Computation of the error function erf in arbitrary precision with correct rounding *

S. Chevillard[1], N. Revol[2]
[1]ENS Lyon, [2]INRIA
Université de Lyon
LIP (CNRS-ENSL-INRIA-UCBL), ENS Lyon, 69364 Lyon Cedex 07, France
Sylvain.Chevillard@ens-lyon.fr, Nathalie.Revol@ens-lyon.fr

Abstract

In this paper, the computation of $\mathrm{erf}(x)$ *in arbitrary precision is detailed. A feature of our implementation is correct rounding: the returned result is the exact result (as if it were computed with infinite precision) rounded according to the specified rounding mode. The four rounding modes given in the IEEE-754 standard for floating-point arithmetic are provided. The algorithm that computes the correctly rounded value of* $\mathrm{erf}(x)$ *for any argument* x *is detailed in this paper. In particular, the choice of the approximation formula, the determination of the order of truncation and of the computing precision are presented. The evaluation formula is written as a partially expanded expression: we explain why it improves the performances in practice. Finally, timings on some experiments are given, and the implementation of the complementary error function* erfc *is then sketched.*

Keywords. Error function, complementary error function, floating-point arithmetic, arbitrary precision, adaptation of the computing precision, correct rounding.

Erratum. A typo has been discovered in Equation (4) in the version of this article published in *Proceedings, 8th Conference on Real Numbers and Computers*, pages 27-36, July 2008. The present document corrects this typo compared to the published version. The authors wish to thank Roman Sirotin for pointing them this mistake.

1 Introduction

The goal of this work is to compute the error function erf using arbitrary precision floating-point arithmetic and to deliver the correctly rounded result. Correct rounding means that the returned result is the exact result (as if it were computed with infinite precision) rounded according to the specified rounding mode. The four rounding modes given in the IEEE-754 standard for floating-point arithmetic, namely rounding to nearest (even), to $+\infty$, to $-\infty$ and

*This work has been realized with the financial help of the ANR EVA-Flo BLAN06-2_135670 2006 project.

to 0 are provided. In particular, having these rounding modes pave the path to an implementation of arbitrary precision interval arithmetic.

The method presented here exhibits two main features. First, the algorithm is completely detailed. Second, remarks on the practical complexity of each kind of operations lead to an evaluation scheme which yields a dramatic reduction of the computing time. The main issue which remains unaddressed is the evaluation of the complementary error function erfc and its use for the evaluation of erf with large arguments.

The error functions are widely used in statistics: the probability that a Gaussian random variable X takes values between $m - a$ and $m + a$ is given by $\mathrm{erf}\left(a/(\sigma\sqrt{2})\right)$, where m is the mean and σ is the standard deviation of the distribution for X. The error function also appears in the solution of the diffusion equation in specific configurations and in other heat transfer problems. For more information on the error function and the complementary error function, see for instance [13, 19].

Various implementations of the error functions can be found [5, 6, 7, 15], but none of the quoted implementations returns the correctly rounded result. Most use the standard floating-point single and double precisions defined by the IEEE-754 standard for floating-point arithmetic [10]. Even if the quality of the approximation can be quantified (by an upper bound on the error), none of the above implementations guarantees the correct rounding of the result. Much less implementations are available for arbitrary precision. Maple and Mathematica evaluate the error functions in arbitrary precision. However, Maple suffers from two problems: it does not guarantee the correct rounding of the result, and it is not even clear what the accuracy of the returned result is. Mathematica offers for most functions an indication of the accuracy of the results (based on first order approximations, *i.e.* not rigorous ones), cf. [18]. The MPFUN package [2] is better specified [3]: if 2^{-n} is the required accuracy, the (relative, except when the result is 0) error between the computed result and the exact value is $O(2^{-n})$. The goal of the DLMF project (*Digital Library of Mathematical Functions*) is to develop a replacement of Abramowitz and Stegun's *Handbook of Mathematical Functions* [1], cf. `http://dlmf.nist.gov/`. Unfortunately, it will not provide an implementation for the evaluation of special functions. The software accompanying the book [8] is not yet available.
The MPFR library (*Multiple Precision Floating-point Reliable library*, [16]) is a library for arbitrary precision floating-point arithmetic with correct rounding. We adopt their motivation for correct rounding: "*As a consequence, applications using such a library inherit the same nice properties as programs using IEEE 754–portability, well-defined semantics, ability to design robust programs and prove their correctness–with no significant slowdown on average, with respect to multiple-precision libraries with ill-defined semantics.*" [16]. MPFR includes several special functions (Gamma, Zeta). Simultaneous with our work, the error function has been implemented in MPFR as `mpfr_erf`. We will compare the implementation of MPFR with ours in Section 4.

We use the MPFR library for our implementation: we use the arbitrary precision arithmetic and algebraic operations and the π constant.

Our algorithm computes an approximation of $\mathrm{erf}(x)$ with an absolute error of ε (we will explain later how we choose ε). This gives us a number y such that $\mathrm{erf}(x) \in [y - \varepsilon, y + \varepsilon]$. In order to know if rounding y in a direction r leads to the same result z as rounding the exact

value $\mathrm{erf}(x)$ in direction r, it suffices to test that $y - \varepsilon$ and $y + \varepsilon$ rounded in direction r are equal to z. This is achieved by a function `can_round`.

The scheme of an algorithm that returns the correctly rounded evaluation of a function f on x with rounding mode r and precision p is thus the following:

1. approximate $f(x)$ with computing precision q (the working precision q is determined by the algorithm, it is larger than the target precision p) with error $\leq \varepsilon$;
2. if `can_round`$(f(x), \varepsilon, r, p)$ then return the correctly rounded value of $f(x)$ in the direction r and precision p;
3. otherwise increase the working precision q, decrease ε and try again.

The error function erf and the complementary error function erfc are introduced in the next Section. Some formulas and properties are given; they are used either to derive an algorithm for the evaluation of erf and erfc or to simplify some cases. Then our algorithm to evaluate erf and to return the correctly rounded result is presented in Section 3. Our implementation is compared to Maple and to the `mpfr_erf` function of MPFR on some typical examples, in Section 4: for large precisions and small arguments, computing times are significantly reduced. Finally, a list of desirable improvements is given in Conclusion.

2 The erf and erfc functions

2.1 Definitions

The error function erf and the complementary error function erfc are defined as:

$$\mathrm{erf} \ : \ x \mapsto \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t, \qquad \mathrm{erfc}\, x \ : \ x \mapsto 1 - \mathrm{erf}\, x = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} \mathrm{d}t. \tag{1}$$

The normalisation factor $2/\sqrt{\pi}$ ensures that erf defines a probability density function.

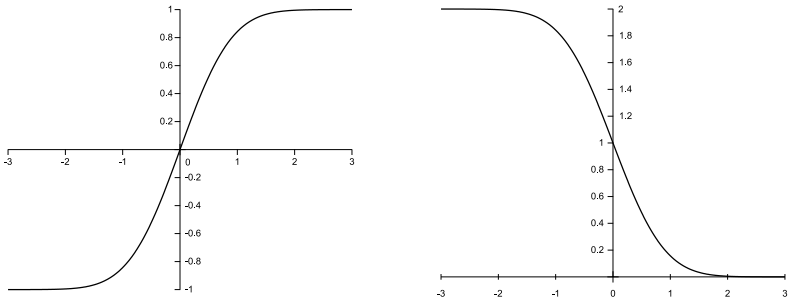

Figure 1. Graph of the erf **and** erfc **functions.**

2.2 Properties

- The erf function is odd: this enables us to consider only nonnegative arguments.
- The value of erf in 0 is $\mathrm{erf}(0) = 0$. Its limit when x tends to $+\infty$ is 1.

- Unfortunately, no properties similar to $\sin(x + 2\pi) = \sin x$ or $\log(x^2) = 2 \log x$ hold for the error functions. Such properties prove extremely useful for the evaluation of elementary functions. They allow to reduce the domain to a small interval. Such *range reduction* is not possible for the erf and erfc functions.

2.3 Approximation formulas

Among the possible formulas that can be used to approximate the erf and erfc functions, the book [1, pp. 297-298] gives the integral representations given above (eq. (7.1.1) and (7.1.2) in the book), series expansions (eq. (7.1.5) and (7.1.6)) for erf among which (eq. (7.1.5)):

$$\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{+\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)}, \tag{2}$$

an asymptotic expansion for erfc (eq. (7.1.23)), a continued fraction for erfc (eq. (7.1.14)) and finally an enclosure for erfc (eq. (7.1.13)) that is intensively used by MPFR for large values of x and small precisions.

3 Algorithm for the erf function

Several approaches could be considered to evaluate $\operatorname{erf} x$ where x is a given floating-point number in arbitrary precision. For instance, a numerical quadrature could be performed [9], using the integral representation (1). This implies a large number of evaluations of the exponential function, which is costly: each evaluation implies the evaluation of a sum of the kind given in the power series (2). Either this sum is evaluated with correct rounding (as in `mpfr_exp`) and such an accurate evaluation is costly, or it is approximated less accurately but then the corresponding error must be carefully taken into account in the total error bound.

Determining the best minimax polynomial approximation is not considered when the precision is variable. When the precision is fixed (single or double IEEE-754 precision), then this polynomial has to be determined only once, but when the precision varies this polynomial has to be recomputed for each precision and this computation is costly. Furthermore, the determination of the approximating polynomial involves the evaluation of the approximated function.

Another solution is to use a Taylor expansion. An advantage of using equation (2) is that it involves only rational calculations, whereas the other expansion given by formula (7.1.6) in [1], with a term e^{-x^2} in factor of the series, requires one evaluation of the exponential function, which is costly. A second advantage of equation (2) is that it is an alternate series and that the remainder of a truncated alternate series is extremely easy to bound: the first neglected term directly provides a bound. On the contrary, the truncation error is not obvious to derive for other expansions. Furthermore, this first neglected term (and thus, the error bound) tends rapidly to 0 as n tends to ∞, at least for small values of x. However, this term of order n can be large when x is large, at least for relatively small values of n. This means that to get a small enough denominator that compensates for the large numerator, it may thus be necessary to truncate after a large number N of terms. Another main drawback of using equation (2) is that summing an alternate series usually implies having cancellation. On the opposite, other

power series should exhibit no problem of numerical stability, since only positive terms are added. We nevertheless decided upon using equation (2) and we will show how to circumvent this difficulty in Section 3.3.

Several problems must then be solved, in order to derive our algorithm:

- at which number N of terms should we truncate?
- what is an upper bound on the remainder, *i.e.* on the error due to the truncation?
- in which order should the sum be computed in order to minimise the effect of the roundoff errors?
- how should the operations be arranged in order to decrease the computing time?
- what is an upper bound on these roundoff errors?
- what computing precision should be used?
- how can the computed result be correctly rounded?

In the following, these questions will receive an answer.

3.1 Truncation order and truncation error

Let us denote by $a_n(x) = \left((-1)^n x^{2n+1}\right) / \left(n!(2n+1)\right)$ and by $S_N(x)$ the truncated sum $\sum_{n=0}^{N} a_n(x)$. The term $|a_n(x)|$ is either decreasing with n if x is small enough ($x \leq \sqrt{3}$), or it begins by increasing, starting from the value $x > \sqrt{3}$ and then it decreases. The proof of this result can be found in [4].

The series is an alternating series. Thus, if N is such that the sequence $(|a_k(x)|)_{k \geq N}$ decreases, it is well known that the truncation error satisfies

$$\forall n \geq N, \left| \operatorname{erf} x - \frac{2}{\sqrt{\pi}} S_n(x) \right| \leq \frac{2}{\sqrt{\pi}} |a_{n+1}(x)|.$$

Hence, if $\varepsilon < 2\sqrt{3}/\sqrt{\pi}$ and if $2/\sqrt{\pi}|a_{N+1}| \leq \varepsilon$, $S_N(x)$ is an approximation of $\operatorname{erf}(x)$ with an absolute error less or equal to ε.

Firstly, if $x \geq 1$ then the exponent of the floating-point representation of $\operatorname{erf} x$ is 0. For smaller x, x is a rough approximation of $\operatorname{erf} x$: we use this value to get an estimation e of the exponent of the floating-point representation of $\operatorname{erf} x$.

If the target precision for the floating-point approximation of $\operatorname{erf} x$ is p bits, we truncate the expansion series for $\operatorname{erf} x$ at an order N that satisfies $2/\sqrt{\pi}|a_{N+1}(x)| \leq 2^{e-p-8}$, *i.e.* we choose $\varepsilon = 2^{e-p-8}$ (note that $\varepsilon < 2\sqrt{3}/\sqrt{\pi}$ for $p \geq 9$. If p is too small, we use a default value).

The 8 extra bits are for safety, they are expected to enable us to round correctly the computed result. We must also take care of the roundoff errors implied by the floating-point summation: this will be discussed later.

To determine such an order N, we simply compute iteratively the consecutive $2/\sqrt{\pi}|a_k(x)|$, $k = 0, 1, \cdots$ until $2/\sqrt{\pi}|a_k(x)| < 2^{e-p-8}$, using a small computing precision. π and $\sqrt{\pi}$ are computed downwards. Then the sequence is computed with rounding upwards. This way, when the process stops, we are sure that $2/\sqrt{\pi}|a_k(x)|$ is actually smaller than 2^{e-p-8}.

For small target precisions p, we use crude majorations to get an inequality between $|a_n|$ and $\lceil \log_2 n \rceil = \alpha$ and we try every α in sequence, until $n = 2^\alpha - 1$ satisfies this inequality. Thus, we try less values and we reduce the time of pre-computations.

3.2 Upper bound on the roundoff error of the summands

Using Higham's notation [11, Chapter 3] and intermediate lemmas [4], we obtained the following bound.

Theorem

If $5\lfloor N/2 \rfloor u < 1$, with $u = \text{ulp}(1) = 1^+ - 1$ where 1^+ is the smallest floating-point number strictly larger than 1 (using the current computing precision), then the absolute error between $2/\sqrt{\pi} S_N(x)$ and its computed value $2/\sqrt{\pi}\widehat{S_N}(x)$ satisfies

$$\left| \frac{2}{\sqrt{\pi}} S_N(x) - \frac{2}{\sqrt{\pi}} \widehat{S_N}(x) \right| \leq \frac{2}{\sqrt{\pi}} \frac{e^{x^2} - 1}{x} \gamma_{5\frac{N}{2}} \tag{3}$$

where $\gamma_k = (ku)/(1 - ku)$.

More refined majorations (not yet publicly available) enable us to reduce this error. In particular, much more elaborated calculations yield an error bound which is close to the running error bound described in Section 3.6.

3.3 Summation order

In [11, Chapter 4], N. Higham devotes a complete chapter to the summation problem. The question is: in which order should k summands $s_1, \ldots, s_k$ be added in floating-point arithmetic to minimise the roundoff error? When all summands are nonnegative, the recommended order is the increasing order on the added values. In our case, summands are of alternating signs. To reduce the problem to the sum of nonnegative terms, a first idea could be to add odd terms on the one hand, even terms on the other hand, each sum using the increasing order of absolute values. However, this strategy does not work because of heavy cancellations. Our solution consists in grouping the summands by pairs of two consecutive terms: if N is odd,

$$\begin{aligned} S_N(x) &= \textstyle\sum_{n=0}^{N} a_n(x) = \sum_{k=0}^{\lceil N/2 \rceil - 1} \left(a_{2k}(x) + a_{2k+1}(x) \right) \\ &= x \textstyle\sum_{k=0}^{\lceil N/2 \rceil - 1} \frac{x^{4k}}{(2k)!} \left(\frac{1}{4k+1} - \frac{x^2}{(2k+1)\times(4k+3)} \right). \end{aligned} \tag{4}$$

In this case, we do not incur cancellation because the subtracted term is much smaller. Depending on the value of x, the first terms may not be nonnegative, but the last terms are positive and are decreasing.

Our summation algorithm proceeds with k decreasing from $\lceil N/2 \rceil - 1$ down to 0. It uses Horner's scheme. The advantages of this strategy, compared to a summation by increasing order of indices, are threefold: the accuracy is good since the computation is done starting from the last (and smallest) terms of a sum of positive non-increasing terms, it is possible to compute a running error bound that is more accurate than the a priori error bound (cf. Section 3.6 and [4]). Finally the use of the Horner scheme yields a reduced number of operations.

However, for small precisions and small values of x it is preferable to avoid pre-computations and sum directly the terms by increasing values of n.

3.4 Summation order: improving the practical performances for large precisions

A careful analysis of the number of operations performed by the Horner scheme applied to the sum given in equation (4) shows that this evaluation scheme requires, roughly speaking,

$N/2$ multiplications, N additions and $3/2N$ divisions by an integer. For large precisions, additions are much cheaper than multiplications and divisions. Moreover, even if divisions have in general the same (asymptotic) complexity as multiplications, the situation is not the same for divisions by a (small) integer. In this case the division is much faster than the multiplication, and even the more so as precision becomes large. That is why the cost of the previous method is approximately $N/2$ times the cost of a high precision multiplication.

We can carry this principle further (described as *concurrent series* summation in [17]). Let assume L is an integer that divides N. If we group the terms of the series L by L (in consecutive order) in order to form N/L groups, then the overall complexity of the evaluation scheme remains dominated by the number of multiplications. This number is N/L for this new Horner scheme, plus L for the pre-computations of the first L powers of x^2, used inside each group. The minimum number of operations is $2\sqrt{N}$ and is obtained for $L \simeq \sqrt{N}$.

3.5 Choice of the computing precision

In order to get a result with p correct bits, one has to perform intermediate computations with $q \geq p$ bits of precision. We choose to have a roundoff error of the same order ε as the truncation error, *i.e.* $\varepsilon = 2^{e-p-8}$ where p is the target precision. Replacing ε by this value and u by 2^{1-q} in equation (3) enables us to deduce the required precision q for the intermediate computations. More precisely, q is the smallest integer such that the roundoff error is less or equal to the truncation error.

More accurate error bounds have been developed both for the case of small argument x and small precision, and for the case of large precision, where the more elaborated Horner scheme presented in Section 3.4 is used. For lack of space, we can only refer the reader to the research report developing these error analyses (to appear).

3.6 Rounding the computed result

We have derived an a priori bound on the error between the computed value $\hat{y}$ and the exact value $y = \operatorname{erf} x$: this bound is 2ε.

It is possible to refine this a priori bound by estimating the error, during the computations. The roundoff error given in (3) is overestimated: in particular (3) does not take into account the order of the summation. We adapted the computation of running error bound explained in [11, Chapter 5] for Horner's scheme to our problem, to get a new error bound, cf. [4] for details.

We can then invoke the `can_round` function with the computed value $\hat{y}$ that approximates $y = \operatorname{erf} x$, this new error bound and the target precision p to determine if rounding $\hat{y}$ in the required direction and in precision p is equivalent to rounding $y = \operatorname{erf} x$. If the answer is negative, we restart this process with a larger truncation order $N' > N$ and a larger computing precision $q' > q$, cf. Section 3.8.

One remaining question is whether this process will always terminate. It is more generally known as the Table Maker's Dilemma [14]. To our knowledge, there exists no theorem stating that our process will eventually stop, *i.e.* stating that there is no floating-point value x

(except $x = 0$) and no precision t such that $\operatorname{erf} x$ is exactly a floating-point value using this precision t. To prevent our implementation from looping indefinitely, we may decide to stop it when the required error bound is too small. Obviously, if the user chooses to do this, the implementation will not guarantee correct rounding any more.

3.7 How to avoid costly computations when x is large

When x is large, the binary representation of $\operatorname{erf}(x)$ has the form

$$\operatorname{erf}(x) = 0.\underbrace{11\ldots11}_{k} b_1 b_2 b_3 \ldots$$

The interesting bits are $b_1 b_2 b_3 \ldots$. Thus computing $\operatorname{erf}(x)$ directly may be a loss of time. For instance, if the required precision is $2k$ bits, a direct computation will spend a lot of time to compute the first k bits (that are all equal to 1) before computing the interesting last k bits.

A strategy would be to write an algorithm for computing directly $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ for large values of x. Thus, $2k$ bits of $\operatorname{erf}(x)$ would be obtained by computing approximately k bits of $\operatorname{erfc}(x)$ and performing only the subtraction $1 - \operatorname{erfc}(x)$ in high precision.

We did not implement this strategy yet. See Section 5 where we explain what we are currently doing in this direction.

3.8 Adaptation of the computing precision

Kreinovich and Rump [12] established an optimal strategy to automatically adapt the computing precision to the computational needs. The strategy consists in choosing a new precision (and, in our case, a new truncation index) such that the time of the computation using the new precision is twice the time of the previous computation. When the time of the multiplication of two q-bits integers is superlinear (and subquadratic), this yields to multiply by $\sqrt{2}$ the order of truncation and the computing precision. In [12], it is proven that this strategy yields the smallest overhead factor, which is 4, compared to the computational time when the best computing precision were known in advance and used.

This is not the strategy adopted by MPFR for the implementation of most of its elementary or special functions: a probabilistic assumption is made, namely that each extra bit of precision divides by 2 the risk of failure. We do not have such probabilistic assumption.

4 Experimental results

We compare the computing times of Maple, MPFR `mpfr_erf` and our implementation of erf on some typical values of x (small, medium and large) and various precisions. Results are given in Table 1. Our experiments were conducted on a Intel Pentium 4, 3GHz, 2048KB cache, with Maple 10, gcc 4.1.2, GMP 4.2.1 and MPFR 2.3.1. The precision p is in bits. For Maple (that works with a decimal format) we set the variable `Digits` to $\lfloor 0.301 \cdot p \rfloor$.

In each line the best time is written in bold. Times are given in milliseconds.

For small precisions and small values of x, our implementation uses basically the same algorithm as MPFR: a direct evaluation of the series without pre-computing anything. Hence our timings are close to the timings of MPFR in these cases.

When $x > 1.5$ and $p > 500$, our implementation spends some time in computing the order of truncation N and an accurate precision q as described before. This lets us use the technique of

Table 1. Comparison of Maple, MPFR and our implementation

p	x	`mpfr_erf` 2.3.0	Maple 10	our erf
100	$\pi/100$	0.016 ms	0.24 ms	**0.010 ms**
100	π	**0.10 ms**	0.33 ms	0.17 ms
100	$2 \cdot \pi$	**0.25 ms**	0.45 ms	0.46 ms
100	$10 \cdot \pi$	$<$ **0.001 ms**	0.015 ms	61 ms
1 000	$\pi/100$	0.6 ms	1.4 ms	**0.25 ms**
1 000	π	2.2 ms	2.9 ms	**1.0 ms**
1 000	$2 \cdot \pi$	3.6 ms	4.1 ms	**2.3 ms**
1 000	$10 \cdot \pi$	$<$ **0.001 ms**	0.02 ms	92 ms
10 000	$\pi/100$	82 ms	120 ms	**10 ms**
10 000	π	230 ms	350 ms	**27 ms**
10 000	$2 \cdot \pi$	320 ms	480 ms	**49 ms**
10 000	$10 \cdot \pi$	1 100 ms	9 900 ms	**500 ms**
100 000	$\pi/100$	19 000 ms	33 000 ms	**940 ms**
100 000	π	45 000 ms	81 000 ms	**2 000 ms**
100 000	$2 \cdot \pi$	56 000 ms	101 000 ms	**2 500 ms**
100 000	$10 \cdot \pi$	116 000 ms	1 710 000 ms	**9 600 ms**

grouping terms by blocks of $\sqrt{N}$ as described in Section 3.4. In these cases our implementation is the most efficient one: the reduction of the number of multiplications really pays, since the cost of each multiplication is costly for large precision.

For large values of x and relatively small precisions, $\operatorname{erf} x$ is so close to 1 that the answer can be given almost instantaneously. Our implementation is slower than the other ones because its computations may be somewhat too involved. We plan to implement the function erfc with an efficient algorithm for large values of x. Such an algorithm should enable us to compute $\operatorname{erf}(x) = 1 - \operatorname{erfc}(x)$ for large values of x by computing $\operatorname{erfc}(x)$ with much less precision.

5 Conclusion and improvements

This work is a first step towards the evaluation of the error functions in arbitrary precisions with correct rounding. Some improvements could yield a better efficiency. A first improvement lies in the complete implementation of erfc as well. We tried the use of continued fraction, because it was easier for us to bound the truncation error, but Brent in [3] and MPFR prefer the use of the asymptotic series for performance reasons. A second improvement lies in a joint use of erf and erfc, depending on small or large arguments. It is preferable to evaluate $\operatorname{erf} x$ when x is small and possibly to compute $\operatorname{erfc} x$ as $1 - \operatorname{erf} x$, since the relevant information lies in $\operatorname{erf} x$. Conversely, it is preferable to evaluate $\operatorname{erfc} x$ when x is large. The computing precision, and consequently the computing time, would be reduced.

The problem of infinite loop of the algorithm should also be handled in a more satisfactory way. Any theoretical result on the Table Maker's Dilemma for the error functions is welcome. If it were known that this process halts, even with a huge (theoretical) computing precision, then this problem would vanish. Finally, overflows and underflows occurring during interme-

diate computations may happen: they should be handled with more care than in our current implementation.

Solving these problems involves a theoretical but also experimental study: thresholds between "small" and "large" precisions and "small" and "'large" arguments must be determined.

6 Acknowledgements

Our thanks go to C. Q. Lauter for his help in checking the proofs but even more for his help in understanding and improving the practical performances, to the referees for their pertinent remarks and to the program committee for giving us extra time to improve this paper.

References

[1] M. Abramowitz M., I. A. Stegun eds. *Handbook of Mathematical Functions.* Dover, New York. (1965, 1972) Originally published by National Bureau of Standards in 1964.

[2] R.P. Brent. *A Fortran Multiple-Precision Arithmetic Package.* ACM TOMS, vol 4, pp 57-70. (1978)

[3] R.P. Brent. *Unrestricted algorithms for elementary and special functions.* IFIP (Information Processing) 80, North-Holland, Amsterdam, pp 613-619. (1980)

[4] S. Chevillard. *Calcul de la fonction erf en précision arbitraire.* Bachelor's thesis, ENS Lyon. (2003) `http://enslyon.free.fr/rapports/info/Sylvain_Chevillard_1.ps.gz`

[5] W.J. Cody. *Rational Chebyshev Approximations for the Error Function.* Mathematics of Computation, vol 23, pp 631–637. (1969)

[6] W.J. Cody. *Performance evaluation of programs for the error and complementary error functions.* ACM TOMS, vol 16, pp 29–37. (1990)

[7] W.J. Cody. *Algorithm 715: SPECFUN - A Portable FORTRAN Package of Special Function Routines and Test Drivers.* ACM TOMS, vol 19, no 1, pp 22–32. (1993)

[8] A. Cuyt, V. Petersen, B. Verdonk, H. Waadeland, W.B. Jones. *Handbook of Continued Fractions for Special Functions.* Springer. (2008)

[9] A. Gil, J. Segura, N. Temme. *Computing special functions by using quadrature rules.* Numerical Algorithms, vol 33, pp 265–275. (2003)

[10] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard, Std 754-1985,* New York. (1985)

[11] N. Higham. *Accuracy and stability of numerical algorithms (2nd edition).* SIAM. (2002)

[12] V. Kreinovich, S. Rump. *Optimal adaptation of the computing precision.* Reliable Computing, vol 12, no 5, pp 365–369. (2006)

[13] N. Lebedev. *Special Functions and their Applications.* Prentice Hall. (1965)

[14] V. Lefèvre, J.-M. Muller, A. Tisserand. *The Table Maker's Dilemma.* IEEE Transactions on Computers, vol 47, no 11, pp 1235–1243. (1998)

[15] D. Lozier, F. Olver. *Numerical Evaluation of Special Functions.* in W. Gautschi ed., *Mathematics of Computation 1943-1993: A Half-Century of Computational Mathematics*, Proc. of Symp. in Applied Math. 48, AMS, pp 79–125. (1994) `http://math.nist.gov/nesf`.

[16] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann. *MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding.* ACM TOMS, vol 33, no 2, article 13. (2006)

[17] D.M. Smith. *Algorithm 693; a FORTRAN package for floating-point multiple-precision arithmetic.* ACM TOMS vol 17, no 2, pp 273–283. (1991)

[18] M. Sofroniou, G. Spaletta. *Precise numerical computations.* J. Logic and Algebraic Programming, vol 64, pp 113–134. (2005)

[19] N. Temme. *Special Functions.* John Wiley and Sons. (1996)