Evaluating a constant expression in multiple precision with a guaranteed error bound

Sylvain Chevillard

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Evaluating a constant expression in multiple precision with a guaranteed error bound

Sylvain Chevillard

N° 7443

Novembre 2010

_ Algorithms, Certification, and Cryptography _

ISSN 0249-6399 ISRN INRIA/RR--7443--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Evaluating a constant expression in multiple precision with a guaranteed error bound

Sylvain Chevillard

Theme : Algorithms, Certification, and Cryptography
Équipe-Projet Caramel

Rapport de recherche n° 7443 — Novembre 2010 — 16 pages

Abstract: The evaluation of special functions often involves the evaluation of numerical constants. When the precision of the evaluation is known in advance (e.g., when developing libms) these constants are simply precomputed once for a while. In contrast, when the precision is dynamically chosen by the user (e.g., in multiple precision libraries), the constants must be evaluated on the fly at the required precision and with a rigorous error bound.

Often, such constants are easily obtained by means of formulas involving simple numbers and functions. In principle, it is not a difficult work to write multiple precision code for evaluating such formulas with a rigorous roundoff analysis: one only has to study how roundoff errors propagate through subexpressions. However, this work is painful and error-prone and it is difficult for a human being to be perfectly rigorous in this process. Moreover, the task quickly becomes impractical when the size of the formula grows. In this article, we present an algorithm that takes as input a constant formula and that automatically produces code for evaluating it in arbitrary precision with a rigorous error bound. It has been implemented in the Sollya free software tool and its behavior is illustrated on several examples.

Key-words: multiple precision, constant expression, rigorous error bounds, roundoff analysis, faithful rounding.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Évaluation d'expressions constantes en précision arbitraire avec bornes d'erreur garanties

Résumé : L'évaluation de fonction spéciales nécessite souvent d'évaluer certaines constantes. Lorsque la précision est connue à l'avance (par exemple, lorsqu'on développe une libm), ces constantes sont simplement précalculées une fois pour toutes. Mais lorsque la précision est fixée par l'utilisateur au moment de l'évaluation (comme c'est le cas pour les bibliothèques en précision arbitraire), les constantes doivent être évaluées à la volée à la précision demandée et en bornant rigoureusement les erreurs.

Souvent, ce genre de constantes est donné par des formules faisant intervenir des fonctions simples et des entiers. En principe, écrire du code en précision arbitraire pour évaluer ce genre de formule avec une analyse rigoureuse des erreurs d'arrondi n'est pas une tâche difficile. Il suffit d'étudier comment les erreurs d'arrondi se propagent à travers les sous-expressions. Cependant, ce travail est ingrat, délicat, et il est difficile pour un être humain de rester parfaitement rigoureux. De plus, la tâche devient vite inabordable lorsque la formule grossit. Dans cet article, nous présentons un algorithme qui prend en entrée une formule constante et qui produit automatiquement du code pour l'évaluer en précision arbitraire et avec une borne d'erreur rigoureuse. Nous l'avons implémenté de façon expérimentale dans l'outil libre Sollya, et nous illustrons son comportement sur plusieurs exemples.

Mots-clés : précision arbitraire, expression constante, borne d'erreur rigoureuse, analyse d'erreur, arrondi fidèle.

1 Introduction

In the past twenty years, several libraries have been developed for performing floating-point computations with higher precisions than the usual *single* or *double* precisions. Such libraries are often called *multiple precision* or *multi-precision* libraries: one may cite (among others) ZMLIB [13], the `Arprec` C++ [1] or NTL libraries, the GNU `MPFR` library [4], or the Python mpmath library[1]. Such libraries turned to be very useful when double precision does not provide a satisfying accuracy: either because one needs more than 53 significant bits or because one wants to solve a very ill-conditioned problem. They are also used as a building block for higher level libraries such as libraries implementing real arithmetic [16, 11].

When one develops a multiprecision library, one often needs to write code for the multiprecision evaluation of expressions defining constants. As an example, π can be evaluated by means of Ramanujan's formula:[2]

$$\pi = \frac{9801}{2\sqrt{2}\displaystyle\sum_{k=0}^{\infty} \frac{(4k)!\,(1103+26390k)}{(k!)^4\,396^{4k}}}. \tag{1}$$

Important constants also appear in Taylor series, used to evaluate functions. For instance consider the Airy Ai function. Its value at 0 is $\mathrm{Ai}(0) = 3^{-2/3}\,\Gamma(2/3)^{-1}$ and it is the first coefficient of its Taylor series. The value $\Gamma(2/3)$ can be efficiently evaluated thanks to Euler's reflection formula $\Gamma(x)\Gamma(1-x) = \pi/\sin(\pi x)$ and thanks to a series due to Brown[3] for evaluating $\Gamma(1/3)$:

$$\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(-1)^k\,(6k)!}{(k!)^3\,(3k)!\,12288000^k}\right)^{1/6}. \tag{2}$$

In these formulas, the series are fairly easy to implement. However, once the series is correctly evaluated, some work remains for computing the value π or $\mathrm{Ai}(0)$. This work is not difficult in theory, but it is painful and error-prone: in order to perform a completely rigorous roundoff analysis, each subexpression must be rigorously bounded (above and below) and second order error terms cannot be neglected. Writing a complete proof is a long, boring and very error-prone process, and it is unlikely that anyone will ever read it carefully.

In this article, we propose an algorithm for automatically implementing a constant expression in multiprecision. It takes as input any expression formed from numerical constants and from predefined functions. The algorithm produces multiprecision code; this code allows for evaluating the expression with any desired final accuracy.

The article is organized as follows: in the next section, we formally state the problem that we intend to solve. In Section 3, we present previous works related to this subject. Section 4 begins with a small reminder of roundoff analysis and defines notations used in the sequel; the rest of the section is devoted to the presentation and the proof of our algorithm. Finally, in Section 5, we illustrate the behavior of our algorithm on several examples.

[1] `http://code.google.com/p/mpmath/`

[2] This is not the best known formula for evaluating π but we give it as an example of the kind of expressions we are interested with.

[3] See `http://www.iamned.com/math/`

2 Statement of the problem

We begin by formally defining what we mean by *constant expression*. We denote by $\mathcal{C} \subset \mathbb{R}$ a set of "predefined constants". It represents the set of built-in constants, that do not need to be evaluated by composition of operations. For instance, it is reasonable to assume that $\mathbb{Z} \subseteq \mathcal{C}$. It is also possible to add other particular constants in $\mathcal{C}$ such as, e.g., π, γ, etc. Moreover we will denote by $\mathcal{B}$ the set of "basic functions", i.e., the set of unary functions used to build expressions. We additionally require that the functions in $\mathcal{B}$ be differentiable. The set $\mathcal{E}$ of expressions that we consider is the smallest set containing $\mathcal{C}$ and such that:

- If $e_1, e_2 \in \mathcal{E}$, then $e_1 \diamond e_2 \in \mathcal{E}$, where $\diamond \in \{+, -, \div, \times\}$.
- If $e \in \mathcal{E}$ and if $f \in \mathcal{B}$, $f(e) \in \mathcal{E}$.

Remark: strictly speaking, there is a difference between an expression e and the real value that it represents. For instance "$4 \times \arctan(1)$" and "$2 \times \arccos(0)$" are two distinct expressions, and they both represent the same value π. These expressions correspond to different ways of evaluating the same constant, and it is important to consider them as distinct. In general, when speaking of e, it is clear from the context whether the expression e or its value should be considered. Hence, when no confusion is possible, we do not explicitly make the distinction.

We can now rigorously formulate our problem:

> *Given a constant expression $e \in \mathcal{E}$, generate a program* `eval_e` *for the multiprecision evaluation of e. More formally, for any integer $p \geq 2$,* `eval_e`(p) *shall return a value y such that $|y-e| \leq 2^{1-p}\,|e|$.*

In order to write the program `eval_e`, we need a multiprecision floating-point environment, where operations are performed with a rigorous control of rounding errors. As a matter of course, the environment shall provide a way of evaluating all the predefined constants $\mathcal{C}$ and all the basic functions $f \in \mathcal{B}$ with an arbitrary and guaranteed relative error. In our current implementation, the code generated by our algorithm uses the `MPFR` library. Our current implementation neglects the possibility of underflows and overflows. As a matter of fact, the exponent range of `MPFR` is very large and can be considered as infinite in practice.

The algorithm itself needs a multiprecision interval arithmetic environment: in such an environment, variables are intervals whose bounds are floating-point numbers with any user-defined precision. The result of the evaluation of a function $f : \mathbb{R}^n \to \mathbb{R}$ on intervals $x_1, \ldots, x_n$ is an interval y such that the exact image $J = f(x_1, \ldots, x_n)$ satisfies $J \subseteq y$. We also suppose that the library is asymptotically exact, i.e., when the precision tends to infinity, y tends to J for every function $f \in \mathcal{B}$ and every operation $\diamond \in \{+, -, \div, \times\}$.

In our current implementation, we use the `MPFI` library. In this case, among other functions, $\mathcal{B}$ contains the trigonometric functions and their reciprocals, the hyperbolic functions and their reciprocals, the exponential function, logarithms in several bases, and for any integer n, the n-th root and $x \mapsto x^n$.

Our algorithm accepts any constant expression of $\mathcal{E}$, but it may abort if the value of a subexpression is exactly 0. This comes from the fact that the algorithm tries to evaluate expressions with a bounded *relative error*: hence, when a subexpression exactly evaluates to 0, the algorithm would have to actually *prove* it and return 0. Unfortunately, proving that an expression is exactly 0 is a difficult problem and algorithms for this purpose generally rely on number theory conjectures [12]. It would be out of our scope to implement such algorithms. Anyway, we consider this limitation as minor for two reasons:

- First, our algorithm always detects the problem and aborts, explicitly giving the subexpression that seems to evaluate to 0. This allows the users to investigate the problem and manually replace the subexpression by 0 if they manage to prove the equality. If, on the contrary, the users manage to prove that the subexpression is not zero, they can re-run the algorithm using a higher precision in interval arithmetic computations: our algorithm will eventually succeed, because the interval arithmetic library is asymptotically exact.
- Second, the primary purpose of our algorithm is only to help people who write multiprecision code for evaluating constants: if a subexpression can be simplified to 0, they would probably be happy that the algorithm warns them and allows them to simplify their formula.

3 Previous Work

Several methods can be used for evaluating an expression in multiple precision. The most generic one is the so-called *exact real arithmetic* [6]. Real arithmetic simulates computations with real numbers, as if they were exactly known (with infinite precision). In this paradigm, each number is represented as a black-box that provides approximations with any desired accuracy. When an operation is performed (e.g., addition, multiplication, evaluation of a function, etc.), the precision of intermediate computations is automatically adjusted, in order to provide the required accuracy.

The libraries of exact real arithmetic are too general for our purpose: they are designed to evaluate any function given by an expression, in particular variables are allowed. Since the order of magnitude of variables is not known in advance, the choice of the intermediate precisions is necessarily made at run-time. Fairly good strategies have been proposed [15] for which optimality can be proven [8] *provided that the precisions are chosen at run-time*. In contrast, we deliberately limit ourself to constant expressions: hence everything is known *a priori* and the choice of intermediate precisions can be made when generating the code, which will hence be more efficient.

Tools have been designed for studying the effects of rounding errors in a code. In Mathematica, the so-called *significance arithmetic* [14] is used to track the propagation of errors through algorithms. This method, though useful, is not rigorous because it only considers first-order errors and neglects higher-order errors. Krämer proposed an algorithm that automatically and rigorously bounds the total roundoff error of a numerical code [9, 10]. The algorithm that we present in the following has similarities with Krämer's work, however they differ by two points. First, Krämer supposes that the same precision is used

through the whole computation. We do not have such an assumption: actually, our algorithm chooses a suitable precision independently for each subexpression, trying to minimize the necessary precision at each step. Second, Krämer's algorithm allows for variables in the code and may hence largely overestimate the total error. In contrast, we can take advantage of the fact that we consider only constant expressions. Finally, Gappa [3] is a powerful tool able to bound the total roundoff error in a code. Moreover it generates a formal proof for the computed bound which offers a strong guarantee. It is able to handle codes involving several precisions. However, these precisions must be statically known. In our context, the precision is a variable and will be chosen by the user at runtime. Moreover, Gappa is designed to analyze existing code, whereas we are interested in directly generating code with well-chosen intermediate precisions. Thus Gappa is not well-suited for us.

4 Description of the algorithm

4.1 Reminder of roundoff analysis

For proving the correctness of our algorithm, we have to study how roundoff errors propagate through the execution of the generated program `eval_e`. This is a classical topic, very well presented in [7]. We recall a few important facts without demonstration.

Definition 1. *If $p \geq 2$ denotes the current precision, the quantity $u_p = 2^{1-p}$ is called the* unit roundoff.
If $x \in \mathbb{R}$, we denote by $\diamond(x)$ a faithful rounding of x (i.e., x itself if x is a floating-point number and any of the two floating-point numbers enclosing x otherwise).

Proposition 1. *For any $x \in \mathbb{R}$, there exists $\delta \in \mathbb{R}$, $|\delta| \leq u_p$ such that $\diamond(x) = x\,(1+\delta)$.*

The relative error counter is a convenient notation for representing the accumulation of errors through divisions and multiplications: we write $\widehat{z} = z\,\langle k \rangle$, meaning that $\widehat{z}$ is an approximate value of z, obtained by k successive multiplications or divisions. More formally, we write $\widehat{z} = z\,\langle k \rangle$ if

$$\exists \delta_1, \ldots, \delta_k \in \mathbb{R},\ s_1, \ldots, s_k \in \{-1, 1\}, \text{ such that } \widehat{z} = z \prod_{i=1}^{k} (1+\delta_i)^{s_i} \quad \text{with } |\delta_i| \leq u_p.$$

We now need a proposition to bound the error corresponding to a given value of the error counter:

Proposition 2. *Let $z \in \mathbb{R}$ and let $\widehat{z}$ be a floating-point number. We suppose that $k \in \mathbb{N}$ satisfies $ku_p \leq 1/2$ and that we can write $\widehat{z} = z\,\langle k \rangle$. Then*

$$\exists \theta_k \in \mathbb{R},\ \textit{such that } \widehat{z} = z(1+\theta_k) \quad \textit{with } |\theta_k| \leq 2ku_p.$$

Definition 2. *For $x \in \mathbb{R} - \{0\}$, we define* $\mathrm{EXP}(x) = 1 + \lfloor \log_2 |x| \rfloor$. *In other words* $\mathrm{EXP}(x)$ *is the unique integer E such that $2^{E-1} \leq |x| < 2^E$. By extension, we let* $\mathrm{EXP}(0) = -\infty$.
For an interval $[a, b]$, we define $\mathrm{MAXEXP}([a, b]) = \max_{x \in [a,\, b]} \mathrm{EXP}(x)$ *and* MINEXP *accordingly.*

4.2 General description of the algorithm

Our algorithm has the following signature: `implement_constant`(e, `name`). Here $e \in \mathcal{E}$ denotes a constant expression and `name` is a string containing the desired name for the output program (for instance `name`="`eval_e`"). The algorithm produces the code of a procedure `eval_e(prec)` that computes an approximate value of e with relative error smaller than $2^{1-\texttt{prec}}$. In order to generate the core of `eval_e`, we use an auxiliary algorithm `implementer`(`var_name`, e, p). The argument `var_name` contains the name of the variable where the result should be stored, and the argument $p \in \mathbb{Z}$ indicates that the result should be an approximation with relative error smaller than $2^{1-p-\texttt{prec}}$ (roughly speaking, it means that the result is computed with p guard bits; however, p is allowed to be negative). Formally, the correction property of `implementer` is the following:

Proposition 3. *A call to `implementer`(`var_name`, e, p) generates code. This code depends on a formal parameter `prec`; when run, it stores in variable `var_name` an approximate value $\widehat{e}$ of e such that*

$$|\widehat{e} - e| \leq 2^{1-\texttt{prec}-p}\,|e|.$$

Hence, the core of `eval_e` is generated by a call to `implementer`("y", e, 0) immediately followed by the instruction "`return y;`". We describe the algorithm `implementer` and prove its correction in the next sections. It proceeds recursively on the structure of e: we distinguish three cases, whether e is a multiplication/division, an addition/subtraction, or a basic function $f \in \mathcal{B}$. Accordingly, the proof of correctness is a structural induction on e.

4.3 Case of a multiplication/division

The case when e is a multiplication/division is the most simple because (roughly) relative errors are added when performing a multiplication/division. If e has the form $e = e' \times e''$ or $e = e' \div e''$, there is a unique way of rewriting e as a maximal product, i.e., with the form

$$e = \frac{e_1 \times \cdots \times e_n}{f_1 \times \cdots \times f_m},$$

where neither the e_i nor the f_j are multiplications/divisions. By recursive calls, we can evaluate the e_i (resp. f_j) with a controlled relative error δ_i (resp. ε_j). When multiplying and dividing the e_i and f_j together, each operation leads to a relative error γ_k directly controlled by the precision used for performing the operation. The overall relative error is henceforth approximately $\left(\sum_{i=1}^{n} \delta_i\right) + \left(\sum_{j=1}^{m} \varepsilon_j\right) + \left(\sum_{k=1}^{n+m-1} \gamma_k\right)$ and we want to ensure that it stays smaller than $2^{1-\texttt{prec}-p}$.

Optimally choosing the δ_i, ε_j and γ_k appears to be a subtle problem. Suppose for instance that e_1 is much more difficult to evaluate than the others e_i and f_j. Then, one should choose $\delta_1 \simeq 2^{1-\texttt{prec}-p}$ and the other error terms very small compared to δ_1. More generally, the idea would be to give a weight to each expression e_i and f_j, depending on the hardness of their evaluation at a given precision. Unfortunately, estimating this hardness seems difficult in practice because it depends (in a complex way) on the underlying multiprecision library and possibly also on the precision $\texttt{prec} + p$. Van der Hoeven [15] proposed that

the weight of an expression is defined by the number of operations in the expression. This allows for a good distribution of the errors when the expression (seen as a tree) is ill-balanced. However, this does not take into account the fact that operations have different practical complexities.

In our current implementation, we chose to apply a simpler strategy. We simply make all error terms of a product nearly equal, independently of the respective sizes of the subexpressions e_i and f_j: we take $\delta_i = \varepsilon_j = \gamma_k \simeq 2^{1-\mathtt{prec}-p}/(2(n+m))$ for all i, j, k. More formally, this leads to the algorithm described in Algorithm 1. We now rigorously prove its correctness.

Algorithm 1: `implementer`: case of a multiplication/division.

Input: `var_name`, $e_1, \ldots, e_n, f_1, \ldots, f_m, p$
Output: code for evaluating the expression $e = (e_1 \cdots e_n)/(f_1 \cdots f_m)$

1 Output code for the test: "if `prec` $+ p \leq 1$ then {`var_name` $\leftarrow 0$; return}";
2 $r \leftarrow \lceil \log_2(m+n) \rceil$;
3 let $a_1, \ldots, a_n, b_1, \ldots, b_m$, `tmp` be fresh names;
4 For each $i \in \{1 \ldots n\}$, recursively call `implementer`("a_i", e_i, $p + r + 2$);
5 For each $j \in \{1 \ldots m\}$, recursively call `implementer`("b_j", f_j, $p + r + 2$);
6 Output code for setting global precision to `prec` $+ p + r + 2$;
7 Output code for computing `var_name` $\leftarrow a_1 \times \cdots \times a_n$;
8 Output code for computing `tmp` $\leftarrow b_1 \times \cdots \times b_m$;
9 Output code for computing `var_name` $\leftarrow$ `var_name` $\div$ `tmp`;

Proof of correctness: in the following, a_i (resp. b_j, resp. `var_name`) represents the approximated value of e_i (resp. f_j, resp. e) computed during an execution of the generated program.

If `prec` $+ p \leq 1$, we have `var_name` $= 0$, hence $|\mathtt{var\_name} - e| = |e| \leq 2^{1-\mathtt{prec}-p}\,|e|$, which proves the correctness. Thus, for now on, we suppose that `prec` $+ p \geq 2$. By recurrence hypothesis, we can write $a_i = e_i\,\langle 1\rangle$ for each i and $b_j = f_j\,\langle 1\rangle$ for each j, assuming a global precision of `prec` $+ p + r + 2$. When computing `tmp` and `var_name`, the multiplications are faithfully rounded; hence, for each multiplication $z = \widehat{x} \times \widehat{y}$, the computed value $\widehat{z}$ satisfies $\widehat{z} = \widehat{x}\,\widehat{y}\,\langle 1\rangle$. The same holds for the final division. Thus, eventually we have

$$\mathtt{var\_name} = e\,\langle 2(n+m) - 1\rangle\,.$$

Using Proposition 2, we see that $|\mathtt{var\_name} - e| \leq e\,(1+\theta)$ with $|\theta| \leq (4(n+m)-2)\cdot 2^{1-\mathtt{prec}-p-r-2}$. By definition $2^{-r} \leq 1/(m+n)$, hence finally $|\theta| \leq 2^{1-\mathtt{prec}-p}$.

□

4.4 Case of an addition/subtraction

In the case when $e = e' + e''$ or $e = e' - e''$, we do the same as what we do for the multiplications/divisions: we look for the maximal sum $e = e_1 + \cdots + e_n$ (where none of the e_i is an addition/subtraction, or the negation of an addition/subtraction) and we equally distribute the errors between terms.

For a sum, it is more natural to consider absolute errors. If the e_i are evaluated with absolute errors δ_i and if the successive additions lead to absolute

errors γ_k, the overall absolute error is given by $\left(\sum_{i=1}^{n} \delta_i\right) + \left(\sum_{k=1}^{n-1} \gamma_k\right)$ and we want to keep it smaller than the absolute error $2^{1-\mathtt{prec}-p}\,|e|$. Again, a rule of thumb indicates that we should take $\delta_i = \gamma_k \simeq 2^{1-\mathtt{prec}-p}\,|e|/(2n-1)$.

However, contrary to the case of a multiplication/division, γ_k does not depend only on the precision used to perform the corresponding addition but also on the order of magnitude of both operands. The practical consequence of this remark is the following: the error terms γ_k depend on the order the operations are performed. It is well known that choosing an optimal order is a NP-hard problem [7, Chap. 4.2].

Thus, we do not try to find an optimal order of summation: we let the user choose the order of summation and simply follow the structure of the expression tree e. In practice, if $e = e' \pm e''$, we determine the number n' of terms in the maximal sum corresponding to e': $e' = e'_1 \pm \cdots \pm e'_{n'}$ and we define n'' accordingly for e''. Then, we evaluate e' with an absolute error smaller than $2^{1-\mathtt{prec}-p}\,|e|\,n'/(n'+n''+1)$ and we evaluate e'' with an absolute error smaller than $2^{1-\mathtt{prec}-p}\,|e|\,n''/(n'+n''+1)$. Finally, we perform the addition/subtraction with an absolute error smaller than $2^{1-\mathtt{prec}-p}\,|e|/(n'+n''+1)$. This strategy exactly corresponds to equally balancing all error terms in the sum $e'_1 + \cdots + e'_{n'} + e''_1 + \cdots e''_{n''}$. Counting the number of terms in the maximal sum of an expression e is performed by means of a procedure `summation_weight(e)`. The formal algorithm is summed up in Algorithm 2.

Algorithm 2: `implementer`: case of an addition/subtraction.

Input: `var_name`, e_1, e_2, p
Output: code for evaluating the expression $e = e_1 \pm e_2$

1 $\mathbf{u_1} \leftarrow e_1$; $\mathbf{u_2} \leftarrow e_2$ evaluated by interval arithmetic;
2 $\mathbf{v} \leftarrow \mathbf{u_1} \pm \mathbf{u_2}$ evaluated by interval arithmetic;
3 **if** $0 \in \mathbf{v}$ **then abort**;
4 $n_1 \leftarrow$ `summation_weight`(e_1); $n_2 \leftarrow$ `summation_weight`(e_2); $n \leftarrow n_1 + n_2 + 1$;
5 $E_1 \leftarrow \mathrm{MINEXP}(\frac{n_1\,\mathbf{v}}{n\,\mathbf{u_1}})$; $E_2 \leftarrow \mathrm{MINEXP}(\frac{n_2\,\mathbf{v}}{n\,\mathbf{u_2}})$;
6 $E \leftarrow \mathrm{MINEXP}(\frac{\mathbf{v}}{n\,(|\mathbf{u_1}|+|\mathbf{u_2}|)})$;
7 let a_1 and a_2 be fresh names;
8 Output code for the following test:
"if `prec` $+\, p + 1 - E_1 \leq 1$ then $a_1 \leftarrow 0$ else"
`implementer`("a_1", e_1, $p + 1 - E_1$);
9 Output code for the following test:
"if `prec` $+\, p + 1 - E_2 \leq 1$ then $a_2 \leftarrow 0$ else"
`implementer`("a_2", e_2, $p + 1 - E_2$);
10 Output code for setting global precision to `prec` $+\, p + 2 - E$;
11 Output code for computing `var_name` $\leftarrow a_1 \pm a_2$;

Proof of correctness: by definition of E_1, we have $E_1 \leq \mathrm{EXP}(\frac{n_1\,e}{n\,e_1})$. A similar identity holds for E_2 and E. Hence

$$\left|\frac{e_1}{e}\right| \leq \frac{n_1}{n} \cdot 2^{1-E_1}, \quad \left|\frac{e_2}{e}\right| \leq \frac{n_2}{n} \cdot 2^{1-E_2}, \quad \text{and} \quad \frac{|e_1| + |e_2|}{|e|} \leq \frac{1}{n} \cdot 2^{1-E}. \tag{3}$$

We consider an execution of the generated program. For both $i = 1$ and $i = 2$, the definition of a_i depends on the test $\texttt{prec} + p + 1 - E_i \leq 1$ but, in any case, we can write $a_i = e_i\,(1 + \varepsilon_i)$ with $|\varepsilon_i| \leq 2^{1-(\texttt{prec}+p+1-E_i)}$: if $\texttt{prec} + p + 1 - E_i > 1$, the result is directly given by the recurrence hypothesis on `implementer`; if, on the contrary, $\texttt{prec}+p+1-E_i \leq 1$, we have $a_i = 0$ and we can write $a_i = e_i\,(1+\varepsilon_i)$ where $\varepsilon_i = -1$, which satisfies $|\varepsilon_i| \leq 2^{1-(\texttt{prec}+p+1-E_i)}$.

We now write the error between `var_name` and e: since the final addition/subtraction is faithfully rounded, we have $\texttt{var\_name} = (a_1 \pm a_2)\,(1 + \varepsilon)$ with $|\varepsilon| \leq 2^{1-(\texttt{prec}+p+2-E)}$. Hence:

$$\left|\frac{\texttt{var\_name} - e}{e}\right| \leq \left|\frac{e_1}{e}\right| 2^{1-(\texttt{prec}+p+1-E_1)} + \left|\frac{e_2}{e}\right| 2^{1-(\texttt{prec}+p+1-E_2)} + \left|\frac{a_1 \pm a_2}{e}\right| 2^{1-(\texttt{prec}+p+2-E)}.$$

We can conclude using (3), provided that we show that $|a_1 \pm a_2| \leq 2(|e_1| + |e_2|)$. By the triangle inequality: $|a_1 \pm a_2| \leq |a_1| + |a_2|$, thus it suffices to prove that $|a_i| \leq 2|e_i|$. If $a_i = 0$, it is obviously true. Otherwise, $\texttt{prec} + p + 1 - E_i > 1$, thus $|\varepsilon_i| \leq 1$ and finally $|a_i| = |e_i\,(1 + \varepsilon_i)| \leq 2|e_i|$.

□

4.5 Case of a basic function

We now study the case when $e = f(e_1)$ where $f \in \mathcal{B}$ is a basic function. The idea of the algorithm is as follows: a recursive call `implementer`(some fresh name "y", e_1, q) allows us to evaluate e_1 as accurately as desired by adjusting q. We denote by h the corresponding absolute error: $y = e_1 + h$. By recurrence hypothesis, $|h| \leq 2^{1-\texttt{prec}-q}\,|e_1|$.

The function f possibly amplifies or contracts this error. Namely, using the mean value theorem,

$$f(y) = f(e_1) + h\,f'(\xi),$$

where ξ lies between y and e_1. Hence, roughly speaking, the final relative error is $h\,f'(\xi)/f(e_1)$, which is close to $h\,f'(e_1)/f(e_1)$.

In principle, this gives us a suitable value for q: it suffices that $|h| \leq |f(e_1)/f'(e_1)|\,2^{1-\texttt{prec}-p}$. Hence, it suffices to choose

$$q \geq p + \mathrm{MAXEXP}(e_1\,f'(e_1)/f(e_1)).$$

This gives us the idea of the algorithm. However, in order to be perfectly rigorous, we have to take into account the fact that ξ is not exactly equal to e_1. In order to handle this problem, we introduce a retro-action loop for choosing q, and we use interval arithmetic for ensuring safety. Moreover, we must take into account the rounding that happens when evaluating f itself. This leads to the algorithm summed up in Algorithm 3.

Proof of correctness: we shall prove that the algorithm terminates and that it is correct. The case when $\texttt{prec} + p \leq 1$ is obvious. So, we suppose in the following that $\texttt{prec} + p \geq 2$. Obviously r strictly increases during the loop. Moreover, when r increases, the width of the interval $[e_1\,(1 - 2^{-p-r}),\ e_1\,(1 +$

Algorithm 3: `implementer`: case of a basic function.

Input: `var_name`, f, e_1, p
Output: code for evaluating the expression $e = f(e_1)$

```
1  Output code for the test: "if prec + p ≤ 1 then {var_name ← 0; return}";
2  u ← f(e1) evaluated by interval arithmetic;
3  if 0 ∈ u then abort;
4  u ← e1/u;
5  v ← u · f'(e1) evaluated by interval arithmetic;
6  r ← 2 + MAXEXP(v);
7  v ← u · f'([e1 (1 − 2^(−p−r)), e1 (1 + 2^(−p−r))] evaluated by interval
   arithmetic;
8  while r < 2 + MAXEXP(v) do
9  |  r ← r + 1;
10 |  v ← u · f'([e1 (1 − 2^(−p−r)), e1 (1 + 2^(−p−r))] evaluated by interval
   |  arithmetic;
11 end
12 let y be a fresh name;
13 implementer("y", e1, p + r);
14 Output code for setting global precision to prec + p + 2;
15 Output code for computing var_name ← f(y);
```

$2^{-p-r})]$ decreases. Hence, during the loop, the interval $\mathbf{v}$ can only shrink and MAXEXP($\mathbf{v}$) can only decrease (or stay equal). This proves that the loop eventually terminates.

By construction, after the loop, $r \geq 2 + \mathrm{MAXEXP}(\mathbf{v})$. Moreover, by recurrence hypothesis, the generated code computes an approximation y of e_1 satisfying

$$|y - e_1| \leq 2^{1-\mathtt{prec}-p-r}\,|e_1|. \tag{4}$$

In particular, $|y - e_1| \leq 2^{-p-r}\,|e_1|$. Thus any value ξ between y and e_1 satisfies $\xi \in [e_1\,(1 - 2^{-p-r}),\, e_1\,(1 + 2^{-p-r})]$.

Moreover, using (4) and the mean value theorem, we have

$$f(y) - f(e_1) = f(e_1)\,\theta \quad \text{where} \quad |\theta| \leq 2^{1-\mathtt{prec}-p-r} \cdot |e_1\,f'(\xi)/f(e_1)|.$$

Since ξ lies between y and e_1, it holds that $e_1\,f'(\xi)/f(e_1) \in \mathbf{v}$. Hence $|\theta| \leq 2^{1-\mathtt{prec}-p-r+\mathrm{MAXEXP}(\mathbf{v})} \leq 2^{1-\mathtt{prec}-p-2}$. In conclusion, assuming a global precision of $\mathtt{prec}+p+2$, we can write $f(y) = f(e_1)\,\langle 1\rangle$. Moreover, since the final evaluation is also performed in precision $\mathtt{prec}+p+2$, we have $\mathtt{var\_name} = f(y)\,\langle 1\rangle$. We conclude using Proposition 2 on $\mathtt{var\_name} = f(e_1)\,\langle 1\rangle\,\langle 1\rangle = f(e_1)\,\langle 2\rangle$.

□

5 Examples

We have implemented our algorithm in the Sollya[4] free software tool, as the experimental `implementconstant` command. We tested it on several expressions; each time, we ran the generated code and checked that the accuracy of

[4] See `http://sollya.gforge.inria.fr/`

its result was correct by comparing it with a high-precision interval evaluation of the expression.

As expected, the algorithm detects when a subexpression is an exact zero. For instance, when run on the expression

$$\sin(1) + \exp\left(\sqrt[3]{\sqrt[5]{32/5} - \sqrt[5]{27/5}} - \frac{1 + \sqrt[5]{3} - \sqrt[5]{9}}{\sqrt[5]{25}}\right),$$

our algorithm aborts with a message indicating that $\sqrt[3]{\sqrt[5]{32/5} - \sqrt[5]{27/5}} - \frac{1+\sqrt[5]{3}-\sqrt[5]{9}}{\sqrt[5]{25}}$ is probably exactly zero.

5.1 First example

Our first example comes from the competition organized at the CCA 2000 conference for testing the efficiency and correctness of multiprecision libraries [2]. The expression to evaluate is $\log(1 + \log(1 + \log(1 + \log(1 + \exp(1)))))$. There is no numerical difficulty in the evaluation of this expression, but it illustrates well how the algorithm tracks the propagation of roundoff errors and adapts the intermediate precisions. The output of the algorithm, as a pseudo-code, is represented in Algorithm 4; the comments indicate what precision is used to perform each operation.

Algorithm 4: Pseudo-code produced by our algorithm on the first example.

Input: prec

$y \leftarrow \exp(1)$; `/* prec+19 */`
$y \leftarrow 1 + y$; `/* prec+18 */`
$y \leftarrow \log(y)$; `/* prec+15 */`
$y \leftarrow 1 + y$; `/* prec+15 */`
$y \leftarrow \log(y)$; `/* prec+11 */`
$y \leftarrow 1 + y$; `/* prec+11 */`
$y \leftarrow \log(y)$; `/* prec+7  */`
$y \leftarrow 1 + y$; `/* prec+7  */`
$y \leftarrow \log(y)$; `/* prec+2  */`
return y

We ran the algorithm with `prec` varying from 2 to 100 000 and considered the actual relative error $\varepsilon(\texttt{prec})$. By construction, this error has the form $\alpha(\texttt{prec})\, 2^{1-\texttt{prec}}$ with $|\alpha(\texttt{prec})| \leq 1$. As a matter of course, since the error analysis is pessimistic $|\alpha|$ can be much smaller than 1 in practice. For instance when $|\alpha| \simeq 1/2^n$, it means that we have n more correct bits than expected. Reciprocally, it means that we might evaluate the expression using lower precisions and still get an accurate enough result. On this example, we observed that the maximum of $|\alpha|$ over the considered range of precision was roughly $1/2^{3.5}$. This indicates that we were not too pessimistic and that we could probably not improve the choice of precisions by more than one or two bits.

5.2 Second example

The second example is an expression constructed to illustrate the fact that, even for evaluating some fairly simple expressions, double precision may be insufficient. It has been presented by Ghazi et al. [5] to demonstrate the usefulness of multiprecision libraries. The expression to evaluate is

$$e = 173746 \sin(1e22) + 94228 \log(171/10) - 78487 \exp(42/100).$$

Ghazi et al. explain that, when double precision is used to evaluate e, the computed result is $2.91\ldots\text{e}{-11}$; if `long double` are used (i.e. 64-bit significand), the computed result is $-1.31\ldots\text{e}{-12}$; and actually, the correct result is $-1.34\ldots\text{e}{-12}$.

The output of our algorithm is represented in Algorithm 5. A quick look to the code immediately shows that the expression is ill-conditioned for an evaluation in double precision: to obtain one bit of accuracy, it is necessary to perform the last operation in precision at least 62. Using `long double`s, one should not expect more than roughly 3 correct bits. But we can also see that, if the expression is naively evaluated by performing all the operations with a given precision $p \geq 61$, one roughly expects to get $p - 61$ correct bits in the result.

Algorithm 5: Pseudo-code produced by our algorithm on the second example.

Input: prec

$t_1 \leftarrow \sin(1e22)$; `/* prec+66 */`
$t_2 \leftarrow 173746\, t_1$; `/* prec+64 */`

$t_3 \leftarrow 171/10$; `/* prec+70 */`
$t_4 \leftarrow \log(t_3)$; `/* prec+67 */`
$t_5 \leftarrow 94228\, t_4$; `/* prec+65 */`
$t_6 \leftarrow t_2 + t_5$; `/* prec+63 */`

$t_7 \leftarrow 42/100$; `/* prec+68 */`
$t_8 \leftarrow \exp(t_7)$; `/* prec+65 */`
$t_9 \leftarrow 78487\, t_8$; `/* prec+63 */`
$y \leftarrow t_6 - t_9$; `/* prec+61 */`

return y

We ran the algorithm with `prec` varying from 2 to 100 000 and measured the corresponding $\alpha(\texttt{prec})$ as we did for the first example. Here, the maximum of $|\alpha|$ was roughly $1/2^{6.9}$. This indicates that we might improve our error analysis a little and save a few bits of precision in the computation.

5.3 Third example

Our last example comes from a real-life problem arising when one wants to implement the Airy Ai function in arbitrary precision by means of its Taylor series: as explained in the introduction, one needs to evaluate $\Gamma(1/3)$. One could compute an approximate value of $1/3$ and use a generic implementation

of the Γ function but it would be much less efficient than using a formula such as Equation (2) given on page 3.

Our algorithm is not able to handle series, but as discussed in Section 2, the sets of basic functions $\mathcal{B}$ and predefined constants $\mathcal{C}$ are arbitrary. Hence, our implementation provides an extension mechanism that allows one to add a new constant in $\mathcal{C}$. We define $\alpha = \sum_{k=0}^{+\infty} a_k$ where $(a_k)_{k\in\mathbb{N}}$ is defined by $a_0 = 1$ and

$$a_{k+1} = \frac{-(6k+1)\,(6k+2)\,(6k+3)\,(6k+4)\,(6k+5)\,(6k+6)}{(k+1)^3\,(3k+1)\,(3k+2)\,(3k+3)\,12288000}\,a_k.$$

Thus Equation (2) becomes $\Gamma(1/3) = (12\,\pi^4\,\alpha/\sqrt{10})^{1/6}$ and can be handled by our algorithm, provided that $\alpha \in \mathcal{C}$. In order to add α to the set $\mathcal{C}$, we only need to provide a procedure `eval_alpha(prec)` that returns an approximate value y of α such that $|y - \alpha| \leq 2^{1-\mathtt{prec}}\,\alpha$. Such a procedure is easy to design: a quick computation shows that $|a_{k+1}| \leq 9\,|a_k|/64000$ for all k. Hence the series converges at least as fast as the geometric series with ratio 9/64000 which remainder is easily bounded. This gives a suitable truncation rank in function of `prec`. Once the truncation rank is known, the sum is evaluated at a suitable precision p obtained by a straightforward roundoff analysis. Once $\alpha \in \mathcal{C}$, our algorithm easily generates code for $\Gamma(1/3)$ (see Algorithm 6).

Algorithm 6: Pseudo-code produced by our algorithm for evaluating $\Gamma(1/3)$ with Equation (2).

Input: prec

$t_1 \leftarrow \pi$; `/* prec + 11 */`
$t_2 \leftarrow t_1^4$; `/* prec + 7 */`
$t_3 \leftarrow$ `eval_alpha(`$\mathtt{prec}+5$`);`
$t_4 \leftarrow \sqrt{10}$; `/* prec + 7 */`
$t_5 \leftarrow 12\,t_2\,t_3/t_4$; `/* prec + 5 */`
$y \leftarrow \sqrt[6]{t_5}$; `/* prec + 2 */`
return y

We ran the algorithm with `prec` varying from 2 to 100 000. The maximum of $|\alpha|$ in this case was roughly $1/2^{3.4}$, which is close to being optimal.

6 Conclusion

We presented an algorithm that automatically generates code for evaluating constant expressions in multiple precision. In our current implementation we generate `MPFR` code, but it would be easy to generate code using other multiple precision libraries, provided that operations with mixed precisions are possible.

The primary goal of the algorithm is to help developers of multiple precision libraries in their work: when developing a function for such a library, it is often necessary to write code for evaluating constants. As an example, the developers of `MPFR` reported that they successfully used our algorithm when they developed the Ai function. As illustrated in Section 5.2, the algorithm may also serve as a way of understanding how rounding errors propagate in the floating-point evaluation of a constant expression and how many correct bits can be expected in the final result if the operations are all performed at a given precision.

Our algorithm could be improved in at least two ways. First, the expressions are currently represented as trees. An improvement would be to represent expressions as directed acyclic graphs, which would avoid recomputing subexpressions that have already been evaluated. Second, we could try to give realistic weights to expressions, that would be used for well-balancing errors in the evaluation of an expression.

Though we took great care in proving and implementing our algorithm, and though we tested it on many examples, it is always possible that a bug remains somewhere. Moreover, the roundoff analysis always considers the worst possible case and is henceforth very pessimistic: so a bug in the generated code would be very difficult to detect because the code will almost always give correct results. A solution consists in modifying our algorithm so that, for each instance, it generates both the code and a formal proof of its correctness. The formal proof can then be checked with an automatic proof checker, thus offering a high guarantee. Generating such a formal proof is an on-going work.

References

[1] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. Arprec: An arbitrary precision computation package. Software and documentation available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[2] J. Blanck. Exact Real Arithmetic Systems: Results of Competition. In J. Blanck, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 389–393, Heidelberg, Germany, 2001. Springer.

[3] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):1–20, 2010.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2), 2007.

[5] K. R. Ghazi, V. Lefèvre, P. Théveny, and P. Zimmermann. Why and How to Use Arbitrary Precision. *Computing in Science and Engineering*, 12(3):62–65, 2010.

[6] P. Gowland and D. Lester. A Survey of Exact Arithmetic Implementations. In J. Blanck, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 30–47, Heidelberg, Germany, 2001. Springer.

[7] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002.

[8] V. Kreinovich and S. Rump. Towards Optimal Use of Multi-Precision Arithmetic: A Remark. *Reliable Computing*, 12(5):365–369, 2006.

[9] W. Krämer. A priori Worst-Case Error Bounds for Floating-Point Computations. *IEEE Transactions on Computers*, 47(7):750–756, 1998.

[10] W. Krämer and A. Bantle. Automatic Forward Error Analysis for Floating Point Algorithms. *Reliable Computing*, 7(4):321–340, 2001.

[11] N. Müller. The iRRAM: Exact Arithmetic in C++. In J. Blanck, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 222–252, Heidelberg, Germany, 2001. Springer.

[12] D. Richardson. How to Recognize Zero. *Journal of Symbolic Computation*, 24(6):627–645, 1997.

[13] D. M. Smith. Algorithm 786: multiple-precision complex arithmetic and functions. *ACM Transactions on Mathematical Software (TOMS)*, 24(4):359–367, 1998.

[14] M. Sofroniou and G. Spaletta. Precise numerical computation. *Journal of Logic and Algebraic Programming*, 64(1):113–134, 2005.

[15] J. van der Hoeven. Computations with effective real numbers. *Theoretical Computer Science*, 351(1):52–60, 2006.

[16] J. van der Hoeven. Effective real numbers in Mmxlib. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 138–145, New York, NY, USA, 2006. ACM.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399