Interval Slopes as Numerical Abstract Domain for Floating-Point Variables

Alexandre Chapoutot

▶ **To cite this version:**

Alexandre Chapoutot. Interval Slopes as Numerical Abstract Domain for Floating-Point Variables. 2010. hal-00469007v1

HAL Id: hal-00469007

https://hal.archives-ouvertes.fr/hal-00469007v1

Preprint submitted on 1 Apr 2010 (v1), last revised 18 Jun 2010 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables

Alexandre Chapoutot
LIP6 - Université Pierre et Marie Curie
4, place Jussieu F-75252 Paris Cedex 05 France
alexandre.chapoutot@lip6.fr

April 1, 2010

Abstract

The design of embedded control systems is mainly done with model-based tools such as Matlab/Simulink. Numerical simulation is the central technique of development and verification of such tools. Floating-point arithmetic, that is well-known to only provide approximated results, is omnipresent in this activity. In order to validate the behaviors of numerical simulations using abstract interpretation-based static analysis, we present, theoretically and with experiments, a new relational abstract domain dedicated to floating-point variables. It comes from interval expansion of non-linear functions using slopes and it is able to mimic all the behaviors of the floating-point arithmetic. It is hence adapted to prove the absence of run-time errors or to analyze the numerical precision of embedded control systems.

1 Introduction

Embedded control systems are made of a software and a physical environment which aim at continuously interact with each other. The design of such systems is usually realized with the model-based paradigm. Matlab/Simulink[1] is one of the most used tool for this purpose. It offers a convenient way to describe the software and the physical environment in an unified formalism. In order to verify that the control law, implemented in the software, fits the specification of the system, several numerical simulations are made under Matlab/Simulink. Nevertheless, this method is closer to test-based method than formal proof. Moreover, this verification method is strongly related to the floating-point arithmetic which provides approximated results.

Our goal is the use of abstract interpretation-based static analysis [1] to validate the design of control embedded software described in Matlab/Simulink. In

[1]Trademarks of The Mathworks$^{\text{TM}}$company.

our previous work [2], we defined an analysis to validate that the behaviors given by numerical simulations are close to the exact mathematical behaviors. It was based on an interval abstraction of floating-point numbers which may produce too coarse results. In this article, our work is focus on a tight representation of the behaviors of the floating-point arithmetic in order to increase the precision of the analysis of Matlab/Simulink models.

To emphasize the poor mathematical properties of the floating-point arithmetic, let us consider the sum of numbers given in Example 1 with a single precision floating-point arithmetic. The result of this sum is $-2.08616257e^{-6}$ due to rounding errors, whereas the exact mathematical result is null.

Example 1

$$\begin{aligned}0.0007 + (-0.0097) + 0.0738 + (-0.3122) + 0.7102 + (-0.5709) + (-1.0953)\\ + 3.3002 + (-2.9619) + (-0.2353) + 2.4214 + (-1.7331) + 0.4121\end{aligned}$$

Example 1 shows that the summation of floating-point numbers is a very ill-conditioned problem [3, Chap. 6]. Indeed, small perturbations on the elements to sum produce a floating-point result which could be far from the exact result. Nevertheless, it is a very common operation in control embedded software. In particular, it is used in filtering algorithms or in regulation processes, such as for example in PID regulation. Remark that depending of the case, the rounding errors may stay insignificant and the behaviors of floating-point arithmetic may be safe. In consequence, a semantic model of this arithmetic could be used to prove the behaviors of embedded control software using floating-point numbers.

The definition of abstract numerical domains for floating-point numbers is usually based on rational or real numbers [4, 5] to cope with the poor mathematical structure of the floating-point set. In consequence, these domains give an over-approximation of the floating-point behaviors. This is because they do not bring information about the kind of numerical instability which appeared during computations. We underline that our goal is not interested in computing the rounding errors but the floating-point result. In others words, we want to compute the bounds of floating-point variables without considering the numerical quality of these bounds.

Our main contribution is the definition of new numerical abstract domain, called Floating-Point Slopes (FPS), dedicated to the study of floating-point numbers. It is based on interval expansion of non-linear function named *interval slopes* introduced by Krawczyk and Neumaier [6] and, as we will show in this article, it is a relational domain. In particular in Proposition 1, we will adapt the interval slopes to deal with floating-point numbers. Moreover, we are able to tightly represent the behaviors of floating-point arithmetic with our domain. Few cases studies will show the practical use of our domain. We can hence prove properties on programs taking into account the behaviors of the floating-point arithmetic such that the absence of run-time errors or the quality of numerical computations.

Content. In Section 2, we will present the main features of the IEEE754 standard of floating-point arithmetic and we will also introduce the interval expansions of functions. We will present our abstract domain FPS in Section 3 before describing experimental results in Section 4. In Section 5, we will reference the related work before concluding in Section 6.

2 Background

We recall the main features of the IEEE754 standard of floating-point arithmetic in Section 2.1. Next in Section 2.2, we present some results from interval analysis, in particular the interval expansion of functions.

2.1 Floating-Point Arithmetic

We briefly present the floating-point arithmetic in this section, more details are available in [3] and the references therein. The IEEE754 standard [7] defines the floating-point arithmetic in base 2 which is used in almost every computer[2].

Floating-point numbers have the following form: $f = s.m.2^e$. The value s represents the *sign*, the value m is the *significand* represented with p bits and the value e is the *exponent* of the floating-point number f which belongs into the interval $[e_{\min}, e_{\max}]$ such that $e_{\max} = -e_{\min} + 1$. There are two kinds of numbers in this representation. *Normalize numbers*, the significand implicitly starts with a 1 and *denormalized numbers* implicitly starts with a 0. The later is used to gain accuracy around zero by slowly degrading the precision.

The standard defines different values of p and $e_{\min}$: $p = 24$ and $e_{\min} = -126$ for the single precision and $p = 53$ and $e_{\min} = -1022$ for the double precision. We call *normal range* the set of absolute real values in $[2^{e_{\min}}, (2 - 2^{1-p})2^{e_{\max}}]$ and the *subnormal range* the set of numbers in $[0, 2^{e_{\min}}[$.

The set of floating-point numbers (single or double precision) is represented by $\mathbb{F}$ which is closed under negation. Few special values represent special cases: the values $-\infty$ and $+\infty$ to represent the negative or the positive overflow; and the value *NaN*[3] represents invalid results such that $\sqrt{-1}$.

The standard defines round-off functions which convert exact real numbers into floating-point numbers. We are mainly concerned by the rounding to the nearest (noted fl), the rounding towards $+\infty$ and rounding toward $-\infty$. The round-off functions follow the correct rounding property, *i.e.* the result of a floating-point operation is the same that the rounding of the exact mathematical result. Note that these functions are monotone. We are interested in this article by computing the range of floating-point variables rounded to the nearest which is the default mode of rounding in computers.

A property of the round-off function fl is given in Equation (1). It characterizes the *overflow*, *i.e.* the rounding result is greater than the biggest element

[2]It also defines this arithmetic in base 10 but it is not relevant for our purpose.
[3]NaN stands for *Not A Number.*

of $\mathbb{F}$ and the case of the generation of 0. This definition only uses positive numbers, using the symmetry property of $\mathbb{F}$, we can easily deduce the definition for the negative part. We denote by $\sigma = 2^{e_{\min}-p+1}$ the smallest positive subnormal number and the largest finite floating-point number by $\Sigma = (2 - 2^{1-p})2^{e_{\max}}$.

$$\forall x \in \mathbb{F}, x > 0, \quad \mathrm{fl}(x) = \begin{cases} +0 & \text{if } 0 < x \leq \sigma/2 \\ +\infty & \text{if } x \geq \Sigma \end{cases} \tag{1}$$

An *underflow* [3, Sect. 2.3] is detected when the rounding result is less than $2^{e_{\min}}$, *i.e.* the result is in the subnormal range.

The errors associated to a correct rounding is defined in Equation (2) and it is valid for all floating-point numbers x and y except $-\infty$ and $+\infty$ (see [3, Chap. 2, Sect. 2.2]). We represent the *relative rounding error unit* by μ. In single precision, $\mu = 2^{-24}$ and $\sigma = 2^{-149}$ and in double precision, $\mu = 2^{-53}$ and $\sigma = 2^{-1074}$. The operation $\diamond$ is in $\{+, -, \times, \div\}$. Note that it is also valid for the square root operation.

$$\mathtt{fl}(x \diamond y) = (x \diamond y)(1 + \epsilon_1) + \epsilon_2 \qquad \text{with } |\epsilon_1| \leq \mu \text{ and } |\epsilon_2| \leq \frac{1}{2}\sigma \tag{2}$$

If $\mathtt{fl}(x \diamond y)$ is in the normal range or if $\diamond \in \{+, -\}$ then ϵ_2 is equal to zero. If $\mathtt{fl}(x \diamond y)$ is in the subnormal range then ϵ_1 is equal to zero.

Numerical instabilities in programs came from the rounding representation of values and they also came from two problems due to finite precision:

Absorption If $|x| \leq \mu|y|$ then it happens that $\mathrm{fl}(x+y) = \mathrm{fl}(y)$. For example, in single precision, the result of $\mathrm{fl}(1^4 - 1^{-4})$ is $\mathrm{fl}(1^4)$.The classical solution in numerical analysis is to sort the sequence of numbers in increasing order. This solution is not applicable when numbers came from the physical environment.

Cancellation It appears in the subtraction $\mathrm{fl}(x - y)$ if $(|x - y|) \leq \mu(|x| + |y|)$ then the relative errors can be arbitrary big. Indeed, the rounding errors take usually place in the least significant digits of floating-point numbers. These errors may become preponderant in the result of a subtraction when the most significant digits of two closed numbers cancelled each others. In numerical analysis, subtraction of numbers coming from long computations are avoided to limit this phenomena. We cannot apply this solution in embedded control systems where some results are used at different instants of time.

2.2 Interval Arithmetic

In this section, we introduce interval arithmetics and in particular, the interval expansion of functions which is an element of our abstract domain FPS.

2.2.1 Standard Interval Arithmetic.

The *interval arithmetic* [8] has been defined in order to avoid the problem of approximated results coming from the floating-point arithmetic. It had also been used as the first numerical abstract domain in [1].

When dealing with floating-point intervals the bounds have to be rounded to outward as in [5, Sect. 3]. In Example 2, we give the result of the interval evaluation in single precision of a sum of floating-point numbers. The exact mathematical result is 11101 while the floating-point result is 11100 due to an absorption phenomena. The floating-point result and the mathematical result are in the result interval but we cannot longer distinguish them.

Example 2 *Using the interval domain for floating-point arithmetic [5, Sect. 3] the result of the sum defined by* $\sum_{i=1}^{10} 1e^1 + \sum_{i=1}^{10} 1e^2 + \sum_{i=1}^{10} 1e^3 + \sum_{i=1}^{1000} 1e^{-3}$ *is* $[11100, 11101.953]$.

A source of over-approximation is known in the interval arithmetic as the *dependency problem* which is also known in static analysis as the non-relational aspect. For example, if X is an interval then the expression $X - X \neq 0$ in general. This problem is addressed by considering interval expansions of functions.

Notations. Interval values are noted $[a, b]$ where a is the lower bound and b is the upper bound of the interval. We denote by $[\mathtt{f}]$ the interval extension of a function $\mathtt{f}$ obtained by substitution of all the arithmetic operations with their equivalent in interval. The center of an interval X is represented by $\mathtt{mid}([a, b]) = a + 0.5 \times (b - a)$.

2.2.2 Extended Interval Arithmetic.

We are interested in the computation of the image of an interval X by a rational function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}$. In order to reduce the over-approximations in the interval arithmetic, few interval expansions have been developed. The first one is based on the *Mean-Value Theorem* and it is expressed as:

$$\mathtt{f}(X) \subseteq \mathtt{f}(z) + [\mathtt{f}'](X)(X - z) \quad \forall z \in X \ . \tag{3}$$

The first-order approximation of the range of a function $\mathtt{f}$ can be defined thanks a bound of its first order derivative $\mathtt{f}'$ over X. We can then approximate $\mathtt{f}(X)$ by a couple $(\mathtt{f}(z), [\mathtt{f}'](X))$ that are the value of $\mathtt{f}$ in the point z and the interval extension of $[\mathtt{f}']$ evaluated over X.

A second interval expansion has been defined by Krawczyk and Neumaier [6] using the notion of slopes. They reduced the approximation of the derivative form with slopes. It is defined by the relation:

$$\begin{aligned} &\mathtt{f}(X) \subseteq \mathtt{f}(z) + [\mathtt{F}^z](X)(X - z) \\ &\qquad\qquad \text{with } [\mathtt{F}^z](X) = \left\{ \frac{\mathtt{f}(x) - \mathtt{f}(z)}{x - z} : x \in X \wedge z \neq x \right\} \ . \end{aligned} \tag{4}$$

We can then represent $\mathtt{f}(X)$ by a couple $(\mathtt{f}(z), [\mathtt{F}^z](X))$ that are the value of $\mathtt{f}$ in the point z and the interval extension of the slope $[\mathtt{F}^z](X)$ of $\mathtt{f}$ over X from z respectively.

Note that the value z is chosen, in general, as the centers of the interval variables taking place in the function $\mathtt{f}$ for the both interval expansions.

An interesting feature is that we can inductively compute the derivative or the slope of a function by using *automatic differentiation* techniques [9]. It is a semantic-based method to compute derivatives. In this context, we call *independent variables* some input variables of a program with respect to derivatives are computed. We call *dependent variables* output variables whose derivatives are desired. A *derivative object* represents derivative information, such as a vector of partial derivatives like $(\partial e/\partial x_1, \ldots, \partial e/\partial x_n)$ of some expression e with respect to a vector x of independent variables. The main idea of automatic differentiation is that every complicated function $\mathtt{f}$, *i.e.* a program, is composed by simplest elements, *i.e.* program instructions. Knowing the derivatives of these elements with respect to some independent variables then following the differential calculus rules we can compute the derivatives or the slopes of $\mathtt{f}$. Furthermore, using interval arithmetic in the differential calculus rules, we can guarantee the result.

We give in Table 1 the rules to compute derivatives or slopes with respect to the structure of arithmetic expressions. We assume that we know the numbers of independent variables in the programs and we denote by n this number. The variable X represents the vector of independent variables with respect to the derivatives are computed. We denote by δ_i the interval vector of length n, having all its coordinates equal to $[0, 0]$ except the ith element equals to $[1, 1]$. So, we consider that all the independent variables are assigned to a unique position i in X and it is initially assigned with a derivative object equal to δ_i. Following Table 1 where $\mathtt{g}$ and $\mathtt{h}$ represent variables with derivative object, a constant value c has a derivative object equal to zero (the interval vector $\mathbf{0}$ has all its coordinates equal to $[0, 0]$). For addition and subtraction, the result is the vector addition or the vector subtraction of the derivative objects. For multiplication and division, it is more complicated but the rules came from the standard rules of the composition of derivatives, *e.g.* $(u \times v)' = u' \times v + u \times v'$. A proof of the computation rules[4] for slopes can be found in [6, Sect. 2]. Note also that we can applied automatic differentiation for other functions, such as the square root, using the rule of function composition, $(f \circ g)'(x) = f'(g(x))g'(x)$. In Section 3, we will also define the square root for our FPS domain.

Table 1: Automatic differentiation rules for derivatives and slopes

Function	Derivative arithmetic	Slope arithmetic
$c \in \mathbb{R}$	$\mathbf{0}$	$\mathbf{0}$
$\mathtt{g} + \mathtt{h}$	$[\mathtt{g}'](X) + [\mathtt{h}'](X)$	$[\mathtt{G}^z](X) + [\mathtt{H}^z](X)$
$\mathtt{g} - \mathtt{h}$	$[\mathtt{g}'](X) - [\mathtt{h}'](X)$	$[\mathtt{G}^z](X) - [\mathtt{H}^z](X)$
$\mathtt{g} \times \mathtt{h}$	$[\mathtt{g}'](X) \times \mathtt{h}(X) + \mathtt{g}(X) \times [\mathtt{h}'](X)$	$[\mathtt{G}^z](X) \times \mathtt{h}(X) + \mathtt{g}(z) \times [\mathtt{H}^z](X)$
$\dfrac{\mathtt{g}}{\mathtt{h}}$	$\dfrac{[\mathtt{g}'](X) \times \mathtt{h}(X) - [\mathtt{h}'](X) \times \mathtt{g}(X)}{\mathtt{h}^2(X)}$	$\dfrac{[\mathtt{G}^z](X) - [\mathtt{H}^z](X) \times \frac{\mathtt{g}(z)}{\mathtt{h}(z)}}{\mathtt{h}(X)}$

These interval expansions of functions, using either $(\mathtt{f}(z), [\mathtt{f}'](X))$ the derivative form or $(\mathtt{f}(z), [\mathtt{f}^z](X))$ the slope form, define a straightforward semantics

[4]In [6, Sect. 2], the authors went also into detail of the complexity of these operations.

of arithmetic expressions which can be used to compute bounds of variables.

Remark 1 *The difference in over-approximated result between the derivative form and the slope form resides in the rules of multiplication and division. In the derivative form, we need to evaluate the two operands while we only need to evaluate one of them in the slope form.*

Example 3 shows that we can encode, in interval slopes, the list of variables contributing in an arithmetic expression. In particular, the vector composing the interval slope of the variable t represents the influence of the variables a, b and c in the value of t. Moreover by computing interval slopes, we build step by step the set of variables related to arithmetic expressions through a program. Interval slopes represent hence relations between the input variables and the result of arithmetic expressions, more generally the program output.

Example 3 *Lets $t = a + b \times c$, we want to compute the interval slope $[\mathtt{T}^z](X)$ of t. We denote by $X = (a, b, c)$ the set of independent variables. We suppose that the interval slope expansions of a, b and c are $(z_a, [\mathtt{A}^z](X) = \delta_1)$, $(z_b, [\mathtt{B}^z](X) = \delta_2)$, and $(z_c, [\mathtt{C}^z](X) = \delta_3)$ and that the interval value associated to c is X_c* i.e. *$X_c = z_c + [\mathtt{C}^z](X)(X - z)$.*

$$\begin{aligned}
[\mathtt{T}^z](X) &= [\mathtt{A}^z](X) + z_b[\mathtt{C}^z](X) + [\mathtt{B}^z](X)\,(z_c + [\mathtt{C}^z](X)(X - z)) \\
&= ([1,1], 0, 0) + z_b \times (0, 0, [1,1]) + (0, [1,1], 0) \times X_c \\
&= ([1,1], [1,1] \times X_c, z_b \times [1,1]) \\
&= ([1,1], X_c, [z_b, z_b])
\end{aligned}$$

3 Floating-Point Slopes

We present in this section, our new abstract domain `FPS`. In Section 3.1, we adapt the computation rules of interval slopes to take into account floating-point arithmetic. Next in Section 3.2, we define the lattice structure of the `FPS` domain. And in Section 3.3, we define an abstract semantics of arithmetic expressions over `FPS` values taking into account the behaviors of floating-point arithmetic.

3.1 Floating-Point Version of Interval Slopes

The definition of interval slope expansion in Section 2.2 manipulates real numbers. In case of floating-point numbers, we have to take into account the round-off function and then the rounding-errors.

We show in Proposition 1 that the range of a rational function **f** of floating-point numbers can be soundly over-approximated by a floating-point slope. The function `f` must respect the correct rounding. In other words, the result of an operation over set of floating-point numbers is over-approximated by the result of the same operation over floating-point slopes by adding a small quantity depending on the relative rounding error unit μ and the absolute error σ.

Remark 2 *As the floating-point version of slopes is based on μ and σ, we can represent the floating-point behaviors depending of the hardware. For example extended precision*[5] *is represented using the values $\mu = 2^{-64}$ and $\sigma = 2^{-16446}$. Furthermore following [10], we can compute the result of a double rounding*[6] *with $\mu = (2^{11}+2)2^{-64}$ and $\sigma = (2^{11}+1)2^{-1086}$.*

Proposition 1 *Let $\mathtt{f} : D \subseteq \mathbb{R}^n \to \mathbb{R}$ be an arithmetic operation of the form $g \diamond h$ with $\diamond \in \{+,-,\times,\div\}$ or $\sqrt{g}$,* i.e. $\mathtt{f}$ *respects the correct rounding. For all $X \subseteq D$ and $z \in D$, we have:*

$$\mathrm{fl}\big(\mathtt{f}(X)\big) \subseteq \mathtt{f}(z)\big(1+[-\mu,\mu]\big) + \Big[-\frac{\sigma}{2},\frac{\sigma}{2}\Big] + [\mathtt{F}^z](X)(X-z)\big(1+[-\mu,\mu]\big) \ .$$

Proof 1

$$\begin{aligned}
\mathrm{fl}\big(\mathtt{f}(X)\big) &= \{\mathtt{f}(x)(1+\varepsilon_x)+\bar{\varepsilon}_x : x \in X\} && \textit{by Eq. } (2)\\
&= \mathtt{f}(X) + \mathtt{f}(X)\{\varepsilon_x : x \in X\} + \{\bar{\varepsilon}_x : x \in X\} &&\\
&\subseteq \big(\mathtt{f}(z) + [\mathtt{F}^z](X)(X-z)\big) + \{\bar{\varepsilon}_x : x \in X\} && \textit{by Eq. } (4)\\
&\quad + \big(\mathtt{f}(z) + [\mathtt{F}^z](X)(X-z)\big)\{\varepsilon_x : x \in X\} &&\\
&\subseteq \mathtt{f}(z)\big(1+\{\varepsilon_x : x \in X\}\big) + \{\bar{\varepsilon}_x : x \in X\} &&\\
&\quad + [\mathtt{F}^z](X)(X-z)\big(1+\{\varepsilon_x : x \in X\}\big) &&\\
&\subseteq \mathtt{f}(z)\big(1+[-\mu,\mu]\big) + \Big[-\frac{\sigma}{2},\frac{\sigma}{2}\Big] && |\varepsilon_x| \le \mu \ \textit{by Eq. } (2)\\
&\quad + [\mathtt{F}^z](X)(X-z)\big(1+[-\mu,\mu]\big) && |\bar{\varepsilon}_x| \le \frac{1}{2}\sigma \ \textit{by Eq. } (2)
\end{aligned}$$

Proposition 1 shows that we can compute the floating-point range of a function $\mathtt{f}$, respecting the correct rounding, using interval slopes expansion. This expansion is represented by the couple:

$$\Big([\mathtt{f}]\,(z)\big(1+[-\mu,\mu]\,\big) + \Big[-\frac{\sigma}{2},\frac{\sigma}{2}\Big], \quad [\mathtt{F}^z](X)\big(1+[-\mu,\mu]\big)\Big) \ .$$

The first element is a small interval rounding to outward around $\mathtt{f}(z)$ for which we have to take into account the possible rounding errors. The second element is the interval slopes which have to take into account of relative errors. Note that this adaptation adds a very little overhead of computations in regards to the definition of interval slopes by Krawczyk and Neumaier.

3.2 Lattice Structure

In this section, we define the lattice structure of the set of floating-point slopes. In particular, this structure is based on the complete lattice of intervals denoted in the following by $\langle \mathbb{I}, \sqsubseteq_{\mathbb{I}}, \bot_{\mathbb{I}}, \top_{\mathbb{I}}, \sqcup_{\mathbb{I}}, \sqcap_{\mathbb{I}} \rangle$. We denote by $\mathbb{S}$ the set of slopes. An

[5]In some hardware, *e.g.* Intel x87, floating-point numbers may be encoded with 80 bits in registers, *i.e.* the significand is 64 bits long.

[6]It may happen on hardware using extended precision. Results of computations are rounded in registers and then with a less precision in memory.

element s of $\mathbb{S}$ is represented by a couple (m, S) where m is a floating-point interval and S is a vector of floating-point intervals.

The abstraction function $\alpha_{\mathbb{S}}$ is defined in Equation (5). It is applied on all the dependent variables of analyzed programs. We denote by X the vector of independent variables. An interval of floating-point numbers $[a, b]_i$ associated to the ith input is abstracted by a couple (m, S). The value m represents the center of $[a, b]$ *i.e.* $m = \texttt{mid}([a, b])$. The value S represents the interval slope associated to the ith independent variable, *i.e.* $S = \delta_i$. Note that we consider single value a has the interval $[a, a]$.

$$\alpha_{\mathbb{S}}([a, b]_i) = (\texttt{mid}([a, b]), \delta_i) \tag{5}$$

For example, the interval $A = [3, 8]_2$ representing the second independent variable of the program, is abstracted by $\alpha_{\mathbb{S}}(A) = (5.5, \delta_2)$ where the value 5.5 is the center of the interval $[3, 8]$.

We assume that the values of independent variables are kept in a separate interval vector V_X for the concretization. The concretization function $\gamma_{\mathbb{S}}$ is defined in Equation (6). From the abstract value (m, S), we represent the set of floating-point numbers by an approximation over slopes. The notation $\texttt{mid}(V_X)$ stands for the component-wise application of the function `mid` on all the components of the vector V_X.

$$\gamma_{\mathbb{S}}(m, S) = m + S \times (V_X - \texttt{mid}(V_X)) \tag{6}$$

We define a partial order, the join and the meet operations between elements of $\mathbb{S}$. All these operations are defined as a component-wise application of the associated operations of the interval domain. We denote by $\dot{\sqsubseteq}_{\mathbb{I}}$ the component-wise application of the interval order. We can define a partial order $\sqsubseteq_{\mathbb{S}}$ between elements of $\mathbb{S}$ in the following manner:

$$\begin{aligned}&\forall (m_g, S_g), (m_h, S_h) \in \mathbb{S},\\ &\qquad (m_g, S_g) \sqsubseteq_{\mathbb{S}} (m_h, S_h) \Leftrightarrow m_g \sqsubseteq_{\mathbb{I}} m_h \wedge (\forall x \in S_g, y \in S_h, \quad x \dot{\sqsubseteq}_{\mathbb{I}} y)\ .\end{aligned} \tag{7}$$

The join operation $\sqcup_{\mathbb{S}}$ over floating-point slopes is defined in Equation (8). We denote by $\dot{\sqcup}_{\mathbb{I}}$ the component-wise application of the operation $\sqcup_{\mathbb{I}}$.

$$\begin{aligned}&\forall (m_g, S_g), (m_h, S_h) \in \mathbb{S}, \quad \big(m_g, S_g\big) \sqcup_{\mathbb{S}} \big(m_h, S_h\big) = \big(m, S\big)\\ &\qquad\qquad \text{with} \quad m = m_g \sqcup_{\mathbb{I}} m_h \quad \text{and} \quad S = S_g \dot{\sqcup}_{\mathbb{I}} S_h\ .\end{aligned} \tag{8}$$

The meet operation $\sqcap_{\mathbb{S}}$ over floating-point slopes is defined in Equation (9). We denote by $\dot{\sqcap}_{\mathbb{I}}$ the component-wise application of the operation $\sqcap_{\mathbb{I}}$.

$$\begin{aligned}&\forall (m_g, S_g), (m_h, S_h) \in \mathbb{S}, \quad \big(m_g, S_g\big) \sqcap_{\mathbb{S}} \big(m_h, S_h\big) = \big(m, S\big) \quad \text{with}\\ &\qquad\qquad m = m_g \sqcap_{\mathbb{I}} m_h \quad \text{and} \quad S = S_g \dot{\sqcap}_{\mathbb{I}} S_h\ .\end{aligned} \tag{9}$$

Proposition 2 states that the set $\mathbb{S}$ with the order $\sqsubseteq_{\mathbb{S}}$ is a complete lattice. We consider that $\bot_{\mathbb{S}}$ is the least element and that $\top_{\mathbb{S}}$ is the upper element of $\mathbb{S}$.

Proposition 2 *The tuple* $\langle \mathbb{S}, \sqsubseteq_{\mathbb{S}}, \perp_{\mathbb{S}}, \top_{\mathbb{S}}, \sqcup_{\mathbb{S}}, \sqcap_{\mathbb{S}} \rangle$ *is a complete lattice.*

Proof 2 *The set* $\mathbb{S}$ *is defined as the finite product of the partial ordered set* $(\mathbb{I}, \sqsubseteq_{\mathbb{I}})$ *where the order is component-wisely defined. The product of complete lattices is a complete lattice.*

3.3 Semantics of Arithmetic Operations

In this section, we define the semantics of arithmetic operations over elements of floating-point slopes domain. Moreover, we define this semantics to mimic the behaviors of the floating-point arithmetic. We begin by defining some auxilary functions.

We can detect overflow and genration of zero result by using the function Φ defined in Equation (10). We have two kinds or rules: the *total* rule when we are certain that a zero or an overflow occur and the *partial* rule when a part of the set described by a floating-point slope generates a zero or an overflow. For an element $(m, S) \in \mathbb{S}$ and following the concrete value $\gamma_{\mathbb{S}}(m, S)$, we can determine if (m, S) represents an overflow or a zero. We represent hence the finite precision of the floating-point arithmetic. We denote by $\mathbf{p}_\infty$ and by $\mathbf{m}_\infty$ the interval vectors with all their components equal to $[+\infty, +\infty]$ and $[-\infty, -\infty]$ respectively. We recall that σ is the smallest denormalized and Σ is the largest floating-point numbers.

$$\Phi(m,S) = \begin{cases} (0, \mathbf{0}) & \text{if } \gamma_{\mathbb{S}}(z,S) \sqsubseteq_{\mathbb{I}} [-\frac{\sigma}{2}, \frac{\sigma}{2}] \\ (\tilde{m}, \mathbf{0} \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \gamma_{\mathbb{S}}(m,S) \sqcap_{\mathbb{I}}]-\frac{\sigma}{2}, \frac{\sigma}{2}[\; \neq \emptyset \\ & \text{and } \tilde{m} = \begin{cases} 0 & \text{if } m \sqsubseteq_{\mathbb{I}}]-\frac{\sigma}{2}, \frac{\sigma}{2}[\\ [0,0] \sqcap_{\mathbb{I}} m & \text{otherwise} \end{cases} \\ (+\infty, \mathbf{p}_\infty) & \text{if } \gamma_{\mathbb{S}}(m,S) \sqsubseteq_{\mathbb{I}}]\Sigma, +\infty] \\ (\tilde{m}, \mathbf{p}_\infty \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \gamma_{\mathbb{S}}(m,S) \sqcap_{\mathbb{I}}]\Sigma, +\infty] \neq \emptyset \\ & \text{and } \tilde{m} = \begin{cases} +\infty & \text{if } m \sqsubseteq_{\mathbb{I}}]\Sigma, +\infty] \\ [+\infty, +\infty] \sqcap_{\mathbb{I}} m & \text{otherwise} \end{cases} \\ (-\infty, \mathbf{m}_\infty) & \text{if } \gamma_{\mathbb{S}}(m,S) \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ (\tilde{m}, \mathbf{m}_\infty \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \gamma_{\mathbb{S}}(m,S) \sqcap_{\mathbb{I}} [-\infty, -\Sigma[\; \neq \emptyset \\ & \text{and } \tilde{m} = \begin{cases} -\infty & \text{if } m \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ [-\infty, -\infty] \sqcap_{\mathbb{I}} m & \text{otherwise} \end{cases} \\ (m, S) & \text{otherwise} \end{cases} \tag{10}$$

Equation (10) is an adaptation to deal with FPS values of the rule defined in Equation (1).

An interesting feature is that interval slopes can be used to encode the results of floating-point operations. In particular, we can mimic the absorption phenomena by setting to zero the interval slope of the absorbed operand. We define the function ρ for this purpose. It will be used to represent the absorption phenomena. The reduction of an abstract value $g = (m_g, S_g)$ in regards to the abstract value $h = (m_h, S_h)$, which is denoted by $\rho(g \mid h)$, is defined in

Equation (11).

$$\rho(g \mid h) = \begin{cases} (0, \mathbf{0}) & \text{if } \gamma_{\mathbb{S}}(m_g, S_g) \sqsubseteq_{\mathbb{I}} \mu\gamma_{\mathbb{S}}(m_h, S_h) \\ (\tilde{m}_g, \mathbf{0} \mathbin{\dot{\sqcup}_{\mathbb{I}}} S_g) & \text{if } \gamma_{\mathbb{S}}(m_g, S_g) \sqcap_{\mathbb{I}} \mu\gamma_{\mathbb{S}}(m_h, S_h) \neq \emptyset \\ & \text{and } \tilde{m}_g = \begin{cases} 0 & \text{if } m_g \sqsubseteq_{\mathbb{I}} \mu\gamma_{\mathbb{S}}(m_h, S_h) \\ [0,0] \sqcap_{\mathbb{I}} m_g & \text{otherwise} \end{cases} \\ (m_g, S_g) & \text{otherwise} \end{cases} \tag{11}$$

Equation (11) models the absorption phenomena by explicitly setting to zero the values of a slope. As mentioned in Section 2.2, a slope shows which variables influence the computation of an arithmetic expression. An absorption phenomena induces that an operand does not influence the result of an addition or a subtraction any more.

Using the function Φ and the function ρ, we inductively define on the structure of arithmetic expressions the abstract semantics $[\![.]\!]^{\sharp}_{\mathbb{S}}$ of floating-point slopes in Figure 1. We denote by $\theta^{\sharp}$ an abstract environment which associates to each program variable a floating-point slope. For each arithmetic operation, we component-wisely combine the elements of the abstract operands $[\![g]\!]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = (m_g, S_g)$ and $[\![h]\!]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = (m_h, S_h)$. The first element is obtained using the interval arithmetic with outward rounding. The second element is computed using the definition of the slope arithmetic defined in Table 1. Note that we also define the semantics of the square root operation. We take into account of the possible rounding errors in the result (m, S) following Proposition 1. In case of addition and subtraction, according to the Equation (2), we do not consider absolute error $\frac{\sigma}{2}$ which is always null. Moreover, in case of addition or subtraction, we handle the absorption phenomena using the function ρ, defined in Equation (11). Finally, we check if a zero or an overflow is generated by applying the function Φ defined in Equation (10).

Remark 3 *The functions Φ and ρ make the arithmetic operations on floating-point slopes non associative and non distributive as in floating-point arithmetic.*

3.4 Note on the Acceleration of Convergence

In order to enforce the convergence of the fixed-point computation, we can define a widening operation $\nabla_{\mathbb{S}}$ over floating-point slopes values. An advantage of our domain is that we can straightforwardly use the widening operations defined for the interval domain denoted by $\nabla_{\mathbb{I}}$. We define the operator $\nabla_{\mathbb{S}}$ in Equation (12) using the widening operator between intervals. The notation $\dot{\nabla}_{\mathbb{I}}$ represents the component-wise application of $\nabla_{\mathbb{I}}$ between the components of the interval slopes vector.

$$\begin{aligned} \forall (m_g, S_g), (m_h, S_h) \in \mathbb{S}, \quad & (m_g, S_g) \nabla_{\mathbb{S}} (m_h, S_h) = (m, S) \\ & \text{with} \quad m = m_g \ \nabla_{\mathbb{I}} \ m_h \quad \text{and} \quad S = S_g \ \dot{\nabla}_{\mathbb{I}} \ S_h \end{aligned} \tag{12}$$

As FPS is based on the interval domain, we can benefit of all the widening of intervals such as the widening with thresholds defined in [5, Sect. 7].

$$
\begin{aligned}
\llbracket g \pm h \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) &= \Phi\left((\tilde{m}_g \pm \tilde{m}_h)(1+[-\mu,\mu]), \quad \left(\tilde{S}_g \pm \tilde{S}_h\right)(1+[-\mu,\mu])\right) \\
&\qquad \text{with} \quad (\tilde{m}_g, \tilde{S}_g) = \rho(g \mid h) \quad \text{and} \quad (\tilde{m}_h, \tilde{S}_h) = \rho(h \mid g) \\
\llbracket g \times h \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) &= \Phi\left(m, \quad \left(S_g \times \gamma_{\mathbb{S}}(m_h, S_h) + m_g \times S_h\right)(1+[-\mu,\mu])\right) \\
&\qquad \text{with} \quad m = (m_g \times m_h)(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right] \\
\left\llbracket \frac{g}{h} \right\rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) &= \Phi\left(m, \quad \frac{S_g - S_h \frac{m_g}{m_h}}{\gamma_{\mathbb{S}}(m_h, S_h)}(1+[-\mu,\mu])\right) \text{with } 0 \notin \gamma_{\mathbb{S}}(m_h, S_h) \\
&\qquad \text{with} \quad m = \frac{m_g}{m_h}(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right] \\
\llbracket \sqrt{g} \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) &= \Phi\left(m, \quad \left(\frac{S_g}{\sqrt{m_g} + \sqrt{\gamma_{\mathbb{S}}(m_g, S_g)}}\right)(1+[-\mu,\mu])\right) \\
&\qquad \text{with} \quad m = (\sqrt{m_g})(1+[-\mu,\mu]) + [-\frac{\sigma}{2}, \frac{\sigma}{2}] \\
&\qquad \text{and} \quad \sqrt{m_g} \sqcap_{\mathbb{I}} [-\infty, 0] = \emptyset \\
&\qquad \text{and} \quad \gamma_{\mathbb{S}}(m_g, S_g) \sqcap_{\mathbb{I}} [-\infty, 0] = \emptyset
\end{aligned}
$$

Figure 1: Abstract semantics of arithmetic expressions on floating-point slopes

4 Case Studies

In this section, we present experimental results of the static analysis of numerical programs using our floating-point slope domain. We based our examples on Matlab/Simulink models which are block-diagrams. We present as examples a second order linear filter and the computation of a square root with a Newton method.

We first give a quick view of Matlab/Simulink models. In a Matlab/Simulink block-diagram, each node represents an operation and each wire represents a value which evolve during time. We consider few operations such that arithmetic operations, gain operation that is multiplication by a constant, conditional statement (called *switch* in Simulink), and unit delay block represented by $\frac{1}{z}$ which acts as a memory. We can hence write discrete-time models thanks to finite difference equations, see [2] for further details.

4.0.1 Linear filter.

We applied the floating-point slope domain on a second order linear filter. It is defined by the following recurrence equation:

$$y_n = x_n + 0.7x_{n-1} + x_{n-2} + 1.2y_{n-1} - 0.7y_{n-2} \ .$$

The Simulink block-diagrams of this filter is given in Figure 2(a). The input belongs into the interval $[0.71, 1.35]$ and its output is given in Figure 2(b). The gray area represents all the possible trajectories of the output corresponding of the set of inputs. We can hence bound the output by the interval $[0.7099, 9.8269]$.

4.0.2 Newton method.

We applied our domain on a Newton algorithm which computes the square root. The square root of a number a is computed using the iterative sequence defined by:

$$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n} \ .$$

We unfold the loop and the results are computed in five iterations. The Simulink model is given in Figure 3(a) and in Figure 3(b), we give the Simulink model associated to one iteration of the algorithm.

For the interval input $[4, 8]$ with the initial value equals to 2, we have the result $[1.8547, 3.0442]$.

5 Related Work

Numerical domains have been intensively studied. A large part of numerical domains concern the polyhedral representation of sets. For example, we have the domain of polyhedron [11] and the variants [12, 13, 14, 15, 16, 17, 18, 19, 20]. We also have the numerical domains based on affine relations between variables [21, 22] or the domain of linear congruences [23]. In general, all these domains are based on arithmetic with "good" properties such that rational numbers or real numbers. A notable exception is the floating-point version of the octagon domain [5] and the floating-point version of the domain of polyhedron [24]. These domains give a sound over-approximation of the floating-point behaviors but they are not empowered to model the behaviors of floating-point arithmetic as we do.

Our FPS domain is more general than numerical abstract domains made for a special purpose. For example, we have the domain for linear filters [25] or for the numerical precision [26] which provide excellent results. Nevertheless as we showed in Section 4, we can handle a large variety of algorithms without loosing too much precision.

6 Conclusion

We have presented a new relational abstract numerical domain called FPS dedicated to floating-point variables. It is based on Krawczyk and Neumaier's work [6] on interval expansion of rational function using interval slopes. This domain is able to mimic the behaviors of the floating-point arithmetic such that the *absorption* phenomena. We have also presented experimental results showing the practical use of this domain in various contexts.

We want to pursue the work on the FPS domain by model more closely the behaviors of floating point arithmetic, for example by taking into account the hardware instructions [27, Sect. 3].

As an other future work, we want to apply FPS domain for the analyses of the numerical precision by combining the FPS domain and domains defined in [28, 29]. An interesting direction should be to make an analysis of the numerical precision by comparing results of the FPS domain and results coming from the other numerical domain which bound the exact mathematical behaviors such that [24]. We can hence avoid to manipulate complex abstract values to represent rounding errors such as in [26, 29].

References

[1] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM, 1977.

[2] A. Chapoutot and M. Martel. Abstract simulation: a static analysis of Simulink models. In *International Conference on Embedded Systems and Software*, pages 83–92. IEEE Press, 2009.

[3] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of floating-point arithmetic*. Birkhauser Boston, 2009.

[4] E. Goubault. Static analyses of floating-point operations. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001.

[5] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[6] R. Krawczyk and A. Neumaier. Interval slopes for rational functions and associated centered forms. *Journal on Numerical Analysis*, 22(3):604–616, 1985.

[7] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. Institute of Electrical, and Electronic Engineers, 2008.

[8] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied interval analysis*. Springer, 2001.

[9] C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107. ACM, 2002.

[10] S. Boldo and T.M.T. Nguyen. Hardware-independant proofs of numerical programs. In *NASA Formal Methods Symposium*, 2010.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, pages 84–97. ACM, 1978.

[12] Axel Simon, Andy King, and Jacob Howe. Two variables per linear inequality as an abstract domain. In *Logic Based Program Synthesis and Transformation*, volume 2664 of *LNCS*, pages 955–955, 2003.

[13] A. Miné. The Octagon abstract domain. *Journal of Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[14] Sriram Sankaranarayanan, Michael Colon, Henny Sipma, and Zohar Manna. Efficient strongly relational polyhedral analysis. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 111–125. Springer Verlag, 2006.

[15] Mathias Péron and Nicolas Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In *Verification, Model Checking and Abstract Interpretation*, volume 4349 of *LNCS*, pages 268–282. Springer, 2007.

[16] Robert Clarisó and Jordi Cortadella. The Octahedron abstract domain. *Journal of Science Computer Programming*, 64(1):115–139, 2007.

[17] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*, pages 184–188. ACM, 2008.

[18] Vincent Laviron and Francesco Logozzo. Subpolyhedra: a (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 229–244, 2009.

[19] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: an abstract domain to infer interval linear relationships. In *Static Analysis Symposium*, volume 5673 of *LNCS*, pages 309–325. Springer, 2009.

[20] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. An abstract domain to discover interval linear equalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *LNCS*, pages 112–128. Springer, 2010.

[21] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[22] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Computer Aided Verification*, pages 627–633, 2009.

[23] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT Vol.1*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[24] Liqian Chen, Antoine Miné, and Cousot Patrick. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 3–18. Springer, 2008.

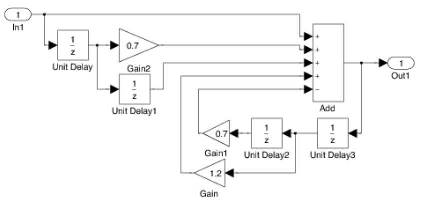
[25] J. Férêt. Static analysis of digital filter. In *European Symposium on Programming*, number 2986 in LNCS, pages 33–48. Springer, 2004.

[26] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.

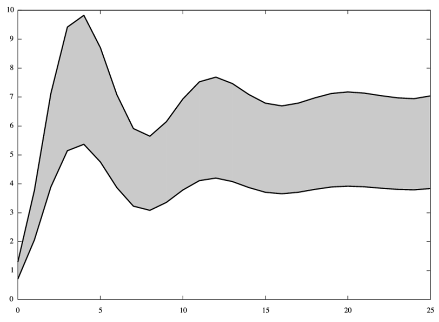
[27] D. Monniaux. Compositional analysis of floating-point linear numerical filters. In *Computer-Aided Verification*, volume 3576 of *LNCS*, pages 199–212. Springer, 2005.

[28] M. Martel. Semantics of roundoff error propagation in finite precision computations. *Journal of Higher Order and Symbolic Computation*, 19(1):7–30, 2004.

[29] Alexandre Chapoutot and Matthieu Martel. Automatic differentiation and Taylor forms in static analysis of numerical programs. *Technique et Science Informatiques*, 28(4):503–531, 2009. in French.

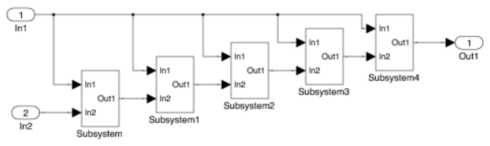


(a) Block-Diagram of the linear filter.

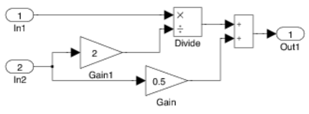


(b) Temporal evolution of the output.

Figure 2: Second order linear filter



(a) Main model.



(b) Content of a subsystem.

Figure 3: Simulink model of the square root computation