Automatic generation of code for the evaluation of constant expressions at any precision with a guaranteed error bound

Sylvain Chevillard
sylvain.chevillard@ens-lyon.org
INRIA Sophia-Antipolis Méditerranée — APICS Project-Team
2004 route des Lioles - BP 93 - 06902 Sophia Antipolis Cedex, FRANCE

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 and for all. 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.

Keywords: multiple precision, constant expression, rigorous error bounds, roundoff analysis, faithful rounding.

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 multiprecision libraries: one may cite (among others) ZMLIB\(^1\), the Arprec\(^2\) or NTL\(^2\) C++ libraries, the GNU MPFR library\(^3\), or the Python mpmath library\(^3\).

Such libraries turned out 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, \(\pi\) can be evaluated by means of Ramanujan’s formula:\(^4\)

\[
\pi = \frac{9801}{2\sqrt{2}} \sum_{k=0}^{\infty} \frac{(4k)! (1103 + 26390k)}{(k!)^4 3964k}. \tag{1}
\]

Important constants also appear in Taylor series, used to evaluate functions. For instance consider the Airy Ai function. Its value at 0 is \(\text{Ai}(0) = 3^{-2/3} \Gamma(2/3)^{-1}\) 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\(^5\) for evaluating \(\Gamma(1/3):\)

\[
\Gamma(1/3) = \left( \frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!}{(k!)^3 (3k)! 1228800k} \right)^{1/6}. \tag{2}
\]

In these formulas, we distinguish two problems: first, one must be able to evaluate the series with a guaranteed accuracy. Writing multiprecision code for the evaluation of a series is a fairly complicated task in general. However, the

\(^1\)http://crd.lbl.gov/~dhbailey/mpdist/
\(^2\)http://www.shoup.net/ntl/
\(^3\)http://code.google.com/p/mpmath/
\(^4\)This is not the best known formula for evaluating \(\pi\) but we give it as an example of the kind of expressions we are interested with.
\(^5\)See http://www.iamned.com/math/
series involved in formulas (1) and (2) are hypergeometric series (i.e., the ratio of two consecutive terms is a rational function of $k$). The best known algorithm for evaluating such series at a given accuracy is called binary splitting and it uses exact rational arithmetic \cite{[2]}. Hence, the only rounding issue is the choice of the truncation rank. We do not address this problem and assume that evaluating the series of formulas (1) and (2) is fairly easy.

Once one has code for correctly evaluating the series, some work remains for computing the value $\pi$ or $\text{Ai}(0)$. 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 finite expression formed from numerical constants and from predefined functions. The algorithm produces dynamic multiprecision code: this code takes as input a target accuracy and delivers an approximate value of the expression with this 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.

2. Statement of the problem

We begin by formally defining what we mean by finite constant expression. We denote by $C \subseteq \mathbb{R}$ 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 $\mathbb{Z} \subseteq C$. It is also possible to add other particular constants in $C$: for instance, one may include the value of a hypergeometric series, provided that we have a black-box for evaluating it. Our algorithm will simply see it as an atomic constant. Moreover we will denote by $B$ the set of “basic functions”, i.e., the set of unary functions used to build expressions. We additionally require that the functions in $B$ be differentiable. The set $E$ of expressions that we consider is the smallest set containing $C$ and such that for all $e, e' \in E$, all $\diamond \in \{+, - , \times , \div \}$ and all $f \in B$, $e \diamond e'$ and $f(e)$ are in $E$.

Formally, we represent an expression $e \in E$ as a tree where the leaves are elements of $C$ and the nodes are either a function of $B$ (with a single child) or an operator $\diamond$ (with exactly two children). We say that $e$ is a basic function (resp., an addition, subtraction, multiplication, or division) when the root of $e$ belongs to $B$ (resp., is $+, -, \times, \div$).

Remark: strictly speaking, there is a difference between an expression $e$ and the real value that it represents. For instance “$4 \times \text{arctan}(1)$” and “$2 \times \arccos(0)$” are two distinct expressions, and they both represent the same value $\pi$. 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 $e \in E$, generate a program $\text{eval}_e$ for the multiprecision evaluation of $e$. More formally, for any integer $p \geq 2$, $\text{eval}_e(p)$ shall return a value $y$ such that $|y - e| \leq 2^{1-p} |e|$.

In order to write the program $\text{eval}_e$, we need a multi-precision 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 $C$ and all the basic functions $f \in B$ with an arbitrary and guaranteed relative error. Moreover, since $\text{eval}_e$ produces dynamic multiprecision code, the environment shall support dynamic setting of the precision, i.e., for each operation performed in the multiprecision environment, we can dynamically set the precision. 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 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 $f \in B$ and every $\diamond \in \{+, - , \times , \div \}$.

In our current implementation, we use the MPFI library. In this case, among other functions, $B$ contains the trigonometric and 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 $E$, but it may abort if the real value of some non-trivial subexpression $e'$ is 0. Actually, our algorithm needs to find the order
of magnitude of $e'$: for this purpose, it evaluates $e'$ by interval arithmetic. When the real value of $e'$ is 0, the computed enclosure is generally not reduced to $[0; 0]$, and it does not allow the algorithm to deduce the order of magnitude of $e'$. The only way of doing it would be to actually prove that $e'$ is exactly 0. Unfortunately, this is a hard decision 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: indeed, our algorithm always detects the problem and aborts with a message indicating that it seems that $e' = 0$. The users can hence investigate the problem and manually replace $e'$ by 0 if they manage to prove the equality (and they would probably be happy that the algorithm indicated to them that unsuspected equality!) If, on the contrary, $e' \neq 0$, it suffices to 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.

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). 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. In order to evaluate an expression $e$ at some precision $\psi$, one has to choose intermediate precisions of the form $\psi + k_i$ for the evaluation of each subexpression $e_i$. The choice of the values $k_i$ is directly related to the order of magnitude of the involved subexpressions. Since the order of magnitude of variables is known only dynamically, the choice of the $k_i$ is necessarily made at run-time. For this general problem, fairly good strategies have been proposed [15] for which optimality can be proved [8]. In contrast, we deliberately limit ourselves to the more specific problem of constant expressions: hence the order of magnitude of each subexpression is known a priori and the choice of the $k_i$ 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 run-time. 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 $\text{eval\_e}$. This is a classical topic, well presented in [7]. We recall a few facts without proof.

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 $\hat{o}(x)$ 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 $\hat{o}(x) = x (1 + \delta)$.

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

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

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 $\hat{z}$ be a floating-point number. We suppose that $k \in \mathbb{N}$ satisfies $ku_p \leq 1/2$ and that we can write $\hat{z} = z(1 + \theta_k)$. Then
\[ \exists \theta_k \in \mathbb{R}, \text{ such that } \hat{z} = z(1 + \theta_k) \text{ with } |\theta_k| \leq 2ku_p. \]

Definition 2. For $x \in \mathbb{R} - \{0\}$, we define $\exp(x) = 1 + |\log_2 |x||$. In other words $\exp(x)$ is the unique integer $E$ such that $2^{E-1} \leq |x| < 2^E$. By extension, we let $\exp(0) = -\infty$.

For an interval $[a, b]$, we define $\text{MAXEXP}([a, b]) = \max_{x \in [a, b]} \exp(x)$ and $\text{MINEXP}$ accordingly.

4.2. General description of the algorithm

Our algorithm has the following signature: implement constant($e$, name). Here $e \in E$ 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 $2^{1-\text{prec}}$. 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 $2^{1-p-\text{prec}}$ (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 $\hat{e}$ of $e$ such that $|\hat{e} - e| \leq 2^{1-\text{prec}-p} |e|$.

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 correctness 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 $e = e' \div e''$, we first find all factors of $e$, i.e. we find the subexpressions $e_1, \ldots, e_n, f_1, \ldots, f_m$ such that neither the $e_i$ nor the $f_j$ are multiplications or divisions and such that $e$ and $(e_1 \cdots e_n)/(f_1 \cdots f_m)$ represent the same real value. The subexpressions $e_i$ and $f_j$ are unique up to a permutation.

By recursive calls, we can evaluate the $e_i$ (resp. $f_j$) with a controlled relative error $\delta_i$ (resp. $\varepsilon_j$). When multiplying and dividing the $e_i$ and $f_j$ together, each operation leads to a relative error $\gamma_k$ directly controlled by the precision used for performing the operation. The overall relative error is henceforth approximately $(\sum_{i=1}^n \delta_i) + (\sum_{j=1}^m \varepsilon_j) + (\sum_{k=1}^{n+m-1} \gamma_k)$ and we want to keep it smaller than $2^{1-\text{prec}-p}$.

Optimally choosing the $\delta_i, \varepsilon_j$ and $\gamma_k$ 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 $\delta_1 \sim 2^{1-\text{prec}-p}$ and the other error terms very small compared to $\delta_1$. 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 $\text{prec}+p$. 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 $\delta_i = \varepsilon_j = \gamma_k \sim 2^{1-\text{prec}-p}/(2(n + m))$ for all $i, j, k$. More formally, this leads to the algorithm described in Algorithm 1. We now rigorously prove its correctness.

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

\[ \text{var\_name} = e (2(n + m) - 1). \]
2. on the order the operations are performed. It is well known \( \gamma \) to perform the corresponding addition but also on the order division/division, \( e \).

For a sum, it is more natural to consider absolute error smaller than \( 2 \cdot 1^{-\text{prec-p}} \cdot |e|/(2n - 1) \) and we define \( n'' \) accordingly for \( e'' \). Then, we evaluate \( e' \) with an absolute error smaller than \( 2^{1-\text{prec-p}} \cdot |e|/(2n'' - 1) \) and we evaluate \( e'' \) with an absolute error smaller than \( 2^{1-\text{prec-p}} \cdot |e|/(2n'' + 2n'' - 1) \). Finally, we perform the addition/subtraction with an absolute error smaller than \( 2^{1-\text{prec-p}} \cdot |e|/(2n'' + 2n'' - 1) \). 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 \( \text{sum_weight} (e) \). The formal algorithm is summed up in Algorithm 2.

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

**Input:** \( \text{var}\_\text{name}, e_1, e_2, p \)

**Output:** code for evaluating the expression \( e = e_1 \pm e_2 \)

1. \( u_1 \leftarrow e_1; u_2 \leftarrow e_2 \) evaluated by interval arithmetic;
2. \( v \leftarrow u_1 \pm u_2 \) evaluated by interval arithmetic;
3. if \( 0 \in v \) then abort;
4. \( n_1 \leftarrow 2 \cdot \text{sum}_{\text{weight}}(e_1) - 1; \)
5. \( n_2 \leftarrow 2 \cdot \text{sum}_{\text{weight}}(e_2) - 1; \)
6. \( E_1 \leftarrow \text{MINEXP}(\frac{n_1 \cdot v}{u_1}); E_2 \leftarrow \text{MINEXP}(\frac{n_2 \cdot v}{u_2}); \)
7. \( E \leftarrow \text{MINEXP}(\frac{v}{n_1 \cdot u_1 + n_2 \cdot u_2}); \)
8. let \( a_1 \) and \( a_2 \) be fresh names;
9. Output code for the following test:
   
   "if \( \text{prec} + p + 1 \leq E_1 \leq 1 \) then \( a_1 \leftarrow 0 \) else"
10. implementer("a_1", e_1, p + 1 − E_1);
11. Output code for the following test:
   "if \( \text{prec} + p + 1 \leq E_2 \leq 1 \) then \( a_2 \leftarrow 0 \) else"
12. implementer("a_2", e_2, p + 1 − E_2);
13. Output code for setting global precision to \( \text{prec} + p + 2 \);
14. Output code for computing \( \text{var}\_\text{name} \leftarrow a_1 \pm a_2; \)

### Proof of correctness: for \( i \in \{1, 2\} \), we have \( E_i \leq \text{EXP}(\frac{n_i \cdot v}{n_i \cdot u_i}) \) by definition of \( E_i \). A similar identity holds for \( E \). Hence

\[
\frac{|e_i|}{e} \leq \frac{n_i}{n} \cdot 2^{1-E_i} \quad \text{and} \quad \frac{|e_1| + |e_2|}{|e|} \leq \frac{1}{n} \cdot 2^{1-E}.
\]  

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

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

\[
\begin{align*}
|\text{var}_i - e| &\leq |e_1| - |e_2| + 2^{1-(\text{prec} + p + 1 - E_i)} + \\
&\leq |a_1 \pm a_2| + 2^{1-(\text{prec} + p + 2 - E)}.
\end{align*}
\]

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, \( \text{prec} + p + 1 - E_i > 1 \), thus \( |\varepsilon_i| \leq 1 \) and finally \( |a_i| = |e_i (1 + \varepsilon_i)| \leq 2|e_i| \).

\[ \square \]

4.5. Case of a basic function

We now study the case when \( e = f(e_1) \) where \( f \in B \) is a basic function. The idea of the algorithm is as follows: a recursive call \( \text{implementer}(\text{some fresh name } "y", e_1, q) \) 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, \( |h| \leq 2^{1-\text{prec} - q}|e_1| \).

The function \( f \) possibly amplifies or contracts this error. Namely, using the mean value theorem, \( f(y) = f(e_1) + h f'(\xi), \) where \( \xi \) 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 \( |h| \leq |f(e_1)| f'(e_1) |/f(e_1)| 2^{1-\text{prec} - p} \). Hence, it suffices to choose

\[ q \geq p + \text{MAXEXP}(e_1 f'(e_1)/f(e_1)). \]

This gives us the idea of the algorithm. However, in order to be perfectly rigorous, we have to take into account the fact that \( \xi \) 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 Algorithm 3.

**Proof of correctness:** we shall prove that the algorithm terminates and that it is correct. The case when \( \text{prec} + p \leq 1 \) is obvious. So, we suppose in the following that \( \text{prec} + p \geq 2 \). Obviously \( r \) strictly increases during the loop. Moreover, when \( r \) increases, the width of the interval \( [e_1 (1 -2-p^-), e_1 (1 +2-p^-)] \) decreases. Hence, during the loop, the interval \( v \) can only shrink and \( \text{MAXEXP}(v) \) can only decrease (or stay equal). This proves that the loop eventually terminates.

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

\[ |y - e_1| \leq 2^{1-\text{prec} - p} |e_1| \tag{4} \]

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

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

\[ |\theta| \leq 2^{1-\text{prec} - p} |e_1 f'(\xi)/f(e_1)|. \]

Since \( \xi \) lies between \( y \) and \( e_1 \), it holds that \( e_1 f'(\xi)/f(e_1) \in v \). Hence \( |\theta| \leq 2^{1-\text{prec} - p - \text{MAXEXP}(v)} \leq 2^{1-\text{prec} - p - 2} \).

In conclusion, assuming a global precision of \( \text{prec} + p + 2 \), we can write \( f(y) = f(e_1) (1) \). Moreover, since the final evaluation is also performed in precision \( \text{prec} + p + 2 \), we have \( \text{var}_i = f(y) (1) \). We conclude using Proposition 2 on \( \text{var}_i = f(e_1) (1) (1) = f(e_1) (2) \).
5. Examples

We have implemented our algorithm in the Sollya\(^6\) free software tool, as the implementconstant command.

We tested it on several expressions; each time, we ran the generated code and checked that the accuracy of 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

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

our algorithm aborts with a message indicating that \(\sqrt{\frac{32}{5}} - \sqrt{\frac{27}{5}} - \frac{1 + \sqrt{3} - \sqrt{9}}{\sqrt{25}}\) 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 [1]. 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 ← exp(1); /* prec+19 */
y ← 1 + y; /* prec+18 */
y ← log(y); /* prec+15 */
y ← 1 + y; /* prec+15 */
y ← log(y); /* prec+11 */
y ← 1 + y; /* prec+11 */
y ← log(y); /* prec+7 */
y ← 1 + y; /* prec+7 */
y ← log(y); /* prec+2 */
return y
```

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...e-11; if long double are used (i.e. 64-bit significand), the computed result is -1.31...e-12; and actually, the correct result is -1.34...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.

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

```
Input: prec

| t1 ← sin(1e22); /* prec+66 */ |
| t2 ← 173746 t1; /* prec+64 */ |
| t3 ← 171/10; /* prec+70 */ |
| t4 ← log(t3); /* prec+67 */ |
| t5 ← 94228 t4; /* prec+65 */ |
| t6 ← t2 + t5; /* prec+63 */ |
| t7 ← 42/100; /* prec+68 */ |
| t8 ← exp(t7); /* prec+65 */ |
| t9 ← 78487 t8; /* prec+63 */ |
| y ← t6 - t9 ; /* prec+61 */ |

return y
```

5.3. Third example

Our last example comes from a real-life problem arising when one wants to implement the Airy \(\text{Ai}\) 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 \(\Gamma\) but it would be much less efficient than using a formula such as Equation (2) given on page 1.

Our algorithm is not able to handle series, but as discussed in Section 2, the sets of basic functions \(B\) and predefined constants \(C\) are arbitrary. Hence, our implementation provides an extension mechanism that allows one to add a new constant in \(C\). We define \(\alpha\) as the value of the series in Equation (2). In order to add \(\alpha\) to the set \(C\), we only need to provide a procedure \text{eval_alpha} (\text{prec}) that returns an approximate value \(y\) of \(\alpha\) such that \(|y - \alpha| \leq 2^{1-\text{prec}} |\alpha|\). It is easy to see that the ratio between two consecutive terms of the series is always smaller than 9/64000, and hence it

\(^6\text{See http://sollya.gforge.inria.fr/}\)
is easy to dynamically find a suitable truncation rank. Then the binary splitting algorithm [2] can be used to evaluate the truncated series without rounding error.

Once $\alpha \in \mathbb{C}$, Equation (2) becomes $\Gamma(1/3) = (12^{\pi^4/\sqrt{10}})^{1/6}$ and can be handled by our algorithm: the generated code is illustrated in Algorithm 6.

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

**Input**: prec

\[
\begin{align*}
  t_1 & \leftarrow \pi; \quad /* \text{prec + 11} */ \\
  t_2 & \leftarrow t_1^4; \quad /* \text{prec + 7} */ \\
  t_3 & \leftarrow \text{eval_alpha(prec + 5)}; \\
  t_4 & \leftarrow \sqrt{t_0}; \quad /* \text{prec + 7} */ \\
  t_5 & \leftarrow t_2t_3/t_4; \quad /* \text{prec + 5} */ \\
  y & \leftarrow \sqrt[4]{t_5}; \quad /* \text{prec + 2} */ \\
\end{align*}
\]

**Return** $y$

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, we used our algorithm when we developed the Ai function in MPFR. 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. A first improvement would be to represent expressions as directed acyclic graphs instead of trees: it 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 could then be checked with an automatic proof checker, thus offering a high guarantee.

References


