Symbolic Round-Off Error between Floating-Point and Fixed-Point

Anh-Hoang Truong, Huy-Vu Tran, Bao-Ngoc Nguyen

VNU University of Engineering and Technology,
144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Abstract

Overflow and round-off errors have been research problems for decades. With the explosion of mobile and embedded devices, many software programs written for personal computers are now ported to run on embedded systems. The porting often requires changing floating-point numbers and operations to fixed-point, and here round-off error between the two versions of the program often occurs. We propose a novel approach that uses symbolic computation to produce a precise representation of the round-off error. From this representation, we can analyse various aspects of the error. For example we can use optimization tools like Mathematica to find the largest round-off error, or we can use SMT solvers to check if the error is always under a given bound. The representation can also be used to generate optimal test cases that produce the worst-case round-off error. We will show several experimental results demonstrating some applications of our symbolic round-off error.

Manuscript communication: received 13 September 2013, revised 25 March 2014, accepted 25 March 2014

Corresponding author: Anh Hoang Truong, hoangta@vnu.edu.vn

Keywords: Round-Off Error, Symbolic Execution, Fixed-Point, Floating-Point

1. Introduction

Traditional round-off error [1] is the difference between the real result and the approximate result that a computer generates. As computers may or may not be equipped with floating-point units (FPU), they may use different numbers representations: floating-point or fixed-point, respectively. In addition, the two types of computers usually have different precisions in their mathematical operations. As a result, the two types of computers may produce different approximation results for the same program executed with the same input data. The difference between the approximation results is another type of round-off errors that we address in this paper. Historically, round-off error has had severe consequences, such as those encountered in a Patriot Missile Failure [2] and Ariane 501 Software Failure [3].

Indeed there are three common types of round-off errors: real numbers versus floating-point numbers, real numbers versus fixed-point numbers, and floating-point numbers versus fixed-point numbers. This paper is based on our previous work [4] where we focused on the last type of round-off errors for two main reasons. Firstly, with the wide-spread use of mobile and embedded devices, many applications developed for personal computers are now run on these platforms. Secondly, even with new applications, it is impractical and time consuming to develop complex algorithms directly on embedded devices. So, many complex algorithms are developed and tested on personal computers that use floating-point numbers before they are ported to embedded devices that use fixed-point numbers.

Our work was inspired by the recent approaches to round-off error analysis [5, 6] that use various kinds of intervals to approximate round-off error. Instead of approximation, we try to build symbolic representation of round-off errors based on the idea of symbolic execution [7]. The symbolic representation, which we called 'symbolic round-off error', is an expression over program parameters that precisely represents the round-off errors of the program.

The symbolic round-off error allows us to analyse various aspects of (concrete) round-off errors. First, to find the maximum round-off error, we only need to find the optima of the symbolic round-off error in the (floating-point) input domain. We usually rely on an external tool such as Mathematica [8] for this task. Second, to check if there is a round-off error above a threshold or to guarantee that the round-off error is always under a given bound we can construct a numerical constraint and use SMT solvers to find the answers. We can also generate test cases that are optimal in terms of producing the largest round-off error.

Our main contributions in this paper is the building of symbolic round-off error between floating-point and fixed-point computation for arithmetic expressions, which is extensible for programs, and the application of the symbolic round-off error in finding the largest round-off error. We also built a tool and do some experimental results which show the advantages and disadvantages our approach.

The rest of the paper is structured as follows. The next section is some background. In Section 3 we extend the traditional symbolic execution to include round-off error information so that we can build a precise representation of the round-off error for a program. Then we present our Mathematica implementation to find the maximal round-off error and provide some experimental results in Section 4. Section 5 discusses related work. Section 6 concludes the paper.

2. Background

IEEE 754 [9, 10] defines binary representations for 32-bit single-precision floating-point numbers with three parts: the sign bit, the exponent, and the mantissa or fractional part. The sign bit is 0 if the number is positive and 1 if the number is negative. The exponent is an 8-bit number that ranges in value from -126 to 127. The mantissa is the normalized binary representation of the number to be multiplied by 2 raised to the power defined by the exponent.

In fixed-point representation, a specific radix point (decimal point) written "." is chosen so there is a fixed number of bits to the right and a fixed number of bits to the left of the radix point. The later bits are called the integer bits. The former bits are called the fractional bits. For a base b (usually base 2 or base 10,) with m integer bits and n fractional bits, we denote the format of the fixed-point by (b, m, n). When we use a base for fixed-point, we also assume the floating-point uses the same base. The default fixed-point format we use in this paper, if not specified, is $(2, 11, 4)$.

Example 1. *Assume we use fixed-point format* $(2, 11, 4)$ *and we have the floating-point number* 1001.010101. *Then the corresponding fixed-point number is* 1001.0101 *and the round-off error is* 0.000001.

Note that there are two types of lost bits in fixed-point computation: overflow errors and round-off errors. We only consider the latter in this work, as they are more subtle to track.

3. Symbolic round-off error

In this section we will first present our idea, inspired from [6], in which we apply symbolic execution [7] to compute a symbolic round-off error for arithmetic expressions. Then we will extend the idea to programs, which will be simplified to a set of arithmetic expressions with constraints for each feasible execution path of the programs.

3.1. Symbolic round-off error

Let $\mathbb{R}$, $\mathbb{L}$ and $\mathbb{I}$ be the sets of all real numbers, all floating-point numbers and all fixed-point numbers, respectively. $\mathbb{L}$ and $\mathbb{I}$ are finite because a fixed number of bits are used for their representations. For practicality, we assume that the number of bits in fixed-point format is not more than the number of significant bits in the floating-point representation, i.e. we assume $\mathbb{I} \subset \mathbb{L} \subset \mathbb{R}$.

Let's assume that we are working with an arithmetic expression over variables $x_1, .., x_n$, denoted by function $y = f(x_1, .., x_n)$ where $x_1, ..x_n$ and y are in $\mathbb{R}$. For a value $x \in \mathbb{R}$ we also denote $x' \in \mathbb{L}$ the rounded floating-point value of x, and $x'' \in \mathbb{I}$ the rounded fixed-point value of x'.

As arithmetic operations on floating-point and fixed-point may be different (in precision), we denote f_l and f_i the floating-point and fixed-point version of f, respectively, where real arithmetic operations are replaced by the corresponding operations in $\mathbb{L}$ and $\mathbb{I}$, respectively. We denote the operations in real as $+, -, \times, \div$, in floating-point as $\{+_l, -_l, \times_l, \div_l\}$ and in fixed-point as $\{+_i, -_i, \times_i, \div_i\}$.

The round-off error analysis in literature usually focuses on the largest error between f and f_l, which can be formalized:

$$\sup_{x_j \in \mathbb{R}, j=1..n} |f(x_1, .., x_n) - f_l(x'_1, .., x'_n)|$$

In our setting we focus on the largest round-off error between f_l and f_i. Since $\mathbb{L}$ is finite, we can use max instead of sup:

$$\max_{x'_j \in \mathbb{L}, j=1..n} |f_l(x'_1, .., x'_n) - f_i(x''_1, .., x''_n)|$$

Alternatively, one may want to check if there exists a round-off error exceeding a given threshold θ. In other words, one wants to find if the following constraint is satisfiable.

$$\exists x'_1, .., x'_n \text{ s.t. } |f_l(x'_1, .., x'_n) - f_i(x''_1, .., x''_n)| > \theta$$

Note that here we have some assumptions which we base on the fact that the fixed-point function is not manually reprogrammed to optimize for fixed-point computation. First, the evaluations of f_l and f_i are the same. Second, the scaling of variables in f_i is uniform, i.e. all values and variables use the same fixed-point format. Third, as mentioned in Section 2, we assume floating-point and fixed-point use the same base.

Because of the differences in floating-point and fixed-point representations, a value $x' \in \mathbb{L}$ usually needs to be rounded to the corresponding value $x'' \in \mathbb{I}$. So we have a non-decreasing monotonic function r from $\mathbb{L}$ to $\mathbb{I}$ and for $x' \in \mathbb{L}$, $x' -_l r(x')$ $(= x' -_l x'')$ is called the conversion error. This error is in $\mathbb{L}$ because we assume $\mathbb{I} \subset \mathbb{L}$. Note that we need to track this error as it will be used when we evaluate in floating-point, but not in fixed-point. In other words, the error is accumulated when we are evaluating the expression in fixed-point computation.

As we want to use the idea of symbolic execution to build a precise representation of round-off errors, we need to track all errors, when they are introduced by rounding and then propagated by arithmetic operations, and also new errors introduced because of the difference between arithmetic operations – $\times_l$ and $\times_i$ in particular.

To track the error, now we denote a floating-point x by (x_i, x_e) where $x_i = r(x)$ and $x_e = x -_l r(x)$. Note that x_e can be negative, depending on the rounding methods (example below). The arithmetic operations with symbolic round-off error between floating-point and fixed-point denoted by $+_s$, $-_s$, $\times_s$ and $\div_s$ are defined in a similar way to [6] as follows. The main idea in all operations is to determine the accumulation of error during computation.

Definition 1 (Basic symbolic round-off error).

$$(x_i, x_e) \ +_s (y_i, y_e) = (x_i +_l y_i, x_e +_l y_e)$$

$$(x_i, x_e) -_s (y_i, y_e) = (x_i -_l y_i, x_e -_l y_e)$$

$$\begin{aligned}(x_i, x_e) \times_s (y_i, y_e) =&(r(x_i \times_l y_i),\\ &x_e \times_l y_i +_l x_i \times_l y_e +_l x_e \times_l y_e\\ &+_l re(x_i, y_i))\end{aligned}$$

$$
\begin{aligned}
(x_i, x_e) \div_s (y_i, y_e) =& (r(x_i, y_i),\\
&(x_i +_l x_e) \div_l (y_i +_l y_e) -_l x_i \div_l y_i\\
&+_l de(x_i, y_i))
\end{aligned}
$$

where $re(x_i, y_i) = (x_i \times_l y_i) -_l (x_i \times_i y_i)$ (resp. $de(x_i, y_i) = (x_i \div_l y_i) -_l (x_i \div_i y_i)$) are the round-off errors between floating-point and fixed-point multiplication (resp. division).

Note that multiplication of two fixed-point numbers may cause round-off error so the round function r is needed in the first part and $re(x_i, y_i)$ in the second part. Similarly we have $de(x_i, y_i)$ in the definition of $\div_s$. Addition and subtraction may cause overflow errors, but we do not consider them in this work.

The accumulated error may not always increase, as shown in the following example.

Example 2 (Addition round-off error). *For readability, let the fixed-point format be (10, 11, 4) and let $x = 1.312543$, $y = 2.124567$. With rounding to the nearest, we have $x = (x_i, x_e) = (1.3125, 0.000043)$ and $y = (y_i, y_e) = (2.1246, -0.000033)$. Apply the above definition with addition, we have:*

$$
\begin{aligned}
&(x_i, x_e) +_s (y_i, y_e) =\\
&(1.3125 +_l 2.1246, 0.000043 +_l (-0.000033) =\\
&(3.4371, 0.00001)
\end{aligned}
$$

Example 3 (Multiplication round-off error). *With x, y in Example 2, for multiplication, we have:*

$$
\begin{aligned}
&(x_i, x_e) \times_s (y_i, y_e)\\
&= (r(1.3125 \times_l 2.1246), 0.000043 \times_l 2.1246\\
&\quad +_l 1.3125 \times_l (-0.000033) +_l 0.000043\\
&\quad \times_l (-0.000033) +_l re(1.3125, 2.1246))\\
&= (r(2.7885375), 0.000048043881\\
&\quad +_l re(1.3125, 2.1246))\\
&= (2.7885, 0.000048043881 +_l (1.3125 \times_l 2.1246)\\
&\quad -_l (1.3125 \times_i 2.1246))\\
&= (2.7885, 0.000048043881 +_l 2.7885375\\
&\quad -_l 2.7885)\\
&= (2.7885, 0.000085543881)
\end{aligned}
$$

As we can see in Example 3, the multiplication of two fixed-point numbers may cause a round-off error, so the second part of the pair needs an additional value re(). This value, like conversion errors, is constrained by a range. We will examine this range in the next section.

3.2. Constraints

In Definition 1, we represent a number by two components so that we can later build symbolic representation for the round-off error (the second component). In this representation, the two components are constrained by some rules.

Let assume that our fixed-point representation uses m bits for the integer part and n bits for the fractional part and is in binary. The first part x_i is constrained by its representation. So there exists $d_1, .., d_{m+n}$ such that

$$
\left(x_i = \sum_{i=1}^{m+n} d_{m-i} 2^{m-i}\right) \text{ where } d_j \in \{0, 1\}.
$$

The $x_e = x -_l r(x)$ is constrained by the 'unit' of the fixed-point in base b, which is b^{-n}, where unit is the absolute value between a fixed-point number and its successor. With rounding to the nearest, this constraint is half of the unit:

$$
|x_e| \le b^{-n}/2.
$$

Like x_e, the $re()$ and $de()$ part also have similar constraints.

3.3. Symbolic round-off error for expressions

In normal symbolic execution, an input parameter is represented by a single symbol. However in our approach, it will be represented by a pair of two symbols and we have the additional constraints on the symbols.

As we are working with arithmetic expressions, the symbolic execution will be proceeded by the replacement of variables in the expression with a pair of symbols, followed by the application of the arithmetic expression according to Definition 1. The final result will be a pair that consists of a symbolic fixed-point result and a symbolic round-off error. The later part will be the one we need for the next step – finding properties of round-off errors, in particular its global optima. But before that we will we discuss the extension of our symbolic round-off error for C programs.

```
/*
format: (2, 11, 4);
threshold: 0.26;
x: [-1, 3];
y: [-10, 10];
*/
typedef float Real;
Real rst;
Real maintest(Real x, Real y) {
   if(x > 0) rst = x*x;
   else rst = 3*x
   rst -= y;
   return rst;
}
```

Fig. 1. An example program.

3.4. Symbolic round-off error for programs

Following [6], we simplify our problem definition as follows. Given a mathematical function (input and output parameters are numeric) in the C programming language, with specifications for initial ranges of input parameters, fixed-point format and a threshold θ, determine if there is an instance of input parameters that causes the difference between the results of the function computed in floating-point and fixed-point above the threshold. Similarly to the related work, we restrict the function to the mathematical functions without unknown loops. That means the program has a finite number of all possible execution paths.

By normal symbolic execution [7] we can find, for each possible execution path, a pair of result as an expression and the corresponding constraints over the input parameters. Now for each pair, we can apply the approach presented in Section 3.1, combining with the path conditions/constraints to produce a symbolic round-off error for the each path.

Figure 1 is the example taken from [6] that we will use to illustrate our approach. In this program we use special comments to specify the fixed-point format, the threshold, and the input ranges of parameters.

3.5. Applications of symbolic round-off error

The symbolic round-off error can have several applications. To find the largest round-off error is only one of them that we focused here. It can be used to check the existence of a round-off error above a given threshold as we mentioned. The application will depend on the power of external SMT solvers as the constraints are non-linear in many programs. Since the symbolic round-off error is also a mathematical function, it can also tells us various information about the properties of the round-off error, such as which variables make significant contribution to the error. It can also be used to compute other round-off error metrics [11], such as the frequency/density of the error above a specified threshold, or the integral of error.

4. Implementation and experiments

4.1. Implementation

We have implemented a tool in Ocaml [12] and used Mathematica for finding optima. The tool assumes that by symbolic execution the C program for each path in the program we already have an arithmetic expression with constraints (initial input ranges and path conditions) of variables in the expression. The tool takes each of these expressions and its constraints and processes in the following steps:

1. Parse the expression and generate an expression tree in Ocaml. We use the Aurochs[1] parser generator for this purpose.
2. Perform symbolic execution on the expression tree with arithmetic operations to produce a symbolic round-off error expression together with constraints of variables in the expression.
3. Simplify the symbolic round-off error and constraints of variables using the Mathematica function `Simplify`. Note that the constants and coefficients in the input expression are also split into two parts: the

[1]http://lambda-diode.com/software/aurochs

fixed-point part and the round-off error part, both of them are constants. When the round-off error is non-zero, the simplification can reduce a lot the size of symbolic round-off error expression.

4. Use Mathematica to find optimum of the round-off error symbolic expression with constraints of variables. We use the Mathematica function `NMaximize` for this purpose. Since Mathematica does not support fixed-point, we need to build some utility functions for converting floating-point numbers to fixed-point numbers, and for emulating fixed-point multiplication (see Algorithm 1 and 2).

Algorithm 1: Rounding a floating-point value to fixed-point value

```
Input : A floating-point value $x$
Output: The converted fixed-point value of $x$
Data  : $bin_x$ stores the binary representation of $x$
Data  : $fp$ is the width of the fractional part
Data  : $x_1$ is the result of $bin_x$ after left shifted
Data  : $ip$ is the integer part of $x_1$
Data  : $digit$ is the value of $n^{th}$ bit
Data  : $fixed$: the result of converting
1  Procedure convertToFixed($x, fp$);
2  begin
3      Convert a floating-point $x$ to binary numbers $bin_x$;
4      $x_1$ = Shift left $bin_x$ by $fp$ bits;
5      $ip$ = Integer part of $x_1$;
6      Take the $(fp + 1)^{th}$ bit of $bin_x$ as $digit$;
7      if $digit$ equals 1 then
8          if $x > 0$ then
9              $ip = ip +_l 1$;
10         else
11             $ip = ip -_l 1$;
12     $fixed$ = Shift right $ip$ by $fp$ bits;
13     return $fixed$
14 end
```

Algorithm 2 emulate multiplication in fixed-point on Mathematica with round to the nearest. Assume that the fixed-point number has fp bits to represent the fractional part. The inputs of multiplication are two floating-point numbers a and b and the output is their product in fixed-point.

First, we shift left each number by fp bits to get two integer numbers. Then we take their product and shift right $2 * fp$ bits to produce the raw result without rounding. With round to the nearest, we shift right the product fp bits, store it in $i\_mul\_shr\_fp$, and take the integer and fractional part of $i\_mul\_shr\_fp$. If the fractional part of $i\_mul\_shr\_fp$ is larger than 0.5 then the integer part of $i\_mul\_shr\_fp$ needs to be increased by 1. We store the result after rounding it in $i\_mul\_rounded$. Shifting left $i\_mul\_rounded$ fp bits produces the result of the multiplication in fixed-point.

4.2. Experimental results

For comparison with [5], we use two examples taken from the paper as shown in Figure 1 and the polynomial of degree 5.

We also experimented with a Taylor series of a sine function to see how the complexity of the symbolic round-off error develops.

4.2.1. Experiment with simple program

For the program in Figure 1, it is easy to compute its symbolic expression for the two possible runs: $(x > 0 \wedge x \times x - y)$ and $(x < 0 \wedge 3 \times x - y)$.

Consider the first one. Combining with the input range of $x \in [0, 3]$ we get $x > 0 \wedge -1 \leq x \leq 3$, which can be simplified to $0 < x \leq 3$. So we need to find the round-off error symbolic expression for $x \times x - y$ where $0 < x \leq 3$ and $-10 \leq y \leq 10$.

Applying Definition 1, we get the symbolic round-off error:

$$(x_i \times_l x_i) -_l (x_i \times_i x_i) +_l 2 \times_l x_i \times_l x_e +_l x_e^2 -_l y_e$$

and the constraints of variables (with round to the nearest) are

$$x_i = \textstyle\sum_{j=1}^{15} d_j\, 2^{11-j} \bigwedge d_j \in \{0, 1\}$$
$$\bigwedge -1 \leq x_i \leq 3 \bigwedge x_i \geq 0 \bigwedge$$
$$-0.03125 \leq x_e, y_e \leq 0.03125$$

Algorithm 2: Fixed-point multiplication emulation in Mathematica

```
Input  : A floating value $a$
Input  : A floating value $b$
Output : The product of $a \times_i b$
Data   : $a\_shl\_fp$ is the result after $a$ after left
         shifted $fp$ bits
Data   : $i\_a\_shl\_fp$ is the integer part of $a\_shl\_fp$
Data   : $b\_shl\_fp$ is the result of $b$ after left
         shifted $fp$ bits
Data   : $i\_b\_shl\_fp$ is the integer part of $b\_shl\_fp$
Data   : $i\_mul$ is the product of $i\_a\_shl\_fp$ and
         $i\_b\_shl\_fp$
Data   : $i\_mul\_shr\_fp$ is the result of $i\_mul$ after
         right shifted $fp$ bits
Data   : $ipart\_i\_mul$ is the integer part of
         $i\_mul\_shr\_fp$
Data   : $fpart\_i\_mul$ is the fraction part of
         $i\_mul\_shr\_fp$
Data   : $truncate\_part$ is result of $fpart\_i\_mul$
         after left shifted 1 bit
Data   : $round\_part$ is the integer part of
         $truncate\_part$
Data   : $i\_mul\_rounded$ is the result after
         rounding
Data   : $result$ is the product of $a$ and $b$ in
         fixed-point

Procedure iMul($a$, $b$);
begin
    Convert a floating-point $x$ to binary numbers $bin_x$;
    $a\_shl\_fp$ = Shift left $a$ by $fp$ bits;
    $i\_a\_shl\_fp$ = Integer part of $a\_shl\_fp$;
    $b\_shl\_fp$ = Shift left $b$ by $fp$ bits;
    $i\_b\_shl\_fp$ = Integer part of $b\_shl\_fp$;
    $i\_mul$ = multiply two integers $i\_a\_shl\_fp$ and $i\_b\_shl\_fp$;
    $i\_mul\_shr\_fp$ = Shift right $i\_mul$ by $fp$ bits
    and then take the integer and the fractional part of $i_mul$ is that $ipart\_i\_mul$ and $fpart\_i\_mul$;
    $truncate\_part$ = Shift left 1 bit $fpart\_i\_mul$;
    $round\_part$ = Take Integer part of $truncate\_part$;
    $i\_mul\_rounded$ = $ipart\_i\_mul$ + $round\_part$; with rounding to the nearest
    $result$ = Shift right $i\_mul\_rounded$ by $fp$ bits;
    return $result$
end
```

Next we convert the round-off error symbolic expression and constraints to Mathematica syntax as in Figure 2. Mathematica found the following optima for the problem:

- With round to the nearest, the maximal error is 0.2275390625. The inputs that cause the maximal round-off error are: $x_i = 2.875; x_e = 0.03125$ so $x = x_i +_l x_e = 2.90625$ and $y_i = 4.125; y_e = -0.03125$ so $y = y_i +_l y_e = 4.09375$.

- With round towards $-\infty$ (IEEE 754 [9]): the error is 0.4531245939250891 with $x_i = 2.8125$; $x_e = \sum_{j=-5}^{-24} 2^j \rightarrow x = x_i +_l x_e = 2.874999940395355225$ and $y_i = 4$; $y_e = -\sum_{j=-5}^{-24} 2^j \rightarrow y = y_i +_l y_e = 3.937500059604644775$.

Comparing to [5] we find that using round to the nearest the error is in $[-0.250976, 0.250976]$ so our result is more precise.

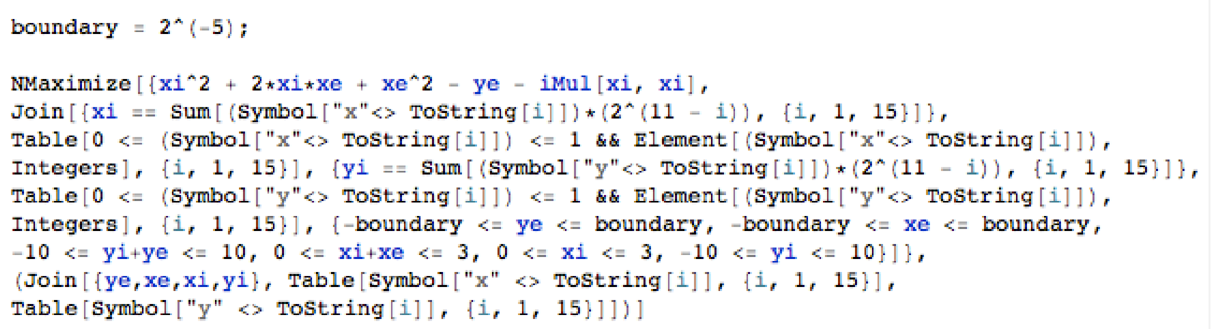
To verify our result, we wrote a test program for both rounding methods that generates 100.000.000 random test cases for $-1 \leq x \leq 3$ and $-10 \leq y \leq 10$ and directly computes the round-off error between floating-point and fixed-point results. Some of the largest round-off error results are shown in Table 1 and in Table 2. The tests were run many times, but we did not find any inputs that caused larger round-off error than predicted by our approach.

4.2.2. *Experiment with a polynomial of degree 5*

Our second experiment is a polynomial of degree 5 taken from [5]:

$$P5(x) = 1 - x + 3x^2 - 2x^3 + x^4 - 5x^5$$

where fixed-point format is $(2, 11, 8)$ and $x \in [0, 0.2]$. After symbolic execution, the symbolic



```
boundary = 2^(-5);

NMaximize[{xi^2 + 2*xi*xe + xe^2 - ye - iMul[xi, xi],
Join[{xi == Sum[(Symbol["x"<> ToString[i]])*(2^(11 - i)), {i, 1, 15}]},
Table[0 <= (Symbol["x"<> ToString[i]]) <= 1 && Element[(Symbol["x"<> ToString[i]]),
Integers], {i, 1, 15}], {yi == Sum[(Symbol["y"<> ToString[i]])*(2^(11 - i)), {i, 1, 15}]},
Table[0 <= (Symbol["y"<> ToString[i]]) <= 1 && Element[(Symbol["y"<> ToString[i]]),
Integers], {i, 1, 15}], {-boundary <= ye <= boundary, -boundary <= xe <= boundary,
-10 <= yi+ye <= 10, 0 <= xi+xe <= 3, 0 <= xi <= 3, -10 <= yi <= 10}]},
(Join[{ye,xe,xi,yi}, Table[Symbol["x" <> ToString[i]], {i, 1, 15}],
Table[Symbol["y" <> ToString[i]], {i, 1, 15}]])]
```

Fig. 2. Mathematica problem for example in Figure 1.

Table 1. Top round-off errors in 100.000.000 tests with round to the nearest

No.	x	y	err
1	2.9061846595979763	-6.530830752525674	0.22674002820827965
2	2.9061341245904635	-4.4061385404330045	0.2267540905421832
3	2.9062223934725107	-3.2184902596952947	0.22711886001638248

round-off error is:

$$\begin{aligned}
&0. +_l 3 \times_l x_i^2 -_l 2 \times_l x_i^3\\
&-_l 5 \times_l x_i^5 -_l x_e +_l 6 \times_l x_i^2\\
&\times_l x_e +_l 4 \times_l x_i^3 \times_l x_e -_l\\
&25 \times_l x_i^4 \times_l x_e +_l 3 \times_l x_e^2\\
&-_l 6 \times_l x_i \times_l x_e^2 +_l 6 \times_l x_i^2\\
&\times_l x_e^2 -_l 50 \times_l x_i^3 \times_l x_e^2 -_l\\
&2 \times_l x_e^3 +_l 4 \times_l x_i \times_l x_e^3\\
&-_l 50 \times_l x_i^2 \times_l x_e^3 +_l x_e^4\\
&-_l 25 \times_l x_i \times_l x_e^4 -_l\\
&5 \times_l x_e^5 -_l 3 \times_l iMul[x_i, x_i]\\
&+_l 2 \times_l iMul[iMul[x_i, x_i], x_i] -_l\\
&iMul[iMul[iMul[x_i, x_i], x_i], x_i] +_l\\
&5 \times iMul[iMul[iMul[iMul[x_i, x_i], x_i], x_i], x_i]
\end{aligned}$$

and the constraints of variables with round to the nearest are

$$\begin{aligned}
x_i = \; & \textstyle\sum_{j=1}^{19} d_j\, 2^{11-j} \bigwedge d_j \in \{0, 1\}\\
& \bigwedge 0 \le x_i \le 0.2\\
& \bigwedge -0.001953125 \le x_e \le 0.001953125\\
& \bigwedge 0 \le x_i +_l x_e \le 0.2\\
& \bigwedge 0.0625 \le y_r \le 0.0625
\end{aligned}$$

For this problem, Mathematica found the maximum 0.007244555 with $x_i = 0.12890625; x_e = -0.001953125$ so $x = x_i +_l x_e = 0.126953125$. In [5], their real error is in $[-0.01909, 0.01909]$ when using round to nearest, so our result is more precise.

We verify our results with round to the nearest by directly computing the difference between the fixed-point and floating-point with 100.000.000 random test cases for $0 \le x \le 0.2$. The largest error we found is 0.00715773548755 which is very close but still under our bound.

4.2.3. *Experiment with Taylor series of sine function*

In the last experiment, we want to see how far we can go with our approach, so we use a Taylor series of sine function. $P7(x) = x - 0.166667x^3 + 0.00833333x^5 - 0.000198413x^7$ where the fixed-point format is $(2, 11, 8)$ and $x \in [0, 0.2]$.

The largest round-off error is 0.00195312 with $x_i = 0; x_e = 0.00195313$ so $x = x_i +_l x_e = 0.00195313$. Comparing to the results in [5], using round to the nearest the error is in $[-0.00647, 0.00647]$ so our result is much better.

We tried with longer Taylor series but Mathematica could not solve the generated problem. We are aware of the scalability of this approach and plan to try with more specialized solvers such as raSAT [13] in our future work.

Table 2. Top round-off errors in 100.000.000 tests with round towards $-\infty$

No.	x	y	err
1	2.8749694304357103	-0.874361299827422	0.4523105257672544
2	2.874964795521085	-5.4371633555888055	0.45258593107439893
3	2.8749437460462444	-1.249785289663956	0.4525868325943687

5. Discussion and related work

5.1. Discussion

We have presented a symbolic round-off error technique that precisely represents round-off error between floating-point and fixed-point versions of a program. The symbolic round-off error enables several applications in error analysis and test cases generation. Note that in the above experiments Mathematica gives us solutions when it found optima. The solutions can be used to generate test cases for the worst round-off error.

We are aware of several advantages and drawbacks with this approach. First, our approach assumes that Mathematica does not over approximate the optima. However, even if the optima is over approximated, the point that produces the optimum is still likely to be the test case we need to identify. We can recompute the actual round-off error when this occurs.

Second, it is easy to see that our approach may not be scalable for more complex programs. The round-off error representation will grow very large in complex programs. Some simplification strategy may be needed, such as sorting the contributions to the round-off error of components in the round-off error expression and removing components that are complex but contribute insignificantly to the error. Alternatively, we can divide the expression into multiple independent parts to send smaller problems to Mathematica.

Third, if a threshold is given, we can combine it with testing to find a solution for the satisfiability problem, or we can use an SMT solver for this purpose. We plan to use raSAT [13] for this application when the tool is available for use.

Finally, we can combine our approach with interval analysis. The interval analysis will be used for complex parts of the program, while other, simpler parts can have precise round-off errors determined.

Note that the largest round-off error is only one of the metrics for the preciseness of the fixed-point function versus its floating-point one. In our previous work [11], we proposed several metrics and the symbolic round-off error seems convenient to compute these metrics as it contains rich information about the nature of the round-off error.

5.2. Related works

Overflow and round-off error analysis has been studied from the early days of computer science because both fixed-point and floating-point number representations and computations have the problem. Most work addresses both overflow and round-off error, for example [10, 9]. Because round-off error is more subtle and sophisticated, we focus on it in this work, but our idea can be extended to overflow error.

As we mentioned, there are three kinds of overflow and round-off errors: real numbers versus floating-point, real numbers versus fixed-point, and floating-point numbers versus fixed-point numbers. Many previous works focus on the first two types of round-off errors, cf. [14]. Here we focus on the last type of round-off errors. The most recent work that we are aware of is of Ngoc and Ogawa [6, 5]. The authors develop a tool called CANA for analyzing overflows and round off errors. They propose a new interval, the extended affine interval (EAI), to estimate round-off error ranges instead of the classical interval [15] and affine interval [16]. EAI avoids the problem of introducing new noise symbols of AI, but it is still as imprecise as our approach.

6. Conclusions

We have introduced symbolic round-off error and instantiated the symbolic round-off error between a floating-point function and its fixed-point version. The symbolic round-off error is based on symbolic execution extended for the round-off error so that we can produce a precise representation of the round-off error. It allows us to determine a precise maximal round-off error and to produce the test case for the worst error. We also built a tool that uses Mathematica to find the worst error from the symbolic round-off error. The initial experimental results are very promising.

We plan to investigate possibilities to reduce the complexity of the symbolic round-off error before sending it to the solver. For example, we might introduce techniques for approximating the symbolic round-off error. For real world programs, especially the ones with loops, we believe that combining interval analysis with our approach may allow us to find a balance between preciseness and scalability. We also plan study the round-off error that may cause the two versions of the program following different execution paths.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to the earlier version of the paper. Special thanks to Prof. Randy Ribler for improving the presentation of the paper.

References

[1] J. Wilkinson, Modern error analysis, SIAM Review 13 (4) (1971) 548–568.
URL `http://dx.doi.org/10.1137/1013095`

[2] N. J. Higham, Accuracy and Stability of Numerical Algorithms, SIAM: Society for Industrial and Applied Mathematics, 2002.

[3] M. Dowson, The ariane 5 software failure, SIGSOFT Softw. Eng. Notes 22 (2) (1997) 84–. doi:10.1145/251880.251992.

[4] A.-H. Truong, H.-V. Tran, B.-N. Nguyen, Finding round-off error using symbolic execution, in: Conference on Knowledge and Systems Engineering 2013 Proceedings, 2013, pp. 105–114. doi:10.1109/SEFM.2009.32.

[5] T. B. N. Do, M. Ogawa, Overflow and Roundoff Error Analysis via Model Checking, in: Conference on Software Engineering and Formal Methods, 2009, pp. 105–114. doi:10.1109/SEFM.2009.32.

[6] T. B. N. Do, M. Ogawa, Combining Testing and Static Analysis to Overflow and Roundoff Error Detection, in: JAIST Research Reports, 2010, pp. 105–114.

[7] J. C. King, J. Watson, Symbolic Execution and Program Testing, in: Communications of the ACM, 1976, pp. 385 – 394. doi:10.1145/360248.360252.

[8] S. Wolfram, Mathematica: A System for Doing Mathematics by Computer, Addison-Wesley, 1991.

[9] D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, in: ACM Computing Surveys, 1991, pp. 5 – 48. doi:10.1145/103162.103163.

[10] W. Stallings, Computer Organization and Architecture, Macmillan Publishing Company, 2000.

[11] T.-H. Pham, A.-H. Truong, W.-N. Chin, T. Aoshima, Test Case Generation for Adequacy of Floating-point to Fixed-point Conversion, Electronic Notes in Theoretical Computer Science 266 (0) (2010) 49 – 61, proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS). doi:10.1016/j.entcs.2010.08.048.

[12] J. B. Smith, Practical OCaml (Practical), Apress, Berkely, CA, USA, 2006.

[13] V.-K. To, M. Ogawa, raSAT: SMT for Polynomial Inequality, Tech. rep., Research report (School of Information Science, Japan Advanced Institute of Science and Technology) (2013).

[14] M. Martel, Semantics of roundoff error propagation in finite precision calculations, in: Higher-Order and Symbolic Computation, 2006, pp. 7 – 30.

[15] A. Goldsztejn, D. Daney, M. Rueher, P. Taillibert, Modal intervals revisited : a mean-value extension to generalized intervals, in: In International Workshop on Quantification in Constraint Programming (International Conference on Principles and Practice of Constraint Programming, CP-2005), Barcelona, Espagne, 2005.
URL `http://hal.inria.fr/hal-00990048`

[16] J. Stolfi, L. d. Figueiredo, An introduction to affine arithmetic, in: Tendencias em Matematica Aplicada e Computacional, 2005.