RangeLab: a Static-Analyzer to Bind the Accuracy of Finite-Precision Computations
Matthieu Martel

To cite this version:
Matthieu Martel. RangeLab: a Static-Analyzer to Bind the Accuracy of Finite-Precision Computations. SYNASC: Symbolic and Numeric Algorithms for Scientific Computing, Sep 2011, Timisoara, Romania. pp.118-122. hal-00813002

HAL Id: hal-00813002
https://hal.archives-ouvertes.fr/hal-00813002
Submitted on 14 Apr 2013

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract—This article introduces RangeLab, a simple tool to validate the accuracy of floating-point or fixed-point computations. Given intervals for the inputs, RangeLab computes the range of the outputs of simple functions with conditionals and loops as well as a range for the roundoff errors arising during the computation. Hence the user not only obtains the range of the result of the computation in the computer arithmetic but also a bound on the difference between the computer result and the result in infinite precision. RangeLab is based on static analysis by abstract interpretation and, in this article, we describe the techniques implemented in the tool. In particular, RangeLab uses a hybrid numerical-formal evaluation technique used to limit the wrapping effect in interval computations.

Keywords—Formal verification, fixed-point arithmetic, floating-point arithmetic, numerical stability.

I. INTRODUCTION

RangeLab is a new interactive tool which automatically computes the ranges of the numerical outputs of simple programs and binds the roundoff errors arising during floating-point or fixed-point computations. Typically, RangeLab takes as entry a simple code, with arithmetic expressions, arrays, assignments and simple control structures. Given some ranges for the inputs, the system computes safe ranges for the outputs, according to the computer arithmetic (IEEE754 floating-point arithmetic [1] or fixed-point arithmetic [2]). In addition, the system computes a safe range of the roundoff error on each result, i.e. a range for the difference between the result in the computer arithmetic and the result in infinite precision. Doing so, RangeLab makes it possible to assert the accuracy of some computations described by simple pieces of code. Related tools are mainly INTLAB [3] for the classical interval arithmetic, Gappa [4] for semi-automatic proofs and Fluctuat [5], [6] for the static analysis of C codes.

In this article, we present the theoretical ideas implemented in RangeLab. In summary, RangeLab uses a variant of interval arithmetic called the arithmetic of finite precision numbers with errors as well as techniques of static analysis by abstract interpretation [7]. Thanks to static analysis, RangeLab can perform interval-like computations for codes containing conditional control structures. In this case, the tool returns an over-approximation of the results in all the branches of the program. In addition, the static analysis of loops contain extrapolation techniques which make it possible to compute an over-approximation of the result of a while loop without entirely executing it.

Our main contribution is technical: the tool itself. At the theoretical level, our contributions in this article are: (i) Refinements of the arithmetic of finite precision numbers with errors [8] with efficiently implementable division and square root operators. (ii) A new abstract semantics, used in the static analysis to improve the accuracy of interval computations. We use semi-formal expressions which are formal expressions of limited height. Formal computations are performed to simplify expressions, limiting the wrapping effect. In addition, we perform interval slicing, i.e. we decompose intervals in partitions of sub-intervals to improve the accuracy of the evaluation of the formal expressions.

II. OVERVIEW OF THE TOOL

In this section, we give an overview of what is computed by RangeLab and of how the tool works. First of all, RangeLab is an interactive system which displays a prompt symbol in order to indicate that the tool is waiting for commands (written in Matlab's syntax).

```matlab
-] 0.1+0.2
ans = float64: [2.999999999999999E-1,
3.000000000000001E-1]
error: [-4.857225732735059E-17,
4.857225732735060E-17]
```
In this article, we always assume that the floating-point computations are rounded to the nearest. In the above session, the system computes the numerical integral $\int_{0.25}^{50} g(x) dx$ of some function $g : \mathbb{R} \to \mathbb{R}$ by means of a procedure trapeze implementing the trapeze rule. The parameters require to truncate the interval $[0.25, 50]$ in 100 steps in order to compute the result. The code of the trapeze method is stored in a separate file trapeze.m whose content is:

```matlab
function r = trapeze(a,b,n)
    r = 0.0; xa=a; h=(b-a)/n;
    while xa<b,
        xb = xa+h;
        if xb>b, xb=b end;
        r = r + ((g(xb)+g(xa))/2)*h;
        xa=xa+h;
    end
end
```

The code of the function $g$ is stored in a file g.m:

```matlab
function y = g(x)
    y = [1.0,2.0.0]/(x*x*x*x)
end
```

The function $g$ has an interval coefficient indicating that $g$ represents the family of functions $G = \{ x \mapsto \frac{y}{x}, 1 \leq a \leq 2 \}$. In the above session, RangeLab computes that the result of trapeze(0.25,50.0,100) is

$$ F = [6.557548333631875 \times 10^{1}, 1.311509666726376 \times 10^{-2}] $$

with an error

$$ E = [-1.4568 \ldots \times 10^{-12}, 1.4577 \ldots \times 10^{-12}] $$

This means that, for any function in $G$, trapeze(0.25,50.0,100) returns a value in $F$ and that the numerical error on this result belongs to $E$, assuming that the errors on the coefficients $a$ is null. We may claim several things from this execution:

- The error is significantly greater than the precision, roundoff error arose during this execution. This is because the integration of $g$ by the trapeze methods leads to the addition of values of different magnitudes since $g$ is decreasing. We can confirm this matter of fact by computing a larger integral, with the same steps than in the former case:

```matlab
{-} trapeze(0.25,75.0,150)
ans = float64: [6.56756562766613 \times 10^{1}, 2.1655284708336 \times 10^{-2}]
```

The error is greater while the magnitude of the floating-point result is almost identical.

Let us also remark that RangeLab does not detect the method error which affects the result of this computation: trapeze(0.25,50.0,100) differs significantly from $\int_{0.25}^{50} g(x) dx$ even in infinite precision.

### III. Inside RangeLab

RangeLab uses an arithmetic in which error terms are attached to the floating-point or fixed-point numbers. They indicate a range for the roundoff error due to the rounding of the exact value in the current rounding mode. The exact error term being possibly not representable in finite precision, we compute an over-approximation and return an interval with bounds made of multiple precision floating-point numbers. Indeed, the error interval may be computed in an arbitrarily large precision since it aims at binding a real number and, in practice, we use the GMP multi-precision library.

Let $x$ and $y$ be to values represented in our arithmetic by the pairs $(f_x, e_x)$ and $(f_y, e_y)$ where $f_x$ and $f_y$ are the floating-point or fixed-point numbers approximating $x$ and $y$ and $e_x$ and $e_y$ the error terms on both operands. Let $o(v)$ be the rounding of the value $v$ in the current rounding mode and let $\varepsilon(v)$ be the roundoff error, i.e. the error arising when rounding $v$ into $o(v)$. We have by definition $\varepsilon(v) = v - o(v)$ and, in practice, when $v$ is an interval, we approximate $o(v)$ by $[-\frac{1}{2}ulp(m), \frac{1}{2}ulp(m)]$ in floating-point arithmetic, or by $[0, ulp(m)]$ in fixed-point arithmetic, where $m$ is the maximal bound of $v$, in absolute value, and $ulp$ is the function which computes the unit in the last place of $m$ [9]. The elementary operations are defined in equations (1) to (5).

For an addition, the errors on the operands are added to the error due to the roundoff of the result. For a subtraction, the errors on the operands are subtracted. The semantics of the multiplication comes from the development of $(f_x + e_x) \times (f_y + e_y)$. Our definition of the division comes from the fact that

$$ \frac{x}{y} = \frac{f_x + e_x}{f_y + e_y} \times \frac{1}{1 + \frac{e_y}{f_y}} $$

(6)
and, using the power series development of \(\frac{1}{1+x^2}\), we obtain

\[
\frac{x}{y} = \left(\frac{f_x}{f_y} + \frac{e_x}{f_y}\right) \left(1 + \frac{q}{1-q}\right).
\]  

Finally, by \(\frac{f_x}{f_y} = o(\frac{f_x}{f_y}) + \epsilon(\frac{f_x}{f_y})\), we obtain Equation (4).

We mentioned earlier that a pair \((f, e)\) made of a floating or fixed-point number \(f\) and an error \(e\) is used to represent the real number \(r = f + e\) rounded to \(f\) in the current computer format. In practice, this is extended to intervals. By definition, the pair of intervals \((\bar{f}, \bar{e})\) represents the set of floating or fixed-point numbers belonging to \([f, \bar{f}]\) whose error with respect to the real number they represent is in \([e, \bar{e}]\):

\[
(\bar{f}, \bar{e}) = \{ (f, e) \in \mathcal{F} \times \mathcal{R} : f \in [f, \bar{f}], e \in [e, \bar{e}] \} \quad (8)
\]

In Equation (8), \(\mathcal{F}\) denotes the current format used to represent the number in memory (a fixed or floating-point format) and \(\mathcal{R}\) denotes the set of real numbers.

RangeLab also performs symbolic computations as long as the height \(\text{height}(e)\) of the resulting expression \(e\) is less than or equal to a user-defined parameter \(h\) (\(h = 5\) by default in our tool). If \(\text{height}(e) > h\) then we evaluate the deepest sub-expressions until we obtain a new expression \(e'\) of height \(h\). The tool uses some rewriting rules to generate a set \(E\) of formal expressions mathematically equivalent to \(e\) and whose sizes are limited to \(h\). These rules correspond to formal simplifications of expressions. All the expressions in \(E\) are mathematically correct and we evaluate them successively in the interval arithmetic to obtain a set \(V\) of results. The values of \(V\) are intervals and they are all correct evaluations of the initial expression \(e\). However the intervals of \(V\) do not have all the same width and we return the smallest as result of our computation.

A second improvement, implemented in our tool, consists of performing the evaluation of the formal expressions of \(E\) by means of a dichotomic slicing of the variables (one variable at the time): instead of evaluating \(e\) with \(x = [a, b]\), we evaluate \(e\) with \(x = [a, \frac{a+b}{2}]\) and then with \(x = [\frac{a+b}{2}, b]\) before joining the partial results. This is repeated recursively \(n\) times as far as it improves the accuracy of the result. Note that only the intervals for the floating-point or fixed-point numbers are sliced, the error term being always the whole interval of error. However, slicing has a strong incidence on the computed errors since they depend on the ulp of the fixed or floating-point values.

RangeLab uses standard abstract interpretation techniques [7] to evaluate control structures such as conditional or loops. Basically, the tool returns a superset of the outputs that we would obtain with a standard semantics by selecting scalar entries in the ranges of the input intervals.

When a condition containing interval variables can evaluate to both true and false, the \texttt{then} branch is evaluated in an environment where the intervals corresponding to the variables occurring in the condition are reduced to the set of values which make the condition true and the \texttt{else} branch is evaluated in the converse environment. Then the values of the variables after execution of the \texttt{then} and \texttt{else} branches are joined. When a condition involves a floating-point or fixed-point value \((f, e)\), only \(f\) can be reduced, the error term \(e\) being left unchanged.

Concerning the \texttt{while} loops, the conditions are interpreted as the conditions of the \texttt{if} statements and the semantics of the \texttt{while}, in an initial environment \(env_0\), is: (1) The entries are restricted to the sub-intervals which make the condition true. This yields an environment \(env_1\) which is used to execute the first iteration of the loop. (2) The environment \(env_2\) containing the values of the variables at the end of the first iteration is joined to \(env_1\) into a new environment \(env_3\). (3) Step (1) is then repeated with the values of \(env_3\) as entries. The process is repeated until \(env_1 = env_2\), i.e. a fixed-point is reached. Finally, the environment at the end of the loop is defined by joining \(env_3\) and \(env_3\) where \(env_3\) and \(env_3\) denote the environments \(env_0\) and \(env_3\) in which the values of the variables are restricted to the sub-intervals which make the condition false. This is illustrated by the example below.

```
-] a=0; x=0;
```
At the end of the loop, the value of $a$ is intersected with the values which invalidate the condition, we obtain $a = [0, 10] \cap [10, +\infty]$. Since $x$ is not constrained by the condition, its value at the end of the loop corresponds to the value of the last assignment $x=a$. In addition, RangeLab implements a widening operator \cite{7} to compute the result of a while loop without entirely executing it. This is as a conservative extrapolation of the result using a kind of convergence acceleration technique. The results are always correct but less accurate.

When floating-point or fixed-point computations are performed inside a while loop, both the values and error terms are widened. However, as for the if construct, the error terms are not constrained by the condition of the loop. Let us consider the following unstable computation:

```matlab
while i<n, i=i+1; x=x*k
end
```

**RangeLab** states that the floating-point value of $x$ at the end of the loop is $9.999999999999999 \cdot 10^{-2}, +\infty$ and that the roundoff error belongs to $]-\infty, +\infty[$. This means that the difference between the computed value and the value that we would obtain in infinite precision is arbitrarily large. Let us consider now a stable computation:

```matlab
while i<n, i=i+1; x=x*k
end
```

This computation is stable, the value of $x$ at the end of the loop being $k^n$ with $k \in [0.8, 0.9]$ and $n \geq 0$. The loop performs only $n = 10$ iterations, but a widening is invoked because $x$ is not constrained by the condition of the loop. RangeLab computes that for any $k \in [0.8, 0.9]$ and for any $0 \leq i \leq 10$, $k^n$ belongs to $[0.0, 0.9]$. The system has extrapolated that $\bigcap_{i=0}^{n} [0.8, 0.9]^i$ goes towards $[0.0, 0.9]$ as $n$ goes towards $+\infty$.

**IV. Experimental Results**

Our first example is a Gaussian elimination procedure. In interval arithmetic, when a function is expansive, as it is the case here, the result is usually a large interval and the user cannot know whether numerical errors arose during the evaluation or not. It is only possible to state that the error may be as large as the width of the resulting interval, which is not quite informative. We use the following function which also gives a flavor of the matrix operations supported by our tool.

```matlab
function [x] = gaussel(A,b) 
N = max(size(A));
for j=2:N, % Gaussian elimination 
    m = A(i,j-1)/A(j-1,j-1);
    A(i,:) = A(i,:) - A(j-1,:)*m;
    b(i) = b(i) - m*b(j-1); 
end
x = zeros(N,1); % Back substitution 
x(N) = b(N)/A(N,N); 
for j=N-1:-1:1, 
    x(j)=(b(j)-A(j,j+1:N)*x(j+1:N))/A(j,j)
end 
end
```

We take the matrix $A$ and vector $b$ defined by:

$$A = \begin{bmatrix} u & 0 & 0 \\ 0 & v & 0 \\ 0 & 0 & u \\ 0 & v & u \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$u = ([4.5454 \cdot 10^{-8}, 1.5454 \cdot 10^{-7}], 
[-3.45 \ldots 10^{-23}, 3.45 \ldots 10^{-23}])$

$v = ([-.12722 \cdot 10^{-7}, -5.4545 \cdot 10^{-8}], 
[-3.57 \ldots 10^{-23}, 3.57 \ldots 10^{-23}])$

RangeLab computes the following result.

```matlab
ans = [2.042984771573605E8, ...
-1.673296672306825E8, ...
2.109099300620420E9, ...
1.172588832487311E7 ]
```

Errors: $[-4.038941344594430E-6, 
-1.223143375128125E-7, 
4.038941344594430E-6, 
1.223143375128125E-7 ]$

Firstly, the system warns about the scalar product done in the back substitution because the roundoff errors arising during this computation depend on the order of the operations performed to compute the scalar product. This order is not unique and the error returned by the tool is correct only if the terms of the product are accumulated in increasing order of the indices. Secondly, RangeLab indicates that while the result of the computation belongs...
to a large interval (for instance the width of the interval is about $2.3\cdot10^9$ in the first dimension), the numerical error is always very small (always less than $e = 4.04 \cdot 10^{-6}$ in absolute value). This means that for any matrix with floating-point coefficients taken in the intervals of $A$, the numerical error on the result of the gaussian elimination is always less than or equal to $e$ in absolute value, which corresponds to the error precision in the IEEE754 double format. In other words, for any matrix with floating-point coefficients taken in the intervals of $A$, the system ensures that the roundoff error on the result of the gaussian elimination are negligible. This result could not be obtained by the usual interval arithmetic.

Our second example concerns the fixed-point arithmetic. A fixed number $[i,f]$ is made of an integer part $i$ and a fractional part $f$. It is written $i.f$ in RangeLab's syntax. For an input value, RangeLab always adjusts the size of $i$ to the minimal size needed to represent the value. The size of the fractional part can be specified by the user (a default size is used instead). Concerning the result of an elementary operation, the integer part is extended if needed, in order to avoid overflows, and the size of the fractional part corresponds to the maximal size of the fractional parts of the operands. Operations are rounded towards zero and introduce roundoff errors, just like floating-point numbers do. For example, the following command adds the values $1.1$ and $1.2$ represented with a fractional part of $16$ bits.

```
[-] l_1#16+1_2#16
ans = fixed(2,16): 2_299987792969
   error: 9.155273438254952E-6
```

The tool states that the result is encoded on a 2-bits integer part and on a 16-bits fractional part and that the roundoff error on the result is $9.15\ldots 10^{-6}$. RangeLab also handles intervals of fixed-point numbers in the same way than intervals of floating-points numbers.

Let us consider now the implementation of Horner's method for the evaluation of a polynomial. The function below takes as arguments an array with the coefficients of the polynomial $p$ and the point $x$ and computes $p(x)$.

```matlab
function y = horner(a,x)
    y = a(1); n = max(size(a));
    for i = 2:n, y = a(i) + x * y, end
end
```

If we set the default size of the fractions to 16, we obtain:

```
[-] set frac 16
[-] a = [1_0.8,1_2, -6_2, -5_8]
    [10_8,1_1.2, -6_2, 5_8];
[-] horner(a,[0_0,4_0])
ans = fixed(6,16): [-62_199996948242,
    50_39990234375]
   error: [0.000000000000000E-1,
    2.593994140625000E-3]
```

The system computes that the result is always representable without overflow on 6 bits for the integer part and that, with 16 bits of fraction the roundoff error on the result of the evaluation is bounded by $2.59 \ldots 10^{-3}$.

V. Conclusion

In this article, we have given an overview of the features of RangeLab, an interactive tool to detect numerical errors in floating-point and fixed-point computations. We have also given a survey of the techniques implemented in the tool. RangeLab handles the elementary arithmetic operations and we would like to add more advanced features in the future but this raises some theoretical issues. Firstly, in order to add the usual elementary mathematical functions (sine, logarithms, etc.), the way roundoff errors are propagated inside these functions must be defined carefully. Secondly, we would like solvers, for example for ODEs. However, in order to estimate the error on the results, guaranteed integration techniques should be employed [10], [11].

References