A study of the Behavior of Floating-Point Errors

Nasrine Damouche

ISAE-SUPAERO, Université de Toulouse,
10 Avenue Edouard Belin, Toulouse, France

Abstract. The dangers of programs performing floating-point computations are well known. This is due to numerical reliability issues resulting from rounding errors arising during the computations. In general, these round-off errors are neglected because they are small. However, they can be accumulated and propagated and lead to faulty execution and failures. Typically, in critical embedded systems scenario, these faults may cause dramatic damages (eg. failures of Ariane 5 launch and Patriot Rocket mission). The ufp (unit in the first place) and ulp (unit in the last place) functions are used to estimate maximum value of round-off errors. In this paper, the idea consists in studying the behavior of round-off errors, checking their numerical stability using a set of constraints and ensuring that the computation results of round-off errors do not become larger when solving constraints about the `ufp` and `ulp` values.

Keywords: Static analysis, floating-point arithmetic, round-off errors, ufp and ulp functions.

1 Introduction

In recent years, the perpetual advances in computer science have resulted in the development of sophisticated devices and complex software. However, this has also brought additional challenges to ensure properties of performance and reliability especially in embedded critical systems such as avionic, automotive and robotic. In particular, the accuracy of numerical computations may have an important impact on the validity of programs based floating-point arithmetic. By way of illustration, it seems mandatory to tackle the problems of inaccurate program computation results or numerical stability issues caused by round-off errors.

Floating-point arithmetic, as specified by the IEEE-754 Standard [2], is widely used for numerical simulations and applied to many industrial domains, such as embedded critical systems (avionics, robotics, autonomous systems). Floating-point numbers are used to encode real numbers. In practice, floating-point numbers are approximation of real numbers. Therefore, this approximation generates round-off errors and makes the floating-point arithmetic prone to numerical accuracy problems. Depending on the critical level of the application [25,29,30], the approximation becomes dangerous when accumulated errors cause damages like the failure of the Ariane 5 launch and the Patriot Rocket mission. Asserting the numerical accuracy of floating-point computations is then a well-known problem with many applications in safety critical systems.

Several techniques have been proposed over the last fifteen years to validate [3,11,13,14,28], to improve [27] the numerical accuracy of programs, and to avoid failures. Many methods are based on static analyses of the numerical accuracy of floating-point computations. These methods compute an over-approximation of the worst error that arises during the program execution. Another family of work improves the accuracy of computations by program transformation [10]. While

these methods compute an over approximation of the worst error arising during the executions of a program, they operate on final codes, during the verification phase and not at implementation time.

The main contribution of this paper is to study the behavior of the round-off errors relying on floating-point computations. Let us precise that detecting accuracy errors in critical embedded systems is a very difficult task since a small error may lead in catastrophic results. Recall that, the existing embedded critical systems suffer from divergence caused by their values which cannot be bounded statically. For this reason, many useful programs see their initial errors grow with the number of iterations. Our purpose is to ensure that even if the round-off error becomes large, it can not be greater than the output of the program (see Figure 2 corresponding to the curve of the while loop example). A key element in their computations is the use of the `ufp` and `ulp` functions. In addition, a set of constraints will be specified and a proof in terms of the behavior of the round-off errors computations will be highlighted. The idea is to compare the results for different number of iterations going from 0 up to 100 iterations. This is important to observe the behavior of our program through the computation of the errors.

The reminder of the article is organized as follows. Section 2 surveys related work. Section 3 introduces the background material and gives some preliminary definitions. Section 4 consists in the presentation of the problem statement, our motivation to tackle this numerical issue and details through a running example our purpose and aim. This section also show the method followed to deduce the constraints set. In Section 5, a proof of correctness of our approach is presented. Section 6 concludes and outlines future work and the potential application of these results.

2 Related Work

Over the past decade, several approaches and tools relying on formal methods and numerical analysis have been proposed. These approaches can be categorized as follows: i) Formal proofs and proof assistants to guarantee the accuracy of finite-precision computations [4,15,19]. ii) Compile-time optimization of programs in order to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [9,27].

Other papers based on static and/or dynamic analysis propose to determine the best floating-point formats as a function of the expected accuracy on the results. In [11] the authors introduced Rosa tool. This source-to-source compiler computes the propagation of errors by using a forward static analysis coupled to a SMT solver. Rosa takes as input real values and error specification and then compute bounds on the accuracy. If the computed bound satisfies the post-conditions then the analysis is run again with a smaller format until founding the best format. In this approach, all the values have the same format. Other static techniques [14,28] could be used to infer the suitable floating-point formats from the forward error propagation. In [22] the authors introduced a new static analyzer that takes as input a set of functional oating-point expressions and generates the round-o error estimations for each function in the program. Another method has been proposed by [6] to allocate a precision to the terms of an arithmetic expression (only). This method uses a formal analysis via Symbolic Taylor Expansions and error analysis based on interval functions to solve a quadratically constrained quadratic program to obtain annotations. The FPTuner [7] tool is a type system based on constraint generation and relies on local optimization by solving quadratic problems for the set of data types. POP [17,1], is a mixed precision tuning static tool for numerical calculations. Based on forward and backward error analysis techniques, the set of constraints are solved by SMT solver [12] and LP solver [20] to find an optimal solution.

Other approaches rely on dynamic analysis. The Precimonious tool attempts to decrease the precision of variables and checks whether the accuracy requirements are still fulfilled or not [26].

In [18], the authors instrument binary codes in order to modify their precision without modifying the source codes. They also propose a dynamic search method to identify the pieces of code where the precision should be modified.

3 Background

Before presenting the main contribution of this paper, let us present the background material according to the IEEE-754 Standard and recall some definitions of the `ulp` and `ufp` functions needed in our study.

3.1 IEEE-754 Standard

This paper is relying on IEEE-754 floating-point arithmetic Standard [2,23] where a float number x is defined by :

$$x = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

with sign $s \in \{-1, 1\}$, mantissa m, basis β, precision p (length of the mantissa) and exponent $e \in [e_{min}, e_{max}]$. Recall that we only consider the case $\beta = 2$. The IEEE-754 Standard specifies several formats for floating-point numbers by providing specific values for p, β, e_{min} and e_{max}. The standard also defines some rounding modes, towards $+\infty$, $-\infty$, 0 and to the nearest, respectively denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$. Interestingly, in this paper we are considering the rounding mode to the nearest.

The IEEE-754 Standard defines the semantics of the elementary operations $\circledast_r \in \{+, -, \times, \div\}$ by:

$$x \circledast_r y = \uparrow_\sim (x * y) \tag{2}$$

where $\circledast_r$ is computed by using the rounding to the nearest and $* \in \{+, -, \times, \div\}$ denotes an exact operation. Because of the round-off errors, the results of the computations are not exact.

Example 1. For instance, let us take this example of two mathematically equivalent functions f and g defined in Equations (3) and (4) to illustrate that computations performing floating-point arithmetic are error prone. Not that, even if these two functions are equivalents, the results may differ.

$$f(x) = x^2 - 2.0 \times x + 1.0 \tag{3}$$

$$g(x) = (x - 1.0) \times (x - 1.0) \tag{4}$$

The computation of $f(0.999)$ and $g(0.999)$ respectively returns $1.00000000002875566e^{-6}$ and $1.00000000000000186e^{-6}$ as results. Even on small computations, different results are obtained. ■

In addition, the propagation of errors arising during the computation is described by the `ulp` (unit in the last place) and `ufp` (unit in the first place) functions [24,21]. These functions are very useful in the error analysis of floating-point computations. Recall that lots of `ulp` definitions are existing in the literature. The one presented in this paper consists of the `ulp` of floating-point number with p significant digits.

The `ulp` (unit in the place) denotes the value of the last bit of the mantissa of a floating-point number or the corresponding value for a real number [5,16]. In other words, the `ulp` of a floating-point number corresponds to the binary exponent of its least significant digit. Note that

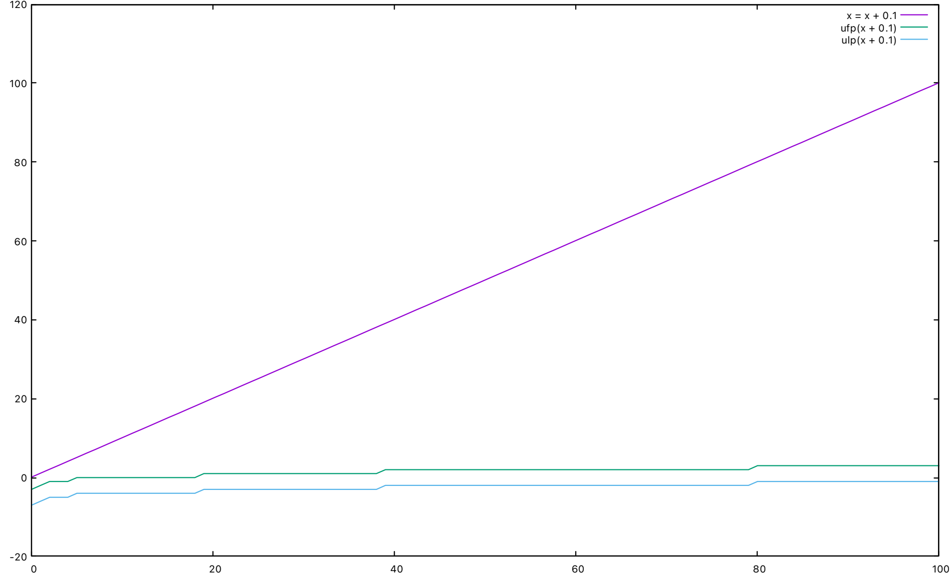


Fig. 1. Representation of ufp, ulp and x values for the control loop example (2).

several definitions of the `ulp` have been given [23]. In [24], the `ulp(x)` function is defined as the distance between two closest representatives floating-point numbers f and f', so that $f < x < f'$ and $f \neq f'$. It is of the form :

$$\texttt{ulp(x)} = \texttt{ulp(s,m,e)} = \texttt{ulp} \times 2^e \tag{5}$$

The `ufp` function computes the unit in the first place. More precisely, the `ufp` of a floating-point number corresponds to the binary exponent of its most significant digit. The `ufp` of a number x is given by Equation (6).

$$\texttt{ufp(x)} = \beta^{\lfloor log_\beta |x| \rfloor} \quad if\ x \neq 0 \quad and\ \texttt{ufp(0)} = 0 \tag{6}$$

Nevertheless, the machine epsilon (named also machine precision) can also be used. The machine epsilon consists in the maximum on the relative approximation error due to rounding in floating-point arithmetic. For example, the representation of a real number x by a machine epsilon x', is given by Equation (7).

$$x' = x + \alpha, with\ \alpha \begin{cases} \alpha \leq \frac{1}{2} ulp(x) & \text{rounding to nearest} \\ \alpha \leq ulp(x) & \text{otherwise} \end{cases} \tag{7}$$

Recall that, the rounding mode used in our case is to nearest. Thus, the associated round-off error α is assumed to be less than or equal to $\frac{1}{2}ulp(x)$.

A program computing the `ufp` and `ulp` values has been implemented using the C language. Figure 1 depicts the comparison between the computed values of the output **x** and its corresponding **ufp** and **ulp** values for the control loop given in Example 2. The implementation clearly shows that even if the values of the **ufp(x)** or the **ulp(x)** grow, they are always under the curve of the computed value for the output x. In other words, even if the associated error grows, it can not exceed the output value of x inside the considered control loop example.

4 Our Approach

Observing the floating-point errors behavior in critical embedded systems consists in a hard task. It permits to manage the computation errors and avoid faulty executions and then catastrophic damages. Also, this behavior observation allows to detect which operation is responsible of such abnormal behavior and then correct it. Nevertheless, the detection of which operation or value inside the program are responsible of the bad behavior of the computations helps programmers to anticipate, correct and improve the behavior of such program.

Our approach consists in observing the behavior of the different program values within the control loop as well as their associated rounding errors. To this aim, the `ufp` and `ulp` functions able to represent the rounding errors are defined. First, a set of constraints have been deduced from the initial program presented in Example 2. Our approach relies on static analysis to deduce the interval values of variables and constraints given by Equation (26).

4.1 Motivating Example

For example, let us consider the program of Example 2 which implements a simple integrator within a control loop. At each iteration n of the while loop, the output x is computed as a function of the former value of x obtained at the previous loop iteration. Then, the value of x is integrated and then summed to 0.1.

Example 2.

$$x = \ 0.0;$$
$$while(x <= 10.0)\{ \quad x = x + 0.1; \quad \}$$

■

First, let us compute the rounding error on the value x within the control loop. The idea is to represent the computed error, denoted by ϵ_x^n, in function of $ulp(x)$. We also assume that $ulp(x)$ is the machine epsilon defined formally in Section 3. More precisely, the error on x within the loop can be written as follows: $\epsilon_x^n = \frac{1}{2}ulp(x)$, because we are using the rounding to the nearest. For simplicity, consider $ulp(x)$ shortened to u be the ulp of x and $ulp(+)$ shortened to ϵ_x be the ulp of the intermediary result. In other words, ϵ_x is the `ulp` of the elementary operation consisting on the result of adding the value 0.1 to x.

Initially, let us unfold the body of the control while loop n times. Thus, the expression of x will be written in the following way:

$$x = \underbrace{0.0 + 0.1 + 0.1 + \cdots + 0.1}_{n\ times} \tag{8}$$

In the following, the associated error ϵ_x at each control point of the Equation (8) is computed. Notice that, for the sake of simplicity, control points have been put only on the arithmetic operations of Equation (9). These arithmetic operations, the additions, are defined as control points of the program and denoted by $+^{(i)}$, for i $\in \{1,..,n\}$ with n $\in \mathbb{N}$.

$$x = \underbrace{0.0 +^{(1)} 0.1 +^{(2)} 0.1 +^{(3)} \cdots +^{(n)} 0.1}_{n\ times} \tag{9}$$

Let ϵ_x^i denote the associated round-off error of the float value x at iteration i, for i $\in \{1,..,n\}$ with n $\in \mathbb{N}$.

For the first iteration $i = 1$; Equation (8) is of the form $x = 0.0 +^{(1)} 0.1$ and its associated round-off error ϵ_x^1 is written as follows:

$$\epsilon_x^1 = ulp(0.0) + ulp(0.1) + ulp(+^{(1)}) \tag{10}$$

In other words, the associated rounding-error is equals to the error associated to each floating-point values (operands) plus the error introduced by the arithmetic operation (the addition). For simplicity, consider `u` be the `ulp(x)` and ϵ_x be the $\texttt{ulp}(+^{(i)})$ and assume that $ulp(0.0)$ is close to $ulp(0.1)$, then $ulp(0.0) \approx ulp(0.1) = u$. Thus, Equation (10) will be written like that :

$$\epsilon_x^1 = u + u + \epsilon_x \tag{11}$$

We simplify Equation (11), we obtain the following formula:

$$\epsilon_x^1 = 2 \times u + \epsilon_x \tag{12}$$

For the second iteration $i = 2$; Equation (8) is of the form $x = 0.0 +^{(1)} 0.1 +^{(2)} 0.1$ and its associated round-off error ϵ_x^2 is formulated as follows:

$$\epsilon_x^2 = \epsilon_x^1 + u + \epsilon_x \tag{13}$$

After replacing ϵ_x^1 obtained in Equation (12) within Equation (13), we obtain :

$$\epsilon_x^2 = 2 \times u + \epsilon_x + u + \epsilon_x \tag{14}$$

After simplification, Equation (14) will be of the form:

$$\epsilon_x^2 = 3 \times u + 2 \times \epsilon_x \tag{15}$$

For the third iteration $i = 3$; we have that $x = 0.0+^{(1)}0.1+^{(2)}0.1+^{(3)}0.1$ and ϵ_x^3 is represented by the following equations:

$$\epsilon_x^3 = \epsilon_x^2 + u + \epsilon_x = 3 \times u + 2 \times \epsilon_x + u + \epsilon_x \tag{16}$$

We replace and simplify the expression of ϵ_x^3, it will be written like this:

$$\epsilon_x^3 = 4 \times u + 3 \times \epsilon_x \tag{17}$$

Let us now generalize to n iterations. The below formula computes the associated round-off error ϵ_x^n as following:

$$\epsilon_x^n = (n + 1) \times u + n \times \epsilon_x \tag{18}$$

By factorizing and simplifying the computations done in Equation (18), the associated error ϵ_x^n is written in the following form:

$$\epsilon_x^n = n \times (u + \epsilon_x) + u \tag{19}$$

Once the associated error ϵ_x^n is computed, let us now tackle the computation of the formal values of variables and constants within the program of the while loop. To this end, consider the previous Example 2, and annotate it with control points (or labels; denoted by ℓ_i, for $\texttt{i}_{\{1,..,n\}} \in \mathbb{N}$) to facilitate the computation of the formal values of x. For a sink of simplicity, we associate control

points just to variables and constants. Thus, we are not considering them for the arithmetic operations (addition).

$$x^{\ell_0} = 0.0;$$
$$while(x^{\ell_1} <= 10.0^{\ell_2})\{ \quad x^{\ell_3} = x + 0.1^{\ell_4}; \quad \}$$

To start, the range of values is given using intervals. From the control loop program, the formal value of the variable `x` at control point ℓ_0, according to Equation (2) and following its initialization is $x^{\ell_0} = [0.0, 0.0]$. At the control point ℓ_1, the formal value of x is given by :

$$x^{\ell_1} = [0.0, 10.0] \tag{20}$$

More precisely, this range is deduced from the initialization of $\mathtt{x}^{\ell_0}$ and the condition of the while loop according to the example 2.

The range of the constant 10.0 at control point ℓ_2 is:

$$x^{\ell_2} = [10.0, 10.0] \tag{21}$$

The range of the constant 0.1 at control point ℓ_4 is:

$$x^{\ell_4} = [0.1, 0.1] \tag{22}$$

Inside the while loop, the formal value of x at control point ℓ_3 is the interval [0.0,10.0]. This range is obtained using the static analysis method by processing as follows:

For the first iteration, the range of x^{ℓ_3} is equal to [0.0,0.1]. This range is obtained by applying a `join` operator, denoted by $\cup$, between $\mathtt{x}^{\ell_0}$ and $\mathtt{x}^{\ell_4}$. In other words, we merged the abstract states for $\mathtt{x}^{\ell_0}$ and $\mathtt{x}^{\ell_4}$. See Equation (23).

$$x^{\ell_3} = [0.0, 0.0] \cup [0.1, 0.1] = [0.0, 0.1] \quad (1^{st}\ iteration) \tag{23}$$

For the next loop iteration, while the condition is statically satisfied, the previous process of joining the former range of x^{ℓ_3} with the value at control point ℓ_4 is applied. We summarize these steps through the following equations:

$$x^{\ell_3} = [0.0, 0.1] \cup [0.1, 0.1] = [0.0, 0.2] \quad (2^{nd}\ iteration)$$
$$x^{\ell_3} = [0.0, 0.2] \cup [0.1, 0.1] = [0.0, 0.3] \quad (3^{rd}\ iteration)$$
$$\ldots$$

In order to reach quickly the fix-point and so terminate the analysis of the while loop quickly, the `widening` is applied, denoted by the symbol ∇, for x^{ℓ_3} using the range values at control points ℓ_1 (see Equation (20)) and ℓ_4.

$$x^{\ell_3} = [0.0, 10.0]\nabla[0.1, 0.1] = [0.0, +\infty] \tag{24}$$

Finally, the intersection of the two ranges of x at control points ℓ_1 and ℓ_3 obtained by widening in Equation (24), gives:

$$x^{\ell_3} = [0.0, 10.0] \cap [0.0, +\infty] = [0.0, 10.0] \tag{25}$$

From the snippet of code given in Example 2, we deduce that the value of x is bounded by the interval $[0.0, 10.0]$ thanks to the static analysis method by abstract interpretation [3,8], that permits to infer safe ranges for the variables and compute errors bounds on them..

At this level, we will go one step further by deducing some constraints from the previous equations presented formerly. Equation (26) shows these constraints.

$$\begin{cases} n > 0, \\ x_0 = 0.0, \\ x \in [0.0, 10.0], \\ x = x_0 + n \times 0.1, \\ \epsilon_x^n = n \times (u + \epsilon_x) + u, \end{cases} \tag{26}$$

- n, with n $\in \mathbb{N}$ is the iterations number of the while loop,
- $x_0 = 0.0$, obtained from program declaration and initialization,
- $x \in [0.0, 10.0]$, obtained by static analysis (see more details of computations in Equations (23) to (25)),
- $x = x_0 + n \times 0.1$, consists in the while loop unfolding,
- ϵ_x, consists in the error computation obtained by Equation (19).

This system of equations is very helpful to go through the main aim of this paper. Recall that the main purpose being to study the behavior of the error ϵ_x^n and the variable x, to prove that at each iteration, the value of the computed error do not exceed the value of variable x.

5 Proof Correctness of the Approach

This section presents a proof correctness of our approach based on the constraints set presented previously by Equation (26) in Section 4.1. The main aim of this proof is to demonstrate that for each iteration n of the control loop, the value of x must be greater than or equal to its associated error computed using the `ulp` and `ufp` functions formulas given by Equation (5). We believe that, even if the computed error is accumulated and propagated, it always stays down the x value within the control loop body and this for each iteration n. In other words, the value of the computed error ϵ_x^n releases always smaller than the value of x.

In section 3, we have that $\alpha \leq \frac{1}{2}ulp(x)$ by definition. For simplicity, the $ulp(x)$ is shortened as u. That gives $\alpha \leq \frac{1}{2}u$. Typically, the error term presented by α formerly, is similar to the error ϵ_x detailed in Section 4.1. Thus, the α formula may be expressed by Equation (27) as following:

$$\epsilon_x \leq \frac{1}{2}u \tag{27}$$

In order to link Equation (27) to the computed error on ϵ_x^1, the term $2u$ is added for each side of the Equation (27). The new equation is :

$$2u + \epsilon_x \leq 2u + \frac{1}{2}u \tag{28}$$

Let us simplify the right side of Equation (27). We obtain :

$$2u + \epsilon_x \leq \frac{5}{2}u \tag{29}$$

From Equation (29), we observe that, the left term of the equation, consists in the definition of the ϵ_x^1. The Equation (29) will be written like follows:

$$\epsilon_x^1 \leq \frac{5}{2}u \tag{30}$$

Now, let us generalize for the ϵ_x^n. From Equation (26), we have that:

$$\epsilon_x^n = n \times (u + \epsilon_x) + u, \forall n \in \mathbb{N} \tag{31}$$

Let us start the first part of our correctness proof. We have that from Equation (27):

$$\epsilon_x \leq \frac{1}{2}u \tag{32}$$

Then, by adding the term u to each side of the Equation (32), we get:

$$u + \epsilon_x \leq u + \frac{1}{2}u \tag{33}$$

After simplification, we obtain:

$$u + \epsilon_x \leq \frac{3}{2}u \tag{34}$$

Now, let us multiply the Equation (34) by n, the Equation (35) will be written like that:

$$n \times (u + \epsilon_x) \leq n \times \frac{3}{2}u \tag{35}$$

Adding the term u to Equation (35), we obtain:

$$n \times (u + \epsilon_x) + u \leq n \times \frac{3}{2}u + u \tag{36}$$

Simplifying the previous Equation (36), we get:

$$\underbrace{n \times (u + \epsilon_x) + u}_{\epsilon_x^n} \leq u(\frac{3}{2}n + 1), \forall n \in \mathbb{N} \tag{37}$$

The left side of Equation (37) consists in the definition of the error ϵ_x^n given in the Equation (31). Therefore, Equation (37) is written as follows:

$$\epsilon_x^n \leq u(\frac{3}{2}n + 1), \forall n \in \mathbb{N}. \tag{38}$$

In the rest of this section, let us prove by induction the correctness of the following property P(n). More precisely, we want to prove that for each $n \in \mathbb{N}$, this property is valid:

$$P(n) : \epsilon_x^n \leq u(\frac{3}{2}n + 1), \forall n \in \mathbb{N}$$

- Initialization : P(1), n = 1.
  The computations of the $ulp(x)$ for the first iteration of the while loop (see Equation (26)) gives that the $ulp(0.1)$ is equal to -9. Recall that, the value of x within the control loop, x = x + 0.1, is equal to 0.1 for the first iteration (when n = 1). Let us replace this value in Equation (38), that gives:

$$\epsilon_x^1 \leq u(\frac{3}{2} \times 1 + 1) \quad \Rightarrow \quad \epsilon_x^1 \leq -\frac{45}{2} \tag{39}$$

$$\Rightarrow \epsilon_x^1 \leq 0.1 \tag{40}$$

  Consequently, from Equation (39), we find that ϵ_x^1 is less or equal to $-\frac{45}{2}$, that means ϵ_x^1 is less than 0.1 (the value of x within the control loop, for the first iteration). To conclude, the property P(1) is satisfied.

– Heredity: P(n) ⇒ P(n+1)
  Let us assume that the property P(n) given by Equation (41) is true and check that the property P(n+1) given by Equation (42) is verified.

$$P(n) : \epsilon_x^n \leq u(\frac{3}{2}n + 1) \tag{41}$$

$$P(n+1) : \epsilon_x^{n+1} \leq u(\frac{3}{2}(n+1) + 1) \tag{42}$$

  To start, let us take the term error ϵ_x^n presented by Equation (26). For n+1, we have :

$$\epsilon_x^{n+1} = (n+1) \times (u + \epsilon_x) + u \tag{43}$$

  We develop the Equation (43), we find:

$$\epsilon_x^{n+1} = n \times u + n \times \epsilon_x + u + \epsilon_x + u \quad \Rightarrow \quad \epsilon_x^{n+1} = \underbrace{n(u + \epsilon_x) + u}_{\epsilon_x^n} + \epsilon_x + u \tag{44}$$

  Then, the ϵ_x^{n+1} is of the form:

$$\epsilon_x^{n+1} = \epsilon_x^n + \epsilon_x + u \tag{45}$$

  We have formerly that

$$\epsilon_x^n \leq u(\frac{3}{2}n + 1) \tag{46}$$

  Now, let us inflate the Equation (46). To begin, recall the equation of ϵ_x term $\epsilon_x \leq \frac{1}{2}u$. After adding the term u to each side of ϵ_x, we have:

$$\epsilon_x + u \leq \frac{1}{2}u + u = \frac{3}{2}u \tag{47}$$

  By summing Equations (46) and (47), we obtain:

$$\underbrace{\epsilon_x^n + \epsilon_x + u}_{\epsilon_x^{n+1}} \leq u(\frac{3}{2}n + 1) + \frac{3}{2}u \quad \Rightarrow \quad \epsilon_x^{n+1} \leq u(\frac{3}{2}n + 1) + \frac{3}{2}u \tag{48}$$

  The factorization and simplification of the right side of Equation (48) gives the following formula:

$$u(\frac{3}{2}n + 1) + \frac{3}{2}u = u(\frac{3}{2}n + \frac{3}{2} + 1) \quad \Rightarrow \quad u(\frac{3}{2}n + 1) + \frac{3}{2}u = u(\frac{3}{2}(n+1) + 1) \tag{49}$$

  Equation (48) will be written like that:

$$\epsilon_x^{n+1} \leq u(\frac{3}{2}(n+1) + 1) \tag{50}$$

  To conclude, the property P(n) ⇒ P(n+1) is verified. Thus, $\forall n \in \mathbb{N}$, the $\epsilon_x^n \leq u(\frac{3}{2}n + 1)$ is verified. Finally, we have demonstrated through the mathematical proof the correctness of our approach based on the study of the behavior of the error.

6 Conclusion

The main idea investigated in this paper is to study the behavior of round-off errors in the case of an integrator in a control loop in order to ensure that the results of computing rounding errors do not become larger than the output of the program. Thanks to the `ulp` and `ufp` functions used to estimate the maximum rounding error value. The motivating example shows that for this kind of computations, the rounding errors is always smaller than the output of the integrator. Of course, errors get larger, but not enough to exceed the program output value. To emphasize our approach, a mathematical proof based on induction has been made. We have demonstrated though it the correctness of our approach.

In future work, this study will be extended to automate the process of generating the programs constraint set and solving them using an SMT solver [12] or Z3 solver [20]. Also, it will be appreciated to experiment the approach on real and more complete programs, linear or not, to generalize this study. Another research consists in studying the impact of predicting the behavior of program rounding errors on the safety and security of critical embedded systems. These two aspects are very sensitive and contribute greatly to avoiding catastrophic damage.

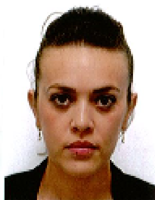
Acknowledgments

This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE Nř 2019 65 0090004707501).

References

1. Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. Fast and efficient bit-level precision tuning. In *Sensors Applications Symposium*, 2021.
2. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.
3. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, and A. Miné. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
4. S. Boldo, J-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.
5. S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH*, pages 243–252, 2011.
6. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Rigorous floating-point mixed-precision tuning. In *POPL*, pages 300–315, 2017.
7. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous floating-point mixed-precision tuning. *SIGPLAN Not.*, 52(1):300315, jan 2017.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252, 1977.
9. N. Damouche and M. Martel. Salsa: An automatic tool to improve the numerical accuracy of programs. In B. Dutertre and N. Shankar, editors, *Automated Formal Methods, AFM@NFM 2017*, volume 5 of *Kalpa Publications in Computing*, pages 63–76, 2017.
10. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *20th International Workshop Formal Methods for Industrial Critical Systems, FMICS*, volume 9128 of *LNCS*, pages 31–46, 2015.
11. E. Darulova and Viktor Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 235–248, 2014.

12. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008.
13. 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 *FMICS'09*, pages 53–69, 2009.
14. E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *LNCS*, pages 1–3, 2013.
15. J. Harrison. Floating-point verification. *J. UCS*, 13(5):629–638, 2007.
16. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.
17. Dorra Ben Khalifa, Matthieu Martel, and Assalé Adjé. Pop: A tuning assistant for mixed-precision floating-point computations. In *International Workshop on Formal Techniques for Safety-Critical Systems*, 2019.
18. Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Supercomputing, ICS'13*, pages 369–378, 2013.
19. Wonyeol Lee, Rahul Sharma, and Alex Aiken. On automatically proving the correctness of math.h implementations. *PACMPL*, 2(POPL):47:1–47:32, 2018.
20. Andrew Makhorin. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. Accessed: 2012/06/23.
21. Matthieu Martel. Floating-point format inference in mixed-precision. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017*, volume 10227 of *LNCS*, pages 230–246, 2017.
22. Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *SAFECOMP*, volume 10488 of *LNCS*, pages 213–229, 2017.
23. 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*. 2010.
24. J-M Muller and Marc Daumas. *Qualité des calculs sur ordinateur : vers des arithmétiques plus fiables ?* 1997.
25. P.G. Neumann. Technical report - uss yorktown dead in water after divide by zero., 1998.
26. Cuong Nguyen, Cindy Rubio-Gonzalez, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Int. Conf. on Software Engineering (ICSE)*, 2016.
27. P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11, 2015.
28. A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550, 2015.
29. Patriot missile defense : Software problem led to system failure at dhahran, saudi arabi., 1992.
30. Ariane 5 flight 501 failure., 1996.



Nasrine Damouche is a Research Engineer specializing in Critical Systems within the Critical Systems Design and Analysis research group at the Department of Complex Systems Engineering at ISAE-SUPAERO, Université de Toulouse, France. She received her Ph.D. in Computer Science from the University of Perpignan. Her research interests include formal methods, abstract interpretation based on static analysis, program semantics, numerical accuracy, compilation and recently model-based systems engineering and timing analysis in real-time systems.