Optimal Approximation for Efficient Termination Analysis of Floating-point Loops

Fonenantsoa Maurica, Frédéric Mesnard, Étienne Payet
University of Reunion Island, France
{fonenantsoa.maurica,frederic.mesnard,etienne.payet}.univ-reunion.fr

***Abstract*—Floating-point numbers are used in a wide variety of programs, from numerical analysis programs to control command programs. However floating-point computations are affected by rounding errors that render them hard to be verified efficiently. We address in this paper termination proving of an important class of programs that manipulate floating-point numbers: the simple floating-point loops. Our main contribution is an *optimal* approximation to the rationals that allows us to efficiently analyze their termination.**

***Index Terms*—Software correctness, Floating-point numbers, Termination analysis, Linear ranking function, Linear approximation.**

I. Introduction

Termination analysis is concerned with determining whether a given program will always stop or could execute forever. The property of termination is not less important than, say, properties concerning the absence of run-time errors. Examples close to the daily life come from the Microsoft products which bugged a few times due to non-expected infinite loops [1] [2] [3]. In this paper, we study termination of programs that use floating-point numbers. We get rid of the difficulties introduced by the rounding errors by linearly approximating the rounding function. Our proposed approximation is optimal in the sense that it cannot be refined anymore.

The rest of the paper is organized as follows. Section II introduces the basics. Section III precises our programs of interest and presents a sufficient condition for their termination. Section IV develops our main contribution which is the optimality result on the approximation we use. Section V briefly surveys the related work. Section VI concludes. Throughout these sections, we pay a constant attention to efficiency.

II. Preliminaries

Consider the two Java programs *pDec* and *pSqrt* presented in Figure 1. Do they always terminate for any possible value supplied by the user through the `input` function? First let us suppose that we do not use floats but rationals or reals. In that case, both programs always terminate. Indeed the variable x of *pDec* cannot infinitely be decreased by $\frac{1}{10}$ while remaining strictly positive. Similarly the difference $x_M - x_m$ in *pSqrt* cannot infinitely be divided by 2 while remaining strictly greater than some strictly positive quantity d. Following the same idea, termination provers attempt to automatically discover functions that are strictly decreasing at each iteration on some set that does not allow infinite descents. These functions are called *ranking functions*: termination of a program is equivalent to the existence of ranking functions for it. In this paper, we focus on a specific class of ranking functions called *Linear Ranking Functions* (LRFs). For the sake of simplicity, we will say that a LRF is just a ranking function of linear form. For example $f(x) = x$ is a LRF for the rational version of *pDec* since it is linear and is such that $f(x) > 0$ and $f(x) \geq f(x') + \frac{1}{10}$. We use the primed notation x' to denote the next value of the variable x after an iteration.

```java
float x = input();
while (x > 0) {
  x = x - 0.1;
}
```

(a) Simply decreasing x

```java
float xm = 1, xM = 2;
float d = input(); // d > 0
do {
  float x = (xm + xM) / 2;
  float hx = x*x - 2;
  if (hx < 0)  xm = x;
  else xM = x;
} while (xM - xm > d);
```

(b) Computing an interval $[x_m, x_M]$ of length d approximating $\sqrt{2}$ using the dichotomy method.

Fig. 1: Two Java programs: *pDec* (left) and *pSqrt* (right). If the variables are rationals or reals then both programs always terminate. However they may not terminate when using floats due to the rounding errors.

Now let us use floats in *pDec* and *pSqrt*: both programs do *not* always terminate for any possible input. Indeed *pDec* terminates if the supplied x is for example 10 but does not if it is 10^7. Similarly *pSqrt* terminates if the supplied d is for example 10^{-3} but does not if it is 10^{-9}. To explain these surprising changes of behaviors, let us give some basic notions on floating-point numbers.

A floating-point number is a rounded representation of a real number. Simply put, we approximate a real number $x = (-1)^s m\beta^e \in \mathbb{R}$ where $s \in \{0, 1\}, m \in \mathbb{R}^+, \beta \in \mathbb{N}, \beta \geq 2, e \in \mathbb{Z}, e \in [e_{min}, e_{max}]$ as in scientific notation by the floating-point number $f = o(x) = (-1)^s \hat{m}\beta^e \in \mathbb{F}$ where $\hat{m}$ is an approximation of m on p digits, $p \geq 2$. The quantities β, p, e_{min} and e_{max} parameterize the considered floating-point type. The function o is called the *rounding function*. Let us suppose in the rest of the paper that o rounds x to the floating-point number that is *nearest* to it. Consider for example the toy floating-point type `myfloat` presented in Figure 2. The

[1]https://azure.microsoft.com/fr-fr/blog/update-on-azure-storage-service-interruption/

[2]http://www.zdnet.com/article/why-the-blue-screen-of-death-no-longer-plagues-windows-users/

[3]https://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/

real number $x = 10\pi = 31.4\cdots = (-1)^0 3.14\cdots 10^1$ is approximated in `myfloat` by $f = o(x) = (-1)^0 3.1 \cdot 10^1 = 31$.

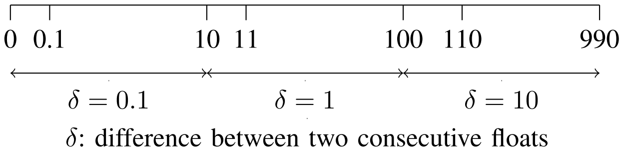


Fig. 2: A personalized floating-point type `myfloat` with $\beta = 10, p = 2, e_{min} = 0$ and $e_{max} = 2$. Symmetry to the origin for the negatives. Call $\mathbb{F}_{myfloat}$ the corresponding set of floats.

Now suppose we use `myfloat` as type of the variables in *pDec* and *pSqrt*. If we supply 20 as value of x then *pDec* does not terminate as $20 - 0.1 = 19.9$ is rounded to 20 itself. Also if we supply 10^{-3} as value of d then *pSqrt* does not terminate as the tightest interval approximating $\sqrt{2}$ that we can obtain with `myfloat` is $[1.4, 1.5]$ which is of length 10^{-1}. Similar phenomena occur when the Java type `float` is used.

Floating-point numbers are standardized by the IEEE-754 norm. Among other things that norm requires the floating-point arithmetic operations to be *correctly rounded*. That is their result must be computed as if in the reals before being rounded. Thus given a real arithmetic operation $\star$, its floating-point equivalent $\circledast$ and 2 floating-point numbers f_1, f_2, the following holds: $f_1 \circledast f_2 = o(f_1 \star f_2)$. For example in `myfloat`, we have $1 \oslash 3 = o(1/3) = o(0.3\dots) = 0.3$.

Last for terminology, if a floating-point number f is such that $|f| \geq \beta^{e_{min}}$ then f is called a *normal number*, otherwise it is called a *subnormal number*. For example in `myfloat`, if $|f| \geq 1$ then f is normal, otherwise it is subnormal.

III. A FLOATING-POINT VERSION OF PODELSKI-RYBALCHENKO

In this paper, we restrict our study to a specific class of programs called *simple loops*. Their study is of great interest as many modern termination analysis techniques consist in splitting the considered program into multiple simple loops that are analyzed separately [1]. Until now literature mainly studied the rational case. A *Simple Rational Loop* (SRL) is a loop defined by a single `while` instruction that contains no nested loop nor branching. Its guard condition and its update relation are conjunctions of linear inequalities as shown in Figure 3.

```
Rational x1 = input(),
         x2 = 0;
while (x1 >= 2) {
  x1 = 0.5 * x1;

  x2 = x2 + 1;
}
```

Set of inequalities describing the loop:

$$\begin{cases} x_1 & & & \geq 2 \\ 0.5 \cdot x_1 & - & x_1' & \geq 0 \\ -0.5 \cdot x_1 & + & x_1' & \geq 0 \\ x_2 & - & x_2' & \geq -1 \\ -x_2 & + & x_2' & \geq 1 \end{cases}$$

Fig. 3: A program computing and storing in x_2 the integer base-2 logarithm of x_1. Call *pIlog* the loop.

Many techniques have been developed for the termination analysis of these SRLs. A notable mention is the Podelski-Rybalchenko (PR) algorithm which *completely* detects LRFs for SRLs in *polynomial time*. Whenever LRFs exist for a given SRL then PR *does* find them. On top of that, it can *synthesize* the detected LRFs. For example if applied to the SRL *pIlog* of Figure 3, PR answers in polynomial time that LRFs exist. Also PR says that a possible LRF f is for example $f(x_1, x_2) = x_1$ which is such that $f(x_1, x_2) \geq 2$ and $f(x_1, x_2) \geq f(x_1', x_2') + 1$. We point out that existence of LRFs implies termination. However the reverse is not always true. Thus if the answer returned by PR is "NO", it only means that no LRF exists for the considered program: we cannot conclude anything regarding its termination. Due to all the reasons mentioned previously, PR is central to modern termination analysis techniques.

Now let us consider the floating-point case. A *Simple Floating-point Loop* (SFL) is like a SRL with the difference that it uses floating-point variables and floating-point operations instead of rational ones. Question arises: is it sound to use PR for analyzing SFL? Obviously no, it isn't. Indeed we have seen in Section II that termination behaviors completely differ depending on the types of the variables. Thus termination of *pIlog* if the variables are rationals is absolutely no guarantee of its termination if the variables are floats.

We point out that our interest in PR is its efficiency: its time complexity is polynomial. Existence of LRFs for SFLs *is* decidable since termination of finite states programs, which is a more general problem, *is* decidable [2, Theorem 1]. Unfortunately that problem lays in *coNP* [2, Theorem 3]. It means that if we suppose $P \neq NP$ then any algorithm that decides existence of LRFs for SFLs is doomed to be exponential in time. In this paper, we are looking for efficiency: *we want to remain polynomial*. Due to the *coNP* limitation, our only solution is to sacrifice completeness.

Following that direction, [2] proposed an adaptation of PR that detects and synthesizes in polynomial time LRFs for SFLs but that is incomplete: sometimes the algorithm answers "I DON'T KNOW". Basically the idea consists in over-approximating the considered SFL by a SRL. That is done by means of what we call *1-piece linear approximation*. A 1-piece linear approximation of the rounding function o on an interval $I = [x_{min}, x_{max}]$ where $x_{min}, x_{max} \in \mathbb{F}$ is a pair of functions μ, ν such that $\forall x \in I : \nu(x) \leq o(x) \leq \mu(x), \mu(x) = ax + b, \mu(x) = cx + d$ where $a, b, c, d \in \mathbb{Q}$. We precise that in this paper, we only consider the case where the bounds x_{min} and x_{max} are floats. Indeed we only use floating-point variables in our programs of interest. Consider for example the rounding function corresponding to `myfloat`. On the interval $I = [100, 200]$, the distance between two consecutive floats is 10 as shown in Figure 2. Thus $\forall x \in I : x - 10 \leq o(x) \leq x + 10$ is a correct 1-piece linear approximation of o. From there, approximating the result of any floating-point arithmetic operation is straightforward by the property of correct rounding. Continuing the example, if $x = 0.5 \odot x_1$

then $\forall x \in I : 0.5 \cdot x_1 - 10 \leq o(x) \leq 0.5 \cdot x_1 + 10$.

That way we approximate to the rationals each operation in the SFL and we obtain a SRL on which we can apply PR. If PR answers "YES" then LRFs exist, otherwise we cannot conclude anything. In the latter case the approximation may be too loose for PR to detect LRFs. Thus we can attempt to refine it. In the previous example, on the interval $I = [100, 200]$ the distance between two consecutive floats is 10 *and* we round to the nearest float. Thus $\forall x \in I : x - \frac{10}{2} \leq o(x) \leq x + \frac{10}{2}$ is a better 1-piece linear approximation of o as it is more precise. Question arises: what is the best, the *optimal* 1-piece linear approximation of o on a given interval?

IV. OPTIMAL 1-PIECE LINEAR APPROXIMATION

In the rest of the paper, we only consider the upper approximation function μ since finding the optimal lower approximation function ν is similar to finding the optimal μ. Now we want to place μ above the rounding function o and as close as possible to it. By characterizing that closeness by the surface between the two functions, we formally define the problem as follows.

Definition 1(*OptMu*). *OptMu* is the problem of finding the affine segment $\mu(x), x \in I, I = [x_{min}, x_{max}]$ where $x_{min}, x_{max} \in \mathbb{F}$ that solves the following optimization problem:

$$\begin{cases} minimize(S) \\ S = \int_I \big(\mu(x) - o(x)\big)dx \\ o(x) \leq \mu(x) \\ \mu(x) = ax + b \\ a \in \mathbb{Q}, b \in \mathbb{Q}, x \in \mathbb{R} \end{cases} \tag{IV.1}$$

The function o rounds the real number x to the nearest element of the considered floating-point type:

$$o(x) = [x]_{ulp(x)} \tag{IV.2}$$

$$\text{where} \quad ulp(x) = \epsilon \cdot \beta^{exp(x)} \tag{IV.3}$$

$$\text{and} \quad exp(x) = \begin{cases} \lfloor log_\beta(|x|) \rfloor \text{ if } |x| \geq \beta^{e_{min}} \\ e_{min} \text{ otherwise} \end{cases} \tag{IV.4}$$

$$\text{and} \quad \epsilon = \beta^{-p+1} \tag{IV.5}$$

$$\text{and} \quad \beta \in \mathbb{N}, \beta \geq 2, p \in \mathbb{N}, p \geq 2, e_{min} \in \mathbb{Z}$$

The notation $[a]_b$ denotes the multiple of b nearest to a while the notation $\lfloor a \rfloor$ denotes the greatest integer smaller or equal to a. ■

Interested readers can find out more about these different functions and quantities involved in the definition of o in [3] and [4, Definition 3].

A. A first solution to OptMu

As expressed in Definition 1, the problem is daunting. The natural question that arises is: can we even solve it? To answer that question, notice first that the rounding function o is actually a piecewise constant function. It is graphically represented by a set of constant segments as shown in Figure 4 and 6. For example, the rounding function corresponding to the simple floating-point type `myfloat` we presented in Figure 2 is defined as follows:

$$\begin{aligned} o \ &: \ \mathbb{R} \ \rightarrow \ \mathbb{F}_{myfloat} \\ x \ &\mapsto \ o(x) = \begin{cases} 990 & 985 < x < 995 \\ ... & \\ 110 & 105 < x < 115 \\ 100 & 99.5 \leq x \leq 105 \\ 99 & 98.5 < x < 99.5 \\ 98 & 97.5 \leq x \leq 98.5 \\ 97 & 96.5 < x < 97.5 \\ ... & \\ -990 & -995 < x < -985 \\ \infty & \text{otherwise (we suppose} \\ & \text{this case never occurs)} \end{cases} \end{aligned}$$

Then notice that placing a segment above a set of segments can be simplified into placing it above the two endpoints, left and right, of each of them. Even better, for the particular case of the set of constant segments defining the rounding function o, we just have to consider the *left endpoints*. Indeed, the right endpoint of a constant segment is always below the left endpoint of the next constant segment as shown in Figure 4. That allows us to transform the constraints $o(x) \leq \mu(x), \mu(x) = ax + b, x \in I$ of Equation IV.1 into a conjunction of linear inequalities. Continuing our example, $o(x) \leq \mu(x), \mu(x) = ax + b, x \in [97, 110]$ is transformed as follows:

$$\begin{cases} 110 & \leq \mu(x) \text{ at } x = 105 \\ 100 & \leq \mu(x) \text{ at } x = 99.5 \\ 99 & \leq \mu(x) \text{ at } x = 98.5 \\ 98 & \leq \mu(x) \text{ at } x = 97.5 \\ \mu(x) & = ax + b \end{cases} \iff \begin{cases} 110 & \leq 105a + b \\ 100 & \leq 99.5a + b \\ 99 & \leq 98.5a + b \\ 98 & \leq 97.5a + b \end{cases}$$

Last notice that the objective function S to minimize is also a linear expression of a and b: $S = \frac{1}{2}(x_{max}^2 - x_{min}^2)a + (x_{max} - x_{min})b$. Thus we managed to completely transform the *OptMu* problem into a linear programming problem. Continuing our example, *OptMu* is reduced to the following problem:

$$\begin{cases} minimize(S) \\ S = 1345.5a + 13b \\ 110 \leq 105a + b \\ 100 \leq 99.5a + b \\ 99 \leq 98.5a + b \\ 98 \leq 97.5a + b \\ a, b \in \mathbb{Q} \end{cases} \tag{IV.6}$$

which we can solve by using for example the Simplex algorithm: $a = \frac{8}{5}$ and $b = -58$, that is the optimal μ is $\mu(x) = \frac{8}{5}x - 58$.

Theorem 1. The *OptMu* problem can be reduced to a linear programming problem that is solvable in polynomial time. ■

Now that we found a way to solve *OptMu*, question arises: is that solution efficient? No, it isn't. Indeed, though the obtained linear programming problem can be solved in polynomial time [5], its size can become astronomical. In our illustrative

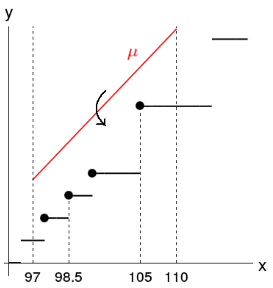


Fig. 4: The *OptMu* problem for the floating-point type `myfloat` and the interval $I = [97, 110]$: we want to lower the affine segment $\mu(x)$ as much as possible while remaining above the four left endpoints.

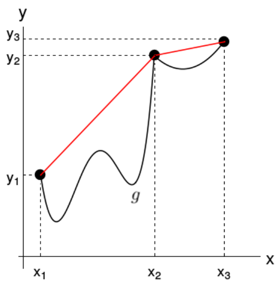


Fig. 5: The endpoints lemma. Placing a segment above the function $g(x)$ on the interval $[x_1, x_3]$ is equivalent to simply placing it above the three points $P_i = [x_i, y_i], i \in \{1, 2, 3\}$.

example, for the floating-point type `myfloat` and the interval $I = [97, 110]$, we obtained a linear programming problem having an objective function subject to four constraints. If we had $I = [97, \mathit{130}]$ instead, we would have to consider *six* constraints as there are six left endpoints in that interval as can be viewed in Figure 6c. Actually we need to consider as many constraints as the number of the floats in I. Thus for the IEEE-754 type `double` which is encoded on 64 bits, we may have to consider up to approximately 2^{64} constraints. Clearly that naive transformation into a linear programming problem is not useful in practice.

B. A second solution to OptMu

We now present an algorithm that solves *OptMu* very efficiently, in constant time regarding the considered floating-point type and the interval I. Our solution relies on the following intermediate result.

Lemma 1(Endpoints lemma). Let g be a real function of $x \in \mathbb{R}$ defined on the interval $I = [x_{min}, x_{max}]$. Let μ be a 1-piece linear upper approximation of g on I: $g(x) \leq \mu(x), x \in I$. If $\mu(x_{min}) = g(x_{min})$ and $\mu(x_{max}) = g(x_{max})$ then μ is optimal: $\int_I \big(\mu(x) - g(x)\big)dx$ is minimal. ■

Simply put, the endpoints lemma just says that if a 1-piece linear upper approximation function μ *touches* the function g it approximates on two points x_1 and x_2, $x_1 < x_2$, then μ is optimal on the interval $[x_1, x_2]$. Indeed in that case, we can show that if μ is placed lower then it will be under g at least at one point. If it is placed higher then the surface between μ and g will increase, rendering μ to be not optimal.

The use of that lemma is as shown in Figure 5. If we can find a set of such "touching points" for the considered function g then we just need to place μ above these points. Using that reasoning we can abstract the rounding function to-nearest o to a set of four points at most, as shown in the following result.

Theorem 2. The following algorithm solves *OptMu* in constant time regarding the considered floating-point type $\mathbb{F}$ and the interval I.

INPUT $I = [x_{min}, x_{max}]$ and $x_{min}, x_{max} \in \mathbb{F}$
OUTPUT $\mu(x)$ solving *OptMu*
BEGIN

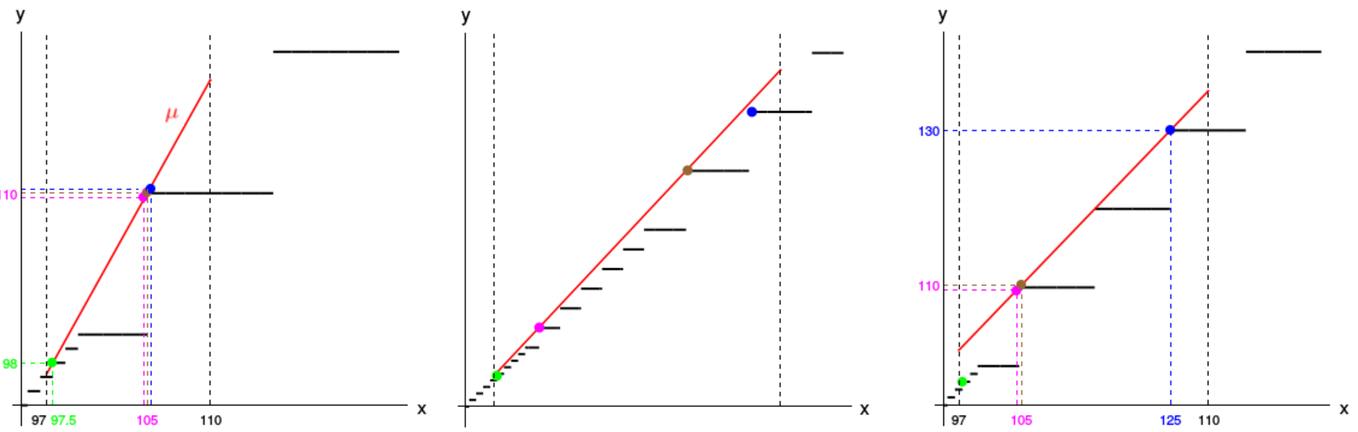
`Step 1`: Determine the four points P_{min}, P_i, P_j and P_{max}
P_{min}: left endpoint [of abscissa] in I closest to the origin
P_{max}: left endpoint in I farthest to the origin
P_i: left endpoint in I closest to the origin and having a greater *ulp* than that of P_{min}
P_j: left endpoint in I closest to the origin and of same *ulp* as P_{max}
LET $p_{min}, p_i, p_j, p_{max}$: abscissa of $P_{min}, P_i, P_j, P_{max}$
IF $p_{min}p_i \leq 0$ THEN $P_i \leftarrow P_j$

`Step 2`: Choose the optimal μ
LET $M = max\big(|p_{min} - p_i|, |p_i - p_j|, |p_j - p_{max}|\big)$
IF $M = |p_{min} - p_i|$ THEN RETURN $\mu(x)$: $(P_{min}P_i)$
ELSE IF $M = |p_i - p_j|$ THEN RETURN $\mu(x)$: (P_iP_j)
ELSE IF $M = |p_j - p_{max}|$ THEN RETURN $\mu(x)$: (P_jP_{max})
END ■

We define the *ulp* of an endpoint as the *ulp* of its abscissa as obtained with Equation IV.3. However for the sake of simplicity, we will just say that the *ulp* of an endpoint is the length of its corresponding constant segment. For example in Figure 6c, the left endpoint at abscissa 105 and the one at 125 are of same *ulp* as their corresponding constant segments are of same length. We emphasize though that that definition based on the length of the corresponding constant segment is *not* equivalent to the correct definition based on Equation IV.3: it only serves for the intuition.

Now where did these four points and that algorithm come from? First let us give more details about the rounding function o. Notice that the rounding function o is such that the more we go far away from zero, the more the distance between two consecutive floats increases. Graphically it means that the more we go far away from the origin, the more the length of the constant segments increases as shown in Figure 6. However that increasing is done in a peculiar way: by a ratio of β every power of β, roughly. As illustrated for example in Figure 6d, when going from the origin to negative infinity, the length of the constant segments remains unchanged for a certain amount of time, then increases after reaching some left endpoint, then remains unchanged again, then increases again after reaching some left endpoint and so forth. Let us call *ulp-increasing endpoints* these left endpoints where the length of the constant segments increases. For example in Figure 6c, the left endpoint at abscissa 105 is an *ulp*-increasing endpoint.

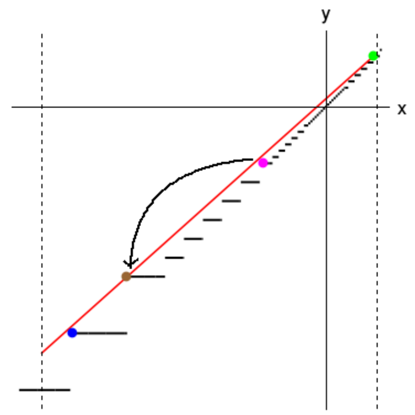
Now we can prove that given two left endpoints P_1 and P_2, the segment $\overline{P_1P_2}$ remains above o for any of the three following cases: (a) P_1 and P_2 are of same *ulp*, (b) P_1 and P_2 are both *ulp*-increasing endpoints, (c) P_2 is the first *ulp*-increasing endpoint after P_1 when leaving the origin. The left endpoints P_{min} and P_i satisfy case (c). The left endpoints P_i and P_j satisfy case (b). The left endpoints P_j and P_{max} satisfy case (a). Thus by the endpoints lemma, the segments $\overline{P_{min}P_i}$, $\overline{P_iP_j}$ and $\overline{P_jP_{max}}$ are optimally placed on their respective intervals. The fact that P_i is merged with P_j if $p_{min}p_i \leq 0$ is because we can show that in that case the segment $\overline{P_{min}P_j}$ remains above o, thus "short-circuiting" $\overline{P_{min}P_i}$ as illustrated on Figure 6d. At this point, we just have to place μ above



(a) $I = [97, 110]$

(b) $I = [5, 120]$, $P_{min} = (5.05, 5.1)$, $P_i = (10.5, 11)$, $P_j = (105, 110)$, $P_{max} = (115, 120)$. Approximative figure.

(c) $I = [97, 130]$, $P_{min} = (97.5, 98)$



(d) $I = [-130, 15]$, $P_{min} = (14.5, 15)$, $P_i = P_j = (-105, -100)$, $P_{max} = (-125, -120)$. The segment $\overline{P_{min}Pi}$ is "short-circuited" by the segment $\overline{P_{min}Pj}$. Approximative figure.

Fig. 6: Finding the affine segment μ that optimally upper approximates the rounding function o for the `myfloat` type and a given interval I

these three segments: we can show that the solution is one of them, depending on the length of their respective intervals.

To avoid making the text cumbersome, we do not give the algebraic values of the coordinates of P_{min}, P_i, P_j and P_{max}. Instead we will graphically illustrate the determination of these points with a couple of examples. We just point out that the computation of these coordinates is similar to the computation of the floats preceding and following a given float. Intuitively we can express the coordinates of the left (resp. right) endpoint corresponding to a given float with the value of the float preceding (resp. following) it. Thus in the same way we know how to compute these straddling floats very easily, see for example the functions `mpfr_nextbelow` and `mpfr_nextabove` of the MPFR library [6], we know how to determine P_{min}, P_i, P_j and P_{max} very efficiently.

Now we give some illustrative examples. In the following, we consider the floating-point type `myfloat` and we want to find the optimal μ for a given interval I.

Example 1 ($\mathbf{I = [97, 110]}$, see Figure 6a).
`Step 1`: Determining P_{min}, P_i, P_j and P_{max}

- P_{min}: left endpoint in I closest to the origin: $(97.5, 98)$
- P_{max}: left endpoint in I farthest to the origin: $(105, 110)$
- P_i (intuitive but approximative definition): left endpoint in I closest to the origin whose corresponding segment is longer than that of P_{min}: $(105, 110)$
- P_j: left endpoint in I closest to the origin and such that its corresponding segment is of same length as that of P_{max}: $(105, 110)$

In this case, P_i, P_j and P_{max} are all the same point.

`Step 2`: Choosing the optimal μ
We have $M = max\big(|p_{min} - p_i|, |p_i - p_j|, |p_j - p_{max}|\big) = max(7.5, 0, 0) = |p_{min} - p_i|$. Thus the line $(P_{min}P_i)$ optimally approximates o on I: $\mu(x) = \frac{8}{5}x - 58$. That is indeed the solution we obtained after solving the linear programing problem of Equation IV.6.

We point out that the optimal μ we obtain here is neither an approximation by the absolute error nor by the relative error as encountered in the literature [7][8][9]. ■

Example 2 ($\mathbf{I = [5, 120]}$, see Figure 6b).
`Step 1`: Determining P_{min}, P_i, P_j and P_{max}. See figure.

`Step 2`: Choosing the optimal μ
We have $M = max\big(|p_{min} - p_i|, |p_i - p_j|, |p_j - p_{max}|\big) = max(5.45, 94.5, 10) = |p_i - p_j|$. Thus the line (P_iP_j) optimally approximates o on I: $\mu(x) = \frac{22}{21}x$.

We point out that the floats in I are normals and the optimal μ we obtain here is the approximation by relative error as preferred in the literature for approximating the normals. Indeed μ is such that $\mu(x) = \big(1 + \mathcal{R}_n^o\big)x$ where $\mathcal{R}_n^o = \frac{\epsilon}{2+\epsilon} = \frac{1}{21}$ is the optimal bound for the relative error for the normals as shown in [10]. The quantity ϵ is as shown in Equation IV.5. ■

Example 3 ($\mathbf{I = [97, 130]}$, see Figure 6c).
`Step 1`: Determining P_{min}, P_i, P_j and P_{max}. See figure: in this case, P_i and P_j are the same point.

`Step 2`: Choosing the optimal μ
We have $M = max(|p_{min} - p_i|, |p_i - p_j|, |p_j - p_{max}|) = max(7.5, 0, 20) = |p_j - p_{max}|$. Thus the line $(P_j P_{max})$ optimally approximates o on I: $\mu(x) = x + 5$.

We point out that though the floats in I are normals, our algorithm says that the best approximation is *not* the approximation by the relative error as in the case of Example 2 and as preferred in the literature. Indeed μ is such that $\mu(x) = x + \mathcal{A}_I^o$ where $\mathcal{A}_I^o = \frac{ulp(x_{max})}{2} = 5$ is the optimal bound for the absolute error for the normals in I. ■

Example 4 ($\mathbf{I} = [-\mathbf{130}, \mathbf{15}]$, see Figure 6d).
`Step 1`: Determining P_{min}, P_i, P_j and P_{max}. See figure: in this case, we have $p_{min} p_i \leq 0$. Thus P_i is merged with P_j.

`Step 2`: Choosing the optimal μ
We have $M = max(|p_{min} - p_i|, |p_i - p_j|, |p_j - p_{max}|) = max(119.5, 0, 20) = |p_{min} - p_i|$. Thus the line $(P_{min} P_i)$ optimally approximates o on I: $\mu(x) = \frac{230}{239}x + \frac{250}{239}$.

We point out that there are both normals and subnormals in I. This is to show that our algorithm handles seamlessly intervals containing normals only, subnormals only or a mix of both. ■

To end this section let us get back to the analysis of the loop *pIlog* presented in Section III when its variables are of `myfloat` type. First we determine the ranges of the variables: say for example $x_1 \in [2, 150]$ due to some restriction on the `input` function and $x_2 \in [0, 990]$. Then we optimally (upper) approximate them: $x_1' \leq \frac{22}{21}(0.5 \cdot x_1)$ and $x_2' \leq (x_2 + 1) + 5$. Last we apply PR on the obtained SRL: PR says LRFs exist. Thus the loop *pIlog* terminates when its variables are of `myfloat` type.

V. Related work

First we remind that we need to know the ranges of the variables. In the same way termination of floating-point programs is decidable, the ranges of their variables is exactly computable. However that is achieved alongside high time complexity. Interpolation operators from the framework of Abstract Interpretation [11][12] allows us to obtain reasonably precise ranges in a reasonable amount of time.

Then we used 1-piece linear approximations in order to remain polynomial. We can have more refined approximations by increasing the number of pieces, that is by using k-pieces linear approximations, $k \in \mathbb{N}^*$. However we can show that for any $k \geq 2$ the obtained approximation is exponential in the size of the considered program [13, Theorem 10]. Techniques based on these k-pieces linear approximations have been recently developed for analyzing termination of floating-point loops [13]. We point out that the idea of using piecewise linear approximations already appeared in [14] where they are used to approximate floating-point implementations of transcendental functions. Actually they can be used for various applications, for example for solving constraints over floating-point numbers [7].

Last there are adornment-based approaches for termination analysis of floating-point computations [15]. Also there are boolean-based ones for termination analysis of finite state-programs in general, including programs that use floating-point numbers [16][17].

VI. Conclusion

In this paper, we presented an efficient way to infer termination of simple floating-point loops. Our approach relies on an approximation to the rationals that allows us to use the Podelski-Rybalchenko (PR) algorithm for detecting Linear Ranking Functions (LRFs) in polynomial time. The existence of these functions is a sufficient condition for termination. However using approximation introduces incompleteness: sometimes LRFs exist but we cannot detect them. We mitigate that shortcoming by making our approximation optimal: it cannot be refined anymore. In that sense, we now have the optimal adaptation of PR to the floating-point case. It could be used as a central piece to termination analysis of complex floating-point loops in the same way PR is used as a central piece to termination analysis of complex integer ones [18].

References

[1] M. Heizmann, N. D. Jones, and A. Podelski, "Size-change termination and transition invariants," in *Proc. of SAS 2010*. http://dx.doi.org/10.1007/978-3-642-15769-1_4

[2] F. Maurica, F. Mesnard, and É. Payet, "On the linear ranking problem for simple floating-point loops," in *Proc. of SAS 2016)*. http://dx.doi.org/10.1007/978-3-662-53413-7_15

[3] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, 1991. http://doi.acm.org/10.1145/103162.103163

[4] J.-M. Muller, "On the definition of $ulp(x)$," INRIA, Tech. Rep., 2005. https://hal.inria.fr/inria-00070503

[5] L. Khachiyan, "Polynomial algorithms in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, 1980. http://www.sciencedirect.com/science/article/pii/0041555380900610

[6] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, 2007. http://doi.acm.org/10.1145/1236463.1236468

[7] M. S. Belaid, C. Michel, and M. Rueher, "Boosting local consistency algorithms over floating-point numbers," in *Proc. of CP 2012*. Available: http://dx.doi.org/10.1007/978-3-642-33558-7_12

[8] S. Boldo and T. M. T. Nguyen, "Proofs of numerical programs when the compiler optimizes," *Innovations in Systems and Software Engineering*, vol. 7, no. 2, 2011. http://dx.doi.org/10.1007/s11334-011-0151-6

[9] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *Proc. of ESOP 2004*. http://dx.doi.org/10.1007/978-3-540-24725-8_2

[10] C.-P. Jeannerod and S. M. Rump, "On relative errors of floating-point operations: optimal bounds and applications," Tech. Rep., 2016. https://hal.inria.fr/hal-00934443

[11] P. Cousot and R. Cousot, "A gentle introduction to formal verification of computer systems by abstract interpretation," in *Logics and Languages for Reliability and Security*, vol. 25, 2010. http://dx.doi.org/10.3233/978-1-60750-100-8-1

[12] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *Proc. of FMICS 2009*. http://dx.doi.org/10.1007/978-3-642-04570-7_6

[13] F. Maurica, F. Mesnard, and É. Payet, "Termination analysis of floating-point programs using parameterizable rational approximations," in *Proc. of SAC 2016*. http://doi.acm.org/10.1145/2851613.2851834

[14] V. Lefèvre, A. Tisserand, and J. Muller, "Towards correctly rounded transcendentals," in *Proc. of ARITH 1997*. http://dx.doi.org/10.1109/ARITH.1997.614888

[15] A. Serebrenik and D. D. Schreye, "Termination of floating-point computations," *Journal of Automated Reasoning*, vol. 34, no. 2, 2005. http://dx.doi.org/10.1007/s10817-005-6546-z

[16] C. David, D. Kroening, and M. Lewis, "Unrestricted termination and non-termination arguments for bit-vector programs," in *Proc. of ESOP 2015*. http://dx.doi.org/10.1007/978-3-662-46669-8_8

[17] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger, "Ranking function synthesis for bit-vector relations," *Formal Methods in System Design*, vol. 43, no. 1, 2013. http://dx.doi.org/10.1007/s10703-013-0186-4

[18] M. Codish, V. Lagoon, and P. J. Stuckey, "Testing for termination with monotonicity constraints," in *Proc. of ICLP 2005*. http://dx.doi.org/10.1007/11562931_25