original sequences: they are constant after the 10th term.

Using the accelerated sequence. When the accelerated sequence reaches the limit (or is sufficiently close to it), we modify the Kleene iteration and we use this limit as a good fixpoint candidate. Formally, if the limit is $(\underline{y}_1, \overline{y}_1, \underline{y}_2, \overline{y}_2, \underline{y}_3, \overline{y}_3)$ and if the current Kleene iterate is $\vec{\mathcal{X}_p}$, we construct the abstract element $\vec{\mathcal{Y}}$ whose bounds are $\underline{y}_1$, $\overline{y}_1, \dots$ and set $\vec{\mathcal{X}}_{p+1} = \vec{\mathcal{X}_p} \dot{\sqcup} \vec{\mathcal{Y}}$ and re-start Kleene iteration from $\vec{\mathcal{X}}_{p+1}$. In this way, we remain sound $(\vec{\mathcal{X}_p} \dot{\sqsubseteq} \vec{\mathcal{X}}_{p+1})$ and we are very close to the fixpoint, as $\vec{\mathcal{Y}} \dot{\sqsubseteq} \vec{\mathcal{X}}_{p+1}$ (note that $\dot{\sqsubseteq}$ and $\dot{\sqcup}$ are, respectively, the inclusion and the union operators over each element of $\vec{\mathcal{Y}}$, see Section 3.1 for more details). If necessary, we can apply the accelerated method after each Kleene iteration until we reach the fixpoint or a post-fixpoint. In this example, modified Kleene iteration stopped after 17 steps and reached a post-fixpoint, which is very close to the one obtained without widening and acceleration. Figure 3 shows the Kleene iteration and the modified one, for the infimum of variable `x1`. Let us recall that the Kleene iteration needed 127 steps to converge, where the modified iteration stops after 17 steps.

3. Theoretical frameworks

On the one hand in Section 3.1, we briefly recall the basics of abstract interpretation, with an emphasis on the widening operator. On the other hand in Section 3.2, we present the theory of sequence transformations in more details.

3.1. Overview of abstract interpretation-based static analysis

As mentioned in Section 1, we only consider in this article numerical programs for which we want to compute numerical invariants, that is the set of values reached by each variable from the set of possible inputs. In our case, numerical invariants are given by an abstract interpretation-based static analysis.

In this section, we recall the main features of the theory of abstract interpretation [11, 12] by considering a small imperative language. For completeness reason, we will detail the construction of an abstract interpretation of this small imperative language while our method is independent of the concrete syntax and is defined on the abstract semantics only.

The considered language is defined by the BNF specification given in Equation (3.1). It is composed of arithmetic expressions e made of variables v taken from a finite set $\mathcal{V}$, constant values $c \in \mathbb{R}$ and arithmetic operations among addition, subtraction, multiplication, division and square root. It is also made of Boolean expressions b that is the constant `true`, the constant `false`, comparison operations between a variable x and a constant c, and the negation. Finally, the language contains instructions i made of assignments, sequence of instructions, branching instructions and loops.

$$\begin{aligned} e &::= v \mid c \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2 \mid \sqrt{e_1} \\ b &::= \texttt{true} \mid \texttt{false} \mid x \le c \mid x < c \mid x \ge c \mid x > c \mid \neg b \\ i &::= x = e \mid i_1 ; i_2 \mid \text{if } (b) \ \{\, i_1 \,\} \text{ else } \{\, i_2 \,\} \mid \text{while } (b) \ \{\, i_1 \,\} \end{aligned} \tag{3.1}$$

To each program P written in this language, we can associate a system of semantic equations [10] denoted by F. Note that this system is equivalent to the representation of program semantics by a discrete transition system associated to an operational semantics. Before defining the construction rules of semantic equations, we introduce the set $\mathcal{L}$ of syntactic control points representing points of interest to compute numerical invariants. Each instruction is enclosed by two elements of $\mathcal{L}$. Moreover, we introduce environments $\sigma : \mathcal{V} \to \mathbb{R}$ mapping a real value $r \in \mathbb{R}$ to each variable $v \in \mathcal{V}$. The set of all environments σ will be denoted by Σ.

The rules to build semantic equations from the syntax and the set $\mathcal{L}$ are given in Figure 4(c). These semantic equations are defined from the semantics of arithmetic expressions defined in Figure 4(a), and the transfer functions defined in Figure 4(b), that is the semantics of basic instructions which are assignments and Boolean expressions.

More precisely, the (*concrete*) semantics of arithmetic expression $[\![.]\!]^\natural_{\mathcal{A}}$ associates to each arithmetic expression a real value r from a given environment σ. In Figure 4(a), $[\![\cdot]\!]^\natural_{\mathcal{A}}$ is defined inductively on the structure of arithmetic expressions and from a given environment σ. The semantics of a constant c is itself for all environments, the semantics of a variable x is its value in σ and the semantics of an arithmetic operation $\diamond$ is the result of the operation $\diamond$ on the results of the interpretation by $[\![.]\!]^\natural_{\mathcal{A}}$ of the two arithmetic sub-expressions. It works in the same way for the square root operation. Remark that in case of division and square root the validity of the results depends on the value taken by the operands. In consequence when an error happen we assume that the analysis will be stopped.

Furthermore, the (concrete) transfer functions $[\![.]\!]^\natural_{\mathcal{T}}$, defined in Figure 4(b), associate the set of environments corresponding to the execution of a basic instruction from a set of environments $S \subseteq \Sigma$. For an assignment $x = e$, the transfer function updates the

$$
[\![e]\!]^{\natural}_{\mathcal{A}}(\sigma) = \begin{cases} c & \text{if } e \equiv c \\ \sigma(x) & \text{if } e \equiv x \\ [\![e_1]\!]^{\natural}_{\mathcal{A}}(\sigma) \diamond [\![e_2]\!]^{\natural}_{\mathcal{A}}(\sigma) & \text{if } e \equiv e_1 \diamond e_2 \text{ with } \diamond \in \{+,-,\times\} \\ [\![e_1]\!]^{\natural}_{\mathcal{A}}(\sigma) \div [\![e_2]\!]^{\natural}_{\mathcal{A}}(\sigma) & \text{if } e \equiv e_1 \div e_2 \text{ and } [\![e_2]\!]^{\natural}_{\mathcal{A}}(\sigma) \neq 0 \\ \sqrt{[\![e_1]\!]^{\natural}_{\mathcal{A}}(\sigma)} & \text{if } e \equiv \sqrt{e_1} \text{ and } [\![e_1]\!]^{\natural}_{\mathcal{A}}(\sigma) \geq 0 \end{cases}
$$

(a) Semantics of arithmetic expressions.

$$
[\![x = e]\!]^{\natural}_{\mathcal{T}}(S) = \Big\{ \sigma[x \leftarrow v] : \sigma \in S,\ v = [\![e]\!]^{\natural}_{\mathcal{A}}(\sigma) \Big\}
$$

$$
[\![b]\!]^{\natural}_{\mathcal{T}}(S) = \begin{cases} \{\sigma : \sigma \in S,\ [\![x]\!]^{\natural}_{\mathcal{A}}(\sigma) \star [\![c]\!]^{\natural}_{\mathcal{A}}(\sigma) = \texttt{true}\} & \text{if } b \equiv x \star c,\ \star \in \{<, \leq, >, \geq\} \\ S & \text{if } b \equiv \texttt{true} \\ \emptyset & \text{if } b \equiv \texttt{false} \\ \Sigma \setminus [\![b_1]\!]^{\natural}_{\mathcal{T}}(S) & \text{if } b \equiv \neg b_1 \end{cases}
$$

(b) Transfer functions.

$$
\frac{\text{①}\ x = e\ \text{②}}{\mathcal{X}^{\natural}_{\text{②}} = [\![x = e]\!]^{\natural}_{\mathcal{T}}(\mathcal{X}^{\natural}_{\text{①}})}
\qquad
\frac{\text{① if } (b)\{\text{②}\ i_1\ \text{③}\} \text{ else } \{\text{④}\ i_2\ \text{⑤}\}\ \text{⑥}}{\begin{cases} \mathcal{X}^{\natural}_{\text{②}} = [\![b]\!]^{\natural}_{\mathcal{T}}(\mathcal{X}^{\natural}_{\text{①}}) \\ \mathcal{X}^{\natural}_{\text{④}} = [\![\neg b]\!]^{\natural}_{\mathcal{T}}(\mathcal{X}^{\natural}_{\text{①}}) \\ \mathcal{X}^{\natural}_{\text{⑥}} = \mathcal{X}^{\natural}_{\text{③}} \cup \mathcal{X}^{\natural}_{\text{⑤}} \end{cases}}
\qquad
\frac{\text{① while ② } (b)\{\text{③}\ i_1\ \text{④}\}\ \text{⑤}}{\begin{cases} \mathcal{X}^{\natural}_{\text{③}} = [\![b]\!]^{\natural}_{\mathcal{T}}(\mathcal{X}^{\natural}_{\text{②}}) \\ \mathcal{X}^{\natural}_{\text{⑥}} = [\![\neg b]\!]^{\natural}_{\mathcal{T}}(\mathcal{X}^{\natural}_{\text{②}}) \\ \mathcal{X}^{\natural}_{\text{②}} = \mathcal{X}^{\natural}_{\text{①}} \cup \mathcal{X}^{\natural}_{\text{④}} \end{cases}}
$$

(c) Semantic equations.

Fig. 4. Semantic model.

values of the variable x with all the possible results v of e, which is denoted by $\sigma[x \leftarrow v]$, evaluated in all the environments $\sigma \in S$. For a Boolean expression $x \star c$, the transfer function acts like a filter which only keeps the environments satisfying the predicate $x \star c$ or in case of a constant Boolean value t it keeps S if $t = \texttt{true}$ and it produces an empty set if $t = \texttt{false}$. The result of the negation expression $\neg b_1$ is the complement set of the interpretation of b_1.

Semantic equations, defined in Figure 4(c), give a constructive way to compute the set of environments $\mathcal{X}^{\natural}_{\odot}$ at each control point $\odot$ of P. This set $\mathcal{X}^{\natural}_{\odot}$ is the invariant associated to the execution of each basic instruction of P from an initial set of environments $\mathcal{X}^{\natural}_{0}$. This system of semantic equations represents the *collecting semantics* [12] of a program. Remark that the system of semantic equations of a loop is recursive. In the sequel of the article, we will denote by $F^{\natural}$ the system of semantic equations associated to a program P.

We know that the system of semantic equations $F^{\natural}$ associated to a program P has a solution. On the one hand, a complete lattice structure can be endowed to the set $\wp(\Sigma)$ of all sets of environments. In other words, the tuple $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$ is the complete lattice of environments with a least element $\emptyset$, a maximal element Σ, a least upper bound $\cup$ and the greatest lower bound $\cap$. We denote by $\wp(E)$ the power set of the set E. On the other hand, the transfer functions $[\![.]\!]^{\natural}_{\mathcal{T}}$ are continuous w.r.t. the order $\subseteq$ [40]. So, applying the Tarski's theorem [39], we know that the solution of the system of semantic equations exists. Moreover, following the Kleene iteration method this solution can be computed as the limit the recursive sequence:

$$
\begin{aligned} \vec{\mathcal{X}}^{\natural}_{0} &= \vec{\emptyset} \\ \vec{\mathcal{X}}^{\natural}_{n+1} &= \vec{\mathcal{X}}^{\natural}_{n}\ \dot{\cup}\ F^{\natural}(\vec{\mathcal{X}}^{\natural}_{n}) \ . \end{aligned} \tag{3.2}
$$

We consider the system of semantic equations $F^\natural$ as a function from $\wp(\Sigma)^p$ to $\wp(\Sigma)^p$ with p the number of control points of P. The vector $\vec{\mathcal{X}}^\natural$ represents the vector of environments associated to each control point. The operation $\dot{\cup}$ is the component-wise application of the join operation $\cup$ of the lattice of environments over each element of $\vec{\mathcal{X}}^\natural$.

More precisely, Kleene iteration provides a constructive method to compute the solution of the system $F^\natural$ of semantic equations which is represented by Algorithm 1. Operations $\dot{\cup}$ and $\dot{\subseteq}$ are understood as the component-wise application of the operations $\cup$ and $\subseteq$ respectively. Note that the solution of $F^\natural$ is interpreted as the solution of a fixpoint equation.

Algorithm 1 Kleene iteration method.

```
1: $\vec{\mathcal{X}}^\natural_0 := \vec{\emptyset}$
2: **repeat**
3:     $\vec{\mathcal{X}}^\natural_i := \vec{\mathcal{X}}^\natural_{i-1} \,\dot{\cup}\, F(\vec{\mathcal{X}}^\natural_{i-1})$
4: **until** $\vec{\mathcal{X}}^\natural_i \,\dot{\subseteq}\, \vec{\mathcal{X}}^\natural_{i-1}$
```

In general, the main drawbacks to implement a static analysis from the collecting semantics are that $S \subseteq \Sigma$ is not representable on computers and Equation (3.2) may converge with an unbounded number of iterations. Abstract interpretation [11, 12] is a general theory that solves these two drawbacks by computing an over-approximation of program semantics defined by a continuous semantic function $F^\natural$, like the system of semantic equations. The two key ideas of abstract interpretation are *safe abstractions* and an *effective computational method* to solve Equation 3.2. We detail these notions in the next two paragraphs.

Safe abstractions of sets of states, i.e. environments, based on, in the more general framework [12, Sect. 7], concretization functions. More precisely let $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$ be the lattice of concrete states and let $\langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be the lattice of abstract states where $\bot$ is the least element, $\top$ is the greatest element, $\sqcup$ is the join operator and $\sqcap$ is the meet operator. The *concretization function* is a monotonic map $\gamma : A \to \wp(\Sigma)$. We consider $x \in A$ as a safe abstraction of $S \in \wp(\Sigma)$ if $S \subseteq \gamma(x)$. Moreover, the abstract continuous semantic function $F^\sharp$ is a safe abstraction of $F^\natural$ if and only if

$$\forall x \in A, F^\natural(\gamma(x)) \subseteq \gamma(F^\sharp(x)) \ .$$

Note that the abstract continuous semantic function $F^\sharp$ is a redesign of the system of semantic equations defined over abstract version of transfer functions depending on A.

In some cases, a strongest notion of abstraction may be defined with Galois connection that is a pair of functions (α, γ) such that:
- $\alpha : \wp(\Sigma) \to A$ is monotonic;
- $\gamma : A \to \wp(\Sigma)$ is monotonic;
- $\forall S \in \wp(\Sigma), x \in A, \quad \alpha(S) \sqsubseteq x$ and $S \subseteq \gamma(x)$.

The function α is the *abstraction function* and the function γ is still the *concretization function.* The notion of safe abstraction is formalized either by $\alpha(S) \sqsubseteq x$ or $S \subseteq \gamma(x)$. The abstract continuous semantic function $F^\sharp$ is a safe abstraction of $F^\natural$ if and only if

$$\forall x \in A, \left(\alpha \circ F^\natural \circ \gamma\right)(x) \subseteq F^\sharp(x) \ .$$

While this definition of abstraction is limited (not all numerical abstract domain has an abstraction function, e.g. the polyhedron abstract domain [14]), we recall it in order to clarify some remarks about Galois connections in Section 4.

Example 3.1. A simple abstraction for set of values is given by the domain of intervals. It is based on the complete lattice of intervals $\langle I, \sqsubseteq_I, \bot_I, \top_I, \sqcup_I, \sqcap_I \rangle$, its definition is recalled at Definition 3.2. Furthermore, a Galois connection exists between the lattice $\langle \wp(\mathbb{R}), \subseteq, \emptyset, \mathbb{R}, \cup, \cap \rangle$ and the lattice of intervals, its definition is recall at Theorem 3.3. Note that we use the fact that $\wp(\Sigma)$ is isomorphic to $\mathcal{V} \to \wp(\mathbb{R})$ to define the abstract domain of intervals whose its elements are functions $\mathcal{V} \to I$.

■

Definition 3.2 (Complete lattice of intervals)**.** Let I be the set of intervals such that:

$$I = \{[a, b] : -\infty \leq a \leq b \leq +\infty\} \quad .$$

The complete lattice of intervals $\langle I, \sqsubseteq_I, \bot_I, \top_I, \sqcup_I, \sqcap_I \rangle$ is defined by:
- an order relation: $[a_1, b_1] \sqsubseteq_I [a_2, b_2] \Leftrightarrow a_1 \geq a_2 \wedge b_1 \leq b_2$;
- a minimal element: $\bot_I$ and a maximal element: $\top_I = [-\infty, +\infty]$;
- a join operator: $[a_1, b_1] \sqcup_I [a_2, b_2] = [\min(a_1, a_2), \quad \max(b_1, b_2)]$;
- a meet operator:

$$[a_1, b_1] \sqcap_I [a_2, b_2] = \begin{cases} \bot_I & \text{if } b_1 < a_2 \text{ or } a_1 > b_2 \\ [\max(a_1, a_2), \quad \min(b_1, b_2)] & \text{otherwise} \end{cases} .$$

Theorem 3.3 (Galois connection for intervals)**.** *The Galois connection*

$$\langle \wp(\mathbb{R}), \subseteq, \emptyset, \mathbb{R}, \cup, \cap \rangle \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \langle I, \sqsubseteq_I, \bot_I, \top_I, \sqcup_I, \sqcap_I \rangle$$

is defined by:
- *the function of abstraction:* $\alpha_I(N \subseteq \mathbb{R}) = [\min_{n \in N} N, \max_{n \in N} N]$ *;*
- *the function of concretization:* $\gamma_I(I \in I) = \{i : i \in I\}$ *.*

An effective computation method to solve Equation (3.2). The abstract program semantics is a set of abstract states $\mathcal{X}^\sharp$ depending on the lattice $\langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$. From the abstract system of semantic equations $F^\sharp$ and for each control point, $\vec{\mathcal{X}^\sharp}$ is given as the limit of the recursive sequence:

$$\begin{aligned} \vec{\mathcal{X}}^\sharp_0 &= \vec{\bot} \\ \vec{\mathcal{X}}^\sharp_i &= \vec{\mathcal{X}}^\sharp_{i-1} \,\dot{\sqcup}\, F^\sharp(\vec{\mathcal{X}}^\sharp_{i-1}) \quad . \end{aligned} \tag{3.3}$$

The sequence $(\vec{\mathcal{X}}^\sharp_n)_{n \in \mathbb{N}}$ defines an increasing chain of elements of A. This chain may be infinite, when abstract semantics are based on infinite height lattices. So to enforce the convergence of this sequence, we substitute the operator $\sqcup$ by a *widening operator* ∇, see Definition 3.4, that is an over-approximation of $\sqcup$ with a guarantee of termination. In consequence, instead of having the least solution of Equation (3.3), we usually get an over-approximation of the least fixpoint.

Definition 3.4 (Widening operator [11])**.** Let $\langle A, \sqsubseteq_A \rangle$ be a lattice. The map $\nabla : A \times A \to A$ is a widening operator if and only if

(1) $\forall v_1, v_2 \in A$, $v_1 \sqcup v_2 \sqsubseteq_A v_1 \nabla v_2$;

(2) For each increasing chain $v_0 \sqsubseteq_A \cdots \sqsubseteq_A v_n \sqsubseteq_A \cdots$ of A, the increasing chain defined by $s_0 = v_0$ and $s_n = s_{n-1} \nabla v_n$ is stationary:

$$\exists n_0, \forall n_1, n_2, (n_2 > n_1 > n_0) \Rightarrow s_{n_1} = s_{n_2} \ .$$

Remark 3.5. In [12], Cousot and Cousot defined the widening operator as an n-ary operator over the abstract lattice A. However, to the best of our knowledge, all the widening operators defined so far are binary operators, that is why we only considered binary widening in Definition 3.4.

The widening operator plays an important role in static analysis because it allows to consider infinite state spaces where the ascending chain condition is not satisfied. Many abstract domains are thus associated with a widening operator; for the interval domain, for example, it is usually defined by:

$$[a,b]\nabla[c,d] = \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases}\right] \ . \tag{3.4}$$

Note that we only consider two consecutive elements of the increasing chain of Kleene iterates to extrapolate the potential fixpoint. The main drawback with this widening is that it may generate too coarse results by quickly going to infinity. For example, the application of the widening operator during an interval analysis of the program given at Figure 1 produce as a result: $[-\infty, \infty]$ for all the variables `x1`, `x2` and `x3`.

A solution of the loss of precision is to add intermediate steps among a finite set T; that is the idea behind the *widening with thresholds* ∇_T. In Algorithm 2, we can see the adapted version of Kleene iteration method using the widening operator with thresholds. Operations $\dot{\nabla}_T$ and $\dot{\sqsubseteq}$ are understood as a component-wise application of operations ∇_T and $\sqsubseteq$ respectively.

Algorithm 2 Kleene iteration algorithm based on widening operator with thresholds.

```
1: $\vec{\mathcal{X}}^\sharp_0 := \vec{\bot}$
2: repeat
3:     $\vec{\mathcal{X}}^\sharp_i := \vec{\mathcal{X}}^\sharp_{i-1} \ \dot{\nabla}_T \ F^\sharp(\vec{\mathcal{X}}^\sharp_{i-1})$
4: until $\vec{\mathcal{X}}^\sharp_i \ \dot{\sqsubseteq} \ \vec{\mathcal{X}}^\sharp_{i-1}$
```

Formally, for each call of ∇_T, the sequence is extrapolated up to an element of T until we reach a post-fixpoint. In some cases, after using all the elements of T, we cannot reach the least fixpoint, so to force termination we put the infinite as a last threshold.

For the interval domain, it is defined [3] by:

$$[a,b]\nabla_T[c,d] = \left[\begin{cases} a & \text{if } a \leq c \\ \max\{t \in T : t \leq c\} & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ \min\{t \in T : t \geq d\} & \text{otherwise} \end{cases}\right] \ . \tag{3.5}$$

While widening with thresholds gives better results, we are facing with the problem to define *a priori* the set T. Finding relevant values for T is a difficult task for which only syntactic-based techniques exist [25, 37]. For example, in the program in Figure 1, if we consider all the program constants as thresholds, this set does not provide any useful information to compute the bounds of the invariants at Line 5.

Example 3.6. We give the different results obtained from the program given at Figure 1 with the widening operator defined at Equation (3.4) and with the widening operator with thresholds defined at Equation (3.5). The results are given in Table 3.6 (cells with - symbol have the same value than the previous cell and cells filled in gray stands for the last iteration of Kleene algorithm). The notation J stands for Kleene iteration with join operator, W stands for for Kleene iteration with widening operator and WT stands for Kleene iteration with widening operator with thresholds. Moreover, we consider two sets of thresholds:

- $T_1 = \{-\infty, -0.4375, -0.2665, 0.0625, 0.1, 0.2665, 0.375, 0.4375, 1, 2, 3, 4, 6, 20, +\infty\}$ is made of all the constants appearing in the source code.
- $T_2 = \{-\infty, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 6, 32, 64, +\infty\}$ is made of powers of two.

The results given in Table 3.6 show that the iterations with classical widening operator converge faster (in 4 iterations) but produce a coarse result. The widening operator with thresholds defined in T_1 converges a little bit slower (in 5 iterations) and does not produce more precise result. Finally, the widening with thresholds in T_2 converges even slower (in 8 iterations) and produces a sharper invariant but which is bigger than the invariants given by the standard Kleene algorithm (the width of intervals is more than twice the width of the most precise interval).

As we can see, the set T of thresholds is an important matter to produce precise invariants. Let us remark that using a narrowing operator [13], we can refine the invariant in case of widening WT with the set T_2 but it needs 99 decreasing iterations to get the same results than the classical Kleene iteration. To be fair with the narrowing operator, we can remark that the invariant produced with only 10 decreasing iterations is: $\mathtt{x}_1 = [-5.68, 9.16]$, $\mathtt{x}_2 = [-3.34, 11.18]$ and $\mathtt{x}_3 = [-5.43, 20]$ but in general using such operator will introduce a new parameter (the number of decreasing iterations) which has to be tuned by the user. The work presented in this article is aimed at improving the widening operator in order to avoid as much as possible the use of a narrowing operator.

■

In the rest of the article, as we only work on abstract semantics we we will refer to abstract elements either by the notation $\mathcal{X}^\sharp$ or by the notation $\mathcal{X}$.

3.2. Acceleration of convergence

Convergence acceleration techniques, also named *sequence transformations*, allow to increase the rate of convergence of a sequence. In numerical analysis, there are several convergence acceleration methods, they transform convergent sequences (which approach their limits slowly) into sequences which converge more quickly to the same limit.

Before giving an overview of these techniques (for more details, we refer to [6]), we recall some basic definitions of *metric spaces* and *sequence convergence* [34].

Definition 3.7 (Metric space)**.** A metric space is a pair $\langle D, d\rangle$ where D is a non-empty set and d is a metric on D, i.e. a function $d : D \times D \to \mathbb{R}^+$, such that $\forall x, y \in D$ we have:

- $d(x, y) = 0 \Leftrightarrow x = y$.
- $d(x, y) = d(y, x)$.
- $d(x, y) \leq d(x, z) + d(z, y), \forall z \in D$.

The set of sequences over D (denoted by $D^{\mathbb{N}}$) is the set of functions between $\mathbb{N}$ and D.

Meth.	Var.	It. 1	It. 2	It. 3	It. 4	It. 5	It. 6	It. 7	It. 8	...	It. 127
J	x_1	[1, 2]	[-0.44, 5.71]	[-2.29, 6.57]	[-3.03, 7.49]	[-3.69, 7.87]	[-4.08, 10.85]	[-4.38, 8.36]	[-4.59, 8.50]	...	[-5.19, 8.87]
	x_2	[1, 4]	[0.86, 7.57]	[0.50, 9.37]	[-0.11, 10.21]	[-0.65, 10.64]	[-1.11, 10.85]	[-1.47, 10.96]	[-1.76, 11.02]	...	[-2.62, 11.12]
	x_3	[1, 20]	[0.20, 20]	[-1.10, 20]	[-1.92, 20]	[-2.63, 20]	[-3.14, 20]	[-3.54, 20]	[-3.84, 20]	...	[-4.71, 20]
W	x_1	[1, 2]	[-∞, +∞]	-	[-∞, +∞]						
	x_2	[1, 4]	[-∞, +∞]	-	[-∞, +∞]						
	x_3	[1, 20]	[-∞, 20]	[-∞, +∞]	[-∞, +∞]						
WT (T_1)	x_1	[1, 2]	[-∞, 6]	[-∞, +∞]	-	[-∞, +∞]					
	x_2	[1, 4]	[-0.2665, 20]	[-∞, 20]	[-∞, +∞]	[-∞, +∞]					
	x_3	[1, 20]	[-0.2665, 20]	[-∞, +∞]	-	[-∞, +∞]					
WT (T_2)	x_1	[1, 2]	[-1, 8]	[-4, 8]	[-8, 16]	[-16, 16]	-	-	[-16, 16]		
	x_2	[1, 4]	[-1, 8]	[-1, 16]	[-2, 16]	[-4, 16]	[-8, 16]	[-16, 16]	[-16, 16]		
	x_3	[1, 20]	[-1, 20]	[-4, 20]	[-4, 20]	[-8, 20]	[-16, 20]	-	[-16, 20]		

Table 1: Sequences of Kleene algorithm with different widening operators.

Definition 3.8 (Convergent sequence)**.** A sequence $(x_n) \in D^{\mathbb{N}}$ converges to $\ell \in D$ if and only if we have:

$$\lim_{n\to\infty} d(x_n, \ell) = 0 \ .$$

Definition 3.9 (Sequence transformation)**.** Let a sequence $(x_n) \in D^{\mathbb{N}}$ converge to $\ell \in D$. A *sequence transformation* is a function $T : D^{\mathbb{N}} \to D^{\mathbb{N}}$ (T denotes a particular acceleration method) such that: if $(y_n) = T(x_n)$ then (y_n) converges to ℓ faster than (x_n) i.e.

$$\lim_{n\to\infty} \frac{d(y_n, \ell)}{d(x_n, \ell)} = 0 \ .$$

This means that (y_n) is asymptotically closer to ℓ than (x_n).

As mentioned before there are several sequence transformations changing convergent sequences into sequences which converge more quickly to the same limit. Nevertheless, each sequence transformation only increases the rate of convergence of some classes of sequences. These classes are related to the *kernel* of transformation methods, formally defined in Definition 3.10.

Definition 3.10 (Kernel of sequence transformations)**.** Let T be a sequence transformation. The kernel θ_T of T is the set of sequences $(x_n) \in D^{\mathbb{N}}$, for which T transforms (x_n) to the constant sequence $\ell \in D$. More precisely,

$$\theta_T = \left\{(x_n) \in D^{\mathbb{N}} : \exists \ell \in \mathbb{R}, \forall n \in \mathbb{N}, T(x_n) = \ell\right\} \ .$$

For the rest of this section, we present only sequence transformations for real sequences, the extension to sequences on any metric space D is straightforward. We now present some acceleration methods that we used in our experiments described in Section 6.

3.2.1. The Aitken Δ^2-method

It is probably the most famous sequence transformation. Given a sequence $(x_n) \in \mathbb{R}^{\mathbb{N}}$, the accelerated sequence (y_n) is defined by:

$$\forall n \in \mathbb{N},\ y_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \ . \tag{3.6}$$

It should be noted that in order to compute y_n for some $n \in \mathbb{N}$, three values of (x_n) are required: x_n, x_{n+1} and x_{n+2}.

The kernel K_{Δ^2} of this method is the set of all sequences of the form $x_n = s + a.\lambda^n$ where s, a and λ are real constants such that $a \neq 0$ and $\lambda \neq 1$ (see [7] for more details).

The Aitken Δ^2-method is an efficient method for accelerating sequences, but it highly suffers from numerical instabilities when x_n, x_{n+1} and x_{n+2} are close to each other.

Example 3.11. To illustrate the Aitken Δ^2-method, we apply it to the following sequence:

$$\forall n \in \mathbb{N}, \quad s_n = 1 + \frac{1}{n+1} \quad \text{with} \quad \lim_{n\to+\infty} s_n = 1 \ .$$

The obtained results are given in Figure 5. We can see that the value of the 7th element of the accelerated sequence is closer to the limit by a factor of two.

■

Rank	Initial sequence	Rank	Accelerated sequence
0	2.0		
1	1.5		
2	1.3333333	0	1.25
3	1.25	1	1.16666667
4	1.2	2	1.125
5	1.1666667	3	1.1
6	1.1428571	4	1.0833333
7	1.125	5	1.0714286
8	1.1111111	6	1.0625
9	1.1	7	1.0555556
⋮	⋮	⋮	⋮

Fig. 5. The results obtained with the Aitken Δ^2-method on the sequence $s_n = 1 + \dfrac{1}{n+1}$.

3.2.2. The ε-algorithm

It is often cited as the best general purpose sequence transformation for slowly converging sequences [41]. The ε-algorithm is an extension of the Aitken Δ^2 method.

From a converging sequence $(x_n) \in \mathbb{R}^{\mathbb{N}}$ with limit ℓ, the ε-algorithm builds the following sequences:

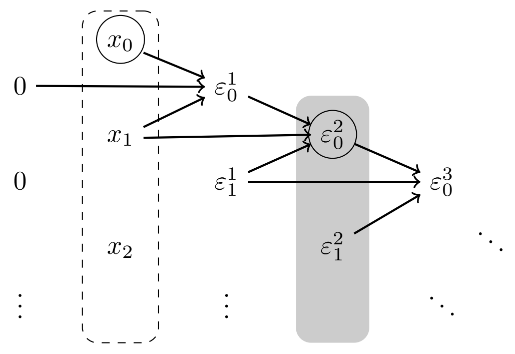
$$(\varepsilon_n^{-1}) : \forall n \in \mathbb{N}, \varepsilon_n^{-1} = 0, \tag{3.7}$$

$$(\varepsilon_n^{0}) : \forall n \in \mathbb{N}, \varepsilon_n^{0} = x_n, \tag{3.8}$$

$$(\varepsilon_n^{k}) : \forall k \geq 1,\ \forall n \in \mathbb{N},\ \varepsilon_n^{k+1} = \varepsilon_{n+1}^{k-1} + \left(\varepsilon_{n+1}^{k} - \varepsilon_n^{k}\right)^{-1} \tag{3.9}$$

For a fixed k, the sequence $((\varepsilon_n^k)_{n\in\mathbb{N}})$ is called the k^{th} column, and its construction can be graphically represented as on Figure 6. The *even columns*, that is the sequences of the form $\left((\varepsilon_n^{2k})_{n\in\mathbb{N}}\right)$ for $k > 0$, converge faster to ℓ. Note that the first column, that is the sequence $\left((\varepsilon_n^{0})_{n\in\mathbb{N}}\right)$, the dashed column on Figure 6, is the initial sequence for which we want to increase the convergence rate. For example, the accelerated sequence $\left((\varepsilon_n^{2})_{n\in\mathbb{N}}\right)$ is the column depicted in gray on Figure 6. Furthermore the *even diagonals*, that is the sequences $\left((\varepsilon_n^{2k})_{k\in\mathbb{N}}\right)$ for $n \geq 0$, made of the elements of the even columns also converge faster to ℓ. For example, the circled elements in Figure 6 represent the first elements of the diagonal sequence $\left((\varepsilon_0^{2k})_{k\in\mathbb{N}}\right)$.

Let us remark that in order to compute the p$^{\text{th}}$ element of the first diagonal sequence $\left((\varepsilon_0^{2k})_{k\in\mathbb{N}}\right)$, $(2p-1)$ elements of (x_n) are required as stated by Proposition 3.12. In consequence, we can see that the computations induced by the ε-algorithm are cheap in time and memory. Note that even if the result of this proposition seems to be known by



Arrows depict dependencies: the element at the beginning of the arrow is required to compute the element at the end. For example, the second element of the diagonal accelerated sequence is:

$$\begin{aligned}\varepsilon_0^2 &= \varepsilon_1^0 + \frac{1}{\varepsilon_1^1 - \varepsilon_0^1} \\ &= x_1 + \frac{1}{\varepsilon_2^{-1} + \frac{1}{\varepsilon_2^0 - \varepsilon_1^0} - \varepsilon_1^{-1} + \frac{1}{\varepsilon_1^0 - \varepsilon_0^0}} \\ &= x_1 + \frac{1}{\frac{1}{x_2 - x_1} - \frac{1}{x_1 - x_0}}\end{aligned}$$

Fig. 6. The ε-table, the circled elements are the ones used to define the diagonal accelerated sequence.

users of numerical acceleration methods, we did not see any proof of it in the literature and thus give one here.

Proposition 3.12. *Let $(x_n) \in \mathbb{R}^{\mathbb{N}}$ be a sequence and let $(y_n) = (\varepsilon_0^{2n})$ be its accelerated version given by the ε-algorithm. Then the p^{th} element of (y_n) is defined by $2p-1$ elements of (x_n).*

Proof. We call $G(n,k)$ the number of elements from (x_n) required to compute the element ε_n^k of indices n and k in the ε-algorithm. By construction, we have:
$\forall k \geq 1,\ \forall n \in \mathbb{N},\ \varepsilon_n^{k+1} = \varepsilon_{n+1}^{k-1} + \left(\varepsilon_{n+1}^k - \varepsilon_n^k\right)^{-1}$.
So, $G(n,k+1) = \max\big(G(n+1,k-1),\quad G(n+1,k),\quad G(n,k)\big)$.
Note that the function G is increasing in n and k, so $G(n,k+1) = G(n+1,k)$. Thus, the function G is defined by:

$$G(n,k) = \begin{cases} n+1 & \text{if } k = 0 \\ G(n+1,k-1) & \text{otherwise} \end{cases} .$$

Moreover following the ε-algorithm, we know that y_p, the element of index p in (y_n), i.e. the $(p+1)^{\text{th}}$ element, is ε_0^{2p}.

(1) We prove by induction that $\forall k \in \mathbb{N},\ (\forall n \in \mathbb{N}, G(n,k) = G(n+k,0))$:
- For $k = 0$ we have $\forall n \in \mathbb{N}, G(n,0) = G(n+0,0)$.
- We assume that $G(n,k) = G(n+k,0)$ and we want to prove that it is true in the case of $G(n,k+1)$.

$$\begin{aligned}\forall n \in \mathbb{N},\ G(n,k+1) &= G(n+1,k+1-1) && \text{(From the definition of } G.\text{)} \\ &= G(n+1,k) \\ &= G(n+k+1,0) && \text{(Induction's hypothesis.)}\end{aligned}$$

The proposition is true for $(k+1)$ so it is true for $\forall k \in \mathbb{N}$.

(2) Now, we prove that $\forall p \in \mathbb{N},\ G(0,2p) = 2p+1$:

$$\begin{aligned} G(0,2p) &= G(1,2p-1) \\ &= G(1+2p-1,0) \text{ (By induction on: G(n,k)=G(n+k,0))} \\ &= G(2p,0) \\ &= 2p+1. \end{aligned}$$

So, to have the element y_p, we need $(2p+1)$ elements from (y_n). We know that the element y_p is the $(p+1)^{\text{th}}$ element of (y_n). So, to obtain p elements of (y_n), $2p-1$ elements of (x_n) are required. $\square$

The kernel K_ε of the ε-algorithm contains an important set of sequences (see [6, Chap. 2, pp. 85–91] for a detailed review of them). In particular, all the convergent linear sequences of the form $x_{n+1} = ax_n + b$ where a and b are a real numbers are in the kernel K_ε. Moreover, *totally monotonic* sequences and *totally oscillating* convergent sequences are also in the kernel K_ε. These two kinds of sequences are very interesting because almost complete results have been obtained. We recall that a sequence (x_n) is totally monotonic sequence if:

$$\forall k \in \mathbb{N}, \forall n \in \mathbb{N}, \quad (-1)^k \Delta^k x_n \geq 0 \ .$$

The operator Δ^k represents the k^{th} forward finite difference operator such that $\Delta^k x_k = \sum_{i=0}^{k} \binom{k}{i} (-1)^{n-i} x_{k+i}$ where $\binom{k}{i}$ is the binomial coefficient. A sequence (x_n) is said totally oscillating if the sequence $((-1)^n x_n)$ is totally monotonic. Note that not all totally oscillating sequences are convergent.

For example, the sequence $x_n = \lambda^n$ with $\lambda \in]0,1[$ is totally monotonic. Moreover, under some conditions, we can build from totally monotonic sequences new totally monotonic sequences. In particular [6, Chap. 2, pp. 88], if x_n is a totally monotonic sequence, if the function f has a series expansion with all coefficients positive and if $x_0 < r$ the radius of convergence of f then $f(x_n)$ is a totally monotonic sequence.

The most important feature with such two kinds of sequences (for more details about these results, see [6, Theorem 2.22, pp. 89 and Theorem 2.24, pp. 90]) is that if the ε-algorithm is applied to a sequence (x_n) (not necessarily in the kernel K_ε) which converges to ℓ and if there exist a and b two real constants such that (ax_n+b) is totally monotonic or totally oscillating and convergent then the even columns and the even diagonal sequences converge more quickly to ℓ.

Example 3.13. Applying the ε-algorithm to the sequence of the Example 3.11, we obtain the results given in the Figure 7. Let us remark that both of the column and the diagonal sequences converge more quickly to 1 (the limit of the initial sequence) than the initial sequence. In this case, the diagonal sequences are the fastest.

■

3.2.3. Acceleration of vector sequences

Many acceleration methods were designed to handle scalar sequences of real numbers. For almost each of these methods, extensions have been proposed to handle vector sequences (see [24] for a review of them). The simplest, yet one of the most powerful, of these methods is the *vector ε-algorithm* (VEA).

ε_n^0	ε_n^2	ε_n^4	ε_n^6	ε_n^8
2.0				
1.5	1.25			
1.3333333	1.1666667	1.1111109		
1.25	1.1249999	1.0833337	1.0624931	
1.2	1.1000001	1.0666663	1.0500028	1.0399799
1.1666667	1.0833333	1.0555556	1.0416545	1.0334257
1.1428571	1.0714287	1.0476161	1.0357504	
1.1250000	1.0624998	1.0416761		
1.1111111	1.0555557			
1.1				

Fig. 7. The results obtained with the ε-algorithm on the sequence $s_n = 1 + \dfrac{1}{n+1}$.

Given a vector sequence $(\vec{x}_n)$, the VEA computes a set of vector sequences $(\vec{\varepsilon}_n^{\,k})$ using Equations (3.7)-(3.9) where:

- The arithmetic operations $+$ and $-$ are computed component-wise.
- The inverse of a vector $\vec{v}$ (according to Brezinski [8]) is computed as $\vec{v}^{-1} = \vec{v}/(\vec{v} \cdot \vec{v})$, with / being the component-wise division and $\cdot$ the scalar product.

The VEA differs from a component-wise application of the (scalar) ε-algorithm as it introduces *relations* between the components of the vector: the scalar product $\vec{v} \cdot \vec{v}$ computes a global information on the vector $\vec{v}$ which is propagated to all components. Our experiments show that this algorithm works better than a component-wise application of the ε-algorithm.

The kernel K_ε of the VEA was less studied than the scalar version of the ε-algorithm. Nevertheless, it contains all sequences of the form $\vec{x}_{n+1} = A\vec{x}_n + B$, where A is a constant matrix and B a constant vector [8].

4. From abstract lattices to metric spaces and back

As explained in Section 3.1, Kleene iteration is widely used in abstract interpretation based static analysis to compute numerical invariants, but it is inefficient as the number of iterations can be very large. Widening operator solves this problem, but it can make the computation imprecise as it makes large over-approximations. In the next two sections, we combine numerical acceleration methods with the abstract fixpoint computation in order to obtain a method that is fast (i.e. reduces the number of iterations) and precise (i.e. computes a post-fixpoint close to the result of Kleene iteration). Our goal is to be as non-intrusive as possible in the classical iterative scheme. In this way, our method can be implemented with minor adaptations in current static analyzers. First, in Section 4, we explain how we make the link between abstract domains which are based on an order relation and numerical sequences for which the notion of distance is fundamental. We give a generic framework and two instantiations for the interval and octagon abstract domains. Then, in Section 5, we give our main algorithm and prove its soundness.

4.1. Methodology

Kleene iteration for finding the least fixpoint is based on abstract values from some abstract lattice A. In order to use numerical acceleration techniques on the abstract iterates, we need to extract a vector of real numbers from the abstract elements $\vec{\mathcal{X}}_n \in A$. Doing this for each iterate, we obtain a sequence of real vectors $(\vec{x}_n)$ that we can accelerate. We thus get a new numerical sequence $(\vec{y}_n)$ that reaches the same limit than $(\vec{x}_n)$ but more quickly. We then construct an abstract element $\vec{\mathcal{X}}$ that corresponds to this limit and we use it as a candidate for the least fixpoint of the semantic function. This process of transforming an abstract value into a real vector and back is formalized by the notion of *extraction* and *combination* functions. These functions make the link between the abstract domain and (vector of) real numbers. In other words, these functions allow a non-trivial communication between ordered sets and metric or topological spaces. Note that they are dependent of the chosen abstract domain. By non-trivial communication, we mean that they should verify a common property in order to make our algorithm relevant. This property (see Property 1) states that the extraction and combination functions must transform an order theoretic convergence on abstract elements into a topological convergence on real numbers.

Property 1 (Asymptotically safe transformation)**.** Let $\langle A, \sqsubseteq_A\rangle$ be an ordered set and $\langle D, d\rangle$ be a metric space. The functions $\phi : A \to D$ and $\psi : D \to A$ form an asymptotically safe transformation if and only if, for all monotonic sequences $(\mathcal{X}_n) \in A^{\mathbb{N}}$ that converge, i.e. such that $\exists \mathcal{X} \in A$ with $\bigsqcup_{n\in\mathbb{N}} \mathcal{X}_n = \mathcal{X}$, we have:
(1) $\big(\phi(\mathcal{X}_n)\big) \in D^{\mathbb{N}}$ is a convergent sequence in D, i.e. $\exists S \in D$ with $\lim_{n\to\infty} \phi(\mathcal{X}_n) = S$
(2) the limit $\mathcal{S}$ of $\big(\phi(\mathcal{X}_n)\big)$ verifies $\mathcal{X} \sqsubseteq_A \psi(S)$.

Property 1 verifies that the asymptotic behaviors of the sequences $(\phi(\mathcal{X}_n))$ and $(\mathcal{X}_n)$ when n tends to $+\infty$ are safe: when $(\mathcal{X}_n)$ (order theoretically) converges towards $\mathcal{X}$, then $(\phi(\mathcal{X}_n))$ topologically converges towards a value that contains at least as much information as $\mathcal{X}$. Intuitively, the sequence $(\mathcal{X}_n)$ will be the sequence of Kleene iterates and will converge towards the least fixpoint of the semantic equations (see Section 3.1). As we will manipulate the numerical sequence $(\phi(\mathcal{X}_n))$, we need it to be convergent (in order to apply the methods for convergence acceleration) and we want to obtain, from its limit, an over-approximation of the fixpoint. This is the reason for the two conditions on ϕ (the extraction function) and ψ (the combination function) that we require in Property 1. We now give the formal definition of the *extraction* and *combination* functions.

Definition 4.1 (Extraction and combination.)**.** Let $\langle A, \sqsubseteq_A\rangle$ be an abstract domain, and let $\langle D, d\rangle$ be a metric space. The functions $\Lambda_A : A \to D$ and $\Upsilon_A : D \to A$ are called extraction and combination function, respectively, if and only if they form an asymptotically safe transformation.

Let us remark that for all extraction and combination functions Λ_A, Υ_A, we have: $\forall \mathcal{X} \in A,\ \mathcal{X} \sqsubseteq_A \Upsilon_A\big(\Lambda_A(\mathcal{X})\big)$ (just take $\mathcal{X}_n$ to be the constant sequence $\mathcal{X}_n = \mathcal{X}$ in Property 1). This is what we call *local safety.* However, the notion of extraction and combination functions is different of the notion of Galois connection (see Section 3.1). Actually, the only point that matters to us is that Λ_A and Υ_A respect the asymptotic behavior of monotone sequences, in particular we do not require both functions to be monotonic as the order on A and the order on D induced by the metric d may be unrelated. Let us also remark that the soundness of our method does not rely on the property of asymptotically safe transformations (see Theorem 5.1 in Section 5). Rather, this property ensures a certain coherence between the abstract and numerical elements and thus ensures the efficiency of our algorithm (if the extraction function produces an accelerable sequence, see discussion at the end of Section 5).

In the rest of this section, we present some extraction and combination functions for different abstract domains. Mainly we believe that such functions can be easily defined for template-based domains. The template-based domains represent the set of domains with a pre-defined shape as a template polyhedra [35]. For these domains, the geometric order is equivalent to a syntactic order on the constraint coefficients. On the other hand, the not template-based domain includes domains with changing shape like polyhedra [14], and zonotope [22]. In Sections 4.1.1 and 4.1.2, we give the extraction and combination functions for two template-based domains, the interval and octagon abstract domains respectively. For these domains the extraction and combination functions are straightforward and they are very precise: the inequality in Property 1 is indeed an equality. We believe that we can obtain similar results for other template-based domains. And in Section 4.1.3, we present some tests that show the difficulty to define the extraction and combination functions for the polyhedra abstract domain.

4.1.1. The interval abstract domain

In this section, we recall the lattice structure of the interval abstract domain and define the corresponding extraction and combination functions. We gave in Section 3.1, Definition 3.2, the formal definition of the interval abstract domain. We recall in Definition 4.2 its lattice structure as we will need it to prove that our extraction and combination functions are asymptotically safe.

Definition 4.2. The complete lattice of intervals $\langle I, \sqsubseteq_I, \perp_I, \top_I, \sqcup_I, \sqcap_I \rangle$ is defined by:
- an order relation: $[a_1, b_1] \sqsubseteq_I [a_2, b_2] \Leftrightarrow a_1 \geq a_2 \wedge b_1 \leq b_2$;
- a minimal element: $\perp_I$ and a maximal element: $\top_I = [-\infty, +\infty]$;
- a join operator: $[a_1, b_1] \sqcup_I [a_2, b_2] = [\min(a_1, a_2), \quad \max(b_1, b_2)]$;
- a meet operator:

$$[a_1, b_1] \sqcap_I [a_2, b_2] = \begin{cases} \perp_I & \text{if } b_1 < a_2 \text{ or } a_1 > b_2 \\ [\max(a_1, a_2), \quad \min(b_1, b_2)] & \text{otherwise} \end{cases}.$$

Extraction and combination functions For the interval domain $I = I^v$, where $v = \mathrm{card}(\mathcal{V})$ is the number of variables of the program and I is the set of floating-point intervals [29, Sect. 4], the extraction and the combination functions are defined in Equations (4.1).

$$\Lambda_I : \begin{cases} I \to \mathbb{R}^{2v} \\ (i_1, \dots, i_v) \mapsto \left(\bar{i}_1, \underline{i}_1, \dots, \bar{i}_v, \underline{i}_v\right) \end{cases} \tag{4.1a}$$

$$\Upsilon_I : \begin{cases} \mathbb{R}^{2v} \to I \\ (x_1, x_2, \dots, x_{2v-1}, x_{2v}) \mapsto \left([x_1, x_2], \dots, [x_{2v-1}, x_{2v}]\right) \end{cases} \tag{4.1b}$$

To show that Λ_I and Υ_I are, respectively, the extraction and the combination functions of the interval abstract domain, we must prove that they form an asymptotically safe transformation. This is stated in the Proposition 4.3. Note that we only give a proof in case of $v = 1$, the generalisation for $v = n$ is very similar.

Proposition 4.3. *Functions Λ_I and Υ_I form an asymptotically safe transformation.*

Proof. We must prove that, for a monotonic sequence $[a_n, b_n] \in I^n$ that

$$[a, b] = \bigsqcup_{n\in\mathbb{N}}{}_{\mathrm{I}} [a_n, b_n] \Rightarrow \left([a, b] \sqsubseteq_I \left[\lim_{n\to+\infty} a_n,\ \lim_{n\to+\infty} b_n\right]\right)$$

or equivalently, that

$$[a, b] = \bigsqcup_{n\in\mathbb{N}}{}_{\mathrm{I}} [a_n, b_n] \Rightarrow \left(a \leq \lim_{n\to+\infty} a_n \wedge b \geq \lim_{n\to+\infty} b_n\right) \ .$$

We know that the sequence of elements $[a_n, b_n]$ is monotonic, then:
we have $\forall n \in \mathbb{N}, [a_n, b_n] \sqsubseteq_I [a_{n+1}, b_{n+1}] \Rightarrow a_{n+1} \leq a_n$ and $b_{n+1} \geq b_n$;
and $[a, b] = \bigsqcup_{\mathrm{I}n\in\mathbb{N}} [a_n, b_n] \Rightarrow [a_n, b_n] \sqsubseteq_I [a, b]$ and $(a_n \geq a \vee b_n \leq b)$.
In consequence we know that:
- the sequence (a_n) is decreasing and bounded thus convergent;
- the sequence (b_n) is increasing and bounded thus convergent.

Moreover, we know that:
$[a, b] = \bigsqcup_{\mathrm{I}n\in\mathbb{N}} [a_n, b_n] \Rightarrow a$ is the infimum of (a_n) and b is the supremum of (b_n).
If a is the infimum of (a_n) then $\forall \varepsilon \in \mathbb{R}, \varepsilon > 0, \exists N \in \mathbb{N}, |a_N - a| < \varepsilon$.
Thus,

$$(\forall n > N, a_n \leq a_N \wedge a_n \geq a) \Rightarrow |a_n - a| < \varepsilon.$$

So, (a_n) converges to a then $\lim_{n\to+\infty} a_n = a$.
In the same way, we can prove that: $\lim_{n\to+\infty} b_n = b$.
In conclusion, we proved that
$[a, b] \sqsubseteq_I \Upsilon_I\big(\lim_{n\to+\infty} \Lambda_I ([a_n, b_n])\big)$. □

Theorem 4.4. *For the interval abstract domain, Λ_I and Υ_I represent, respectively, the extraction and the combination functions.*

Remark 4.5. Note that if there is a Galois connection (α_I, γ_I) between a domain $\langle A, \sqsubseteq_A\rangle$ and the interval domain $\langle I, \sqsubseteq_I\rangle$, the extraction and combination functions can be defined as $\Lambda_A = \Lambda_I \circ \alpha_I$ and $\Upsilon_A = \gamma_I \circ \Upsilon_I$.

4.1.2. The octagon abstract domain

In this section, we recall the main features of the octagon abstract domain and we define its associated extraction and combination functions.

The octagon abstract domain [30] is a relational domain, i.e. it allows to represent the existing relations between pairs of program variables. These relations are given by constraints of the form $\pm x \pm y \leq c$ with $c \in \mathbb{R}$. These constraints are encoded as potential constraints in order to use the difference bound matrix (DBM), and so facilitate their manipulations. If we have a program with $v = \mathrm{card}(\mathcal{V})$ variables, then a conjunction of their octagonal constraints can be represented as a DBM of dimension $2v$, that is, a

$$\begin{cases} x - y \leq 3 \\ x + y \leq 0 \\ -x - y \leq -3 \\ x \leq 1 \end{cases}$$

(a) Octagon constraints.

$$\begin{cases} x_2 - y_2 \leq 3 \\ y_1 - x_1 \leq 3 \\ x_2 - y_1 \leq 0 \\ y_2 - x_1 \leq 0 \\ x_1 - y_2 \leq -3 \\ y_1 - x_2 \leq -3 \\ x_2 - x_1 \leq 2 \end{cases}$$

(b) Potential constraints.

$$\begin{array}{c|cccc} & x_1 & x_2 & y_1 & y_2 \\ \hline x_1 & +\infty & 2 & 3 & 0 \\ x_2 & +\infty & +\infty & -3 & +\infty \\ y_1 & +\infty & 0 & +\infty & +\infty \\ y_2 & -3 & 3 & +\infty & +\infty \end{array}$$

(c) Difference Bound Matrix.

Fig. 8. Example of transformation of an octagon constraints to DBM.

$2v \times 2v$ matrix with elements in $R = \mathbb{R} \cup \{+\infty\}$. We use the infinite coefficient when the associated constraint is not considered. Let us remark that the DBM is symmetric so we can use only half a matrix to encode it (it is actually how it is done in our implementation), but for more clarity we give the whole DBM in this section. To illustrate this domain, an example is given in Figure 8. In this example, we give the octagon constraints of a program with two variables x and y (see Figure 8(a)). Each constraint is transformed into two potential ones, see Figure 8(b) where $x_1 = -x$, $x_2 = x$, $y_1 = -y$ and $y_2 = y$. The DBM is built from these potential constraints (see Figure 8(c)).

In order to avoid confusion between indices related to the elements of the DBM and the indices related to the elements of real sequences, we denote by m^{ij} the element at row i and column j of the DBM m. An element of a real sequence is still denoted by a subscript.

Definition 4.6. Formally, let $\langle \mathbb{O}, \sqsubseteq_{\mathbb{O}} \rangle$ be the octagon abstract domain such that: $\mathbb{O} = M_{2v,2v}(R)$ the set of square matrices of size $2v \times 2v$ with coefficients in R. The complete lattice of octagon $\langle \mathbb{O}, \sqsubseteq_{\mathbb{O}}, \bot_{\mathbb{O}}, \top_{\mathbb{O}}, \sqcup_{\mathbb{O}}, \sqcap_{\mathbb{O}} \rangle$ is defined by:
- an order relation: $\forall p, q \in \mathbb{O}, p \sqsubseteq_{\mathbb{O}} q \Leftrightarrow \forall i, j \in [1, 2v], p^{ij} \leq q^{ij}$;
- a minimal element: $\bot_{\mathbb{O}}$ and a maximal element: $\top_{\mathbb{O}}$;
- a join operator: $\forall p, q, r \in \mathbb{O}, \quad r = p \sqcup_{\mathbb{O}} q \Leftrightarrow \forall i, j \in [1, 2v], r^{ij} = \max(p^{ij}, r^{ij})$;
- a meet operator: $\forall p, q, r \in \mathbb{O}, \quad r = p \sqcap_{\mathbb{O}} q \Leftrightarrow \forall i, j \in [1, 2v], r^{ij} = \min(p^{ij}, q^{ij})$.

For the rest of this section, let $\langle \mathbb{O}, \sqsubseteq_{\mathbb{O}} \rangle$ be the octagon abstract domain, and m a DBM of the octagon constraints with $w = 2v$ and such that:

$$m = \begin{pmatrix} m^{11} & \dots & m^{1w} \\ \vdots & \ddots & \vdots \\ m^{w1} & \dots & m^{ww} \end{pmatrix} .$$

Extraction and combination functions . Each of these functions can be defined for the octagon abstract domain [30]: the function $\Lambda_{\mathbb{O}}$ associates to a *difference bound matrix* a vector containing all its coefficients. The function $\Upsilon_{\mathbb{O}}$ transforms the vector into a DBM. The extraction and the combination functions are defined in Equations (4.2) where $w = 2v$ and $v = \mathrm{card}(\mathcal{V})$ is the number of variables of the program and $R = \mathbb{R} \cup \{+\infty\}$.

Remark 4.7. In the case where the coefficients in the DBM are infinite, the extracted sequence will contain infinite values. We do not modify the acceleration process, so that infinite values are propagated to the accelerated sequence.

$$\Lambda_{\mathbb{O}} : \begin{cases} \mathbb{O} \to R^{w^2} \\ \begin{pmatrix} m^{11} & \dots & m^{1w} \\ \vdots & \ddots & \vdots \\ m^{w1} & \dots & m^{ww} \end{pmatrix} \mapsto \left(m^{11}, \dots, m^{1w}, \dots, m^{w1}, \dots, m^{ww}\right) \end{cases} \tag{4.2a}$$

$$\Upsilon_{\mathbb{O}} : \begin{cases} R^{w^2} \to \mathbb{O} \\ \left(x_1, \dots, x_w, \dots, x_{w^2}\right) \mapsto \begin{pmatrix} x_1 & \dots & x_w \\ \vdots & \ddots & \vdots \\ x_{w^2-w} & \dots & x_{w^2} \end{pmatrix} \end{cases} \tag{4.2b}$$

Like with the interval domain, the functions of extraction and of combination for the octagon domain verify Property 1, as stated by Proposition 4.8.

Proposition 4.8. *Functions $\Upsilon_{\mathbb{O}}$ and $\Lambda_{\mathbb{O}}$ form an asymptotically safe transformation.*

Proof. Let $p \in \mathbb{O}$ and $(m_n) \in \mathbb{O}^{\mathbb{N}}$ be an order monotonic sequence of DBM, we must show that:

$$p = \bigsqcup_{n\in\mathbb{N}}^{\mathbb{O}} m_n \Rightarrow p \sqsubseteq_{\mathbb{O}} \begin{pmatrix} \lim\limits_{n\to+\infty} m_n^{11} & \dots & \lim\limits_{n\to+\infty} m_n^{1w} \\ \vdots & \ddots & \vdots \\ \lim\limits_{n\to+\infty} m_n^{w1} & \dots & \lim\limits_{n\to+\infty} m_n^{ww} \end{pmatrix}$$

or equivalently, that:

$$p = \bigsqcup_{n\in\mathbb{N}}^{\mathbb{O}} m_n \Rightarrow \forall i,j \in [1,2v], \quad p^{ij} \geq \lim_{n\to+\infty} m_n^{ij} \ .$$

In the octagon domain, we know that the sequence (m_n) is order monotonic, then:
we have $\forall n \in \mathbb{N}, m_n \sqsubseteq_{\mathbb{O}} m_{n+1} \Rightarrow \forall n \in \mathbb{N}, \forall i,j \in [1,2v], \quad m_n^{ij} \leq m_{n+1}^{ij}$
and $p = \bigsqcup_{\mathbb{O}\, n\in\mathbb{N}} m_n \Rightarrow \forall n \in \mathbb{N}, \forall i,j \in [1,2v], \quad m_n^{ij} \leq p^{ij}$.
Thus, the sequences (m_n^{ij}) are increasing and bounded so they are convergent.
Moreover, as the value p^{ij} is the supremum of the sequence (m_n^{ij}), we have:
$\forall \varepsilon \in \mathbb{R}, \varepsilon > 0, \exists N \in \mathbb{N}, \quad |m_N^{ij} - p^{ij}| < \varepsilon$
$\Rightarrow \forall n > N, \left(m_n^{ij} > m_N^{ij} \wedge m_n^{ij} < p^{ij}\right) \Rightarrow |m_n^{ij} - p^{ij}| < \varepsilon.$
Then, each of the sequences (m_n^{ij}) converges to p^{ij}, so $\lim_{n\to+\infty} m_n^{ij} = p^{ij}$.
In conclusion, we proved that
$p \sqsubseteq_{\mathbb{O}} \Upsilon_{\mathbb{O}}\left(\lim_{n\to+\infty} \Lambda_{\mathbb{O}}(m_n)\right)$. □

Theorem 4.9. *The functions $\Upsilon_{\mathbb{O}}$ and $\Lambda_{\mathbb{O}}$ are, respectively, the extraction and the combination functions for the octagon abstract domain.*

4.1.3. Remarks on not template-based domains

In this section, we give examples of extraction and combination functions that are less precise and that operate on the polyhedra abstract domain [14]. We do not give the proofs that the functions are asymptotically safe, our intention is to give an intuition on the kind of functions that can be used in our method. We also give an example of a pair of functions that is not asymptotically safe while locally safe (i.e. verifying $\forall \mathcal{X} \in A,\ \mathcal{X} \sqsubseteq_A \Upsilon_A \circ \Lambda_A(\mathcal{X})$).

For simplicity reasons, we only give examples on two-dimensional polyhedra like [38], but they generalize very easily to the n-dimensional case. A polyhedra P in two variables x and y is then a conjunction of constraints of the type $\alpha x + \beta y \leq c$ for $\alpha,\ \beta,\ c \in \mathbb{R}$. We note it $P = \bigwedge_{i=1}^{m} \alpha_i x + \beta_i y \leq c_i$, P represents the valuations of both variables that verify all the constraints, written $\gamma(P) = \big\{(x, y) \in \mathbb{R}^2,\ |\ \forall i \in [1, m],\ \alpha_i x + \beta_i y \leq c_i\big\}$. On the set of 2-dimensional polyhedra, noted $\mathcal{P}$, we can define the extraction/combination functions in different manners, three of them are presented in Case 1 to Case 3.

Case 1. The extraction and the combination functions $\Lambda_1,\ \Upsilon_1$ follow Equations (4.3).

$$\Lambda_1 : \begin{cases} \mathcal{P} \to \mathbb{R}^4 \\ P \mapsto \langle \underline{x}, \overline{x}, \underline{y}, \overline{y} \rangle \quad \text{with} \quad \forall x, y \in \gamma(P),\ \underline{x} \leq x \leq \overline{x} \wedge \underline{y} \leq y \leq \overline{y} \end{cases} \tag{4.3a}$$

$$\Upsilon_1 : \begin{cases} \qquad\quad \mathbb{R}^4 \to \mathcal{P} \\ \langle \underline{x}, \overline{x}, \underline{y}, \overline{y} \rangle \mapsto \big(-x \leq -\underline{x}\big) \wedge \big(x \leq \overline{x}\big) \wedge \big(-y \leq -\underline{y}\big) \wedge \big(y \leq \overline{y}\big) \end{cases} \tag{4.3b}$$

Intuitively, these functions associate a polyhedra with the infima and suprema of variables x and y. If we use these functions in our framework, we will thus be able to quickly compute the bounding box of the least fixpoint, but we will lose all relations between x and y because the combination function discards all relations between x and y.

Remark 4.10. We can compute the lower and upper bounds of all variables defined by polyhedra using *linear programming.*

Case 2. The extraction and the combination functions $\Lambda_2,\ \Upsilon_2$ follow Equations (4.4).

$$\Lambda_2 : \begin{cases} \mathcal{P} \to \mathbb{R} \\ P \mapsto r, \quad \text{with } \forall x, y \in \gamma(P),\ ||(x, y)|| \leq r \end{cases} \tag{4.4a}$$

$$\Upsilon_2 : \begin{cases} \mathbb{R} \to \mathcal{P} \\ \ r \mapsto \diamond(S_r) \end{cases} \tag{4.4b}$$

In these equations, S_r is the 2-dimensional circle of ray r and $\diamond(.)$ is any polyhedral enclosure of a circle, while $||(x, y)|| = \sqrt{x^2 + y^2}$ is the norm of the vector (x, y). The composition $\Upsilon_2 \circ \Lambda_2$ is locally safe (see Figure 9) and it is easy to see that these functions verify Property 1. Actually, for any converging monotone sequence of polyhedra P_n, the sequence $\Lambda_2(P_n)$ is monotone and bounded, thus convergent. If we use these functions in our method, the extracted sequence will contain the distance of the extreme point of the polyhedra to 0, and we will thus quickly compute the distance of the extreme point of the fixpoint to 0, but when we use this distance using the combination function, we again lose all relations between variables (or more precisely, we introduce new, arbitrary ones, see Figure 9).

Remark 4.11. To compute Λ_2, we must iterate over all the generator of a polyhedra. These can be computed using Chernikova algorithm [27].

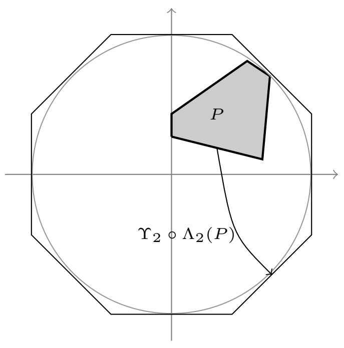


Fig. 9. Effect of the functions Λ_2 and Υ_2 on a polyhedra.

Case 3. In order to keep the relations between x and y, one can try to extract from a polyhedra the coefficients of the constraints. As there can be arbitrary many constraints, there can be arbitrary many coefficients, so we cannot keep all of them. For example, we can try to only keep the coefficients of the first four constraints of any polyhedra, in this way the extraction function associates a polyhedra P with 12 real numbers (3 coefficients per constraints). This is formalized in Equations (4.5).

$$\Lambda_3 : \begin{cases} \mathcal{P} \to \mathbb{R}^{12} \\ \left(\bigwedge_{i=1}^{n} \alpha_i x + \beta_i y \leq c_i\right) \mapsto \langle \alpha_1, \beta_1, c_1, \dots, \alpha_4, \beta_4, c_4 \rangle \end{cases} \tag{4.5a}$$

$$\Upsilon_3 : \begin{cases} \mathbb{R}^{12} \to \mathcal{P} \\ \langle \alpha_1, \beta_1, c_1, \dots, \alpha_4, \beta_4, c_4 \rangle \mapsto \bigwedge_{i=1}^{4} \alpha_i x + \beta_i y \leq c_i \end{cases} \tag{4.5b}$$

If a polyhedra has less than four constraints, then the remaining coefficients are set to zero. The functions Λ_3 and Υ_3 make a locally safe transformation, i.e. $\forall P \in \mathcal{P}$, $P \subseteq \Upsilon_3 \circ \Lambda_3(P)$ as $\Upsilon_3 \circ \Lambda_3(P)$ contains a subset of the constraints of P. However, they do not verify Property 1: we give in Appendix A an example of a sequence of polyhedra that is monotone, bounded and convergent but for which the extracted sequence is not convergent. The main reason is that there is no link between the geometric order on polyhedra and the relative value of the coefficient of the constraints. This was not true for the functions Λ_1/Υ_1 and Λ_2/Υ_2 as the monotonicity of P_n implied the monotonicity of $\Lambda_1(P_n)$ and $\Lambda_2(P_n)$.

Remark 4.12. To define Λ_3, we assumed an order on the constraints of the polyhedra. This order can be derived from an order on the program variables. However, whatever the order chosen for the constraints, the (Λ_3/Υ_3) functions will not be asymptotically safe.

As these three examples should have demonstrated, the definition of extraction/combination functions for complex abstract domains like polyhedra is not an easy task. The main problem is that the numerical values that are easily extractable from a polyhedra (i.e. the coefficients of the constraints or the coordinates of the generators) are not correlated with the geometric order: bigger constraints coefficients does not imply polyhedra inclusion. This is why we need to define functions that only extract some information about the polyhedra (like the bounding box or the coordinates of the most extreme point), keeping in mind that this information must be coherent with the geometric order. Finding an extraction function that is able to keep some relations and that is more precise than Λ_1 or Λ_2 is one of our future work.

5. Acceleration of the fixpoint computation

In this section, we describe the use of the numerical sequence transformation in the abstract fixpoint computations, more precisely its insertion in the Kleene iteration process as in Algorithm 3. We compute in parallel the sequence $(\vec{\mathcal{X}}^\sharp_n)$ coming from the abstract Kleene's iteration and the accelerated sequence $(\vec{y}_n)$ computed from an accelerated method applied on $\Lambda_A(\vec{\mathcal{X}}^\sharp_n)$. Once the sequence $(\vec{y}_n)$ seems to converge, that is, the distance between two consecutive elements of $(\vec{y}_n)$ is smaller than a given value δ, we combine the two sequences. That is we compute the upper bound of the two elements of the current iteration. In this step, $\Upsilon_A(\vec{y}_i)$ is similar to a threshold defined in the widening with thresholds. The difference lies in the fact that the threshold used here is automatically computed, while in the widening with thresholds the set of thresholds is statically given by the programmer before running the analysis. Note that the monotonicity of the computed sequence $(\vec{\mathcal{X}}^\sharp_n)$ is still guaranteed.

Algorithm 3 Accelerated abstract fixpoint computation.

```
1: $\vec{\mathcal{X}}^\sharp_0 := \vec{\bot}$
2: $i := 1$
3: **repeat**
4:     $\vec{\mathcal{X}}^\sharp_i := \vec{\mathcal{X}}^\sharp_{i-1} \mathbin{\dot\sqcup} F^\sharp(\vec{\mathcal{X}}^\sharp_{i-1})$
5:     $\vec{y}_i :=$ Accelerate $\Big(\Lambda_A(\vec{\mathcal{X}}^\sharp_0), \ldots, \Lambda_A(\vec{\mathcal{X}}^\sharp_i)\Big)$
6:     **if** $d(\vec{y}_i - \vec{y}_{i-1}) \ \leq\ \delta$ **then**
7:         $\vec{\mathcal{X}}^\sharp_i := \vec{\mathcal{X}}^\sharp_i \mathbin{\dot\sqcup} \Upsilon_A(\vec{y}_i)$
8:     **end if**
9: **until** $\vec{\mathcal{X}}^\sharp_i \mathrel{\dot\sqsubseteq} \vec{\mathcal{X}}^\sharp_{i-1}$
```

So the use of acceleration methods may be seen as an automatic delayed application of the widening with thresholds. Let us remark that we are not guaranteed to terminate in finitely many iterations: we know that asymptotically, the sequence $\vec{y}_i$ from Algorithm 3 gets closer and closer to the fixpoint, but we are not guaranteed that it reaches it. To guarantee termination of the fixpoint computation, we have to use more "radical" widening thresholds, as in [23, Sect 2.4], for example after n applications of the accelerated method. So this method cannot be a substitute for widening, but it improves it by reducing the number of parameters (delay and thresholds) that a user must define.

Theorem 5.1. *The post-fixpoint obtained with the Algorithm 3 is sound i.e. if $\vec{\mathcal{K}}^\sharp$ is the fixpoint obtained with Kleene iteration, and if $\vec{\mathcal{A}}^\sharp$ is the post-fixpoint obtained in the Algorithm 3, then we have $\vec{\mathcal{K}}^\sharp \mathrel{\dot\sqsubseteq} \vec{\mathcal{A}}^\sharp$.*

Proof. The soundness of our method relies on the fact that when we use the accelerated sequence y_i, we compute the join with the current iterate (Line 7 on Algorithm 3). So if

$\vec{\mathcal{K}}_i^\sharp$ are the iterates of Kleene iteration and $\vec{\mathcal{A}}_i^\sharp$ are the iterates of the accelerated iteration, then we have: $\forall i \in \mathbb{N},\ \vec{\mathcal{K}}_i^\sharp \,\dot{\sqsubseteq}\, \vec{\mathcal{A}}_i^\sharp$, which proves that $\vec{\mathcal{K}}^\sharp \,\dot{\sqsubseteq}\, \vec{\mathcal{A}}^\sharp$. $\square$

Remark 5.2. Note that for the Algorithm 3:
- the relevance of its results depends on the three function Λ_A, Υ_A and *Accelerate*;
- it does not work for all programs, because the function *Accelerate* cannot accelerate all sequences;
- in the case where the sequence $\left(\Lambda_A(\vec{\mathcal{X}}_0^\sharp), \ldots, \Lambda_A(\vec{\mathcal{X}}_i^\sharp)\right)$ belongs to the kernel of the *Accelerate* method, the Algorithm 3 is very efficient.

Example 5.3. To illustrate this method, let us apply it to the example of the introductive example (see Figure 1) with the interval abstract domain and using the ε-algorithm for the `Accelerate` method. The obtained results are given in the Figure 10. Using the Algorithm 3 with the interval abstract domain, the post-fixpoint is reached after only 16 iterations, where the Kleene iteration need 127 iterations.
In this example, we consider the first diagonal (ε_0^{2k}) as the accelerated sequence, so to compute each element of this sequence (noted $\vec{y}_i$ in the Algorithm 3) we need 3 elements from the initial one, noted $(\vec{\mathcal{X}}_n)$. We remark that the sequence $\left(\Upsilon_I(\vec{y}_n)\right)$ need some iterations (in this example, 5 iterations) to stabilize. Every time that a new $\vec{y}_i$ is computed, the algorithm compare this result with the lastest element of the accelerated sequence, when the difference is lower than δ, we consider this result as a threshold that we put back into the abstract sequence, that is the Kleene iteration. In our example this re-injection is activated when the maximum of the distance between upper and lower bound of two successive elements $\vec{y}_i$ becomes lower than 10^{-3}. Let us remark, that in the Figure 10 the first threshold is obtained after 11 iterations. After its union with the last element in $(\vec{\mathcal{X}}_n)$, we apply again the Algorithm 3. The second threshold is not interesting, because its lower than the last element in the modified sequence. In this case, the accelerated sequence needs some iterations to stabilize after the re-injection of the threshold to the initial sequence. The algorithm stops after 16 iterations, and reach the post-fixpoint which is very close to the fixpoint obtained with the Algorithm 1 (the difference between them is of the order of 10^{-14}).
Note that if we use the set of thresholds obtain in Figure 10 in Algorithm 2 using widening with thresholds, the result would be the same but the difficulty is to define the good set of thresholds statically.

$\blacksquare$

6. Experimentation

In this section, we present the performance of our algorithm on some typical numerical programs that are complicated to analyze using standard widening/narrowing techniques. The programs we address are infinite loops that iterate some linear or nonlinear functions (see Figure 11(a) for example). The termination of the analysis thus does not depend on the guard of the loop, which is what make widening based analyzers fail to obtain precise invariants. Moreover, the classical threshold approach is not of any help as no constraint in the programs can help to determine the good thresholds to choose. Let us also remark that we do not consider, in this article, loops with conditional branches

$\vec{\mathcal{X}}_n$	$\Upsilon_A(\vec{y}_n)$	$d(y_i - y_{i-1})$	Thresholds ($\delta = 10^{-3}$)
$[1.000000, 2.000000]$	$[1.000000, 2.000000]$		
$[-0.447300, 5.716500]$			
$[-2.291255, 6.573381]$	$[6.280857, 6.830145]$	$\max(5.280857, 4.830145)$	
$[-3.038029, 7.492492]$			
$[-3.695558, 7.871720]$	$[-4.663282, 8.463092]$	$\max(10.944139, 2.367053)$	
$[-4.084503, 8.185954]$			
$[-4.386442, 8.369221]$	$[-5.189239, 8.851176]$	$\max(0.525957, 0.388084)$	
$[-4.594886, 8.508279]$			
$[-4.751445, 8.603235]$	$[-5.197089, 8.872567]$	$\max(0.00785, 0.021391)$	
$[-4.865385, 8.673918]$			
$[-4.950393, 8.725095]$	$[-5.197506, 8.873307]$	$\max(0.000417, 0.00074)$	$[-5.197506, 8.873307]$
$[\mathbf{-5.197506, 8.873307}]$			
$[-5.197506, 8.873307]$	$[-5.197089, 8.872567]$	$\max(0.000417, 0.00074)$	$[-5.197089, 8.872567]$
$[-5.197567, 8.873341]$			
$[-5.197616, 8.873348]$	$[-5.198638, 8.849716]$	$\max(0.001549, 0.022851)$	
$[-5.197598, 8.873362]$			

Fig. 10. The results of application of the Algorithm 3 to the introductive example.

inside the loop body. Such loops would be difficult to handle using acceleration methods only as the dynamics of the program variables may be completely different in one branch or another, thus making the acceleration process perform poorly. However, we believe that our technique can be used together with policy iteration [9, 16] or the guided static analysis framework [19] to handle such programs, we discuss this in Section 7.

Our experimentation aims at comparing the number of iterations needed to obtain a post-fixpoint using our accelerated algorithm and using Kleene algorithm without widening. We measured both the number of iterations and the computation time, and we also compare the accuracy of the post-fixpoint obtained by both methods. To do so, we integrated our algorithm on the interval abstract domain in the Interproc analyzer [1]. Interproc is an inter-procedural analyzer that uses the APRON library [26] of abstract numerical domains. We extended the interval library of APRON to add the extraction and combination methods, and added our algorithm in the FIXPOINT library. We are currently working on making this process more generic to handle other abstract domains (such as the octagon abstract domain). However, we were able to measure the number of iterations needed by our algorithm on the octagon abstract domain by using Interproc for computing the sequences of Kleene iterates and patching it, using scripts, to implement the extraction, acceleration and combination functions. Our algorithm on octagons is thus not completely integrated into Interproc, so we could not compare the execution time, but the results we present in this article are correct: we used the `assume` assertions of Interproc in order to simulate the join of the current iterate and the combination of the accelerated sequence.

In the rest of this section, we present our results. In short, we prove that our algorithm works very well for linear programs, which was expected as for such programs, the extracted sequences are within the kernel of the acceleration process we use (we only used the ϵ-algorithm for these experiments). For such programs, the number of iterations is greatly reduced both for the interval and octagon domains. The reduction in term of

[1] `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

computation time is also very interesting, at least for programs that need many iterations to converge. For the programs that converge quickly with the Kleene algorithm (in less that 100 iterations), it is less interesting to use the acceleration algorithm as the gain in the number of iterations is not sufficient to significantly reduce the computation time. We also show that for some nonlinear examples, our method succeeds in greatly reducing the number of iterations while preserving the precision of the analysis.

Remark 6.1. We give in this section the programs written in the C syntax. The same programs written in the SYNTAX language of Interproc can be found at:
`http://www.lix.polytechnique.fr/~bouissou/jsc`

6.1. Linear examples

We now consider four linear programs: the running example, two linear filters (a Butterworth filter of order 1 and 2) and a program that implements a Gauss-Seidl resolution scheme. Let us remark that the Gauss-Seidl program can be found in industrial programs to compute fast approximations of usual functions (square-root for example). We were able to analyze these programs using the interval and octagon abstract domains and thus present both results. We also show, for the Butterworth order 1 program, the influence of the δ parameter of our algorithm on the number of iterates and the accuracy of the post-fixpoint computed by our algorithm.

6.1.1. The programs

The file `contraction.c` (see Figure 1) is the C version of the running example. It is a contracting linear program, and the analysis converges in the interval abstract domain. This example originates from a classical set of benchmarks for the guaranteed solution of ordinary differential equations.

The file `butter1.c` (see Figure 11(a)) is an order 1 Butterworth filter. This kind of low-pass filter is very much used in the signal processing community as its response is very flat. The file `butter2.c` (see Figure 11(b)) is the same kind of filter but of order 2. We modified the classical order 2 Butterworth filter so that the analysis converges with the interval abstract domain.

Finally, the `gauss-seidl.c` file (see Figure 11(c)) represents an implementation of the Gauss-Seidl method to solve the linear system $AX = B$ with:

$$A = \begin{pmatrix} 4 & 21 & 1 \\ -1 & 2 & 0 \\ 2 & 1 & 4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 2 \\ 4 \\ 9 \end{pmatrix} .$$

This programs converges quickly (in only 24 iterates) but the acceleration methods still manages to improve it by a factor 2 while computing the same fixpoint.

6.1.2. Results using the interval abstract domain

In Table 2, we present the results that we obtain on the four programs of Figure 1 and Figure 11. This table should be read as follows: for each program, we give the number of iterations and the computation time necessary to reach a post-fixpoint both for standard Kleene iteration and for the accelerated algorithm. Then we give the results for the accuracy of the analysis. The accuracy is computed as the (Haussdorf) distance between the box computed using Kleene iteration and the one computed using the accelerated fixpoint

```
void main() {
  float x1,xn1,y,u;
  while (1) {        /* u between 1 and 2 */
    xn1 = 0.90480*x1 + 0.95240*u;
    y   = 0.09524*x1 + 0.04762*u;
    x1 = xn1;
}}
```

(a) butter1.c, Butterworth order 1 filter.

```
void main() {
  float x1,x2,xn1,xn2,y,u;     x1=0;x2=0;u=1;
  while(1) {
    xn1 = 0.8027   *x1 + -0.07314*x2 + 0.9314   *u;
    xn2 = 0.07314  *x1 + 0.8053   *x2 + 0.04657  *u;
    y   = 0.004657*x1 + 0.09977  *x2 + 0.002328*u;
    x1 = xn1;      x2 = xn2;
}}
```

(b) butter2.c, Butterworth order 2 filter.

```
void main () {
  double x=0, y=0, z=0;
  double xn=0, yn=0, zn=0;
  while(1) {
    xn = 1 - 0.5 * y - 0.25 * z;
    yn = 1 + 0.5 * xn;
    zn = 2.25 - 0.5 * xn - 0.25 * yn;
    x = xn; y = yn; z = zn;
}}
```

(c) gauss-seidl.c, a Gauss-Seidl method.

Fig. 11. Linear programs.

method. On the interval abstract domain, the Hausdorff distance is very easily computed; we give the absolute distance (column labeled `Distance (Abs.)`) and the distance relative to the value of the box given by Kleene (column labeled `Distance (Rel.)`).

Table 2 shows that our method works very well for such linear programs. We reduce the number of iterations by a large factor (between 6 and 43) while computing almost exactly the same fixpoint (the overhead is neglectful).

Let us remark that the convergence of Kleene iteration for these programs is rather slow because of the semantics of floating-point numbers. The iteration computes quickly (in 30 or 40 iterates) a value very near the least fixpoint, i.e. at a distance around 10^{-3}, but then takes a large number of iterations to actually compute the least fixpoint. It is because all bits of the floating-point representation must stabilize before getting the least fixpoint. This problem was mentioned by Miné in [29, Sect. 7], where the iteration function is modified (each iterate is enlarged by a small factor) in order to avoid these numerical instabilities. In our setting, the accelerated Kleene iteration avoids this problem

Table 2. Gain in number of iterations for simple linear programs, using the interval abstract domain (with $\delta = 10^{-3}$).

	Efficiency (Iterations - Computation time)		Accuracy	
Program name	Kleene	Accel.	Distance (Abs)	Distance (Rel)
`contraction.c`	127 - 0.014s	17 - 0.009s	2.10^{-13}	8.10^{-12}%
`butter1.c`	346 - 0.018s	8 - 0.003s	6.10^{-14}	6.10^{-12}%
`butter2.c`	180 - 0.012s	16 - 0.006s	10^{-14}	5.10^{-13}%
`gauss-seidl.c`	24 - 0.004s	12 - 0.004s	10^{-16}	5.10^{-14}%

as the acceleration techniques stabilize, in once, many bits of floating-point representation by computing a very precise approximation of the least fixpoint.

6.1.3. Results using octagon abstract domain

We conducted the same study for the octagon abstract domain. On octagons, we did not measure the precision using the Hausdorff distance as it is complicated to compute. Rather, we define the distance between two octagons given by their DBM m and n by:

$$d(m, n) = \max_{i,j} |m^{ij} - n^{ij}| \ .$$

This is a good indication of how close two octagons are. The relative distance is then this distance relative to the lowest, non-zero value of the matrix.

The results are given in Table 3: they show that on these examples, our algorithm performs as well on the octagon abstract domain as on the interval abstract domain. The reduction factor in the number of iterations is still very important and the accuracy of the accelerated analysis remains very high. To make our algorithm works on the octagon abstract domain, we had to slightly modify our method such that it only accelerates the unstable coefficients of the DBM. Actually we noticed that some of the coefficients of the DBM are almost immediately stable (i.e. after one or two iterations), so there is no need to apply acceleration on them. In Algorithm 3, this is equivalent to having an `Accelerate` function that behaves differently on each coefficient of the matrix.

Let us remark that the acceleration performs a little better (in the number of iterates) on intervals than on octagons. We believe that it is mainly due to the better implementation of our algorithm in the interval abstract domain than with octagons.

6.1.4. Influence of δ

We now study the influence of the parameter δ of the Algorithm 3. This is the only parameter that our algorithm needs: it is a measure of the precision that we want to achieve, i.e. the smaller δ, the better the approximation of the limit by the accelerated sequence y_n will be, and we should obtain a value for $\mathcal{X}_i$ that is very close to the limit of Kleene iteration. On the contrary, the bigger δ the sooner the join of $\mathcal{X}_i$ with $\Upsilon(y_i)$ will take place, so we may need many iterations after to get to a fixpoint.

However, as it is shown on Figure 12, the influence of δ on the accuracy and speed of the algorithm is in practice limited. We can see that the number of iterates grows linearly in δ, but a very small δ is not needed to obtain good performances: on the `gauss-seidl.c` example, as soon as $\delta \leq 10^{-4}$, the results are almost constant (in terms

Table 3. Gain in number of iterations for simple linear programs in the octagon abstract domain (with $\delta = 10^{-3}$).

	Efficiency (Iterations)		Accuracy	
Program name	Kleene	Accel.	Distance (Abs)	Distance (Rel)
`contraction.c`	127	21	8.10^{-14}	$5.10^{-12}\%$
`butter1.c`	346	10	3.10^{-14}	$2.10^{-12}\%$
`butter2.c`	181	21	2.10^{-14}	$2.10^{-12}\%$
`gauss-seidl.c`	24	16	10^{-8}	$10^{-7}\%$

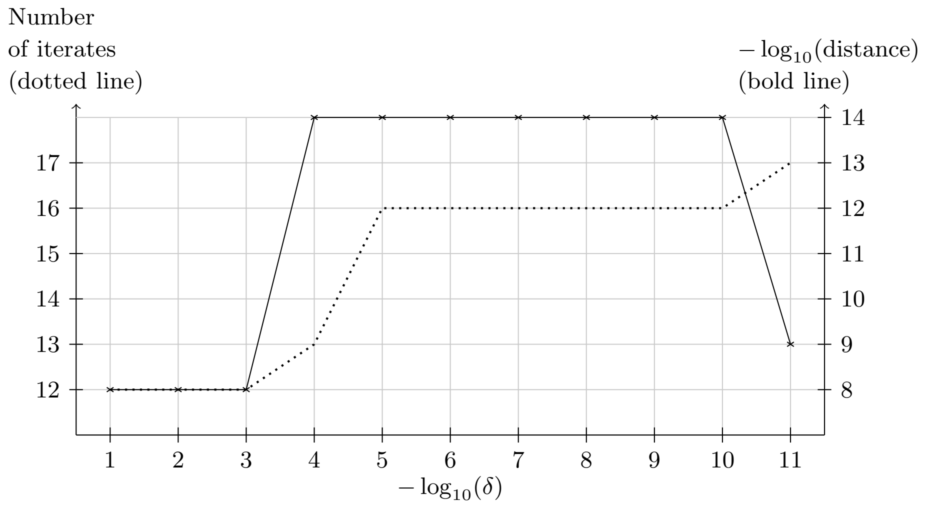


Fig. 12. Evolution of the number of iteration and accuracy with δ for the `gauss-seidl.c` file.

of accuracy). It should be noted that a too small δ can be problematic: for $\delta = 10^{-11}$, the accuracy actually decreases. This is because the accelerated sequence cannot stabilize up to δ, so that (probably because of numerical instabilities) it goes beyond the limit of the extracted sequence. In this way, we obtain a post-fixpoint which is a little bit further from the least-fixpoint than with $\delta = 10^{-8}$. For the other programs, the algorithm behaves in a similar way with respect to δ.

6.2. Influence of the acceleration process

As explained in Section 4, our accelerated algorithm depends on the `Accelerate` method chosen to accelerate the extracted sequence (see Algorithm 3). In this section, we study the influence of this function on the performance of our algorithm, both in terms of number of iterations and accuracy of the analysis. We compare the three acceleration processes we described in Section 3.2, i.e. the ϵ-algorithm, the Aitken Δ^2 method and the Vector ϵ-algorithm. Results are given in Table 4.

This table shows that the choice of the acceleration process is important for the overall performance of the algorithm. On some programs (like the `butter2.c` program), the

Table 4. Influence of the acceleration process on the number of iterates and accuracy of the accelerated fixpoint computation.

	Number of iterations				Accuracy		
Program name	Kleene	Δ^2	ϵ-Algo	VEA	Δ^2	ϵ-Algo	VEA
`contraction.c`	127	21	17	23	10^{-4}	3.10^{-14}	3.10^{-5}
`butter1.c`	346	6	8	7	10^{-14}	10^{-14}	10^{-14}
`butter2.c`	180	33	16	20	10^{-3}	10^{-13}	1.10^{-12}
`gauss-seidl.c`	24	14	12	17	4.10^{-10}	10^{-8}	2.10^{-14}

performance is significantly increased when using the ϵ-algorithm compared to the Δ^2-Aitken method. This is normal as the extracted sequence in this case is within the kernel of the ϵ-algorithm and not in the kernel of the Δ^2-Aitken method. To our surprise, the Vector ϵ-algorithm is not better than the scalar ϵ-algorithm. The difference between both lies in the implementation of the `Accelerate` function that takes as input a sequence of real vectors and accelerates it. In the vector case, it consists in applying the VEA directly on this sequence while in the scalar case, we apply the ϵ-algorithm independently on each coordinate of the vector sequence. So in the vector case, our algorithm tries to accelerate simultaneously the upper and lower bounds of all variables while in the scalar case, each bound is accelerated independently. We believe that the VEA will be mostly useful for accelerating the convergence of Kleene iteration with relational abstract domains and for programs that do not converge in the interval domain.

6.3. Nonlinear examples

In this section, we present the results of our algorithm on three nonlinear examples given at Figures 13(a) to 13(c). These programs were chose to test our algorithm on various nonlinear operators (square-root, polynomial, division). We chose them so that the analysis using the interval abstract domain converges; in order to test our method on more complicated programs (for example, a Newton method for computing the inverse as in [21]), it is necessary to use relational abstract domains. We are confident that an extension of our work on zonotopic [22] or polyhedral [14] domains will be able to treat these cases.

The program of Figure 13(a) iterates the sequence: $x_{n+1} = 0.5 \times \left(x_n + x_n^{-1}\right)$ with $x_0 \in [3, 4]$. Its analysis using the interval abstract domain and Kleene iteration needs 58 iterations to converge to the least fixpoint which is $x \in [0.25, 4]$, it is the lower bound of x which takes a long time to converge. Let us note that no widening technique can help compute the same fixpoint unless we explicitly give 0.25 as a threshold.

The program of Figure 13(b) is associated to the sequence $x_{n+1} = \sqrt{x_{n-1} \times x_n}$ with $x_0 = 1$ and $x_1 = 2$. Using the interval abstract domain, the static analysis based on the Kleene iteration requires 53 iterations to converge to the least fixpoint $x_i \in [1, 2]$ with $i \in \{1, 2, 3\}$. Once again, it is the lower bound of the variables which takes some time to converge.

The program of Figure 13(c) is an order 2 polynomial of a single variable x. Its analysis using the interval abstract domain takes 1929 iterations to converge to the least fixpoint $x \in [-58.823529417650402, -7]$.

Table 5. Gain in number of iterations and computation time for nonlinear programs.

	Iterations		Computation time (s)	
Program name	Kleene	Accel	Kleene	Accel
`nonlinear1.c`	58	17	0.004	0.003
`nonlinear2.c`	53	12	0.004	0.003
`nonlinear2.c`	1929	7	0.059	0.003

```
void main() {
  float x;
  while (1){
    /* x between 3 and 4 */
    x = 0.5 * (x + 1 / x);
  }
}
```

(a) `nonlinear1.c`, a rational function.

```
void main () {
  float x1, x2, x3;
  x1 = 1; x2 = 2;
  while (1){
   x3 = sqrt (x1 * x2);
   x1 = x2; x2 = x3;
  }
}
```

(b) `nonlinear2.c`, a square root function.

```
void main(){
  float x;
  x = -7;
  while (1){
    x = x * (0.001 * x + 0.99) - 1;
  }
}
```

(c) `nonlinear3.c`, a polynomial function.

Fig. 13. Non-linear programs.

In Table 5 we present the results of our algorithm using the ϵ-algorithm on these examples. In all cases we significantly reduced the number of iterations to compute a fixpoint (for the program `nonlinear3.spl`, we reduced it from 1929 to 7). These results were obtain using the ϵ-algorithm and choosing the parameter δ to 10^{-3}. Note that our method computed the same fixpoint as Kleene iteration for programs `nonlinear1.spl` and `nonlinear2.spl`, while the over-approximation is very small for `nonlinear3.spl`: the distance with the least fixpoint is 3.10^{-10}.

6.4. A note on other abstract domains

The extraction and combination functions must be defined for each abstract domain. We showed how to define them for the interval and octagon abstract domains, and we believe that the same can be done for other relational abstract domains with a *predefined shape* like the templates abstract domain [35]. However, using Remark 4.5, we can easily define extraction and combination functions on any domain that has a Galois connection with the interval abstract domain. In this way, we will accelerate the computation of the bounding boxes of the Kleene iterates and we might loose the relational information between the program variables. It is an easy way to define the extraction and combination functions for any abstract domain and might perform well for programs where the most influential constraints are the ones on the bounds of the variables. We tested this technique on the affine sets abstract domain [21] of Fluctuat [20] on the modified version of the Butterworth order 2 filter (see Figure 11(b)). We were able to accelerate the computation of the fixpoint by a large factor, but the post-fixpoint we obtain is, as expected, not as satisfactory as with the previous domains (see Table 6). We believe that the accuracy of the accelerated analysis can be improved by defining ad-hoc extraction and acceleration functions that keep the relations between variables.

Table 6. Gain in number of iterations for the Butterworth filter using Fluctuat and the affine forms abstract domain (with $\delta = 10^{-3}$).

	Efficiency (iterations)		Accuracy	
Program name	Kleene	Accel.	Distance (Abs)	Distance (Rel)
`butter2.c`	179	20	$1.4.10^{-1}$	31%

7. Related work

Most of the work in abstract interpretation based static analysis concerned the definition of new abstract domains (or improvements of existing ones), and the abstract fixpoint computation remained less studied. Initial work from Cousot and Cousot [13] discussed various methods to define widening operators. Bourdoncle [5] presented different iteration strategies that helps to reduce the over-approximation introduced by widening. These methods are complementary to our technique: as explained in Section 4, acceleration should be done at the same control point as the one chosen for widening, and does not replace standard widening as the termination of the fixpoint computation is not guaranteed. However, acceleration methods greatly improve widening by dynamically and automatically finding good thresholds.

Our method works particularly well for programs that iterate a linear map, such as linear filters. For such programs, specific domains and analyzes were designed in [15] or [31]. These techniques produce very precise results by considering the structure of the filter in order to compute invariants on the filter outputs. The work by Feret [15] has been implemented in ASTREE [2] for second order filters, and the work of Monniaux [31] considers the composition of filters written in a higher level description language. Both techniques however suffer from their lack of automization and their specialization to linear filters. As our experimentation shows, our method performs well for other kinds of programs, even nonlinear programs (see Section 6.3).

In [9, 16], the authors try to improve the computation of the least fixpoint by simplifying the semantic equation of the program. This equation is represented as an infimum of a finite set of simpler maps, called policies. The policy iteration algorithm calculate, in every iteration, the least fixpoint of one of the policies (chosen from the policies set) using a specific analysis. The algorithm terminates when the computed fixpoint is also the fixpoint of the initial semantic equation. The authors prove that the least fixpoint of the initial semantic equation is the minimum of the set of least fixpoint of policies. This method allows to improve the fixpoint computation by changing the semantic structure of the program, while our method do the same by keeping the semantics properties of the program.

Gopan and Reps in their *guided static analysis* framework [18] improve the precision of the widening by two instantiations. The first treats the widening in loops with multiple phases, it is a generalization of *the lookahead widening technique* [19]. In their work, the authors used the idea of computing in parallel the main iterates and a guide that shows where the iterates are going. The precision of the fixpoint computation is increased by computing a *pilot value* that explores the state space using a restricted version of the iteration function. Once this pilot has stabilized, it is used to accelerate the main iterates; in a sense, this pilot value is very similar to the value $\vec{y}_i$ of Algorithm 3, but we do not modify the iteration function as done in [18]. The second instantiation deals with the problem of the widening in loops with non-deterministically chosen behavior. In this case, the iteration function is derived into restrictions, where each one corresponds to a particular loop behaviors. The restrictions are analyzed separately before the union of their results. Nevertheless, our method follows the same spirit of the work [18, 19] by limiting the modifications on existing static analyzers.

Maybe the work that is the closest to ours is the use of acceleration techniques in model checking [1], that have recently been applied to abstract interpretation [17, 28, 36]. In this framework, the term acceleration is used to describe techniques that try to predict the effect of a loop on an abstract state: the whole loop is then replaced with just one transition that safely and precisely approximates it. These techniques perform very well for sufficiently simple loops working on integer variables, and gives exact results for such cases. Again, this method is complementary to our usage of acceleration: it *statically* modifies the iteration function by replacing simple loops with just one transition, while our method *dynamically* predicts the limit of the iterates. We believe that our method is more general, as it can be applied to many kinds of loops and is not restricted to a specific abstract domain (changing the abstract domain only requires changing the Λ_A and Υ_A functions).

Note also that the computation of symbolic loop invariants such that [32, 33] produces precise results. Nevertheless, they have to limit the constructions of analyzed programs unlike our approach. In particular, in [33] right-hand side expressions of assignments have to be *solvable mappings with positive rational eigenvalues* which are extensions to polynomials of invertible linear functions. This assumption may be complex to check during the analysis that is why we believe our method is more practicable.

8. Conclusion

We presented a technique to accelerate abstract fixpoint computations in the Kleene iteration using numerical acceleration methods. This technique consists in building a numerical sequence by extracting from the abstract element, at each iteration, a numerical element, and we do this for every variable of the program. We then applied convergence acceleration methods to the obtained sequences, which allow us to get significantly closer to the numerical limit (even in some cases reach it quickly). The resulting numerical limit is transformed into an abstract element by the combination function. To make sure that the fixpoint returned by the accelerated method is indeed the fixpoint of the abstract semantics, we re-inject it in the static analyzer. This guarantees us the fast stop of the analyzer with a very tight over-approximation of the fixpoint. The experiments made on a certain number of examples (linear and nonlinear programs) show a good acceleration of the fixpoint computation especially when we use the ε-algorithm. Let us note that we have assumed in this article that the sequences of iterates and the corresponding vector sequences converge towards a finite limit. In case of diverging sequences, traditional widening can be used as sequence transformation will not perform as well as for converging ones.

For now, the Algorithm 3 is fully implemented for the interval abstract domain in Interproc analyzer. For the octagon abstract domain, we made the experimentations using two separate programs: one that computes the Kleene iterates and extract the numerical sequences, and one that accelerates these sequences. Its automatization in *APRON* library and *Interproc* analyzer is the object of our current work. Our future work will also consist in extending this technique to other relational domains such as polyhedra. To treat a larger set of programs, we will also try to use other numerical accelerated methods with different kernels. The experiments show that acceleration behaves well on loops iterating a nonlinear function. It will be more difficult to treat loops that are less regular, e.g. loops with if statements, as the extracted sequences are less regular. However we believe that our technique can be mixed with guided static analysis [19] to improve the fixpoint computation in the analysis of every program's restriction. We also expect that we can use our method in the policy iteration algorithm [9, 16] to accelerate the fixpoint computation of every policy, which is calculated by using the Kleene iteration in the initial method.

Acknowledgements

The authors want to thank X. Allamigeon and E. Goubault for their helpful discussions and precious advices and S. Zennou and B. Jeannet for their technical help on the development of our analyzer. The authors are also thankful to the anonymous reviewers for their helpful comments and suggestions.

References

[1] Bardin, S., Finkel, A., Leroux, J., Petrucci, L., 2008. FAST: acceleration from theory to practice. Journal on Software Tools for Technology Transfer 10 (5), 401–424.

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation: Complexity, Analysis, Transformation. Vol. 2566 of LNCS. Springer, pp. 85–108.

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2003. A static analyzer for large safety-critical software. In: Programming Language Design and Implementation. ACM Press, pp. 196–207.

[4] Bouissou, O., Seladji, Y., Chapoutot, A., 2010. Abstract fixpoint computations with numerical acceleration methods. Electronic Notes in Theoretical Computer Science 267 (1), 29 – 42, proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

[5] Bourdoncle, F., 1993. Efficient chaotic iteration strategies with widenings. In: International Conference on Formal Methods in Programming and their Applications. Springer-Verlag, pp. 128–141.

[6] Brezinski, C., Redivo Zaglia, M., 1991. Extrapolation Methods-Theory and Practice. North-Holland.

[7] Brezinski, C., Redivo Zaglia, M., 2007. Generalizations of Aitken's process for accelerating the convergence of sequences. Computational and Applied Mathematics.

[8] Brezinski, C., Redivo Zaglia, M., 2008. A review of vector convergence acceleration methods, with applications to linear algebra problems. International Journal of Quantum Chemistry 109 (8), 1631–1639.

[9] Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S., 2005. A policy iteration algorithm for computing fixed points in static analysis of programs. In: Computer Aided Verification. Vol. 3576 of LNCS. Springer, pp. 462–475.

[10] Cousot, P., 1981. Semantic foundations of program. In: Program Flow Analysis: Theory and Applications. Prentice-Hall, Ch. 10, pp. 303–342.

[11] Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages. ACM Press, pp. 238–252.

[12] Cousot, P., Cousot, R., 1992. Abstract interpretation frameworks. Journal of Logic and Computation 2 (4), 511–547.

[13] Cousot, P., Cousot, R., 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Programming Language Implementation and Logic Programming. Vol. 631 of LNCS. Springer, pp. 269–295.

[14] Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: Symposium on Principles of Programming Languages. ACM Press, pp. 84–97.

[15] Feret, J., 2004. Static analysis of digital filter. In: European Symposium on Programming. No. 2986 in LNCS. Springer.

[16] Gaubert, S., Goubault, E., Taly, A., Zennou, S., 2007. Static analysis by policy iteration on relational domains. In: European Symposium on Programming. Vol. 4421 of LNCS. Springer, pp. 237–252.

[17] Gonnord, L., Halbwachs, N., 2006. Combining widening and acceleration in linear relation analysis. In: Static Analysis Symposium. Vol. 4134 of LNCS. Springer, pp. 144–160.

[18] Gopan, D., Reps, T. W., 2006. Lookahead widening. In: Computer Aided Verification. Vol. 4144 of LNCS. Springer, pp. 452–466.

[19] Gopan, D., Reps, T. W., 2007. Guided static analysis. In: Static Analysis Symposium. Vol. 4634 of LNCS. Springer, pp. 349–365.

[20] Goubault, E., Martel, M., Putot, S., 2002. Asserting the precision of floating-point computations: a simple abstract interpreter. In: European Symposium on Programming. Vol. 2305 of LNCS. Springer, pp. 209–212.

[21] Goubault, E., Putot, S., 2006. Static analysis of numerical algorithms. In: Symposium on Static Analysis. Vol. 4134 of LNCS. Springer, pp. 18–34.

[22] Goubault, E., Putot, S., 2011. Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. A. (Eds.), VMCAI. Vol. 6538 of Lecture Notes in Computer Science. Springer, pp. 232–247.

[23] Goubault, E., Putot, S., Baufreton, P., Gassino, J., 2007. Static analysis of the accuracy in control systems: principles and experiments. In: Formal methods for industrial critical systems. Vol. 4916 of LNCS. Springer-Verlag, pp. 3–20.

[24] Graves-Morris, P. R., 1992. Extrapolation methods for vector sequences. Numerische Mathematik 61 (4), 475–487.

[25] Halbwachs, N., Proy, Y.-E., Roumanoff, P., 1997. Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11 (2), 157–185.

[26] Jeannet, B., Miné, A., 2009. APRON: A library of numerical abstract domains for static analysis. In: Computer Aided Verification. Vol. 5643 of LNCS. pp. 661–667.

[27] Le Verge, H., Feb. 1992. A note on Chernikova's Algorithm. Tech. Rep. 635, IRISA, Rennes, France.

[28] Leroux, J., Sutre, G., 2007. Accelerated data-flow analysis. In: Static Analysis Symposium. Vol. 4634 of LNCS. Springer, pp. 184–199.

[29] Miné, A., 2004. Relational abstract domains for the detection of floating-point run-time errors. In: European Symposium on Programming. Vol. 2986 of LNCS. Springer, pp. 3–17.

[30] Miné, A., 2006. The octagon abstract domain. Higher-Order and Symbolic Computation 19 (1), 31–100.

[31] Monniaux, D., 2005. Compositional analysis of floating-point linear numerical filters. In: Computer-aided verification. No. 3576 in Lecture Notes in Computer Science. Springer Verlag, pp. 199–212.

[32] Müller-Olm, M., Seidl, H., 2004. Computing polynomial program invariants. Information Processing Letters 91 (5), 233–244.

[33] Rodriguez-Carbonell, E., Kapur, D., 2007. Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42 (4), 443–476.

[34] Rudin, W., 1976. Principles of Mathematical Analysis. McGraw-Hill.

[35] Sankaranarayanan, S., Sipma, H. B., Manna, Z., 2005. Scalable analysis of linear systems using mathematical programming. In: Verification, Model Checking, and Abstract Interpretation. Vol. 3385 of LNCS. Springer, pp. 25–41.

[36] Schrammel, P., Jeannet, B., 2010. Extending abstract acceleration methods to data-flow programs with numerical inputs. Electronic Notes in Theoretical Computer Science 267 (1), 101 – 114, proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

[37] Simon, A., King, A., 2006. Widening polyhedra with landmarks. In: Asian Symposium on Programming Languages and Systems. Vol. 4279 of LNCS. Springer, pp. 166–182.

[38] Simon, A., King, A., Howe, J., 2003. Two variables per linear inequality as an abstract domain. In: Logic Based Program Synthesis and Transformation. Vol. 2664 of LNCS. pp. 955–955.

[39] Tarski, A., 1955. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309.

[40] Winskel, G., 1993. The formal semantics of programming languages. MIT Press.

[41] Wynn, P., 1961. The epsilon algorithm and operational formulas of numerical analysis. Mathematics of Computation 15 (74), 151–158.

A. A counter example for extraction and combination on polyhedra.

In this section, we consider the set $\mathcal{P}$ of two-dimensional polyhedra on the variables x and y and the two functions Λ_3, Υ_3 defined by Equations (4.5) that we remind here:

$$\Lambda_3 : \begin{cases} \mathcal{P} \to \mathbb{R}^{12} \\ \left(\bigwedge_{i=1}^{n} \alpha_i x + \beta_i y \le c_i\right) \mapsto \langle \alpha_1, \beta_1, c_1, \ldots, \alpha_4, \beta_4, c_4 \rangle \end{cases}$$

$$\Upsilon_3 : \begin{cases} \mathbb{R}^{12} \to \mathcal{P} \\ \langle \alpha_1, \beta_1, c_1, \ldots, \alpha_4, \beta_4, c_4 \rangle \mapsto \bigwedge_{i=1}^{4} \alpha_i x + \beta_i y \le c_i \end{cases}$$

We now build a sequence of polyhedra $(P_n) \in \mathcal{P}^{\mathbb{N}}$ such that:

(1) $\forall n \in \mathbb{N},\ P_n \subseteq P_{n+1}$,

(2) $\bigcup_{n \in \mathbb{N}} P_n = P$ is bounded,

(3) $\Lambda_3(P_n) \in \left(\mathbb{R}^{12}\right)^{\mathbb{N}}$ is not convergent.

The first elements of the sequence (P_n) are depicted on Figure A.1. They are build as follows. Let $k_n = 1 - \frac{1}{2n}$ for all $n \ge 1$. We define the sequences $a_n,\ b_n,\ a'_n$ and b'_n by:

$$a_n = \frac{k_{n+1} - 2k_n}{k_{n+1}},\ b_n = k_n,\ a'_n = \frac{k_{n+1}}{k_{n+1} - 2k_n},\ b'_n = -\frac{k_n k_{n+1}}{k_{n+1} - 2k_n}\ .$$

We then define the sequences of polyhedra:

$$R_n = \left(-x \le 0 \wedge -y \le 0 \wedge y + x \le k_n\right)$$

and

$$Q_n = \left(-x \le 0 \wedge -y \le 0 \wedge y - a_n x \le b_n \wedge y - a'_n x \le b'_n\right)\ .$$

The polyhedra sequence we consider is then (P_n) defined by:

$$\forall n \in \mathbb{N},\ \begin{cases} P_{2n} = R_n \\ P_{2n+1} = Q_n \end{cases}\ . \tag{A.1}$$

The sequence (P_n) is monotone and, if we note $P = -x \le 0 \wedge -y \le 0 \wedge x + y \le 1$, we have $\bigcup_{n \ge 1} P_n = P$. However, the extracted sequence $\Lambda_3(P_n)$ is given by:

$$\Lambda_3(P_{2n}) = \langle -1, 0, 0, 0, -1, 0, 1, 1, k_n, 0, 0, 0 \rangle$$

and

$$\Lambda_3(P_{2n}) = \langle -1, 0, 0, 0, -1, 0, 1, -a_n, b_n, 1, -a'_n, b'_n \rangle\ .$$

We thus have, for all $n \in \mathbb{N}$, $d\left(\Lambda_3(P_{2n}), \Lambda_3(P_{2n+1})\right) \ge 1$, so the sequence $\Lambda_3\left(P_n\right)$ is not convergent.

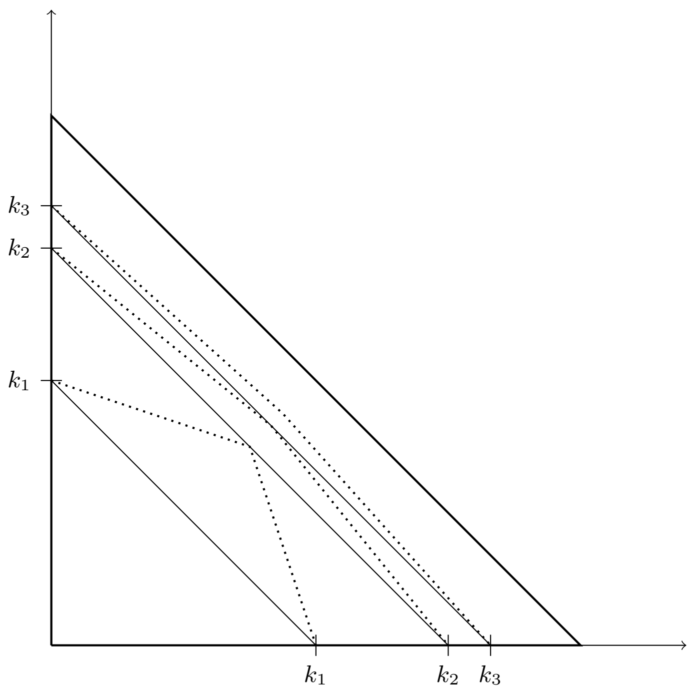


Fig. A.1. The first 6 elements of the sequence of polyhedra P_n of Equation (A.1). The dotted lines represent the polyhedra P_{2n+1} and the full lines are the polyhedra P_{2n}. The thick line represents the limit of the sequence.