Automatic numerical expansions

by Joris van der Hoeven
LIX
École Polytechnique
91128, Palaiseau
France
Email: vdhoeven@lix.polytechnique.fr

October 1994

Abstract

Let exp-log constants be those real numbers which are built up from the rationals by field operations, exponentiation and logarithm. Theoretically, the sign of an exp-log constant can be determined by floating point computations, whenever we can decide whether the constant is zero or not. However, this method has a complexity which is worse than any iterated exponential, even in concrete cases. We propose an algorithm, combining floating point evaluations with asymptotic methods, which is expected to be efficient in practice.

Key words: Numerical expansion, exp-log constant, floating point evaluation, asymptotics, algorithm.

1 Introduction

Let exp-log constants be those real numbers which are built up from the rationals by field operations, exponentiation and logarithm. The zero equivalence problem of these constants is related to very hard conjectures in number theory, notably Schanuel's conjecture (see [CavPr 78]). Modulo this conjecture, Richardson established a zero equivalence algorithm (see [Rich 92], [Rich 94]). We consider the problem of determining the signs of exp-log constants, assuming that we are given a zero equivalence algorithm.

Theoretically, this can be done by floating point computations. However, this method has a complexity which is worse than any iterated exponential. For instance, consider constants like

$$e^{\exp_n x+(\exp_n x)^{-1}} - e^{\exp_n x}.$$

Here $\exp_n x$ denotes the n-th iterated exponential of x. In some cases, where important simplifications take place for number theoretical reasons, we do not expect that there exist good alternatives to floating point evaluations. However, in the above example, the simplifications are rather of a formal nature. The object of this article is to show that in such cases it is possible to combine floating point methods with asymptotic methods to yield an efficient algorithm.

More precisely, we will adapt the asymptotic expansion algorithm for exp-log functions from [VdH 94a], which generalizes Gonnet and Gruntz' algorithm (see [GoGr 92]. See also [GeGo 88], [Sh 90]). We will briefly recall this algorithm in section 2. We discuss how to adapt the symbolic algorithm to the numerical case in section 3. The "numerical expansion" algorithm for exp-log constants is given in section 4. In fact, numerical expansions contain much more detailed information than just the sign of a constant. See also section 5 for a discussion.

An implementation of Richardson's zero-test algorithm together with the numerical expansion algorithm should be available before the end of 1995.

2 Recall of the symbolic expansion algorithm

During this section $\mathfrak{T}$ denotes the field of *exp-log functions*, that is, the set of functions built up from x and $\mathbb{Q}$ by field operations, exponentiation and logarithm (of positive elements). Modulo Schanuel's conjecture (see section 5) all the above operations, as well as zero-equivalence testing, can be performed algorithmically in $\mathfrak{T}$ (see [Rich 94], [Sh 89]). In [VdH 94a] we generalized the limit computation algorithm due to Gonnet and Gruntz (see [GoGr 92]) to yield a symbolic expansion algorithm. We will briefly recall this algorithm, without going into details. All expansions will be done in a neighbourhood of $+\infty$. An example of how the algorithm works is given at the end of this section.

We start with some definitions. Let A be some abelian totally ordered group. We will say that a subset $S \subseteq A$ is *finitely generated*, if there exist $x_1, \cdots, x_n > 0$ in A and $a_1, \cdots, a_n$ in $\mathbb{Z}$, such that each $x \in S$ can be written as $x = b_1 x_1 + \cdots + b_n x_n$, with $a_i \leq b_i \in \mathbb{Z}$, for each $1 \leq i \leq n$.

If f and g are exp-log functions, then we will write $f \lll g$, whenever $\ln |f| = o(\ln |g|)$. Intuitively, this means that f is of a smaller asymptotic scale than g (for example $e^x \lll x^x$, but $x \lll x^{100}$ is false). Suppose now that $g_1 \lll \cdots \lll g_n$ are positive infinitely big exp-log functions. Then a *multiseries* in $g_1^{-1}, \cdots, g_n^{-1}$ over a subfield K of $\mathfrak{T}$ is a series f of the form

$$f = \sum_{\alpha_1, \cdots, \alpha_n \in \mathbb{R}} f_{\alpha_1, \cdots, \alpha_n} g_1^{-\alpha_1} \cdots g_n^{-\alpha_n},$$

with finitely generated support, and the $f_{\alpha_1, \cdots, \alpha_n}$'s in K. Here the support (i.e. those $(\alpha_1, \cdots, \alpha_n)$'s, with $f_{\alpha_1, \cdots, \alpha_n} \neq 0$) is a finitely generated subset of $\mathbb{R}^n$ with the lexicographical ordering. The set $K [\![g_1^{-1}; \cdots; g_n^{-1}]\!]$ forms a field: in fact, multiseries slightly generalize formal power series and all operations on multiseries are defined by the usual formulas.

We remark that a multiseries $f \in K[\![g_1^{-1}; \cdots ; g_n^{-1}]\!]$ can also be interpreted as a multiseries in $K[\![g_1^{-1}; \cdots ; g_{n-1}^{-1}]\!][\![g_n^{-1}]\!]$. If $g_1, \cdots, g_n$ and f are exp-log functions, we say that f is an *automatic multiseries*, if

(a) The elements of its support in g_n^{-1} are in $\mathfrak{T}$ and can be listed in increasing order by some algorithm.
(b) The corresponding coefficients are exp-log functions, which can be computed effectively.
(c) The coefficients are automatic multiseries in $g_1^{-1}, \cdots, g_{n-1}^{-1}$.

By convention, f is assumed to be a constant if $n = 0$. We emphasize that an automatic multiseries is merely an *algorithm*, capable of expanding the multiseries to any desired order. The automatic multiseries also form a field, stable under exponentiation of bounded elements and logarithm of elements with a strictly positive limit.

The problem of expanding an exp-log function f now reduces to finding appropriate $g_1 \lll \cdots \lll g_n$, such that f is an automatic multiseries in $g_1^{-1}, \cdots, g_n^{-1}$. Moreover, the g_i's need to be of a special form, so that we can read all relevant asymptotic information from the expansion. More precisely, the g_i's are assumed to be either iterated logarithms or of the form e^{h_i}, where h_i is an automatic multiseries in $g_1^{-1}, \cdots, g_{i-1}^{-1}$. Such a set $B = \{g_1, \cdots, g_n\}$ is called a *reduced base*.

The expansion algorithm takes an exp-log expression f (considered as a tree) as input. Then it expands all subexpressions of f, starting with the leafs. The algorithm disposes of a global variable B which contains a reduced base, which is gradually enlarged during the execution. In fact, except when we need to insert a new iterated logarithm or a new exponential into B, all operations are trivial.

The result of the expansion algorithm is another exp-log expression, from which the expansion of f can be reconstructed in a straightforward way. More precisely, all logarithms occuring in the result are iterated logarithms of x or logarithms of the form $\ln(1+\varepsilon)$, with $\varepsilon \lll 1$. Similarly, exponentials are either constant powers of elements from the reduced base, or of the form $\exp \varepsilon$, with bounded ε. Therefore, the expansion of f to any order can be computed from the result by using only operations on multiseries.

Algorithm expand(f). The algorithm takes an exp-log expression f on input and rewrites it to an expression from which the expansion of f to any order can be reconstructed by using only operations on multiseries. The algorithm uses a global variable B, which is initialized by $B := \{x\}$.

case $f = x$ or $f \in \mathbb{Q}$: **return** f.

case $f = f_1 \top f_2,\ \top \in \{+, -, \cdot, /\}$:
  if $\top = /$ and $f_2 = 0$ **then raise** "division by zero"
  return expand(f_1) $\top$ expand(f_2)

case $f = \ln f_1$:

$\bullet$Compute the expansion of f_1 and denote $\{g_1 = e^{h_1}, \cdots, g_n = e^{h_n}\} = B$.

$\bullet$Compute $c, \alpha_1, \cdots, \alpha_n$, with $f_1 = c g_1^{\alpha_1} \cdots g_n^{\alpha_n}(1+\varepsilon)$, where $\varepsilon \prec\!\!\prec 1$.

if $f_1 = 0$ or $c < 0$ **then raise** "logarithm: out of domain"

if $\alpha_1 \neq 0$ **then** $B := B \cup \{h_1\}$

return $\ln(1 + \text{expand}(\varepsilon)) + \ln c + \alpha_1 \text{expand}(h_1) + \cdots + \alpha_n \text{expand}(h_n)$

case $f = \exp f_1$:

$\bullet$Compute the expansion of f_1 and denote $\{g_1 = e^{h_1}, \cdots, g_n = e^{h_n}\} = B$.

if f_1 is bounded **then return** $\exp(\text{expand}(f_1))$

if $\exists 1 \leq i \leq n \quad f_1 \asymp h_i$ **then**

$\bullet$Compute $\lambda = \lim f_1/h_i$.

return $g_i^\lambda \text{expand}(\exp(f_1 - \lambda h_i))$

$\bullet$Compute φ and ε, such that $f_1 = \varphi + \varepsilon$ and $\varphi = (f_1)_{0, \overset{n+1-i \text{ times}}{\cdots}, 0}$.

$B := B \cup \{\exp |\varphi|\}$

return $(\exp|\varphi|)^{\mathbf{sign}\,\varphi} \exp \text{expand}(\varepsilon)$

Remark. The computations of $c, \alpha_1, \cdots, \alpha_n$ in the case $f = \ln f_1$ can be done recursively. Indeed, we already know how to expand f_1. Similarly, in the case of exponentiation, we can check whether f_1 is bounded, whether it is asymptotic to one of the h_i's, etc.

Example 1. Suppose that we wish to expand

$$f(x) = \ln\ln(xe^{e^x} + 1) - \exp\exp\left(\ln\ln x + \frac{1}{xe^x}\right).$$

At the beginning, $B := \{x\}$. The expansions of the subexpressions are computed in the following order, keeping track of the changes of B:

$B := \{x\}$

$x \to x$

$e^x \to e^x$

$B := \{x, e^x\}$ (x is not asymptotic to $\ln x$)

$e^{e^x} \to e^{e^x}$

$B := \{x, e^x, e^{e^x}\}$ (e^x is not asymptotic to $\ln x$ nor to x)

$xe^{e^x} \to xe^{e^x}$

$1 \to 1$

$xe^{e^x} + 1 \to xe^{e^x} + 1$

$$\ln(xe^{e^x} + 1) \to e^x + \ln x + \ln\left(1 + \frac{1}{xe^{e^x}}\right) = e^x + \ln x + \frac{1}{xe^{e^x}} + \cdots$$

$B := \{\ln x, x, e^x, e^{e^x}\}$ ($\ln x$ is not yet in B)

$$\ln\ln(xe^{e^x} + 1) \to x + \ln\left(1 + \frac{1}{e^x}\left[\ln x + \ln\left(1 + \frac{1}{xe^{e^x}}\right)\right]\right)$$

$$
\begin{aligned}
&\quad = x + \frac{\ln x}{e^x} - \frac{\ln^2 x}{2e^{2x}} + \cdots + \frac{1}{xe^x e^{e^x}} + \cdots \\
&\ln x \to \ln x \\
&\ln\ln x \to \ln\ln x \\
B := &\{\ln\ln x, \ln x, x, e^x, e^{e^x}\} \quad (\ln\ln x \text{ is not yet in } B) \\
&xe^x \to xe^x \\
&\frac{1}{xe^x} \to \frac{1}{xe^x} \\
&\ln\ln x + \frac{1}{xe^x} \to \ln\ln x + \frac{1}{xe^x} \\
&\exp\left(\ln\ln x + \frac{1}{xe^x}\right) \to \ln x \exp\left(\ln\ln x + \frac{1}{xe^x} - \ln\ln x\right) \\
&\qquad\qquad = \ln x + \frac{\ln x}{xe^x} + \frac{\ln x}{2x^2 e^{2x}} + \cdots \\
&\exp\exp\left(\ln\ln x + \frac{1}{xe^x}\right) \to x \exp\left[\ln x \exp\left(\frac{1}{xe^x}\right) - \ln x\right] \\
&\qquad\qquad = x + \frac{\ln x}{e^x} + \frac{\ln^2 x}{2xe^{2x}} + \frac{\ln x}{2xe^{2x}} + \cdots \\
&f(x) \to x + \ln\left(1 + \tfrac{1}{e^x}\left[\ln x + \ln\left(1 + \tfrac{1}{xe^{e^x}}\right)\right]\right) - x\exp\left[\ln x \exp\left(\tfrac{1}{xe^x}\right) - \ln x\right] \\
&\quad = -\frac{\ln^2 x}{2e^{2x}} - \frac{\ln^2 x}{2xe^{2x}} + \cdots
\end{aligned}
$$

3 The exp-log constant comparison problem

Let us now consider the problem of comparing exp-log constants. Under the hypothesis that we have a zero-equivalence algorithm, it suffices to do floating point arithmetic at a sufficient precision. However, these floating point operations might be very expensive in time. The first reason is number theoretical: we do not have bounds how close we can come to zero with non-zero exp-log constant expressions of bounded size, and such that $|f|$ is bounded as well for all subexpressions of the form e^f. A second reason is that whenever we take an exp-log function $f(x)$ and we substitute a very large constant C for x, then the sign of $f(C)$ is expected to be as hard to determine as the sign of $f(x)$.

The number theoretic problem is expected to be very hard, because good bounds would in particular imply Schanuel's conjecture (see section 5). However, the second reason shows us that brute force floating point arithmetic is a method that is expected to fail frequently, even in practical situations such as the introductory example. In fact, it is possible to refine the floating point method, by adapting the symbolic expansion algorithm. Algorithms, which are obtained in this way will be called *numerical expansion algorithms*. The analogies between symbolic expansions and numerical expansions are given by the following table:

symbolic	numerical
exp-log function exp-log constant	exp-log constant exp-log constant c, with $2^{-D} < \lvert c\rvert < 2^{D}$ D: Precision in number of digits.
$f \prec\!\!\prec g$ $f \prec\!\!\prec\!\!\prec g$ $f \asymp g$	$\lvert f\rvert \leq (2N^2+2)\lvert g\rvert$ $\lvert \ln\lvert f\rvert\rvert \leq (2N^2+2)\lvert \ln\lvert g\rvert\rvert$ $\lvert g\rvert/N \leq \lvert f\rvert \leq N\lvert g\rvert$ N: Maximal length and density of multiseries.
$B = \{g_1, \cdots, g_n\}$ reduced base $1 \prec\!\!\prec g_i$, $0 < g_i$ $g_1 \prec\!\!\prec\!\!\prec \cdots \prec\!\!\prec\!\!\prec g_n$	$B = \{g_1, \cdots, g_n\}$ reduced base. $2^{(2N+1)D} \leq g_i$. $g_1^{2N^2+2} \leq g_2, \cdots, g_{n-1}^{2N^2+2} \leq g_n$.

Clearly, the "constants" in numerical expansions have to be small, without being too small. Otherwise, later terms in the expansion might contribute more than earlier terms. We will call this phenomenon *numerical interference*. The above kind of numerical interference is due to the fact that D was chosen too small.

Similarly, only the first terms of a numerical multiseries make sense, in contrast to the case of usual expansions. Indeed, consider the numerical expansion of

$$e^{-100} \overset{100 \text{ times}}{\cdots} e^{-100} - 2e^{-10000},$$

where $D = 8$, $N = 5$, $g_1 = e^{100}$, $g_2 = e^{10000}$. Here $e^{-100} \overset{100 \text{ times}}{\cdots} e^{-100}$ will be interpreted as g_1^{-100}, while e^{-10000} will be interpreted as g_2^{-1}. We have again numerical interference between the asymptotic scales, because of g_1^{-100}, which occurs before g_2^{-1} in the expansion, although they have the same order. In fact, the "length" of the multiseries in g_1 is too big here ($100 > 5$), which is due to the fact that N was chosen too small. This length is defined to be the maximal absolute value of an exponent of g_1, occuring in the relevant part of the expansion. By relevant we mean that we only need to look at those terms which are actually needed in the computation.

There is a last, more troublesome kind of numerical interference. Consider for example the numerical expansion of

$$e^{100+\frac{1}{100}} - e^{\frac{1}{99}} e^{100},$$

with $D = 8$, $N = 5$ and $g_1 = e^{100}$. If we had directly translated the symbolic expansion algorithm, the above expression would have been expanded as $g_1^{1.0001} - e^{1/99} g_1$ and we would have been unable to determine the sign from this numerical expansion. This is due to the fact that a multiseries is not just a generalized Laurent series, as we may have non integer exponents. In the above case, the "density" of the relevant terms of the multiseries is too high ($1/(1.0001-1) > N = 5$). Here, the density in g_1 is the maximal inverse value of the differences between exponents of g_1 occuring in the relevant part of the expansion. In the above example, we only need to consider

$g_1^{1.0001}$ and g_1. This type of numerical interference is more troublesome, because it is not efficient to merely increase N. Instead, $e^{100.01}$ should be expanded as $e^{100}e^{0.01}$.

Let us now discuss how to adapt the symbolic expansion algorithm. One thing becomes simpler in the numerical case: we do not need analogues for the iterated logarithms of x in the reduced base. Indeed, as they are all small, they are computed numerically. In particular, the reduced base is empty at the beginning. The main difficulty with respect to the symbolic case is that we should be able to read the sign of an exp-log constant from its numerical expansion. This means that we need to ensure that no numerical interference occurs and that we should be able to obtain error bounds. In fact, the three kinds of numerical interference mentioned above correspond to three types of exceptions: "numerical", "series length" and "series density". The two first can be captured in a main loop, in which D, resp. N is augmented whenever a "numerical", resp. "series length" exception is raised. The third type of exception needs a more subtile treatment.

First, we need to encode information in a different way. Instead of rewriting the input exp-log function f into another exp-log function, from which the expansion of f can be reconstructed by operations on multiseries only, we will store the dominant monomial of each exponential occuring in the computation in a so called exponentiation table T. More precisely, whenever we expand e^f, we can write $f = \alpha_1 h_1 + \cdots + \alpha_n h_n + b$, after a potential insertion of a new element into the reduced base. Here b is a "bounded element" (that is $|b| \leq D \ln 2$). Then set $T[f] := g_1^{\alpha_1} \cdots g_n^{\alpha_n}$. As before, the exponentiation table allows us to reconstruct all relevant numerical expansions by doing operations on multiseries only.

Moreover, we can extract generators for the supports of all multiseries from the exponentiation table. Indeed, denote by $\gamma_1, \cdots, \gamma_p$ the different γ_j's, so that $g_i^{\gamma_j}$ occurs in (array elements of) T. Then the support of each multiseries in g_i which can potentially be considered during the computations is in the additive subgroup generated by the γ_j's. Moreover, we can compute integers $k_1, \cdots, k_p$, with $\alpha = k_1\gamma_1 + \cdots + k_p\gamma_p$, for each α in the support of such a multiseries. Indeed, it suffices to propagate this knowledge during the expansion. Alternatively, the LLL-algorithm can be used (see [LLL 82]).

Now suppose that we encountered a "series density" exception during our computations. This is due to the fact that two consecutive terms of a multiseries in g_i have exponents which differ by a number $\varepsilon \leq 1/N$. If this is the case, we can write $\varepsilon = k_1\gamma_1 + \cdots + k_p\gamma_p$, with the above notations. Without loss of generality, we may assume that $k_p \neq 0$ and $\gamma_p \neq 1$. Then it suffices to systematically replace $g_i^{\gamma_p}$ by $T[\varepsilon h_i/k_p] g_i^{-(k_1\gamma_1 + \cdots + k_{p-1}\gamma_{p-1})/k_p}$ in T, after having expanded $\exp(\varepsilon h_i/k_p)$. In the case of the above example, this will replace $g_1^{1.0001}$ by $T[g_1^{0.0001}] g_1 = g_1$.

We finally need to discuss how we obtain error bounds, which will ensure that the sign of an expression is the same as the sign of the dominant coefficient of its numerical expansion. In fact, this is almost automatically the case in absence of numerical interference. More rigourously, a precise bound for the sum of the rest terms of an expansion can be computed, as we will show in the next section. If this bound is insufficient, an exception raised.

4 The numerical expansion algorithm

In this section we describe the actual numerical expansion algorithm. Taking into account the discussion from the previous section, we have four global variables D, N, B and T. Here D contains the numerical precision in the number of digits and N the maximal length as well as density of a numerical multiseries. Furthermore, B is the reduced base and T stands for the exponentiation table. Finally, we will use a **raise** -**catch** formalism for our exception handling: errors can be raised using **raise** at any place in any subalgorithm, where an argument indicating the nature of the exception is given. Errors are captured by the **catch** statement, where the variable 'exception' contains the nature of the exception.

Algorithm main_loop(f). The algorithm takes an exp-log constant f on input and computes its sign.

$D := 32$, $N := 4$, $B := \emptyset$, $T := \emptyset$
while catch
    expand(f)
    $s :=$ sign(f)
do
    if exception="numerical" **then** $D := 2D$
    if exception="series length" **then** $N := 4N$
    if exception $\notin$ {"numerical","series length"} **then raise** exception
    $B := \emptyset$, $T := \emptyset$
return s

The subalgorithm 'expand' is the adaptation of the symbolical expansion algorithm to the numerical case and the subalgorithm 'sign' computes the sign of f using its numerical expansion. Modulo Schanuel's conjecture and the correctness of these subalgorithms, the above algorithm terminates. Indeed, if N is very big, then the reduced base is empty and only floating point operations will be used. From that moment on, D increases until the necessary floating point precision is reached.

Algorithm expand(f). The algorithm takes an exp-log constant f on input and extends the exponentiation table T, so that a numerical expansion of f can be reconstructed from T.

case $f \in \mathbb{Q}$: Do nothing.

case $f = f_1 \top f_2,\ \top \in \{+,-,\cdot,/\}$:

  if $\top = /$ and $f_2 = 0$ **then raise** "division by zero"

  expand(f_1)

  expand(f_2)

case $f = \ln f_1$:

  if $f_1 \leq 0$ **then raise** "logarithm: out of domain"

  expand(f_1)

case $f = \exp f_1$:

  •Compute the expansion of f_1 and denote $\{g_1 = e^{h_1}, \cdots, g_n = e^{h_n}\} = B$.

  if $|f_1| \leq D \ln 2$ **then** $T[f_1] := 1$, **return**

  if $|f_1| \leq (2N+1) D \ln 2$ **then raise** "numerical"

  if $\exists 1 \leq i \leq n \quad h_i/N \leq |f_1| \leq N h_i$ **then**

    •Compute the dominant coefficient λ of f_1/h_i.

    expand($\exp(f_1 - \lambda h_i)$), $T[f_1] := g_i^{\lambda} T[f_1 - \lambda h_i]$, **return**

  if $\exists 1 \leq i \leq n \quad h_i/(2N^2+2) \leq |f_1| \leq (2N^2+2) h_i$ **then raise** "series length"

  •Compute φ and ε, such that $f_1 = \varphi + \varepsilon$ and $\varphi = (f_1)_{0, \overset{n+1-i \text{ times}}{\cdots}, 0}$.

  $B := B \cup \{\exp |\varphi|\}$

  $T[f_1] := (\exp |\varphi|)^{\mathbf{sign}\, \varphi}$.

As before, tests for inequalities, which reduce to sign computations, can be resolved recursively. Indeed, we first apply 'expand' to the expression whose sign has to be determined, and then apply 'sign' to determine its sign.

The "series density" exception is handled in 'sign' (and more generally in every algorithm, which exploits numerical expansions). Whenever this exception is raised, ε and i are passed as arguments, where $\varepsilon < 1/N$ denotes the difference between two exponents, encountered during a computation with a multiseries in g_i^{-1}. The exception is handled by changing T as described in the previous section and by reexecuting 'sign' afterwards.

Algorithm sign(f). Computes the sign of f, after having applied 'expand' to f.

while catch

  $c := f$

  for i:=n **downto** 1 **do**

    $\alpha :=$ lazy_next_exponent($c, i, -\infty$)

    $dummy :=$ lazy_bound(c, i, α)

    $c :=$ lazy_coefficient(c, i, α)

  return sign c

do

  if exception $\neq$ "series density" **then raise** exception

  $(\varepsilon, i) :=$ exception_arguments

  •Let $\gamma_1, \cdots, \gamma_p$ be those exponents of g_i occuring in the exponentiation table T.

  •Compute integers $k_1, \cdots, k_p$, such that $\varepsilon = k_1\gamma_1 + \cdots + k_p\gamma_p$ and with k_p minimal, such that $\gamma_p \neq 1$.

  expand(exp(ε/k_p))

  •Systematically replace all occurences of $g_i^{\gamma_p}$ by $T[\varepsilon h_i/k_p]g_i^{-(k_1\gamma_1+\cdots+k_{p-1}\gamma_{p-1})/k_p}$ in the exponentiation table T.

We need to specify 'lazy_next_exponent', 'lazy_coefficient' and 'lazy_bound'. lazy_next_exponent(f, i, α) determines the first exponent bigger than α in the expansion of f with respect to g_i^{-1} and lazy_coefficient(f, i, α) computes the coefficient of $g_i^{-\alpha}$ in f. They can be implemented in a straightforward way by using the usual formulas for series expansions. We will not detail them any further here, although we mention their additional features:

(a) Whenever a "constant" coefficient is encountered, which exceeds 2^D or 2^{-D} in absolute value, a "numerical" exception is raised.
(b) Whenever an exponent in the support of a multiseries is encountered, which exceeds N in absolute value, a "series length" exception is raised.
(c) Whevener two exponents in the support of a multiseries in g_i^{-1} are encountered, which have a distance $0 < \varepsilon \leq 1/N$, a "series density" exception is raised, with ε and i as arguments.
(d) The algorithms use Richardson's algebraic zero-test algorithm for exp-log constants. It is convenient to use it only, when the first terms of the expansion cancel. This avoids performing unnecessary and expensive zero-tests.

Finally, lazy_bound(f, i, α) computes an admissible upper bound $M_{f,\alpha}$ for $|\sum_{\beta>\alpha} f_\beta g_i^{-\beta}|$, where f is a multiseries in $g_1^{-1}, \cdots, g_i^{-1}$. Here an *admissible bound* is a bound of the form $M_{f,\alpha} = c g_1^{-\beta_1} \cdots g_i^{-\beta_i}$, where $c \leq 2^D$, $\beta_i - \alpha > 1/N$ and such that none of the $|\beta_i|$'s exceeds N. We use admissible bounds for technical reasons, as will become clear from the proof of theorem 1 below. If no admissible bound can be obtained, an exception is raised. We remark that we merely need the existence of a good bound of the sum of the rest terms in 'sign', so that we do not care about its value, which explains the use of the dummy variable in the algorithm.

Algorithm lazy_bound(f, i, α): At the input, we have an expression which is a numerical multiseries in $g_1, \cdots, g_i$, where $B = \{g_1, \cdots, g_n\}$. The algorithm returns an admissible bound $M_{f,\alpha}$ for $|\sum_{\beta>\alpha} f_\beta g_i^{-\beta}|$.

case $f = cg_1^{-\beta_1} \cdots g_i^{-\beta_i}$:
    if $\alpha \geq \beta_i$ **then return** 0
    if $\beta_i - \alpha \leq 1/N$ **then raise** "series density"$(\beta_i - \alpha, i)$
    return f

case $f = f_1 \pm f_2$: **return** bound(lazy_bound(f_1, i, α) + lazy_bound(f_2, i, α))

case $f = f_1 f_2$:
    $b := 0$
    $\mu_1 :=$ lazy_next_exponent$(f_1, i, -\infty)$
    $\mu_2 :=$ lazy_next_exponent$(f_2, i, -\infty)$
    while $\mu_2 + \mu_1 \leq \alpha$ **do**
        $b_1 :=$ lazy_bound(lazy_coefficient$(f_1, i, \mu_1), i - 1, -\infty$)
        $b_2 :=$ lazy_bound$(f_2, i, \alpha - \mu_1)$
        $b :=$ bound$(b + b_1 b_2)$
        $\mu_1 :=$ lazy_next_exponent(f_1, i, μ_1)
    return bound(b + lazy_bound(f_1, i, μ_1)lazy_bound$(f_2, i, \alpha - \mu_1)$)

case $f = 1/(1 - \varepsilon) \quad (|\varepsilon| < 2^{-D})$:
    $\mu :=$ lazy_next_exponent$(\varepsilon, i, -\infty)$
    if $\mu = 0$ **then**
        $\varepsilon_0 :=$ lazy_coefficient$(\varepsilon, i, 0)$
        $\varepsilon_r := \varepsilon - \varepsilon_0$
        $Q := \frac{1}{1-\varepsilon_0}$
        return lazy_bound $\left(Q \frac{1}{1-Q\varepsilon_r}\right)$
    $M_\varepsilon :=$ lazy_bound$(\varepsilon, i, 0)$
    if $\alpha < 0$ **then return** bound$(1 + 2^{-D})$
    $k := \lfloor \alpha/\mu \rfloor$
    $S := \sum_{i=0}^{k} \varepsilon^i$
    return bound(lazy_bound$(S, i, \alpha) + (1 + 2^{-D}) M_\varepsilon^{k+1}$)

case $f = \ln(1 + \varepsilon)$, $f = \exp \varepsilon \quad (|\varepsilon| < 2^{-D})$: Similar to the above case.

The subalgorithm 'bound' is used to compute admissible bounds for sums and products of admissible bounds. The implementation is straightforward and an exception is raised if no admissible bound can be obtained.

Remark. All computations on the "small but not too small exp-log constants" (which replace the exp-log constants from the symbolical expansion) in this section are done by floating point computations. All computations with exponents occuring in the multiseries are done in a similar way. During these computations, we keep track of the numerical inprecision and whenever we test for exceptions, the most pessimistic estimate is used.

Theorem 1. *Modulo Schanuel's conjecture, the above algorithm computes the sign of any exp-log constant f.*

Proof. Let us prove the termination of our algorithm (modulo Schanuel's conjecture). Assume that we enter in an infinite loop for fixed values of D and N. From a certain moment on, more and more elements are inserted into the reduced base, each of which are smaller than elements which are already in the reduced base. This is impossible.

Let us finally prove the correctness of our algorithm. Let us denote $f^{(n)} = f$ and denote by $f^{(i-1)}$ the dominant coefficient of the multiseries $f^{(i)}$ in g_i^{-1} and $\mu^{(i)}$ the corresponding exponent. We claim that the signs of $f^{(n)}$ and $f^{(0)}$ are the same. Now we have an admissible error bound $M_{f^{(i)},\mu^{(i)}} = c g_1^{-\beta_1} \cdots g_i^{-\beta_i}$ for $\sum_{\alpha > \mu^{(i)}} f_\alpha^{(i)} g_i^{-\alpha}$. In particular, $\beta_i - \mu^{(i)} > 1/N$. Let us show that

$$\frac{c g_1^{-\beta_1} \cdots g_i^{-\beta_i}}{f^{(0)} g_1^{-\mu^{(1)}} \cdots g_i^{-\mu^{(i)}}} \leq g_{i-1}^{-1/N}.$$

By convention, $g_0 = 2^{ND}$. We have

$$\begin{array}{rcl} g_i^{-\beta_i - \mu^{(i)}} & \leq & g_{i-1}^{-(2N+1/N)} g_{i-1}^{-1/N} \\ g_{i-1}^{-(\beta_{i-1} - \mu^{(i-1)} + 2N + 1/N)} & \leq & g_{i-2}^{-(2N+1/N)} \\ g_{i-2}^{-(\beta_{i-2} - \mu^{(i-2)} + 2N + 1/N)} & \leq & g_{i-3}^{-(2N+1/N)} \\ & \vdots & \\ c/f^{(0)} g_1^{-1/N} & \leq & 2^{-D} = g_0^{-1/N} \end{array}$$

The relation we just proved can be reformulated:

$$\frac{f^{(i)} g_{i+1}^{-\mu^{(i+1)}} \cdots g_n^{-\mu^{(n)}} - f^{(i-1)} g_i^{-\mu^{(i)}} \cdots g_n^{-\mu^{(n)}}}{f^{(0)} g_1^{-\mu^{(1)}} \cdots g_n^{-\mu^{(n)}}} \leq g_{i-1}^{-1/N}.$$

Now our claim follows by applying the triangular inequality.

5 Perspectives and conclusion

Let us now discuss the complexity of our algorithm. Theoretically, we do not even know whether the algorithm terminates. This shows that it will be very hard to obtain good bounds for the complexity. Indeed, we have proved correctness and termination of the algorithm modulo Schanuel's conjecture:

Conjecture 1. ***(Schanuel)*** *If $\alpha_1, \cdots, \alpha_n$ are $\mathbb{Q}$-linearly independent complex numbers, then the transcendence degree of $\mathbb{Q}[\alpha_1, \cdots, \alpha_n, e^{\alpha_1}, \cdots, e^{\alpha_n}]$ over $\mathbb{Q}$ is at least n.*

In fact, a feature of Richardson's zero-test algorithm is that it only fails (in the sense that it does not terminate) if we gave a counterexample to Schanuel's conjecture on input. This shows that the algorithm is expected to fail extremely seldomly, and that in practice, all failures will rather be due to expression swell.

A more relevant question is to examine in which practical cases our algorithm is expected to be slow. Two situations, in which the algorithm is slow, are intrinsic to the type of method used. First, suppose that we have a very small exp-log constant, whose subexpressions are all "small without being to small". Then floating point computations at a very high precision are needed to determine its sign. Secondly, it can happen that Taylor series expansion is exponential in time, as shows the following example:

$$\frac{1}{1-\varepsilon}-\frac{1}{1-\varepsilon^2}-\frac{\varepsilon}{1-\varepsilon^4}-\cdots-\frac{\varepsilon^{2^{n-1}-1}}{1-\varepsilon^{2^n}}.$$

Another case in which our numerical expansion algorithm is expected to be slow, is when we have a lot of "numerical interference between the different asymptotic scales". Consider for example the numerical expansion of

$$e+e^2+e^4+\cdots+e^{2^n}$$

This constant is expanded by using floating point computations only. This is not very efficient, if we merely want to compute its sign. Similarly, if we want to compute the sign of the sum of a number and a much smaller number, then we do not really need to compute a complete numerical expansion for the smaller number. In fact, in the case of sign computations we are only interested in the first term of a numerical expansion and a convenient error bound. We have some new results for optimizing the sign computation algorithm in this direction, which will be presented in a forthcoming paper.

An important advantage of our numerical expansion algorithm is that if $f(x)$ is an exp-log function and C a huge constant, then the numerical expansion of $f(C)$ usually takes about the same time as the symbolic expansion of $f(x)$. This shows that for this type of constant our algorithm is very efficient. Despite the drawbacks of our algorithm mentioned above, we expect that it behaves correctly for intermediate cases between floating point computations and the above situation. We will check this, as soon as we have implementations.

To conclude, the main point of this article is to combine numerical and symbolic computation methods to solve problems. Probably due to technical reasons, few such algorithms have been developed until now. However, both numerical analysis and computer algebra could potentially benefit from a successfull combination of these methods. Numerical computations can often be accelerated by decomposing numbers in parts of different orders of magnitude. Symbolic computations can often be accelerated if we dispose of some numerical information. For instance, if we have a positive lower bound of a function on some domain, we know that the function does not vanish on this domain. We expect that such combinations of methods will become more and more frequent in the future.

Acknowledgement

We would like to thank the referees for many detailed corrections and suggestions.

6 Bibliography

[CavPr 78] B.F. Caviness, M.J. Prelle. A note on algebraic independence of logarithmic and exponential constants. *SIGSAM Bulletin 12, (p 18-20).*

[Ec 92] J. Ecalle. *Introduction aux fonctions analysables et preuve constructive de la conjecture de Dulac.* Hermann, collection: Actualités mathématiques.

[GeGo 88] K.O. Geddes, G.H. Gonnet. A new algorithm for computing symbolic limits using hierarchical series. *Proc. ISSAC 1988, Lecture notes in computer science 358, (p 490-495).*

[GoGr 92] G.H. Gonnet, D. Gruntz. Limit computation in computer algebra. *Report technique 187 du ETH, Zürich.*

[LLL 82] A.K. Lenstra, H.W. Lenstra, L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann. 261 (p 515-534).*

[Rich 92] D. Richardson. The elementary constant problem. *Proc. ISSAC 92 (p 108-116).*

[Rich 94] D. Richardson. How to recognize zero. *Preprint, University of Bath, England.*

[Sal 91] B. Salvy. Asymptotique automatique et fonctions génératrices. *PhD. Thesis, Ecole Polytechnique (Appendix B), France.*

[Sh 89] J. Shackell. A differential-equations approach to functional equivalence. *Proc. ISSAC 89, Portland, Oregon, A.C.M., New York, (p 7-10).*

[Sh 90] J. Shackell. Growth estimates for exp-log functions. *Journal of symbolic computation 10 (p 611-632).*

[VdH 94a] J. van der Hoeven. General algorithms in asymptotics I, Gonnet and Gruntz's algorithm. *Research report LIX/RR/94/10, Ecole Polytechnique, France.*

[VdH 94b] J. van der Hoeven. General algorithms in asymptotics II, Common operations. *Research report LIX/RR/94/10, Ecole Polytechnique, France.*