Boosting domain filtering over floating-point numbers with safe linear approximations

Mohammed Said Belaid, Claude Michel, Michel Rueher

Boosting domain filtering over floating-point numbers with safe linear approximations*

Mohammed Said Belaid, Claude Michel, and Michel Rueher

I3S (UNS/CNRS)
2000, route des Lucioles - Les Algorithmes - bt. Euclide B - BP 121
06903 Sophia Antipolis Cedex - France
{MSBelaid, Claude.Michel}@i3s.unice.fr, Michel.Rueher@gmail.com

Abstract. Solving constraints over floating-point numbers is a critical issue in numerous applications notably in program verification. Capabilities of filtering algorithms for constraints over the floating-point numbers have been so far limited to 2b-consistency and its derivatives. Though safe, such filtering techniques suffer from the well known pathological problems of local consistencies, e.g., inability to efficiently handle multiple occurrences of the variables. These limitations also take roots in the strongly restricted floating-point arithmetic. To circumvent the poor properties of floating-point arithmetic, we propose in this paper to build various relaxations over the reals of the problem over the floats. We show that using linear programming (LP) to shrink the domains with safe linearisations of such relaxations can be very effective for boosting filtering techniques for constraints over the floats. Preliminary experiments on a limited but relevant set of benchmarks are very promising.

1 Introduction

Modern software is more and more relying on floating-point computations. These pieces of software are often the weakest link in critical system. Thus, the verification of programs performing floating-point computations is a key issue in the development of critical software. In order to increase the reliability of such systems, tools to test and verify automatically numerical programs are required.

Methods for verifying programs performing floating-point computations are mainly derived from standard program verification methods. Bounded model checking (BMC) techniques have been widely used for finding bugs in hardware designs [3] and software [10]. SMT solvers are now used in most of the state-of-the-art BMC tools to directly work on high level formula (see [2, 8, 10]. Until now, integration of floating-point computation in BMC tools is not always satisfactory. For instance, the bounded model checker CBMC interprets the floating-point numbers as fixed point numbers [6], which is far from being safe. Tools based on abstract interpretation [9, 19] can show the absence of run-time errors (e.g., division by zero) on program working with floating-point numbers. Tools

* This work was partially supported by ANR-07-SESUR-003 project CAVERN.

based on abstract interpretation are safe since they over-approximate floating-point computations. However, over-approximations may be very large and these tools reject many valid programs. Constraint programming has also been used for program testing [12, 13] and verification [7]. Constraint programming offers many benefits like the capability to deduce information from partially instantiated problems or to exhibit counter-examples. The constraint programming framework is very flexible and it is very easy to integrate new solvers for handling a specific domain.

The point is that floating-point arithmetic can not be correctly handled by solvers over the reals. Dedicated constraint solvers are required in safe CP-based framework and BMC-SMT tools for testing[1] or verifying numerical software.

Available techniques to correctly solve floating-point constraints are based on an adaptation of classical consistencies over the reals. [18] adapts box-consistency to the floats while [17, 5] rely on 2B-consistency to correctly solve constraints over the floats. However solvers based on these techniques have difficulties to scale up and to handle large pieces of software. Part of these limitations lie in the well known pathological problems of local consistencies, e.g., inability to handle efficiently multiple occurrences of the variables. They also roots in the poorness of floating-point arithmetic.

That is why we introduce here a new method to solve constraints over the floating-point numbers which can take advantage of solvers over the reals. The basic tenet is to build correct but tight approximations over the reals of the floating-point operations. To ensure the tightness of the result, each floating-point operation is approximated according to its rounding mode. For example, assume that x and y are positive normalised floating-point numbers, then the floating-point addition $x \oplus y$ with a rounding mode set to $-\infty$, is bounded by

$$\alpha \times (x + y) < x \oplus y \leq x + y$$

where $\alpha = 1/(1 + 2^{-p+1})$ and p is the size of the significant[2] Approximations for special cases have also been refined, e.g., for the addition with a rounding mode set to zero, or for the multiplication by a constant.

Thanks to these approximations, a problem over the floating-point numbers is translated into a set of constraints over the reals. A linearisation of the nonlinear constraints is then applied to obtain a fully linear problem over the reals. This last set of constraints can directly be solved by available linear solvers over reals which are relieved from the burden of floating-point arithmetic. Preliminary experiments are very promising and these new filtering techniques could really help to scale up all verifications tools using floating-point solver. These new approximations could also help to refine the over-approximations computed by abstract interpretation tools [22].

---

[1] see FPSE, a tool designed to solve floating-point constraints coming from C programs (http://www.irisa.fr/celtique/carlier/fpse.html).

[2] A floating-point number is a triple (s, e, m) where s is the sign, e the exponent and m the significant. Its value is given by $(-1)^s m 2^e$.

1.1 An illustrative example

Before going into the details, let us illustrate our approach on a very simple example. Consider the simple constraint

$$z = x + y - x \tag{1}$$

where x, y and z are 32 bits floating-point variables. Over the real numbers, such an expression can be simplified to $z = y$. However, this is not true with floating-point numbers. For example, over the floats and with a rounding mode set to to the nearest, $10.0 + 1.0e-8 - 10.0$ is not equal to $1.0e-8$ but to 0. This absorption phenomena illustrates why expressions over the floating-point numbers can not be simplified in the same way than expressions over the real numbers.

Let us assume that $x \in [0.0, 10.0]$, $y \in [0.0, 10.0]$ and $z \in [0.0, 1.0e8]$. FP2B, a 2B-consistency algorithm adapted to floating-point constraints, first performs forward propagation of the domains of x and y on the domain of z using an interval arithmetic where interval bounds are computed with a rounding mode set to the nearest. Backward propagation being of no help here, the filtering process yields:

$$x \in [0.0, 10.0],\ y \in [0.0, 10.0],\ z \in [0.0, 20.0]$$

This result highlights the inability of classical algorithms to handle multiple occurrences. A stronger consistency like 3B-consistency would have reduced the domain of z to $[0.0, 10.018352508544921 88]$. However, 3B-consistency would fail to reduce the domain of z when x and y occur more than two times like in $z = x + y - x - y + x + y - x$.

Linear programming provides a more convenient way to handle such problems. Moreover, it offers the opportunity to take advantage of information coming from other constraints. In other words, it take benefit of a global view of the constraint system. To use from linear programming, we must first build safe relaxations over the reals of the problem over the floats. Thus, each basic operation has to be approximated according to the rounding mode. These approximations will produce the following constraint system over the reals:

$$\begin{cases} (2 - \frac{1}{1-2^{-p}})(x+y) \leq tmp1 \leq (2 - \frac{1}{1+2^{-p}})(x+y) \\ (2 - \frac{1}{1-2^{-p}})(tmp1 - x) \leq tmp2 \leq (2 - \frac{1}{1+2^{-p}})(tmp1 - x) \\ z = tmp2 \end{cases}$$

where p is the size of the significant of the floating-point variables. For instance, if x and y are in single precision, then $p = 24$. $tmp1$ approximates the result of the operation $x+y$ by means of two planes over the reals which encompass all the results of this addition over the floats. $tmp2$ does the same for the subtraction. Note that some relaxations like the product includes some nonlinear terms. In such a case, a linearisation process is required. Once the problem is fully linear, a linear solver like CPLEX can be used to reduce the domain of each variable, respectively, minimizing and maximising it.

FPLP, which stands for floating-point linear program, implements the algorithm previously sketched. A call to FPLP on constraint (1) immediately yields:

$$x \in [0, 10],\ y \in [0, 10],\ z \in [0, 10.0000023841859]$$

which is a much tighter result than the one computed by FP2B. Note that, contrary to 3B-consistency, FPLP still gives the same result with $z = x + y - x - y + x + y - x$.

1.2 Outline of the paper

The rest of this paper is organized as followed: section 2 introduces the relaxations over the reals of the constraints over the floats while section 3 shows how linear approximations of the non linear terms of the relaxations are built. Section 4 details the filtering algorithm and section 5 gives the results of our experiments. Finally, section 6 concludes the paper.

2 Relaxations of floating-point constraints

In this section, we describe approximations over the reals of floating-point constraints. These approximations which are the cornerstone of the filtering process introduced in this paper must be

- *correct*, i.e., they must preserve the whole set of solutions of the initial problem,
- *tight*, i.e., they should enclose the smallest amount of non floating-point solutions.

The approximations are built using two tools: the *relative error* and the *correctly rounded* operations. The former is a technique frequently used to analyse the precision of the computation. The latter property is ensured by any IEEE 754 [21] compliant implementation of the floating-point arithmetic: a correctly rounded operation is an operation whose result over the float is equal to the rounding of the result of the equivalent operation over the reals. In other word, let x and y be two floating-point numbers, $\odot$ and $\cdot$, be, respectively, an operation over the float and its equivalent over the real, if $\odot$ is correctly rounded then, $x \odot y = round(x \cdot y)$.

In rest of this section, we first detail how to build these approximations for a specific case before giving the approximations for all possible cases. Then, we will show how the different cases could be simplified.

2.1 A specific case

In order to explain how these over-approximations are built, let us consider the case where an operation is computed with a rounding mode set to $-\infty$ and the result of this operation is a positive and normalised floating-point number.

Such an operation, denoted $\odot$, could be any of the four basic binary operations from the floating-point arithmetic. The operands are all supposed to have the same floating-point type, i.e., either float, double or long double. In this case, the following property holds:

Proposition 1. *Let x and y be two floating-point numbers whose significant is represented by p bits. Assume that the rounding mode is set to $-\infty$ and that the domain of z, the result of $x \odot y$, belongs to the normalised positive floating-point numbers, then the following property holds:*

$$\frac{1}{1+2^{-p+1}}(x \cdot y) < x \odot y \leq (x \cdot y)$$

where $\odot$ is a basic operation over the floating-point numbers and, $\cdot$ is the equivalent operation over the real numbers.

Proof. Since IEEE 754 basic operations are correctly rounded and the rounding mode is set to $-\infty$, we have:

$$x \odot y \leq x \cdot y < (x \odot y)^{+} \tag{2}$$

$(x \odot y)^{+}$, the successor of $(x \odot y)$ within the set of floating-point numbers, can be computed by

$$(x \odot y)^{+} = (x \odot y) + ulp(x \odot y)$$

as, by definition, $ulp(x) = x^{+} - x$. Thus, it results from (2) that

$$x \odot y \leq x \cdot y < (x \odot y) + ulp(x \odot y)$$

From the second inequality, we have

$$\frac{1}{x \odot y + ulp(x \odot y)} < \frac{1}{x \cdot y}$$

By multiplying each side of the inequality by $x \odot y$ – which is a positive number – we get

$$\frac{x \odot y}{x \odot y + ulp(x \odot y)} < \frac{x \odot y}{x \cdot y}$$

By multiplying each side of the above inequality by -1 and by adding one to each side, we obtain

$$1 - \frac{x \odot y}{x \cdot y} < 1 - \frac{x \odot y}{x \odot y + ulp(x \odot y)} = \frac{ulp(x \odot y)}{x \odot y + ulp(x \odot y)} \tag{3}$$

Now, consider ϵ, the relative error defined by

$$\epsilon = \left| \frac{real\_value - float\_value}{real\_value} \right|$$

ϵ is the absolute value of the difference between the result over the reals and the result over the floats divided by the result over the reals. In the considered case, as $z > 0$ and $x \cdot y \geq x \odot y$, the relative error is given by

$$0 \leq \epsilon = \frac{x \cdot y - x \odot y}{x \cdot y} = 1 - \frac{x \odot y}{x \cdot y}$$

Thus, thanks to (3), we have

$$0 \leq \epsilon < \frac{ulp(x \odot y)}{x \odot y + ulp(x \odot y)}$$

z, the result of the operation $x \odot y$, is a binary positive and normalised floating-point number that can be written $z = m_z 2^{e_z}$. Moreover, $ulp(z) = 2^{-p} 2^{e_z}$. Therefore,

$$0 \leq \epsilon < \frac{2^{-p} 2^{e_z}}{m_z 2^{e_z} + 2^{-p} 2^{e_z}} = \frac{2^{-p}}{m_z + 2^{-p}}$$

The value of the significant of a normalised floating-point number belongs to the interval $[1.0, 2.0[$. An upper bound of the relative error ϵ is given by the minimun of $m_z + 2^{-p}$ which is reached when $m_z = 1$. Thus

$$0 \leq \epsilon < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

Since we have

$$\epsilon = \frac{x \cdot y - x \odot y}{x \cdot y}$$

we have

$$0 \leq \frac{x \cdot y - x \odot y}{x \cdot y} < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

and

$$0 \leq x \cdot y - x \odot y < (x \cdot y) \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

By multiplying each side of the inequality by -1 and adding $x \cdot y$ to each side, we finally obtain

$$\frac{1}{1 + 2^{-p+1}} (x \cdot y) < x \odot y \leq x \cdot y$$

□

2.2 Generalisation

Table 1 summarizes the over-approximations for each rounding mode and each possible cases, i.e., positive or negative floating-point numbers, as well as, normalised and denormalised floating-point numbers. Each possible case has a dedicated correct and tight approximation built in a similar way than the one of the case detailed in the previous subsection.

Rounding mode	Negative normalised	Negative denormalised	Positive denormalised	Positive normalised
to $-\infty$	$[(1+2^{-p+1})z_r, z_r]$	$[z_r - min_f, z_r]$	$[z_r - min_f, z_r]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
to $+\infty$	$[z_r, \frac{1}{(1+2^{-p+1})}z_r]$	$[z_r, z_r + min_f]$	$[z_r, z_r + min_f]$	$[z_r, (1+2^{-p+1})z_r]$
to 0	$[z_r, \frac{1}{(1+2^{-p+1})}z_r]$	$[z_r - min_f, z_r]$	$[z_r, z_r + min_f]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
to nearest	$[(1+\frac{2^{-p}}{(1+2^{-p})})z_r,$ $(1-\frac{2^{-p}}{(1-2^{-p})})z_r]$	$[z_r - \frac{min_f}{2},$ $z_r + \frac{min_f}{2}]$	$[z_r - \frac{min_f}{2}$ $z_r + \frac{min_f}{2}]$	$[(1-\frac{2^{-p}}{(1-2^{-p})})z_r,$ $(1+\frac{2^{-p}}{(1+2^{-p})})z_r]$

Table 1. Approximation of $x \odot y$ for each rounding mode where $z_r = x \cdot y$.

Note that tighter approximations for specific cases could also be computed. For example, the approximation of an addition with a rounding mode sets to $\pm\infty$ could be slightly improved. In a similar way, the structure of the problem is another source of improvements of the approximations. For example, the computation of $2 \otimes x$ being always exact[3], $2 \times x$ gives a better approximation of this constraint.

2.3 Simplified relaxations

The main issue of the previous approximations is that the solving process will have to handle the different cases. As a result, for n basic operations, the solver has to deal with 4^n potential combinations of the approximations. To decrease substantially this complexity, we provide here a combination of the four cases of each rounding mode into a single case.

Let us first analyse the case where the rounding mode is set to $-\infty$:

Proposition 2. *Let x and y be two floating-point numbers whose significant size is p and, assume that the rounding mode is set to $-\infty$, then,*

$$z_r - 2^{-p+1}|z_r| - min_f \leq x \odot y \leq z_r$$

where min_f is the smallest positive floating-point number, $\odot$ and $\cdot$ are respectively a basic binary operation over the float and its equivalent over the reals, and $z_r = x \cdot y$.

Proof. In a first step, the normalised and denormalised approximations are combined. If $z_r > 0$ then $\frac{1}{1+2^{-p+1}}z_r < z_r$. Thus,

$$\frac{1}{1+2^{-p+1}}z_r - min_f < z_r - min_f$$

and

$$\frac{1}{1+2^{-p+1}}z_r - min_f < \frac{1}{1+2^{-p+1}}z_r$$

[3] Provided that no overflow occurs.

Rounding mode	The approximation of $x \odot y$
to $-\infty$	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_f, z_r]$
to $+\infty$	$[z_r, z_r + 2^{-p+1}\lvert z_r\rvert + min_f]$
to 0	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_f, z_r + 2^{-p+1}\lvert z_r\rvert + min_f]$
to nearest	$[z_r - \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert - \frac{min_f}{2}, z_r + \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert + \frac{min_f}{2}]$

Table 2. Simplified approximations of $x \odot y$ for each rounding mode (with $z_r = x \cdot y$).

Therefore,

$$\frac{1}{1+2^{-p+1}} z_r - min_f < x \odot y \leq z_r,\ z_r \geq 0$$

When $z_r \leq 0$, we get

$$(1+2^{-p+1}) z_r - min_f < x \odot y \leq z_r,\ z_r \leq 0$$

These two approximations can be rewritten as follows,

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} z_r - min_f < x \odot y \leq z_r, z_r \geq 0 \\ z_r + 2^{-p+1} z_r - min_f < x \odot y \leq z_r, \quad z_r \leq 0 \end{cases}$$

To combine the negative and positive approximations together we can use the absolute value:

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} \lvert z_r\rvert - min_f < x \odot y \leq z_r, z_r \geq 0 \\ z_r - 2^{-p+1} \lvert z_r\rvert - min_f < x \odot y \leq z_r, \quad z_r \leq 0 \end{cases}$$

As $max\{\frac{2^{-p+1}}{1+2^{-p+1}}, 2^{-p+1}\} = 2^{-p+1}$, we get

$$z_r - 2^{-p+1}\lvert z_r\rvert - min_f \leq x \odot y \leq z_r$$

□

The same reasoning holds for other rounding modes. Table 2 summarize the simplified approximations for each rounding mode. Note that these approximations define concave sets.

3 Problem linearisation

The relaxations introduced in the previous section contain nonlinear terms that can not be directly handled by a linear program solver. In this section, we describe how these terms are approximated by sets of linear constraints.

3.1 Absolute value linearisation

To solve constraints with a linear solver, we have to linearise the absolute value. For this, we use the classical *bigM* rewriting method as follows.

$$\begin{cases} z = z_p - z_n \\ |z| = z_p + z_n \\ z_p \leq M \times b \\ z_n \leq M \times (1 - b) \end{cases}$$

where b is a boolean variable, z_p and z_n are real positive variables and, M is a very big floating-point number.

This method consists in the decomposition of z into two parts, the positive part z_p and the negative part z_n. Thanks to the two last constraints, either z_p or z_n is activated. So if z is positive then $z_n = 0$ and $|z| = z_p = z$. On the other hand, if z is negative then $z_p = 0$ and $|z| = z_n = -z$.

To avoid *bigM* method limitations some linear solvers allow to use the indicator constraints. In such a case, the two last constraints are replaced by:

$$\begin{cases} b = 0 \rightarrow z_+ = 0 \\ b = 1 \rightarrow z_- = 0 \end{cases}$$

3.2 Linearisation of the product

To linearise bilinear terms, square terms, and quotient, we use the standard techniques introduced by Sahinidis et al [23]. They have been also used in the Quad system [14] which have been designed to solve constraints over the real numbers. $x \times y$ is linearised according to Mc Cormick [15].

Let $x \in [\underline{x}, \overline{x}]$ and $y \in [\underline{y}, \overline{y}]$, then

$$\begin{cases} L1 : z - \underline{x}y - \underline{y}x + \underline{x}\underline{y} \geq 0 \\ L2 : -z + \underline{x}y + \overline{y}x - \underline{x}\overline{y} \geq 0 \\ L3 : -z + \overline{x}y + \underline{y}x - \overline{x}\underline{y} \geq 0 \\ L4 : z - \overline{x}y - \overline{y}x + \overline{x}\overline{y} \geq 0 \end{cases}$$

These linearisation have been proved to be optimal by Al-Khayyal and Falk [1].

3.3 Linearisation of x^2

Each time $x = y$, i.e., in case of $x \otimes x$, the linearisation can be improved. x^2 is underestimated by all the tangents at x^2 curve between $\underline{x}$ and $\overline{x}$. A good balance is obtained with the two tangents at the bounds of x. Thus, x^2 is linearised by

$$\begin{cases} L1 : y + \underline{x}^2 - 2\underline{x}x \geq 0 \\ L2 : y + \overline{x}^2 - 2\overline{x}x \geq 0 \\ L3 : (\underline{x} + \overline{x})x - y - \underline{x}\overline{x} \geq 0 \\ L4 : y \geq 0 \end{cases}$$

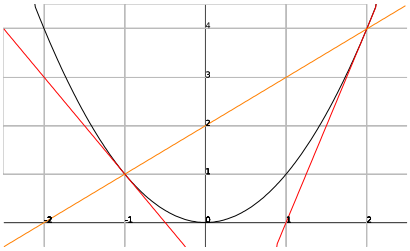

Fig. 1. Linearisation of x^2 with $x \in [-1, 2]$

$L3$ overestimates x^2 with the line that join $(\underline{x}, \underline{x}^2)$ to $(\overline{x}, \overline{x}^2)$. For example, figure 1 shows the curve of x^2 with $x \in [-1, 2]$ and the corresponding linearisations:

$$\begin{cases} L1 : y + 1 + 2x \geq 0 \\ L2 : y + 4 - 4x \geq 0 \\ L3 : x - y + 2 \geq 0 \\ L4 : y \geq 0 \end{cases}$$

3.4 Linearisation of x/y

To linearise the division, we can take advantage of the properties of real arithmetic. The essential observation is that $z = x/y$ is equivalent to $x = zy$ provided $y \neq 0$.

To generate the linearisation of $z \times y$, we need the bounds of z. These bounds can directly be computed by interval arithmetic:

$$[\underline{z}, \overline{z}] = [\nabla(min(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y})), \triangle(max(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}))]$$

with $\overline{y} \neq 0$ and $\underline{y} \neq 0$. ∇ and $\triangle$ are respectively the rounding modes towards $-\infty$ and $+\infty$.

So the linearisation of x/y yields the following set of constraints:

$$\begin{cases} L1 : x - \underline{z}y - \underline{y}z + \underline{z}\underline{y} \geq 0 \\ L2 : -x + \underline{z}y + \overline{y}z - \underline{z}\overline{y} \geq 0 \\ L3 : -x + \overline{z}y + \underline{y}z - \overline{z}\underline{y} \geq 0 \\ L4 : x - \overline{z}y - \overline{y}z + \overline{z}\overline{y} \geq 0 \end{cases}$$

4 Filtering algorithm

The filtering process relies on the linearisations of the relaxations over the reals of the initial problem to attempt to shrink the domain of the variables by means of a linear solver. Algorithm 1 details the steps of the filtering process.

Algorithm 1 FPLP

Function $FPLP(\mathcal{V}, \mathcal{D}, \mathcal{C}, \epsilon)$
% $\mathcal{V}$: floating-point variables
% $\mathcal{D}$: Domains of the variables
% $\mathcal{C}$: Constraints over floating-point numbers
% ϵ: Minimal reduction between two iterations
$\mathcal{C}' \leftarrow$ **Approximate** $(\mathcal{C})$;
$\mathcal{C}'' \leftarrow$ **Linearise** $(\mathcal{C}', \mathcal{D})$;
$reduction \leftarrow 0$;
repeat
  $\mathcal{D}' \leftarrow$ **FP2B**$(\mathcal{V}, \mathcal{D}, \mathcal{C}, \epsilon)$; % 2B consistency over the floats
  $\mathcal{C}'' \leftarrow$ **UpdateLinearisations**$(\mathcal{C}'', \mathcal{D}')$; % Update $\mathcal{C}''$ according to $\mathcal{D}'$
  $oldReduction \leftarrow reduction$;
  for all $x \in \mathcal{V}$ **do**
    $[\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] \leftarrow [$**safeMin**$(x, \mathcal{C}''), -$**safeMin**$(-x, \mathcal{C}'')]$;
  end for
  $reduction \leftarrow \sum_{x \in \mathcal{V}} ((\overline{x}_{\mathcal{D}} - \underline{x}_{\mathcal{D}}) - (\overline{x}_{\mathcal{D}'} - \underline{x}_{\mathcal{D}'}))$;
  $\mathcal{D} \leftarrow \mathcal{D}'$
until $reduction \leq \epsilon \times oldReduction$;
return $\mathcal{D}$;

First step transforms floating-point constraints to constraints over the reals using relaxations detailed in section 2. The second step consists in the linearisations of the nonlinear terms using the linearisations of section 3.

A call to FP2B, a filtering process relying on an adaptation of 2B-consistency to floating-point constraints, does a first attempt to reduce the bounds of the variables. The cost of this filtering process is quite light (w is set to 10%). Note that the first call will allow to propagate bound values to intermediate variables.

After that, we call a mixed integer linear programming (MILP) to compute the bounds of the variables. A first call to the MILP is used to get the lower bound of the domain and a second call is required to get the upper bound.

This process is repeated until the percentage of reduction of the domains of the variables is lower than the given ϵ.

Using an efficient linear solver like CPLEX to filter the domains of the variables raises two important issues related to floating-point computations.
First, linearisation coefficients are computed with floating-point arithmetic and are subject to rounding errors. Therefore, to avoid the loss of solutions, special attention must be paid to the rounding directions. Correct linearisations rely on floating-point computations done using the right rounding directions. For instance, consider the linearisation of x^2 where $\underline{x} \geq 0$ and $\overline{x} \geq 0$:

$$\begin{cases} L1 : y + \triangle(\underline{x}^2) - \triangle(2\underline{x})x \geq 0 \\ L2 : y + \triangle(\overline{x}^2) - \triangle(2\overline{x})x \geq 0 \\ L3 : \triangle(\underline{x} + \overline{x})x - y - \nabla(\underline{x}\overline{x}) \geq 0 \\ L4 : y \geq 0 \end{cases}$$

Algorithm 2 Absorption

```
1: Function Absorption
2: float x ← [0.0, 1e10];
3: float y ← [100.0, 1e5];
4: if x + y == x then
5:    print ("x >> y");
6: else
7:    if x + y = y then
8:       print ("y >> x");
9:    else
10:      print ("No significant absorption");
11:   end if
12: end if
```

where ∇ and $\triangle$ are respectively the rounding modes towards $-\infty$ and $+\infty$. This process ensures that all the linearisations are safe. For more details on how to compute safe coefficients see [16, 4].
Second, efficient linear solvers do use floating-point arithmetic. Thus, the computed minimizer might be wrong. The unsafe linear solver is made safe thanks to the correction procedure introduced in [20]. It consists in computing a safe lower bound of the global minimizer.

5 Experiments

This section compares the results of different filtering techniques for floating-point constraints with the method introduces in this paper. Note that experiments have been done on a laptop with an Intel Duo Core at 2.8Ghz and 4Gb of memory running under Linux.

Our experiments are based on the following set of benches:

- `Absorb 1` is a path of the program 2 which detects if there is an absorption in a simple addition. `Absorb 1` corresponds to the case where x absorbs y.
- `Absorb 2` corresponds to the case where y absorbs x in program 2.
- `Fluctuat 1` is a path of program `Fluctuat` which comes from a presentation of Fluctuat tool in [11].
- `Fluctuat 2` is another path of program `Fluctuat`.
- `Gotlieb` is a program from [5] which shows that some paths are not executable with real arithmetic while they are executable with floating-point arithmetic.
- `Cosine` is a program that computes the function $cos()$ with a Taylor formula.
- `Sqrt` (see program 3) computes the square root of a real number in $[0.5, 2.5]$.
- `MeanValue` returns true if an interval contains a mean value and false otherwise.

Table 3 summarizes experiment results for the following filtering methods:

Algorithm 3 Sqrt

Function float $Sqrt(x \leftarrow [0.5, 1.5])$
$an1 \leftarrow x$;
$cn1 \leftarrow x - 1$;
while $an1 - an > 0.01$ **do**
  $an \leftarrow an1$;
  $cn \leftarrow cn1$;
  $an1 \leftarrow an - an * cn/2$;
  $cn1 \leftarrow cn * cn * cn/4 - cn * cn * 3/4$;
  $i = i + 1$;
end while
return $an1$;

	FP2B		FP3B		FPLP without 2B			FPLP	
Program	size	time	size	time	size	time	iter	time	iter
`Absorb 1`	99900	TO	99900	TO	1092.093	3	1	3	1
`Absorb 2`	99900	1	97.55	18	0.011923	3	1	3	1
`Fluctuat 1`	100	1	1.0029	35	1.00000047	395	11	187	7
`Fluctuat 2`	1	1	1	16	1	145	3	26	2
`Gotlieb`	7.27E-12	1	7.27E-12	1	11.92	3	1	3	1
`Cosine`	1.50	1	1.00019	118	1.000039	243	4	61	2
`Sqrt`	2.24	1	0.527	3284	0.524	2105	6	655	2
`MeanValue`	8	1	2.65	48	2.64	1658	26	81	3

Table 3. Experiments

- FP2B is an adaptation of 2B consistency to floating-point constraints with $w_{2B} = 0$.
- FP3B is an adaptation of 3B consistency to floating-point constraints with $w_{3B} = 0.05$ and $w_{2B} = 0$.
- FPLP without 2B implements algorithm 1 without the call to FP2B.
- FPLP implements algorithm 1

For each filtering method, table 3 gives the size of the domains, as well as, the amount of milliseconds required to filter the constraints. Note that the size of the domains for FPLP is the same than for FPLP without 2B. For the two last filtering methods, table 3 also gives the number of iterations.

The results from table 3 show that FPLP can provides better domain reductions than the other filtering methods. This is exemplified by the two `Absorb` benches. Here, FP2B can not filter the constraints for the first case, and the size given in the second case is much bigger than the result given by FPLP. This is due to the multiple occurrence of the variables.

FPLP competes well with a 3B-consistency: it almost always provides smaller domains in less time.

A comparison of FPLP with and without a call to FP2B shows the benefit of the cooperation of these two filtering methods: it can significantly decreases the time needed to filter as well as the number of calls to the LP. Note that this improvement does not change its capability to reduce the domains of the variables.

6 Conclusion

In this paper, we have introduced a new filtering algorithm for constraints over the floating-point numbers. This algorithm take advantage of a linearisation of a relaxation of the problem over the reals to reduce the domain of the variables by mean of LP solver. Experiments shows that FPLP can significantly improve the filtering process, especially, when combined with an FP2B filtering process. However, more experiments are required to better understand the interactions between the two algorithms.

Acknowledgements

We like to thank Yahia Lebbah for its fruitful discussion.

References

1. F.A. Al-Khayyal and J.E. Falk. Jointly constrained biconvex programming. *Mathematics of Operations Research*, pages 8:2:273–286, 1983.
2. Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11:69–83, January 2009.
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.
4. Glencora Borradaile and Pascal Van Hentenryck. Safe and tight linear estimators for global optimization. *Mathematical Programming*, 2005.
5. Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.
6. Edmund Clarke and Daniel Kroening. Ansi-c bounded model checker user manual. Technical report, School of Computer Science, Carnegie Mellon University, August 2006.
7. Hélène Collavizza, Michel Rueher, and Pascal Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.
8. Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, May 2011.

9. Patrick COUSOT, Radhia COUSOT, Jerome FERET, Antoine MINE, Laurent MAUBORGNE, David MONNIAUX, and Xavier RIVAL. Varieties of static analyzers: A comparison with astree. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 3–20, Washington, DC, USA, 2007. IEEE Computer Society.
10. Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 794–801, New York, NY, USA, 2006. ACM.
11. K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *Computer Aided Verification*, pages 212–226. Springer, 2010.
12. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.
13. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A clp framework for computing structural test data. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2000.
14. Yahia Lebbah, Claude Michel, Michel Rueher, David Daney, and Jean-Pierre Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM J. Numer. Anal*, 42:2076–2097, 2005.
15. G.P. McCormick. Computability of global solutions to factorable nonconvex programs – part i – convex underestimating problems. *Mathematical Programming*, 10:147–175, 1976.
16. C. Michel, Y. Lebbah, and M. Rueher. Safe embedding of the simplex algorithm in a CSP framework. In *Proc. of 5th Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems CPAIOR 2003, CRT, Université de Montréal*, pages 210–220, 2003.
17. Claude Michel. Exact projection functions for floating point number constraints (pdf). In *AMAI*, 2002.
18. Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2001.
19. Atoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.
20. A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer programming. *Math. Programming A.*, page 99:283 296, 2004.
21. IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 2008.
22. Olivier Ponsini, Claude Michel, and Michel Rueher. Refining abstract interpretation-based approximations with a floating-point constraint solver. In *NSV-2011: Fourth International Workshop on Numerical Software Verification*, Cliff Lodge, Snowbird, Utah, July 2011.
23. Hong S. Ryoo and Nikolaos V. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, pages 107–138, 1996.