The functions erf and erfc computed with arbitrary precision

Sylvain Chevillard

Laboratoire de l'Informatique du Parallélisme

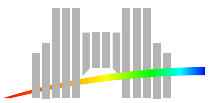
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

The functions erf *and* erfc *computed with arbitrary precision*

Sylvain Chevillard January 2009

Research Report N° RR2009-04

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



The functions erf and erfc computed with arbitrary precision

Sylvain Chevillard

January 2009

Abstract

The error function erf is a special function. It is widely used in statistical computations for instance, where it is also known as the standard normal cumulative probability. The complementary error function is defined as $\operatorname{erfc}(x) = \operatorname{erf}(x) - 1$.

In this paper, the computation of $\operatorname{erf}(x)$ and $\operatorname{erfc}(x)$ in arbitrary precision is detailed: our algorithms take as input a target precision t' and deliver approximate values of $\operatorname{erf}(x)$ or $\operatorname{erfc}(x)$ with a relative error bounded by $2^{-t'}$.

We study three different algorithms for evaluating erf and erfc. These algorithms are completely detailed. In particular, the determination of the order of truncation, the analysis of roundoff errors and the way of choosing the working precision are presented.

We implemented the three algorithms and studied experimentally what is the best algorithm to use in function of the point x and the target precision t'.

Keywords: Error function, complementary error function, erf, erfc, floating-point arithmetic, arbitrary precision, multiple precision

Résumé

La fonction d'erreur erf est une fonction spéciale. Elle est couramment utilisée en statistiques par exemple. La fonction d'erreur complémentaire est définie comme $\operatorname{erfc}(x) = \operatorname{erf}(x) - 1$.

Dans cet article, nous détaillons l'évaluation de $\operatorname{erf}(x)$ et $\operatorname{erfc}(x)$ en précision arbitraire : nos algorithmes prennent en entrée une précision cible t' et fournissent en retour une valeur approchée de $\operatorname{erf}(x)$ ou $\operatorname{erfc}(x)$ avec une erreur relative bornée par $2^{-t'}$.

Nous étudions trois algorithmes différents pour évaluer erf et erfc. Ces algorithmes sont expliqués en détails. En particulier, nous décrivons comment déterminer l'ordre de troncature et la précision intermédiaire de travail. De plus, nous fournissons une analyse rigoureuse des erreurs d'arrondis.

Nous avons implémenté les trois algorithmes ; nous concluons par une étude expérimentale qui montre quel algorithme est le plus rapide en fonction du point x et de la précision cible t'.

Mots-clés: fonctions d'erreur, erf, erfc, arithmétique flottante, précision arbitraire, multiprécision

1 Introduction

The error function, generally denoted by erf is defined as

$$\operatorname{erf} : x \mapsto \frac{2}{\sqrt{\pi}} \int_0^x e^{-v^2} \, \mathrm{d}v.$$

Sometimes, it is called the *probability integral*, in which case, erf denotes the integral itself without the normalisation factor $2/\sqrt{\pi}$ [8]. The complementary error function denoted by erfc is defined as $\operatorname{erfc} = 1 - \operatorname{erf}$. These two functions are defined and analytic on the whole complex plane. Nevertheless we will consider them only on the real line herein.

These functions are important because they are encountered in many branches of applied mathematics, in particular probability theory. Namely, if X is a gaussian random variable with mean 0 and standard deviation $1/\sqrt{2}$, the probability $P(-x \leq X \leq x)$ is equal to $\operatorname{erf}(x)$ (Figure 1). See [8] for instance for other applications.

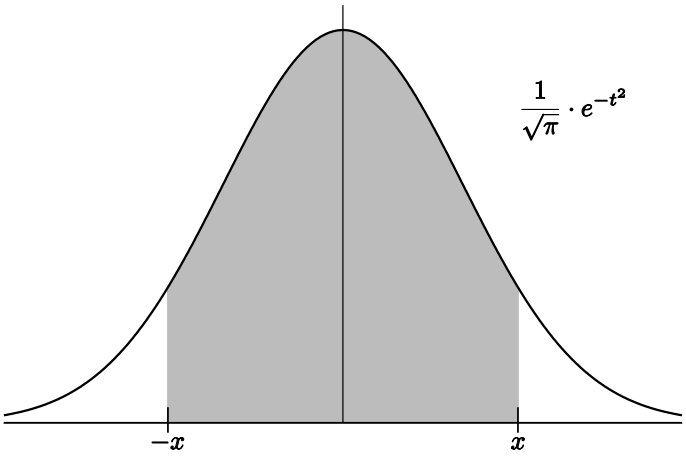


Figure 1: $\operatorname{erf}(x)$ is the probability that a certain gaussian random variable lies in $[-x, x]$

In this article we describe the numerical implementation of erf and erfc in floating-point arithmetic with arbitrary precision. Such an arbitrary precision implementation is useful in several cases including:

- when highly accurate values of the functions are needed;
- for testing the quality of lower precision libraries;
- for building good approximating polynomials with a given accuracy.

A good overview of the applications of arbitrary precision is given in the introduction of [4].

We approximate the real numbers by floating-point numbers with arbitrary precision with radix 2; more precisely: let $t \in \mathbb{N}^\star$, the set of floating-point numbers with precision t is the set

$$\mathcal{F}_t = \{0\} \cup \left\{ \pm \frac{m}{2^t} \cdot 2^e,\ e \in \mathbb{Z},\ m \in [\![2^{t-1},\ 2^t - 1]\!] \right\} \quad \text{where} \quad [\![2^{t-1},\ 2^t - 1]\!] = [2^{t-1},\ 2^t - 1] \cap \mathbb{Z}.$$

The integer e is called the exponent of the floating-point number. For practical reasons, it is usually bounded in the implementation. However, in general, in multiple precision libraries,

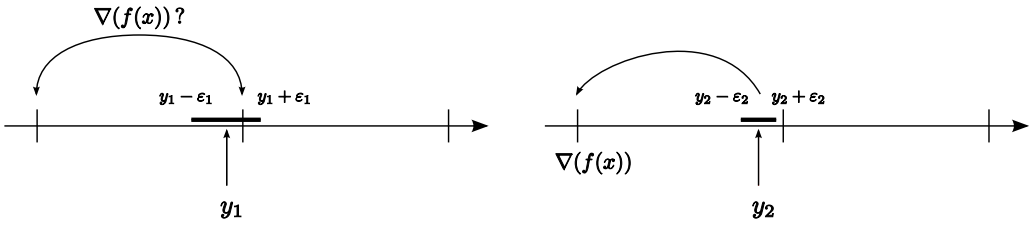
its range is extremely large (typically $e \in [-2^{32}; 2^{32}]$) and may be considered as practically unbounded. We will assume in this paper that e is unbounded. The number $m/2^t$ is called the mantissa and is more conveniently written as $0.1b_2 \dots b_t$ where $b_2, \dots, b_t$ are the bits of its binary representation. The mantissa lies in the interval $[1/2, 1)$: this is the convention used by the library MPFR and we will adopt it here. Note that the IEEE-754 standard takes another convention and suppose that the mantissa lies in $[1, 2)$.

Our goal is the following: given t and t' in $\mathbb{N}^\star$ and $x \in \mathcal{F}_t$, compute a value y approximating $\operatorname{erf}(x)$ with a relative error less than $2^{-t'}$. More formally,

$$\exists \delta \in \mathbb{R},\ \operatorname{erf}(x) = y(1+\delta) \quad \text{where} \quad |\delta| \leq 2^{-t'}.$$

Arbitrary precision floating-point arithmetic is already implemented in several software tools and libraries. Let us cite Brent's historical Fortran package `MP` [3], Bailey's `Arprec` C++ library [2], the MPFR library [6], and the famous tools Mathematica and Maple. MPFR provides correct rounding of the functions: four rounding-modes are provided (rounding up, down, towards zeros, and to nearest) and the returned value y is the rounding of the exact value $\operatorname{erf}(x)$ to the target precision t'. Other libraries and tools usually only ensure that the relative error between y and the exact value is $O(2^{-t'})$.

Remark that MPFR may not finish on some inputs. Assume for instance that rounding downwards is used and that f is a function implemented in MPFR. Moreover, suppose that there exist t and t' and $x \in \mathcal{F}_t$ and $y \in \mathcal{F}_{t'}$ such that $y = f(x)$. Except if the case is known in advance and managed separately this will lead to an infinite loop. Indeed, in general MPFR uses Ziv' strategy [10]: an intermediate precision $t_1 \geq t'$ is chosen and an approximation y_1 of $f(x)$ is computed using working precision t_1. Simultaneously, a bound ε_1 is computed such that $y_1 \in [f(x) - \varepsilon_1,\ f(x) + \varepsilon_1]$. If $f(x) - \varepsilon_1$ and $f(x) + \varepsilon_1$ round to the same value in precision t', this value is to be returned. Otherwise, one cannot decide the value to be returned. A new precision $t_2 > t_1$ is chosen, a new approximation y_2 is computed, together with a bound ε_2. The test is performed again, etc. This process eventually stops if and only if $f(x)$ is not exactly representable with a finite number of bits. This is known as the Table Maker's Dilemma (TMD) and is illustrated in Figure 2.



(a) Rounding may not be decided (b) Rounding may be decided

Figure 2: Illustration of the Table Maker's Dilemma with rounding downwards

For functions such as exp, sin, log, etc. the list of exact cases is known (namely, $\exp(0) = 1$, $\sin(0) = 0$, $\log(1) = 0$, etc.). For erf and erfc, 0 is an obvious exact case, but there is no proof that other exact cases do not exist (though it is unlikely). On such unknown exact cases, MPFR would run indefinitely.

This is why we chose to provide an implementation that guarantees that the relative error is less than $2^{-t'}$. This is a stronger condition than asking for the error to be $O(2^{-t'})$ since it

gives an explicit bound on the error. Unlike MPFR, our implementation is proven to finish on all inputs.

The novelty of our work does not lie in the techniques used: we use classical formulas for approximating erf and erfc, we implement a summation technique proposed by Smith in 1989 [9] and we analyse the roundoff errors using classical techniques. The originality comes from a careful study of each approximation method. We compare them in order to have a very efficient final implementation. It may also be considered as a detailed example than can be used as a general scheme for the implementation of other functions.

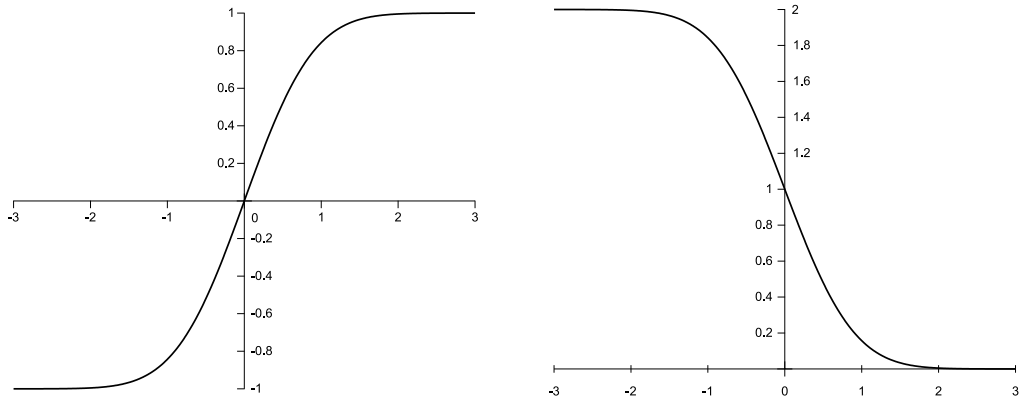
Section 2 of the paper is a general discussion about erf and erfc and the ways of computing them with arbitrary precision. Section 3 is devoted to some reminders on classical techniques required for performing the roundoff analysis. In Section 4 the algorithms are completely described and the roundoff analysis is detailed. Our implementation is written in C and built on top of MPFR. It is distributed under the GPL and freely available. Section 5 shows which algorithm is the best in function of the point x and the target precision t', based on experimental results.

2 General overview of the algorithms

It is easy to see that erf is odd and increasing. Thus we restrict to computing $\mathrm{erf}(x)$ when $x > 0$ without loss of generality. Except if it is explicitly mentioned, x will always be positive in the following. Moreover, $\mathrm{erf}(0) = 0$ and $\mathrm{erf}(x)$ approaches 1 as $x \to +\infty$. Thus for large x, the binary representation of $\mathrm{erf}(x)$ looks like

$$0.\underbrace{11\ldots11}_{\text{many 1s}}b_1b_2b_3\ldots$$

This is why, for large x, it is more convenient to consider the complementary error function $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$. The graphs of these two functions are represented in Figure 3.



(a) Graph of erf (b) Graph of erfc

Figure 3: Graphs of the functions erf and erfc

Among the formulas given in [1] we retain the following ones, suited for the computation in arbitrary precision (Equations 7.1.5, 7.1.6, 7.1.23 and 7.1.24 of [1]):

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

$$\operatorname{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{+\infty} \frac{(2x^2)^n}{1\cdot 3\cdot 5\cdots(2n+1)}, \tag{2}$$

$$\operatorname{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \left(1 + \sum_{n=1}^{N-1} (-1)^n \cdot \frac{1\cdot 3\cdot 5\cdots(2n-1)}{(2x^2)^n}\right) + \varepsilon_N^{(3)}(x) \tag{3}$$

$$\text{where } |\varepsilon_N^{(3)}(x)| \leq \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot \frac{1\cdot 3\cdot 5\cdots(2N-1)}{(2x^2)^N}. \tag{4}$$

Equation (1) is mostly interesting for small values of x. The series is alternating and the remainder is thus bounded by the first neglected term. The ratio between two consecutive terms is, roughly speaking, x^2/n. Thus, if $x < 1$, both x^2 and n contribute to reduce the magnitude of the term and the convergence is really fast. For larger values of x, the convergence is slower since only the division by n ensures that the term decreases. Though, the main problem with large arguments is not the speed of the convergence.

The main drawback of (1) for large arguments comes from the fact that the series is alternating: this implies cancellation of bits during the subtractions that occur in the computation. This leads to an important loss of accuracy. Hence, the computations must be performed with a much higher precision than the target precision. We will quantify this phenomenon in Section 4.3: as we will see, as x increases, the use of Equation (1) becomes quickly impractical.

Equation (2) does not exhibit the problem of cancellations. Its evaluation does not require much more precision than the target precision. However, the convergence is a bit slower. Besides, it requires to compute e^{-x^2} (the complexity of computing it is somewhat the same as computing erf itself).

Equation (3) gives a very efficient way of evaluating erfc and erf for large arguments. However $\varepsilon_N^{(3)}(x)$ cannot be made arbitrary small by increasing N (there is an optimal value reached at $\lfloor x^2 + 1/2 \rfloor$). If $\operatorname{erfc}(x)$ is to be computed with a bigger precision, one has to switch back to Equation (1) or (2).

2.1 Evaluation scheme

The three sums described above exhibit the same general structure: they are polynomials or series (in the variable x^2, $2x^2$ or $1/(2x^2)$) with coefficients given by a simple recurrence involving multiplications or divisions by integers.

Note that these integers will fit in a 32-bit or 64-bit machine integer. It is well known [3] that the multiplication (or division) of a t-bit floating-point number by a machine integer can be performed in time $O(t)$. This should be compared with the multiplication of two t-bit floating-point numbers which is $O(t\log(t)\log(\log(t)))$ asymptotically (and only $O(t^2)$ in practice for small precisions). Additions and subtractions of two t-bit floating-point numbers are also done in time $O(t)$. Hence the cost of an algorithm is generally mainly given by the number of high-precision multiplications.

We may take advantage of the structure of the sums involved here and reduce the numbers of high precision multiplications. This technique has been described by Smith [9] as a *concurrent series summation*. Let us assume that we want to evaluate the following polynomial in y:

$$S(y) = \alpha_0 + \alpha_0\alpha_1 \cdot y + \ldots + \alpha_0\alpha_1 \cdots \alpha_{N-1} \cdot y^{N-1}. \tag{5}$$

We suppose that α_i $(i = 0, \ldots, N-1)$ are small integers or inverse of integers. Hence, a multiplication by one of the α_i is fast. Let L be an integer parameter. For the sake of simplicity, we will assume that N is a multiple of L. The general case is easy to deduce from this particular case. Smith remarks that $S(y)$ may be expressed as follows:

$$\begin{array}{rcl@{\quad}l@{\quad}l@{\quad}l@{\quad}l@{\quad}l@{\quad}l}
S(y) & = & 1 & \cdot \Big(& \alpha_0 & + \alpha_0\cdots\alpha_L(y^L) & + \ldots + & \alpha_0\cdots\alpha_{N-L}(y^L)^{N/L-1} & \Big) \\
& + & y & \cdot \Big(& \alpha_0\alpha_1 & + \alpha_0\cdots\alpha_{L+1}(y^L) & + \ldots + & \alpha_0\cdots\alpha_{N-L+1}(y^L)^{N/L-1} & \Big) \\
& + & \ldots & & & & & & \\
& + & y^{L-1} & \cdot \Big(& \alpha_0\cdots\alpha_{L-1} & + \alpha_0\cdots\alpha_{2L-1}(y^L) & + \ldots + & \alpha_0\cdots\alpha_{N-1}(y^L)^{N/L-1} & \Big).
\end{array}$$

We denote by S_i the sum between the parentheses of the i-th line of this array: thus

$$S(y) = S_0 + S_1 \cdot y + \cdots + S_{L-1} \cdot y^{L-1}.$$

The sums S_i are computed concurrently: a variable is used to compute successively

$$\alpha_0, \quad \alpha_0\alpha_1, \quad \ldots \quad , \quad \alpha_0\cdots\alpha_{L-1}, \quad \alpha_0\cdots\alpha_L(y^L), \quad \text{etc.}$$

and the sums S_i are accumulated accordingly. The power y^L is computed once in the beginning in time $\log(L)$. The multiplications (or divisions) involved are all multiplications by integers, except the multiplications by y^L that occur $N/L - 1$ times.

Finally, the polynomial $S(y) = S_0 + S_1 \cdot y + \cdots + S_{L-1} \cdot y^{L-1}$ is evaluated by Horner's rule and requires $L - 1$ high-precision multiplications.

The complete algorithm requires $N/L + L - 2$ high-precision multiplications (and $O(N)$ additions and multiplications/divisions by small integers). The optimal value is obtained with $L \simeq \sqrt{N}$. The total cost is then approximately $2\sqrt{N}$ slow multiplications whereas a straightforward evaluation would lead to approximately N slow multiplications. Note that this method requires extra space to store the values S_i until they are used in the final Horner evaluation (Lt bits are needed). The algorithm is summed up in Figure Algorithm 1.

Let us do a final remark about this algorithm: consider the value of the variable acc just after the line 6 of the algorithm was performed. Note that $k = \lfloor k/L \rfloor L + i$. The variable acc is an approximation to $\alpha_0 \cdots \alpha_k (y^L)^{\lfloor k/L \rfloor}$. Therefore $(\mathsf{acc} \cdot y^i)$ is an approximation to the coefficient of order k of the polynomial $S(y)$. This means that N does not really need to be known precisely in advance: a test of the form *sum the terms until finding one whose absolute value is smaller than a given bound* can be used. Of course, $(\mathsf{acc} \cdot y^i)$ is only an approximation of the actual coefficient. If we are not careful enough, we might underestimate the absolute value and stop the summation too soon. Hence, the rounding mode should be chosen carefully when updating acc, in order to be sure to always get an upper-estimation of the absolute value.

3 Error analysis

Since the operations in Algorithm 1 are performed with floating-point arithmetic, they are not exact and roundoff errors must be taken into account. We have to choose carefully

```
   Input: y, L, N, t
   Output: the sum S(y) of Equation (5)
   /* each operation is performed in precision t                          */
 1 z ← power(y, L) ;                    /* obtained by binary exponentiation */
 2 S ← [0, ..., 0] ;                    /* array of L floating-point numbers */
 3 acc ← 1 ;
 4 i ← 0 ;                             /* indicates the S_i currently updated */
 5 for k ← 0 to N − 1 do
 6 |   acc ← acc ∗ α_k ;
 7 |   S[i] ← S[i] + acc ;
 8 |   if i = L − 1 then
 9 |   |   i ← 0 ;
10 |   |   acc ← acc ∗ z ;
11 |   else
12 |   |   i ← i + 1 ;
13 |   end
14 end
   /* now S(y) is evaluated from the S_i by Horner's rule                 */
15 R ← S[L − 1] ;
16 for i ← L − 2 downto 0 do
17 |   R ← S[i] + y ∗ R ;
18 end
19 return R;
```

Algorithm 1: ConcurrentSeries()

the precision t that is used for the computations in order to keep the roundoff errors small enough. Techniques make it possible to bound such roundoff errors rigorously. In his book *Accuracy and Stability of Numerical Algorithms* [7], Higham explains in great details what should be known in this domain. We recall a few facts here.

Definition 1. *If $x \in \mathbb{R}$, we denote by $\diamond(x)$ a rounding of x (i.e. x itself if x is a floating-point number and one of the two floating-point numbers enclosing x otherwise).*

If t denotes the current precision, the quantity $u = 2^{1-t}$ is called the unit roundoff.
We will use the convenient notation $z = z' \langle k \rangle$ for meaning that

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

Remark that for any k and k' such that $k' \geq k$, if we can write $z' = z \langle k \rangle$, we can also write $z' = z \langle k' \rangle$.

This notation corresponds to the accumulation of k successive relative errors. The following proposition justifies it.

Proposition 1. *For any $x \in \mathbb{R}$, there exists $\delta \in \mathbb{R}$, $|\delta| \leq u$ such that $\diamond(x) = x \cdot (1+\delta)$. If $\oplus$ denotes the correctly rounded addition, the following holds:*

$$\forall (x, y) \in \mathbb{R}^2,\ x \oplus y = \diamond(x+y) = (x+y)(1+\delta) \quad \text{for a given } |\delta| \leq u.$$

The same holds for the other correctly rounded operations $\ominus$, $\otimes$, $\oslash$, etc.

Let us do a complete analysis on a simple example. Consider a, b, c, d, e f six floating-point numbers. We want to compute $S = a \cdot b + c \cdot d \cdot e + f$. In practice, we may compute for instance $\widehat{S} = ((a \otimes b) \oplus ((c \otimes d) \otimes e)) \oplus f$. We can analyse the roundoff errors with the following simple argument:

$$\begin{array}{rcccccc}
\widehat{S} & = & (a \otimes b) & \oplus & ((c \otimes d) \otimes e) & \oplus & f \\
& = & (a \cdot b)\langle 1 \rangle & \oplus & (c \cdot d \cdot e)\langle 2 \rangle & \oplus & f \\
& = & (a \cdot b)\langle 2 \rangle & \oplus & (c \cdot d \cdot e)\langle 2 \rangle & \oplus & f\langle 2 \rangle \\
& = & \big((a \cdot b)\langle 2 \rangle & + & (c \cdot d \cdot e)\langle 2 \rangle\big)\langle 1 \rangle & \oplus & f\langle 2 \rangle \\
& = & \big((a \cdot b)\langle 3 \rangle & + & (c \cdot d \cdot e)\langle 3 \rangle\big) & \oplus & f\langle 2 \rangle \\
& = & (a \cdot b)\langle 4 \rangle & + & (c \cdot d \cdot e)\langle 4 \rangle & + & f\langle 4 \rangle .
\end{array}$$

We now need another proposition:

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

$$\exists \theta_k \in \mathbb{R},\ \text{such that } z' = z(1+\theta_k) \quad \text{with } |\theta_k| \leq \gamma_k = \frac{ku}{1-ku}.$$

In particular, as soon as $ku \leq 1/2$, $\quad \gamma_k \leq 2ku$.

Using this proposition, we can write $\widehat{S} = (a \cdot b)(1+\theta_4) + (c \cdot d \cdot e)(1+\theta'_4) + f(1+\theta''_4)$. Finally, we get

$$\left|\widehat{S} - S\right| \leq \gamma_4 \cdot (|a \cdot b| + |c \cdot d \cdot e| + |f|)\,.$$

Note the importance of $\widetilde{S} = |a \cdot b| + |c \cdot d \cdot e| + |f|$. If the terms of the sum are all nonnegative, $S = \widetilde{S}$ and the relative error for the computation of the sum is bounded by γ_4. If some terms are negative, the relative error is bounded by $\gamma_4 \widetilde{S}/S$. The ratio $\widetilde{S}/S$ may be extremely large: this corresponds to the phenomenon of *catastrophic cancellations* [5], when terms of the sum cancel with each other while the errors accumulate.

4 Practical implementation

4.1 General scheme

The implementation of the three formulas will follow the same general scheme. The inputs are the point $x \in \mathbb{R}$ where erf or erfc must be evaluated and the target relative error t'.

Whatever the formula we use, we have to truncate the sum at a certain order $N - 1$. Hence, we will have two errors of different natures:

- the approximation error: it comes from the fact that we ignore the remainder of the series (Equation (3) is not actually a series but $\varepsilon_N^{(3)}(x)$ plays the role of a remainder);
- the roundoff error: it comes from the accumulation of errors when using floating-point arithmetic for the evaluation of the finite sum.

The final relative error must be smaller that $2^{-t'}$, that is to say that the absolute error must be smaller than $2^{-t'} \cdot \mathrm{erf}(x)$ (or $2^{-t'} \cdot \mathrm{erfc}(x)$ for erfc). We split it into two equal parts and choose the truncation rank N and the working precision t in such a way that both absolute approximation error and roundoff error are smaller than $2^{-t'-1} \cdot \mathrm{erf}(x)$ (or $2^{-t'-1} \cdot \mathrm{erfc}(x)$).

The approximation error is controlled by the truncation rank. Assume that we know an upper-bound ε_N on the remainder and a positive lower bound $f(x)$ of $\mathrm{erf}(x)$ (or $\mathrm{erfc}(x)$). It is sufficient to find N such that $\varepsilon_N \leq 2^{-t'-1} \cdot f(x)$.

In a first step, we invert this formula and obtain a rough upper-estimation of N. This estimation does not need to be very accurate: we will use it to choose the parameter L of Algorithm 1 and to choose the working precision. When running Algorithm 1, we will loop over k until $\varepsilon_k \leq 2^{-t'-1} \cdot f(x)$ (ε_k will be easily expressed with the coefficient of order k of the series). This way, we will sum approximately the optimal number $N^\star$ of terms.

Suppose for instance that N is an upper-estimation of $N^\star$ by a factor 4. By choosing $L \simeq \sqrt{N} = \sqrt{4N^\star}$ we will eventually perform $N^\star/L + L \simeq 5\sqrt{N^\star}/2$ slow multiplications, which is not so far from the optimal. As we will see, only the logarithm of N is useful for choosing the working precision t. Hence, an upper-estimation by a factor 4 will only lead to use 2 extra bits, which is insignificant.

In a second time, we perform an error analysis of the algorithm (similar to the example presented in Section 3) and get a bound on the final roundoff error. It will typically be of the form

$$|S - \widehat{S}| \leq \gamma_{aN} \widetilde{S}$$

where a is an integer (the order of magnitude of a is 1 to 10 approximately), S is the exact sum, $\widehat{S}$ is the computed value and $\widetilde{S}$ is the sum of the absolute values. Therefore it is sufficient to choose t such that $(2aN)2^{1-t} \cdot \widetilde{S} \leq 2^{-t'-1} \cdot f(x)$ or equivalently

$$t \geq t' + 3 + \log_2(a) + \log_2(N) + \log_2(\widetilde{S}) - \log_2(f(x)).$$

4.2 Important technical results

Before giving the details of the implementation of Equations (1), (2) and (3), we need a few technical lemmas.

Lemma 1 (Propagation of errors through a square root)**.** *Let $k \in \mathbb{N}^\star$, z and z' two numbers such that we can write $z' = z\,\langle k\rangle$. Then we can write $\sqrt{z'} = \sqrt{z}\,\langle k\rangle$.*

Proof. Left to the reader. □

Lemma 2 (Propagation of errors through exp)**.** *Let $z \in \mathbb{R}$. We denote by E its exponent: $2^{E-1} \leq z < 2^E$. Let t be a precision. We note $y = z^2$ and we suppose that $\widehat{y} = \diamond(z^2)$, the operation being performed with a precision larger than $t + 2E$. Then*

$$e^{-\widehat{y}} = e^{-z^2}(1+\delta)^s \quad \text{where } |\delta| \leq 2^{1-t} \text{ and } s \in \{-1, 1\}.$$

In other words, we can write $e^{-\widehat{y}} = e^{-z^2}\,\langle 1\rangle$ in precision t.

Proof. Left to the reader. □

For bounding the remainder of the series, we will need to approximate $n!$. We use the following estimations:

Lemma 3 (Rough estimation of $n!$)**.** *The following inequalities hold for all $n \geq 1$:*

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq e\sqrt{n}\left(\frac{n}{e}\right)^n.$$

Proof. Consider the sequence defined for $n \geq 1$ by

$$u_n = \frac{\sqrt{2\pi n}(n/e)^n}{n!}.$$

We show that this sequence is increasing by considering u_{n+1}/u_n. It is well known (Stirling formula) that $u_n \to 1$ as $n \to +\infty$. Therefore, for all $n \geq 1$, $u_1 \leq u_n \leq 1$. This gives the result. □

The rough estimation of N will be obtained by inverting a certain relation. This relation involves the function $v \mapsto v\log_2(v)$. The following lemmas gives an estimation of the inverse of this function.

Lemma 4 (Inverse of $v\log_2(v)$)**.** *The function $v \mapsto v\log_2(v)$ is increasing for $v \geq 1/e$. We denote by φ its inverse: $\varphi : w \mapsto \varphi(w)$ defined for $w \geq -\log_2(e)/e$ and such that for all w, $\varphi(w)\log_2(\varphi(w)) = w$. The function φ is increasing.*

The following inequalities hold:

$$\begin{array}{lrcccl}
\text{if } w \in [-\log_2(e)/e,\, 0], & 2^{ew} & \leq & \varphi(w) & \leq & 2^w\,; \\
\text{if } w \in [0,\, 2], & 2^{w/2} & \leq & \varphi(w) & \leq & 2^{1/4}\cdot 2^{w/2}\,; \\
\text{if } w \geq 2, & w/\log_2(w) & \leq & \varphi(w) & \leq & 2w/\log_2(w).
\end{array}$$

Proof. Showing that $v \mapsto v\log_2(v)$ is increasing for $v \geq 1/e$ is easy and left to the reader. We only prove the second inequality. The others are proven using the same technique.

We denote $\varphi(w)$ by v for convenience. If $w \in [0,\, 2]$, it is easy to see that $v \in [1,\, 2]$. Now, since $\log_2(v) = w/v$, we get $w/2 \leq \log_2(v) \leq w$. Therefore

$$2^{w/2} \leq v \leq 2^w, \tag{6}$$

which gives the lower bound.

We can refine this identity: using again that $\log_2(v) = w/v$, we get

$$\frac{w}{2^w} \leq \log_2(v) \leq \frac{w}{2^{w/2}} \quad \text{and thus} \quad v \leq 2^{(w\cdot 2^{-w/2})}.$$

For finishing the proof, we only need to show that for any $w \in [0,\, 2]$, $\quad w \cdot 2^{-w/2} \leq 1/4 + w/2$.

The Taylor expansion of $2^{-w/2}$ is

$$2^{-w/2} = \sum_{n=0}^{+\infty} (-1)^n \frac{(w\ln(2))^n}{2^n \cdot n!}.$$

The series is alternating with a decreasing general term when $w \in [0,\, 2]$. Hence

$$2^{-w/2} \leq 1 - \frac{w\ln(2)}{2} + \frac{w^2\ln(2)^2}{8}.$$

From this inequation, we deduce $w \cdot 2^{-w/2} \leq 4/(27\ln(2)) \simeq 0.2137\ldots$. It finishes the proof.
Remark: the last inequality is proved using $v \leq v\log_2(v) \leq v^2$ for $v \geq 2$. We rewrite it $\sqrt{w} \leq v \leq w$ and then we apply the same technique. □

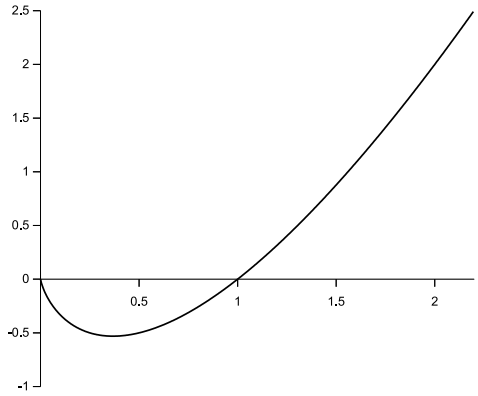


Figure 4: Graph of the function $v \mapsto v\log_2(v)$.

Lemma 5 (Inverse of $v\log_2(v)$, the other branch)**.** *The function $v \mapsto v\log_2(v)$ is decreasing for $0 \leq v \leq 1/e$. We denote by φ_2 its inverse: $\varphi_2 : w \mapsto \varphi_2(w)$ such that $\varphi_2(w)\log_2(\varphi_2(w)) = w$. The value $\varphi_2(w)$ is defined for $-\log_2(e)/e \leq w \leq 0$ and is decreasing.*

The following inequalities give an estimate of $\varphi_2(w)$:

$$\forall w \in \left[-\frac{\log_2(e)}{e},\, 0\right), \qquad \frac{1}{3}\cdot\frac{w}{\log_2(-w)} \;\leq\; \varphi_2(w) \;\leq\; \frac{w}{\log_2(-w)}.$$

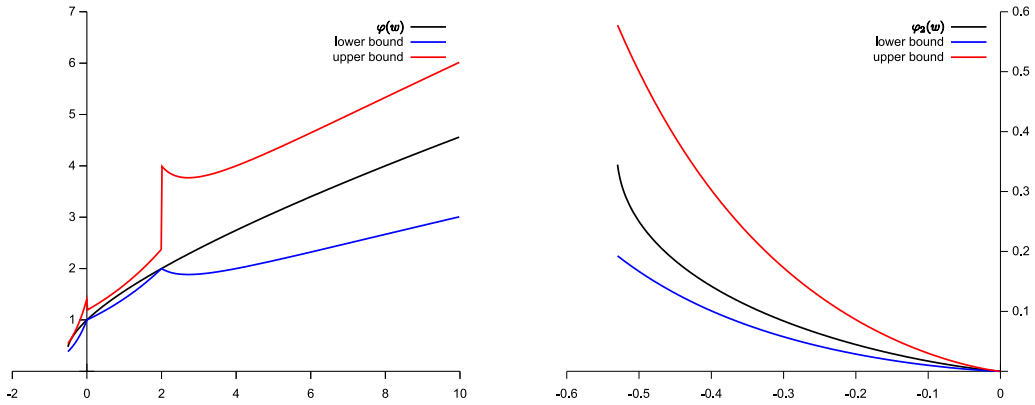
Proof. Showing that $v \mapsto v \log_2(v)$ is decreasing for $v \in [0, 1/e]$ is easy and left to the reader. We will use the following notations that are more convenient:

$$\begin{aligned} \omega &= 1/(-w), \\ \nu &= 1/\varphi_2(w). \end{aligned}$$

Hence $\omega \geq e/\log_2(e)$ and $\nu \geq e$. Moreover, by hypothesis, $\nu/\log_2(\nu) = \omega$. It is easy to show that for any $\nu \geq e$,

$$\nu^{1/3} \leq \frac{\nu}{\log_2(\nu)} \leq \nu.$$

Therefore, $\omega \leq \nu \leq \omega^3$ and thus $\log_2(\omega) \leq \log_2(\nu) \leq 3 \cdot \log_2(\omega)$. We conclude by using $\nu = \omega \cdot \log_2(\nu)$. □



(a) Graph of φ and its bounds

(b) Graph of φ_2 and its bounds

Figure 5: Illustration of Lemmas 4 and 5

When bounding relative errors, we need to estimate the values of $\mathrm{erf}(x)$ and $\mathrm{erfc}(x)$. The next lemma gives such estimates.

Lemma 6. *The following inequalities hold:*

$$\begin{array}{lrcl} \text{if } 0 < x < 1, & x/2 \leq \mathrm{erf}(x) & \leq & 2x, \\ & 1/8 \leq \mathrm{erfc}(x) & \leq & 1; \\ \\ \text{if } x \geq 1, & 1/2 \leq \mathrm{erf}(x) & \leq & 1, \\ & e^{-x^2}/(4x) \leq \mathrm{erfc}(x) & \leq & e^{-x^2}/(x\sqrt{\pi}). \end{array}$$

Proof. When $0 < x < 1$, the series given in Equation (1) is alternating with a decreasing general term. Hence

$$\frac{2x}{\sqrt{\pi}} \cdot \left(1 - \frac{x^2}{3}\right) \leq \mathrm{erf}(x) \leq \frac{2x}{\sqrt{\pi}}.$$

The inequalities for $\mathrm{erf}(x)$ are easily obtained from it.

Since erfc is decreasing, $\mathrm{erfc}(1) \leq \mathrm{erfc}(x) \leq \mathrm{erfc}(0) = 1$. Taking one more term in the series of $\mathrm{erf}(x)$, we get

$$\mathrm{erf}(x) \leq \frac{2x}{\sqrt{\pi}} \cdot \left(1 - \frac{x^2}{3} + \frac{x^4}{10}\right).$$

Applying it at $x = 1$, we get $\mathrm{erfc}(1) = 1 - \mathrm{erf}(1) \geq 1/8$.

When $x > 1$, since erf is increasing and goes to 1 as $x \to +\infty$, we have $\mathrm{erf}(1) \leq \mathrm{erf}(x) \leq 1$. This gives the inequalities for $\mathrm{erf}(x)$.

We now prove the last two inequalities of the lemma. By definition,

$$\begin{aligned}\mathrm{erfc}(x) &= \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-v^2}\,\mathrm{d}v \\ &= \frac{2}{\sqrt{\pi}} \int_x^{+\infty} \frac{-2v \cdot e^{-v^2}}{-2v}\,\mathrm{d}v \\ &= \frac{2}{\sqrt{\pi}} \cdot \left(\frac{e^{-x^2}}{2x} - \int_x^{+\infty} \frac{e^{-v^2}}{2v^2}\,\mathrm{d}v \right).\end{aligned}$$

This gives the upper-bound. For the lower-bound, we use the change of variable $v \leftarrow v + x$:

$$\int_x^{+\infty} \frac{e^{-v^2}}{2v^2}\,\mathrm{d}v = \int_0^{+\infty} \frac{e^{-x^2-2vx-v^2}}{2(v+x)^2}\,\mathrm{d}v.$$

Since $(v+x)^2 \geq x^2$ and $-x^2 \leq 0$,

$$\int_0^{+\infty} \frac{e^{-x^2-2vx-v^2}}{2(v+x)^2}\,\mathrm{d}v \leq \frac{e^{-x^2}}{2x^2} \cdot \int_0^{+\infty} e^{-2vx}\,\mathrm{d}v = \frac{e^{-x^2}}{4x^3}.$$

Therefore,

$$\begin{aligned}\mathrm{erfc}(x) &\geq \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot \left(1 - \frac{1}{2x^2}\right) \\ &\geq \frac{e^{-x^2}}{x \cdot 2\sqrt{\pi}} \quad \text{since } x \geq 1.\end{aligned}$$

We conclude by remarking that $2\sqrt{\pi} \simeq 3.544 \leq 4$. □

4.3 Practical implementation of Equation (1)

Here, we assume that Equation (1) is used to obtain an approximate value of $\mathrm{erf}(x)$ (as before, we suppose $x > 0$):

$$\mathrm{erf}(x) = \frac{2x}{\sqrt{\pi}} \left(\sum_{n=0}^{N-1} (-1)^n \frac{(x^2)^n}{(2n+1)n!} \right) + \varepsilon_N^{(1)}(x)$$

where $\varepsilon_N^{(1)}(x)$ is the remainder.

We first express a relation that ensures that $\varepsilon_N^{(1)}(x)$ is smaller than $2^{-t'-1} \cdot \mathrm{erf}(x)$ (remember that t' is the target precision, given as an input).

Proposition 3. *Let E be the exponent of x. If N satisfies*

$$\frac{N}{ex^2} \log_2\left(\frac{N}{ex^2}\right) \geq \frac{t' + \max(0,\, E)}{ex^2},$$

the remainder is bounded by $\mathrm{erf}(x) \cdot 2^{-t'-1}$.

Proof. Remark first that for $N \geq 1$,

$$\frac{2}{\sqrt{\pi}} \cdot \frac{1}{(2N+1)\sqrt{2\pi N}} \leq \frac{1}{4}. \tag{7}$$

We distinguish the cases when $x < 1$ and when $x \geq 1$:

- if $x < 1$, $E \leq 0$ and $\mathrm{erf}(x)$ is greater than $x/2$. The hypothesis becomes $(ex^2/N)^N \leq 2^{-t'}$ and thus
$$\frac{x^{2N+1}}{2} \cdot \left(\frac{e}{N}\right)^N \leq 2^{-t'} \cdot \mathrm{erf}(x). \tag{8}$$
- if $x \geq 1$, $E > 0$ and $\mathrm{erf}(x)$ is greater than $1/2$. The hypothesis becomes $(ex^2/N)^N \leq 2^{-t'-E}$ and thus
$$\frac{x^{2N}}{2} \cdot \left(\frac{e}{N}\right)^N \leq 2^{-t'-E} \cdot \mathrm{erf}(x).$$
Since $x < 2^E$ we obtain Inequality (8) again.

From Inequalities (7) and (8) we deduce that

$$\frac{2}{\sqrt{\pi}} \cdot \frac{x^{2N+1}}{2N+1} \cdot \left(\frac{e}{N}\right)^N \cdot \frac{1}{\sqrt{2\pi N}} \leq 2^{-t'-1} \cdot \mathrm{erf}(x).$$

Using Lemma 3, we get

$$\frac{2}{\sqrt{\pi}} \cdot \frac{x^{2N+1}}{(2N+1)N!} \leq 2^{-t'-1} \cdot \mathrm{erf}(x).$$

Remark that the left term of the inequality is the absolute value of the general term of the series. Since it is smaller than $2^{-t'-1} \cdot \mathrm{erf}(x)$, it is in particular smaller than $1/2$. Since the series is alternating, we can bound the remainder by the absolute value of the first neglected term as soon as the term decreases in absolute value.

In our case, the absolute value of the general term may begin by increasing before to decrease. But when it increases, it is always greater than 1. Therefore, since here it is smaller than $1/2$, we are in the decreasing phase and we can write $\varepsilon_N^{(1)}(x) \leq 2^{-t'-1} \cdot \mathrm{erf}(x)$. □

We use this proposition and Lemma 4 for obtaining an upper-estimation of N. We first evaluate $a = (t' + \max(0,\, E)) \oslash (\diamond(e) \otimes x \otimes x)$. Rounding modes are chosen in order to be sure that a is an upper bound of the exact value. Then we choose N using this recipe.

If $a \geq 2$,	N	$\geq$	$2(t' + \max(0,\, E))/\log_2(a)$
If $a \in [0,\, 2]$,	N	$\geq$	$ex^2 \cdot 2^{1/4} \cdot 2^{a/2}$

Again, these formulas are evaluated with appropriate rounding modes, in order to ensure that N is really greater than the actual value.

Then, we need to choose a working precision t. This precision depends on the errors that will be accumulated during the evaluation. So, we first sketch the details of the evaluation.

We compute

$$S(x) = \sum_{n=0}^{N-1} (-1)^n \frac{2}{\sqrt{\pi}} \cdot \frac{x^{2n+1}}{(2n+1) \cdot n!}$$

using Algorithm 1 with parameters $y = x^2$, $\alpha_0 = 2x/\sqrt{\pi}$ and for $k \geq 1$, $\alpha_k = \dfrac{-1}{k} \cdot \dfrac{2k-1}{2k+1}$. In the following, the variables acc, tmp, i, k, etc. are the variables introduced in Algorithm 1 on page 6.

In practice, we do not compute the ratio $(2k-1)/(2k+1)$. We use the variable acc to compute $(y^L)^{\lfloor k/L \rfloor}/k!$ and we use a temporary variable tmp for the division by $2k+1$. S_i is updated by alternatively adding or subtracting tmp (instead of acc).

In the beginning, $y = x^2$ and $z = y^L$ are computed with rounding upwards. When computing α_0, the rounding modes are chosen in such a way that the computed value is greater than the exact value $2x/\sqrt{\pi}$. The variables acc and tmp are also updated with rounding upwards. Hence, the following always holds:

$$y^i \cdot \mathsf{tmp} \geq \frac{2}{\sqrt{\pi}} \cdot \frac{x^{2k+1}}{(2k+1)k!}.$$

Let F be the exponent of y: $y < 2^F$. Using the fact that $\operatorname{erf}(x) \geq x/2$ when $x < 1$ and $\operatorname{erf}(x) \geq 1/2$ when $x \geq 1$, it is easy to show that we can stop the loop as soon as

$$k \geq N \quad \text{or} \quad \mathsf{tmp} \cdot 2^{Fi} < 2^{-t'+\min(E-1,0)-2}.$$

The complete algorithm is summed up in Figure Algorithm 2.

The roundoff errors are bounded using the following proposition.

Proposition 4. *If Algorithm 2 is used to compute an approximation $\widehat{S(x)}$ of the sum $S(x)$, the following holds:*

$$\widehat{S(x)} = \sum_{n=0}^{N-1} (-1)^n \frac{2x}{\sqrt{\pi}} \cdot \frac{x^{2n}}{(2n+1)\cdot n!} \langle 8N \rangle .$$

Thus

$$\left| \widehat{S(x)} - S(x) \right| \leq \gamma_{8N} \left(\frac{2x}{\sqrt{\pi}} \sum_{n=0}^{N-1} \frac{x^{2n}}{(2n+1)\cdot n!} \right) \leq \gamma_{8N} \frac{2}{\sqrt{\pi}} \int_0^x e^{v^2} \, \mathrm{d}v.$$

The bound γ_{8N} could be made tighter. However, we cannot hope a better value than γ_N since we do $O(N)$ operations. As we will see, only the logarithm of this value will be of interest for choosing the working precision t. By working more carefully, we would not get more than replacing $\log(8N)$ *by* $\log(N)$ *and it would not be of any practical benefit.*

Proof. For proving this result, we use the same techniques as those presented in the example of Section 3. The main arguments are the following.

- Line 1 of the algorithm leads to one error;
- Line 4 is obtained by binary exponentiation: by counting the number of multiplications, it is easy to show that $\widehat{z} = z \langle 2L-1 \rangle$;
- Line 6 involves an approximation of π ($\widehat{\pi} = \pi \langle 1 \rangle$) and a square root. Using Lemma 1 we get $\mathsf{acc} = \diamond(\sqrt{\widehat{\pi}}) = \diamond(\sqrt{\pi} \langle 1 \rangle) = \sqrt{\pi} \langle 2 \rangle$;
- Line 7 involves a division (the multiplication by 2 is exact);

Input: a floating-point number x,
the working precision t,
the target precision t',
$\mathsf{L} \in \mathbb{N}^\star$, $\mathsf{N} \in \mathbb{N}^\star$.
Output: an approximation of erf(x) with relative error less than $2^{-t'-1}$ obtained using Equation (1)

```
/* each operation is performed in precision t                    */
y ← x * x ; // rounded upwards
F ← exponent(y) ;
if x < 1 then G ← exponent(x) − 1 else G ← 0 ;
z ← power(y, L) ; // computed with rounding upwards
S ← [0, ..., 0] ;
acc ← √π ; // rounded downwards
acc ← 2 * x/acc ; // rounded upwards
i ← 0 ;
k ← 0 ;
tmp ← acc ;
repeat
    if (k mod 2) = 0 then S[i] ← S[i] + tmp else S[i] ← S[i] − tmp ;
    k ← k + 1;
    if i = L − 1 then
        i ← 0 ;
        acc ← acc * z ; // rounded upwards
    else
        i ← i + 1 ;
    end
    acc ← acc/k ; // rounded upwards
    tmp ← acc/(2 * k + 1) ; // rounded upwards
until k = N or exponent(tmp) < G − t' − 2 − F * i ;
/* now S(y) is evaluated from the S_i by Horner's rule           */
R ← S[L − 1] ;
for i ← L − 2 downto 0 do
    R ← S[i] + y * R ;
end
return R;
```

Algorithm 2: erfByEquation1()

- acc is updated at lines 16 and 20 of the algorithm. At line 16, it accumulates $2L$ errors ($2L-1$ due to z and one due to the multiplication itself). At line 20, it accumulates one error. It is hence easy to show that we can always write

$$\widehat{\mathsf{acc}} = \mathsf{acc}\,\langle \underbrace{3}_{\text{initialization}} + \underbrace{N}_{\text{line 20 occurs at most } N \text{ times}} + \underbrace{2L\lfloor N/L\rfloor}_{\text{line 16 occurs at most } \lfloor N/L\rfloor \text{ times}} \rangle.$$

  We simplify it in $\widehat{\mathsf{acc}} = \mathsf{acc}\,\langle 3+3N\rangle$;

- We can always write $\widehat{\mathsf{tmp}} = \mathsf{tmp}\,\langle 4+3N\rangle$ since tmp is obtained from acc by one division ($2k+1$ is computed exactly in integer arithmetic);

- Eventually, $S[i]$ is obtained by less than N additions using variable tmp which allows to write

$$S_i = \mathsf{tmp}_i\,\langle 4N+4\rangle + \mathsf{tmp}_{i+L}\,\langle 4N+4\rangle + \cdots + \mathsf{tmp}_{i+N-L}\,\langle 4N+4\rangle$$

  where tmp_k denotes the value of variable tmp at step k of the algorithm;

- The evaluation by Horner's rule at line 25 accumulates 3 more errors per step (one comes from the fact that $y = x^2\,\langle 1\rangle$, one comes from the multiplication and one comes from the addition). Therefore, during the complete loop, $3(L-1)$ errors are accumulated per term of the sum. We bound it by $3N-3$ and finally get the result with $\langle 7N+1\rangle$. We bound it by $\langle 8N\rangle$.

□

We just need to see how to choose the working precision t and we will be done. We estimate it with the following lemma.

Lemma 7. *The following inequalities hold:*

$$\begin{array}{lrcl} \text{if } 0<x<1, & x & \leq \displaystyle\int_0^x e^{v^2}\,\mathrm{d}v & \leq 2x; \\ \text{if } x\geq 1, & \dfrac{1}{e^2}\cdot\dfrac{e^{x^2}}{x} & \leq \displaystyle\int_0^x e^{v^2}\,\mathrm{d}v & \leq \dfrac{e^{x^2}}{x}. \end{array}$$

Proof. For the first inequality, the lower-bound is obtained by replacing e^{v^2} by 1 in the integral. The upper bound is obtained by replacing e^{v^2} by $e^{0.55^2}$ on $[0,\,0.55]$ and by e on $[0.55,\,1]$.

For the second inequality, the upper bound is obtained by replacing e^{v^2} by e^{vx} in the integral. The lower bound is obtained by considering the integral restricted to the interval $[x-1/x,\,x]$. □

Finally, an appropriate working precision t is given by the following recipe (remember that E is the exponent of x).

$t \geq t' + 9 + \lceil \log_2 N\rceil$	when $x<1$
$t \geq t' + 9 + \lceil \log_2 N\rceil - E + x^2\log_2(e)$	when $x\geq 1$

Proof. We show the result for the first inequality. The second one is obtained the same way. We suppose that $0 < x < 1$ and $t \geq t' + 9 + \lceil \log_2 N \rceil$. Thus

$$2^{-t+8} \cdot N \leq 2^{-t'-1}.$$

Using the fact that $\gamma_{8N} \leq 16Nu$ (Proposition 2), we get

$$2\gamma_{8N}(2x) \leq 2^{-t'-1} \cdot \frac{x}{2}.$$

We conclude by using $(x/2) \leq \mathrm{erf}(x)$ (Lemma 6), $(2/\sqrt{\pi}) \leq 2$ and $\displaystyle\int_0^x e^{v^2}\,\mathrm{d}v \leq 2x$ (Lemma 7). □

In practice $\log_2(e)$ is replaced by a value precomputed with rounding upwards. The factor $x^2 \log_2(e)$ that appears when $x > 1$ highlights the fact that the series is ill-conditioned for large values of x.

4.4 Practical implementation of Equation (2)

Here, we assume that Equation (2) is used to obtain an approximate value of $\mathrm{erf}(x)$:

$$\mathrm{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \left(\sum_{n=0}^{N-1} \frac{(2x^2)^n}{1 \cdot 3 \cdot 5 \cdots (2n+1)} \right) + \varepsilon_N^{(2)}(x)$$

where $\varepsilon_N^{(2)}(x)$ is the remainder.

We follow the same method as for Equation (1). The first thing we need is a way of bounding the remainder. This is achieved by the following lemma.

Lemma 8. *If $N \geq 2x^2$, the following inequality holds:*

$$\varepsilon_N^{(2)}(x) \leq 2 \cdot \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1 \cdot 3 \cdots (2N+1)}.$$

Proof. By definition,

$$\begin{aligned} \varepsilon_N^{(2)}(x) &= \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \sum_{n=N}^{+\infty} \frac{(2x^2)^n}{1 \cdot 3 \cdots (2n+1)} \\ &= \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \cdot \frac{(2x^2)^N}{1 \cdot 3 \cdots (2N+1)} \left(1 + \frac{2x^2}{2N+3} + \frac{(2x^2)^2}{(2N+3)(2N+5)} + \cdots \right). \end{aligned}$$

We bound it by the geometric series with a common ratio of $(2x^2)/(2N+3)$ and a first term equal to 1:

$$\varepsilon_N^{(2)}(x) \leq \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \cdot \frac{(2x^2)^N}{1 \cdot 3 \cdots (2N+1)} \left(1 + \frac{2x^2}{2N+3} + \frac{(2x^2)^2}{(2N+3)(2N+3)} + \cdots \right)$$

Since $N \geq 2x^2$, $(2x^2)/(2N+3) \leq 1/2$ and the sum of the series is bounded by 2. □

The relation between N and t' is given by the following proposition.

Proposition 5. *Let E be the exponent of x. If N satisfies $N \geq 2x^2$ and*

$$\frac{N}{ex^2}\log_2\left(\frac{N}{ex^2}\right) \geq \frac{t'+3+\max(0,\,E)-x^2\log_2(e)}{ex^2}$$

the remainder is bounded by $\operatorname{erf}(x)\cdot 2^{-t'-1}$.

Proof. We use the same kind of arguments as for Proposition 3. The product $1\cdot 3\cdots(2N+1)$ is equal to $(2N+1)!/(2^N\cdot N!)$. This lets us write

$$\left|\varepsilon_N^{(2)}(x)\right| \;\leq\; 4\cdot\frac{xe^{-x^2}}{\sqrt{\pi}}\cdot\frac{(2x^2)^N\cdot N!\cdot 2^N}{(2N+1)!} \;\leq\; 4\cdot\frac{xe^{-x^2}}{\sqrt{\pi}}\cdot\frac{(2x^2)^N\cdot N!\cdot 2^N}{(2N)!}.$$

We conclude using Lemmas 3 and 6. □

We first evaluate $b = x\otimes x\otimes\diamond(\log_2(\diamond(e)))$ with rounding downwards and $a = (t'+3-b+\max(0,\,E))\oslash(\diamond(e)\otimes x\otimes x)$. We deduce the formulas for computing an upper-estimation of N:

if $a \geq 2$,	N	$\geq$	$2(t'+3+\max(0,\,E)-b)/\log_2(a)$
if $a \in [0,\,2]$,	N	$\geq$	$ex^2\cdot 2^{1/4}\cdot 2^{a/2}$
if $a \in [-\log_2(e)/e,\,0]$,	N	$\geq$	$ex^2\cdot 2^a$

The third case may lead to a value N that is smaller than $2x^2$. In this case, we take $N = \lceil 2x^2\rceil$, for ensuring our hypothesis.

We compute

$$S(x) = \sum_{n=0}^{N-1}\frac{2xe^{-x^2}}{\sqrt{\pi}}\cdot\frac{(2x^2)^n}{1\cdot 3\cdots(2n+1)}$$

using Algorithm 1 with parameters $y = 2x^2$, $\alpha_0 = 2xe^{-x^2}/\sqrt{\pi}$ and for $k \geq 1$, $\alpha_k = 1/(2k+1)$. As in the implementation of Equation (1), we use upward roundings and a test for stopping the loop as soon as possible. In this case, the criterion becomes

$$k \geq N \quad \text{or} \quad \left(k \geq 2x^2 \;\text{ and }\; \mathsf{acc}\cdot 2^{Fi} < 2^{-t'+\min(E-1,\,0)-3}\right).$$

The complete algorithm is summed up in Figure Algorithm 3.

The roundoff errors are bounded using the following proposition.

Proposition 6. *If Algorithm 3 is used to compute an approximation $\widehat{S(x)}$ of the sum $S(x)$, the following holds:*

$$\widehat{S(x)} = \sum_{n=0}^{N-1}\frac{2xe^{-x^2}}{\sqrt{\pi}}\cdot\frac{(2x^2)^n}{1\cdot 3\cdots(2n+1)}\,\langle 16N\rangle\,.$$

Thus

$$\left|\widehat{S(x)} - S(x)\right| \leq \gamma_{16N}\cdot S(x) \leq \gamma_{16N}\cdot\operatorname{erf}(x).$$

Input: a floating-point number x,
the working precision t,
the target precision t',
$\mathsf{L} \in \mathbb{N}^\star$, $\mathsf{N} \in \mathbb{N}^\star$.
Output: an approximation of erf(x) with relative error less than $2^{-t'-1}$ obtained using Equation (2)

```
/* each operation is performed in precision t                    */
y ← 2 * x * x ; // rounded upwards
E ← exponent(x) ;
F ← exponent(y) ;
if x < 1 then G ← E − 1 else G ← 0 ;
z ← power(y, L) ; // computed with rounding upwards
S ← [0, ..., 0] ;
acc ← √π ; // rounded downwards
acc ← 2 * x/acc ; // rounded upwards
tmp ← x * x ; // performed in precision t + max(2 · E, 0), rounded downwards
tmp ← exp(−tmp) ; // rounded upwards
acc ← acc * tmp ; // rounded upwards
i ← 0 ;
k ← 0 ;
repeat
    S[i] ← S[i] + acc ;
    k ← k + 1;
    if i = L − 1 then
        i ← 0 ;
        acc ← acc * z ; // rounded upwards
    else
        i ← i + 1 ;
    end
    acc ← acc/(2 * k + 1) ; // rounded upwards
until k = N or ((k ≥ y) and (exponent(acc) < G − t' − 3 − F * i)) ;
/* now S(y) is evaluated from the S_i by Horner's rule           */
R ← S[L − 1] ;
for i ← L − 2 downto 0 do
    R ← S[i] + y * R ;
end
return R;
```

Algorithm 3: erfByEquation2()

Proof. In fact, using the same techniques as in Proposition 4, one proves that

$$\widehat{S(x)} = \sum_{n=0}^{N-1} \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1 \cdot 3 \cdots (2n+1)} \langle 7N+3 \rangle .$$

We bound it by $\langle 16N \rangle$ because it is more convenient to use powers of 2.

Note that at line 9 of the algorithm, x^2 is computed in precision $t + \max(2E, 0)$. Using Lemma 2, it allows for writing $\widehat{\alpha_0} = \alpha_0 \langle 6 \rangle$. □

Finally, an appropriate precision t is given by the following recipe:

$$\boxed{t \geq t' + 7 + \lceil \log_2 N \rceil}$$

4.5 Implementation of erfc using Equation (1) or (2)

In this section, we do not suppose that $x > 0$ anymore.

Since $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$, we can use the previous algorithms for evaluating erfc. However, we have to take care of two things:

- firstly, the approximation of $\operatorname{erf}(x)$ should be computed with an appropriate relative error 2^{-s}. Since $\operatorname{erf}(x)$ and $\operatorname{erfc}(x)$ do not have the same order of magnitude, 2^{-s} has no reason to be the same as the target relative error $2^{-t'}$;
- secondly, contrary to erf, erfc is not odd (nor even). In particular, $\operatorname{erfc}(-x)$ and $\operatorname{erfc}(x)$ do not have the same order of magnitude and this should be considered when estimating the relative error.

We evaluate $\operatorname{erfc}(x)$ in two steps: first we compute an approximation R of $\operatorname{erf}(x)$ with a relative error less than a certain bound 2^{-s} (this is performed by one of the previous algorithms). Then, we compute $1 \ominus R$ with precision $t' + 3$.

Lemma 9. *If s is chosen according to the following recipe, $|R - \operatorname{erf}(x)| \leq 2^{-t'-1} \cdot \operatorname{erfc}(x)$.*

$s \geq t' + 1$	*when* $x \leq -1$
$s \geq t' + 2 + E$	*when* $-1 < x < 0$
$s \geq t' + 5 + E$	*when* $0 \leq x < 1$
$s \geq t' + 3 + E + x^2 \log_2(e)$	*when* $x \geq 1$

Proof.

First case.

We suppose $x \leq -1$ and $s \geq t' + 1$. From the hypotheses we get

$$\big|R - \operatorname{erf}(x)\big| \leq 2^{-s} \cdot |\operatorname{erf}(x)| \leq 2^{-t'-1} \cdot |\operatorname{erf}(x)|.$$

Since $x < 0$, $\operatorname{erfc}(x) \geq 1$. Moreover, $|\operatorname{erf}(x)| \leq 1$. This gives the result.

Second case.

We suppose $-1 < x < 0$ and $s \geq t' + 2 + E$. From the hypotheses we get

$$\big|R - \operatorname{erf}(x)\big| \leq 2^{-s} \cdot |\operatorname{erf}(x)| \leq 2^{-t'-1} \cdot 2^{-E-1} \cdot |\operatorname{erf}(x)|.$$

Since $x < 0$, $\mathrm{erfc}(x) \geq 1$. Moreover, $|\mathrm{erf}(x)| = \mathrm{erf}(|x|) \leq 2|x| \leq 2^{E+1}$. This gives the result.

Third case.
It is the same as the second case but using $\mathrm{erfc}(x) \geq 1/8$.

Fourth case.
Here $x \geq 1$ and $s \geq t' + 3 + E + x^2 \log_2(e)$. Hence

$$\left|R - \mathrm{erf}(x)\right| \leq 2^{-s} \cdot \mathrm{erf}(x) \leq 2^{-t'-1} \cdot 2^{-E-2} \cdot e^{-x^2} \cdot |\mathrm{erf}(x)|.$$

Since $x \geq 1$, $\mathrm{erf}(x) \leq 1$ and $\mathrm{erfc}(x) \geq e^{-x^2}/(4x) \geq e^{-x^2}/(4 \cdot 2^E)$. This gives the result. □

As a consequence of the lemma,

$$|(1 - R) - \mathrm{erfc}(x)| \leq 2^{-t'-1} \cdot \mathrm{erfc}(x).$$

It follows that $|(1 - R)| \leq 2\,\mathrm{erfc}(x)$. Now, since $(1 \ominus R) = (1 - R)\,\langle 1 \rangle$,

$$\left|(1 \ominus R) - (1 - R)\right| \;\leq\; |(1 - R)| \cdot 2^{1-(t'+3)} \;\leq\; 2^{-t'-1} \cdot \mathrm{erfc}(x).$$

Finally $\left|(1 \ominus R) - \mathrm{erfc}(x)\right| \leq \left|(1 \ominus R) - (1 - R)\right| + \left|(1 - R) - \mathrm{erfc}(x)\right| \leq 2^{-t'}\,\mathrm{erfc}(x)$ which proves that $(1 \ominus R)$ is an approximation of $\mathrm{erfc}(x)$ with a relative error bounded by $2^{-t'}$.

4.6 Practical implementation of Equation (3)

Here, we assume that Equation (3) is used to obtain an approximate value of $\mathrm{erfc}(x)$ (we suppose again that $x > 0$):

$$\mathrm{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \left(1 + \sum_{n=1}^{N-1} (-1)^n \cdot \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{(2x^2)^n} \right) + \varepsilon_N^{(3)}(x)$$

where $\varepsilon_N^{(3)}(x)$ is the remainder, bounded thanks to Inequality (4).

The particularity of this formula comes from the fact that the remainder cannot be made arbitrarily small. In fact, x being given, $\varepsilon_N^{(3)}(x)$ is first decreasing until it reaches an optimal value with $N = \lfloor x^2 + 1/2 \rfloor$. For larger values of N, it increases. Hence, given a target relative error $2^{-t'}$, it may be possible that no value of N is satisfying. In this case, Equation (3) cannot be used. Note in particular that this formula is useless for $0 < x < 1$. Until the end of this section, we will suppose that $x \geq 1$.

Conversely, when the relative error can be achieved, we can choose any value of N between two values $N_{\min}$ and $N_{\max}$. Obviously, we are interested in the smallest one. For finding an upper bound of $N_{\min}$ we will use Lemma 5.

We just give the main results needed for the implementation. The techniques for proving them are exactly the same as the one used with Equations (1) and (2).

Lemma 10. *The following inequality holds for any $x \geq 1$:*

$$|\varepsilon_N^{(3)}(x)| \leq \frac{e^{-x^2}}{x} \left(\frac{N}{ex^2} \right)^N.$$

Proof. The result is obtained from the bound given in Equation (4). The product $1\cdot 3\cdots(2N-1)$ equals $(2N)!/(2N\cdot 2^N\cdot N!)$. We bound it by $(2N)!/(2^N\cdot N!)$ and use the bounds given in Lemma 3. □

Proposition 7. *If N satisfies*

$$\frac{N}{ex^2}\cdot\log_2\left(\frac{N}{ex^2}\right)\leq\frac{-t'-3}{ex^2}$$

the remainder is bounded by $\operatorname{erfc}(x)\cdot 2^{-t'-1}$.

Proof. We use the previous lemma and the inequality $\operatorname{erfc}(x)\geq e^{-x^2}/(4x)$ given in Lemma 6. □

Using this proposition and Lemma 5 we obtain an upper-estimation of N. We compute $a=(-t'-3)\oslash(\diamond(e)\otimes x\otimes x)$. Rounding downwards is used for the whole computation. Hence the computed value a is a lower bound for the exact value.

If $a\in[-\log_2(e)/e,\,0]$, $\quad N\quad\geq\quad(-t'-3)/\log_2(-a)$
Otherwise Equation (3) cannot be used

Remark that this estimation may be too large because of the overestimation of φ_2. We are always sure that $\lfloor x^2+1/2\rfloor$ is an upper bound for N. So we actually take the minimum of $(-t'-3)/\log_2(-a)$ and $\lfloor x^2+1/2\rfloor$.

We evaluate the sum

$$S(x)=\frac{e^{-x^2}}{x\sqrt{\pi}}+\sum_{n=1}^{N-1}(-1)^n\cdot\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}$$

using Algorithm 1 with parameters $y=1/(2x^2)$, $\alpha_0=e^{-x^2}/(x\sqrt{\pi})$ and for $k\geq 1$, $\alpha_k=-(2k-1)$. In practice, we use $\alpha_k=(2k-1)$ and we alternatively add and subtract acc to the partial sum (again the variables acc, tmp, i, k, etc. are the variables introduced in Algorithm 1 on page 6).

When computing α_0, the rounding modes are chosen in such a way that the computed value is an upper bound for the actual value. Besides, when acc is updated, rounding upwards is used. Hence, we can stop the loop as soon as

$$k=N\quad\text{ or }\quad\mathsf{acc}\cdot 2^{Fi}<2^{-t'-1}\cdot e^{-x^2}/(4x)$$

where F is the exponent of y. The algorithm is summed up in Figure Algorithm 4.

The roundoff errors are bounded using the following proposition.

Proposition 8. *If Algorithm 4 is used to compute an approximation $\widehat{S(x)}$ of the sum $S(x)$, the following holds:*

$$\widehat{S(x)}=\frac{e^{-x^2}}{x\sqrt{\pi}}\langle 16N\rangle+\sum_{n=1}^{N-1}(-1)^n\cdot\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\langle 16N\rangle\,.$$

Thus

$$\left|\widehat{S(x)}-S(x)\right|\leq\gamma_{16N}\cdot\frac{e^{-x^2}}{x\sqrt{\pi}}\left(1+\sum_{n=1}^{N-1}\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\right)\leq\gamma_{16N}\cdot\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{3}{2}.$$

Input: a floating-point number x,
the working precision t,
the target precision t',
$\mathsf{L} \in \mathbb{N}^\star$, $\mathsf{N} \in \mathbb{N}^\star$.
Output: an approximation of erfc(x) with relative error less than $2^{-t'-1}$ obtained using Equation (3)

```
/* each operation is performed in precision t                    */
E ← exponent(x);
G ← ⌈x * x * log2(e)⌉ ; // computed with rounding upwards
acc ← x * x ; // performed in precision t + 2 · E, rounded downwards
y = 2 * acc;
y ← 1/y ; // rounded upwards
acc ← exp(−acc) ; // rounded upwards
tmp ← x * √π ; // rounded downwards
acc ← tmp/acc ; // rounded upwards
F ← exponent(y) ;
z ← power(y, L) ; // computed with rounding upwards
S ← [0, ..., 0] ;
i ← 0 ;
k ← 0 ;
repeat
    if (k mod 2) = 0 then S[i] ← S[i] + acc else S[i] ← S[i] − acc ;
    k ← k + 1;
    if i = L − 1 then
        i ← 0 ;
        acc ← acc * z ; // rounded upwards
    else
        i ← i + 1 ;
    end
    acc ← acc * (2 * k − 1) ; // rounded upwards
until k = N or exponent(acc) < −t' − 3 − F * i − G − E ;
/* now S(y) is evaluated from the S_i by Horner's rule           */
R ← S[L − 1] ;
for i ← L − 2 downto 0 do
    R ← S[i] + y * R ;
end
return R;
```

Algorithm 4: erfcByEquation3()

Proof. In this algorithm, $\widehat{y} = y\,\langle 2\rangle$, hence the binary exponentiation leads to $\widehat{z} = z\,\langle 3L-1\rangle$. The final bound is

$$\widehat{S(x)} = \frac{e^{-x^2}}{x\sqrt{\pi}}\,\langle 8N+3\rangle + \sum_{n=1}^{N-1}(-1)^n \cdot \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot \frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\,\langle 8N+3\rangle\,.$$

We bound $\langle 8N+3\rangle$ by $\langle 16N\rangle$.

We now prove that

$$1 + \sum_{n=1}^{N-1} \frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n} \le \frac{3}{2} \quad \text{for } N \le \lfloor x^2 + 1/2\rfloor.$$

Since $N \le \lfloor x^2+1/2\rfloor$, $N \le x^2+1$. Moreover, the general term is decreasing. Hence, we can write

$$\sum_{n=1}^{N-1} \frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n} \;\le\; (N-1)\frac{1}{2x^2} \;\le\; \frac{1}{2}.$$

□

Using this proposition, we obtain a suitable working precision t:

$$\boxed{t \ge t' + 6 + \lceil \log_2(N)\rceil}$$

4.7 Implementation of erf with Equation (3)

We finish our study by explaining briefly how Equation (3) is used to compute $\operatorname{erfc}(x)$ when $x \le -1$ and to compute $\operatorname{erf}(x)$ for $x \ge 1$ (the symmetrical case $x < -1$ is the same).

Note that $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = 1 + \operatorname{erf}(-x) = 2 - \operatorname{erfc}(-x)$. We obtain $\operatorname{erfc}(x)$ by computing an approximation R of $\operatorname{erfc}(-x)$ with a relative error smaller than an appropriate bound 2^{-s} and computing $2 \ominus R$ in precision $t'+3$.

The same way, since $\operatorname{erf}(x) = 1 - \operatorname{erfc}(x)$, we obtain $\operatorname{erf}(x)$ by computing an approximation R of $\operatorname{erfc}(x)$ with a relative error smaller than an appropriate bound 2^{-s} and computing $1 \ominus R$ in precision $t'+3$.

The appropriate values for s are given in the two following lemmas. The proofs are left to the reader.

Lemma 11. *If $x \ge 1$ and if s is chosen according to the following recipe, $|R - \operatorname{erfc}(x)| \le 2^{-t'-1} \cdot \operatorname{erfc}(-x)$.*

$$\boxed{s \ge t' + 2 - E - x^2\log_2(e)}$$

Remark that whenever $t' + 2 - E - x^2\log_2(e) \le 1$, it is not necessary to compute an approximation of $\operatorname{erfc}(-x)$ and to perform the subtraction: 2 can be directly returned as a result. Indeed, $t' + 2 - E - x^2\log_2(e) \le 1$ implies that $e^{-x^2}/x \le 2^{-t'}$ and hence

$$\operatorname{erfc}(x) \le 2^{-t'} \le 2^{-t'}\operatorname{erfc}(-x).$$

Since $\operatorname{erfc}(x) = 2 - \operatorname{erfc}(-x)$, it means that 2 is an approximation of $\operatorname{erfc}(-x)$ with a relative error less than $2^{-t'}$.

Lemma 12. *If $x \geq 1$ and if s is chosen according to the following recipe, $|R - \mathrm{erfc}(x)| \leq 2^{-t'-1} \cdot \mathrm{erf}(x)$.*

$$\boxed{s \geq t' + 3 - E - x^2 \log_2(e)}$$

The same remark holds: if $t' + 3 - E - x^2 \log_2(e) \leq 1$, the value 1 can be directly returned as an approximation of $\mathrm{erf}(x)$ with a relative error less than $2^{-t'}$.

5 Experimental results

We gave all the details necessary for implementing each of the three equations (1), (2), and (3). They can be used for obtaining approximate values of either $\mathrm{erf}(x)$ or $\mathrm{erfc}(x)$. We also gave estimations of the order of truncation N and of the working precision t. In each case $O(\sqrt{N})$ multiplications at precision t and $O(N)$ additions and multiplications/divisions by small integers are needed for evaluating the sum. Hence, for each equation, the complexity of the algorithm is $O(\sqrt{N} \cdot M(t))$ where $M(t)$ denotes the complexity of a product of two numbers of precision t.

However, quite different behaviours are hidden behind this complexity. Indeed, the inputs of our algorithms are the point x and the target precision t'. Hence, N and t are functions of x and t'. As can be seen in previous sections, these functions highly depend on the equation used to evaluate $\mathrm{erf}(x)$ or $\mathrm{erfc}(x)$.

5.1 Choosing the best equation

Of course, given an input couple (x, t'), we would like to choose automatically the equation to be used, in order to minimise the computation time. For this purpose, we need to compare the complexity of the three equations. It seems quite hard to perform such a comparison theoretically:

- firstly, the estimations of N and t are different for each equation. They depend on x and t' in a complicated way. Besides, they are defined piecewise, which implies that there are many cases to study;
- secondly, comparing the three methods requires to set some assumptions on the implementation of the underlying arithmetic (e.g. complexity of the multiplication; complexity of the evaluation of exp).

Usually, the multiplication between two numbers of precision t is performed differently whether t is large or not: see Section 2.4 of [6] for an overview of what is used in MPFR. Three different algorithms are used in MPFR, each one depending on the underlying integer multiplication (that can be performed by at least four different algorithms).

In MPFR, the evaluation of $\exp(x)$ is performed by three different algorithms, depending on the required precision t (see the end of Section 2.5 of [6] for a brief overview).

- A naive evaluation of the series is used when t is smaller than a parameter t_0;
- the grouping technique of Smith is used when t lies between t_0 and a second threshold t_1;

- finally, if $t \geq t_1$, a binary splitting method is used. The values t_0 and t_1 have been tuned for each architecture.

This shows that choosing the best equation between the three equations proposed in this paper is a matter of practice and not of theoretical study. We implemented the three algorithms in C, using MPFR for the floating-point arithmetic. Our code is distributed under the GPL and freely available.

Figure 6: Comparison of the execution times of the three implementations of erf

In order to see experimentally which equation is the best for each couple (x, t'), we ran the three implementations for a large range of values x and t'. For each couple, we compared the execution time of each implementation when evaluating $\mathrm{erf}(x)$. The experimental results are summed up in Figure 6. The colours indicate which implementation is the fastest. The experiments were performed on a 32-bit 3.00 GHz Intel Pentium D with 2.00 GB of RAM running Linux 2.6.26 and MPFR 2.3.1. The thresholds used by MPFR for this architecture are $t_0 = 528$ and $t_1 = 47\,120$.

The boundary between Equations (1) and (2) seems to exhibit three phases, depending on t'. These phases approximately match the thresholds used by MPFR for the implementation of exp. Since Equation (2) relies on the evaluation of $\exp(-x^2)$ whereas Equation (1) does not, this is probably not a coincidence and we just see the impact of the evaluation of $\exp(-x^2)$ on the whole computation time.

The boundary between Equations (2) and (3) is more regular. In fact, we experimentally observe that as soon as Equation (3) is useable, it is more interesting than the other equations. Figure 7 shows the timings for $t' = 632$ in function of x: for small values of x, Equation (3) cannot achieve the target precision. But for $x \simeq 15$, it becomes useable and is immediately five times faster than the others. Hence, the domain where Equation (3) should be used is given by the points where the inequality of Proposition 7 has a solution, i.e. if and only if

$$\frac{-s - 2E - 6}{ex^2} \geq \frac{-\log_2(e)}{e},$$

where $s \gtrsim t' + 3 - E - x^2 \log_2(e)$ is the intermediate precision given in Lemma 12. Thus the equation of the boundary is approximately $\log(t') \simeq 2\log(x)$, which corresponds to the observations.

5.2 Comparison with other implementations

We compared our implementation of erf and erfc with two others reference implementations: MPFR and Maple. MPFR is probably the most relevant since we indeed compare two comparable things: since our implementation is written using MPFR, the underlying arithmetic is the same in both cases. Therefore, the difference of timings between our implementation and MPFR is completely due to the difference between the algorithm used.

Maple uses a decimal floating-point format. Besides, it is an interpreted language. Hence, the comparison between our implementation and Maple is not completely relevant. However, Maple is widely used and is one of the rare implementations of erf and erfc in arbitrary precision.

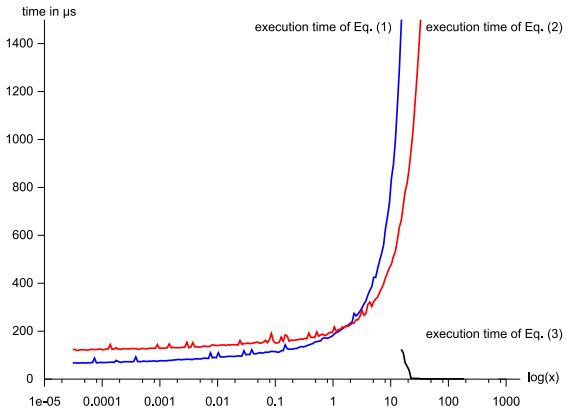


Figure 7: Execution times of the three implementations, in function of $\log(x)$, when $t' = 632$.

Our experiments are related in table Figure 8. The experiments were performed on a 32-bit 2.40 GHz Intel Xeon with 2.00 GHz of RAM running Linux 2.6.22. We used MPFR 2.3.1 and Maple 11. The values of x are chosen randomly with the same precision t' as the target precision. The table only indicates an approximate value. The target precision t' is expressed in bits. Maple is run with the variable Digits set to $\lfloor t'/\log_2(10) \rfloor$. This corresponds approximately to the same precision expressed in a decimal arithmetic. Maple remembers the values already computed. It is thus impossible to repeat the same evaluation several times for measuring the time of one single evaluation by considering the average timing. In order to overcome this difficulty we chose to evalute successively $\mathrm{erf}(x)$, $\mathrm{erf}(x+h)$, $\mathrm{erf}(x+2h)$, etc. where h is a small increment.

The cases when Equation (3) cannot achieve the target precision are represented by the symbol "-". Our implementation is the fastest except in a few cases. In small precisions and small values of x, MPFR is the fastest because it uses a direct evaluation that is a bit faster when the truncation rank is small.

The case $x \simeq 88.785777$ and $t' = 7139$ has another explanation. Actually, the situation corresponds to the remark following Lemma 12: $t' + 3 - E - x^2 \log_2(e) \leq 1$. Hence, there is nothing to compute and 1 can be returned immediately. In our implementation $t' + 3 - E - x^2 \log_2(e)$ is computed using MPFR in small precision. This takes 46 µs. MPFR performs the same kind of test but using hardware arithmetic. This explains that it can give an answer in 2 µs.

6 Conclusion and perspectives

We proposed three algorithms for efficiently evaluating the functions erf and erfc in arbitrary precision. These algorithms are based on three different summation formulas whose coefficients have the same general recursive structure. For evaluating the sum, we take advantage of this structure by using an algorithm due to Smith that makes it possible to perform only $O(\sqrt{N})$ slow multiplications instead of the generic $O(N)$ classical approach.

We gave closed formulas for upper-bounding the truncation rank N and we completely studied the effects of roundoff errors in the summation. We derived from this study closed formulas for the required working precision.

x	t'	Eq. 1	Eq. 2	Eq. 3	MPFR	Maple
0.000223	99	29 µs	47 µs	-	**14 µs**	283 µs
0.005602	99	34 µs	48 µs	-	**17 µs**	282 µs
0.140716	99	40 µs	53 µs	-	**25 µs**	287 µs
3.534625	99	96 µs	**84 µs**	-	125 µs	382 µs
88.785777	99	277 520 µs	6181 µs	**< 1 µs**	2 µs	18 µs
0.000223	412	**55 µs**	87 µs	-	76 µs	739 µs
0.005602	412	**62 µs**	94 µs	-	104 µs	783 µs
0.140716	412	**88 µs**	109 µs	-	186 µs	870 µs
3.534625	412	246 µs	**198 µs**	-	663 µs	1 284 µs
88.785777	412	289 667 µs	9 300 µs	**< 1 µs**	2 µs	21 µs
0.000223	1 715	**311 µs**	562 µs	-	1 769 µs	2 513 µs
0.005602	1 715	**393 µs**	616 µs	-	2 490 µs	2 959 µs
0.140716	1 715	**585 µs**	748 µs	-	4 263 µs	3 968 µs
3.534625	1 715	1 442 µs	**1 260 µs**	-	11 571 µs	8 850 µs
88.785777	1 715	343 766 µs	22 680 µs	**< 1 µs**	2 µs	42 µs
0.000223	7 139	**3 860 µs**	7 409 µs	-	28 643 µs	38 846 µs
0.005602	7 139	**4 991 µs**	7 959 µs	-	40 066 µs	51 500 µs
0.140716	7 139	**7 053 µs**	9 227 µs	-	64 975 µs	79 308 µs
3.534625	7 139	14 744 µs	**14 144 µs**	-	157 201 µs	191 833 µs
88.785777	7 139	628 527 µs	96845 µs	46 µs	**2 µs**	213 µs
0.000223	29 717	**63 ms**	108 ms	-	654 ms	1 140 ms
0.005602	29 717	**79 ms**	119 ms	-	881 ms	1 539 ms
0.140716	29 717	**108 ms**	137 ms	-	1 375 ms	2 421 ms
3.534625	29 717	202 ms	**198 ms**	-	2 968 ms	5 320 ms
88.785777	29 717	2 005 ms	**898 ms**	-	39 760 ms	243 690 ms

Figure 8: Timings of several implementations of erf

We implemented the three algorithms in C and compared the efficiency of the three methods in practice. This shows that the asymptotic expansion is the best method to use, as soon as it can achieve the target accuracy. Whenever the asymptotic expansion cannot be used, one must choose between the two others equations. The domain where it is more interesting to use one than the other depends on the underlying arithmetic. In practice, well-chosen thresholds must be chosen for each architecture. We also compared our implementation with the implementation of erf provided in MPFR and Maple. Our implementation is almost always the fastest one. It represents a condiderable improvement for intermediate and large precisions.

We must remark that our analysis is based on the hypothesis that no underflow or overflow occurs during the evaluation. Though unlikely, this could happen and should be taken into account for really guaranteeing the quality of the final result.

Since the method of evaluation used in this paper requires extra space for storing partial sums, the algorithms may become inefficient for very large precisions, if the memory has to be swapped on the hard disk. In this case, the sums may be evaluated by the straightforward algorithm: compute iteratively the coefficients of the sum and accumulate the result on the fly. This is slower but does not require extra memory. Besides, for large precisions, techniques based on binary splitting are usually faster and should be prefered.

There is another idea that may lead to a significant improvement: if the point x is of the form p/q where p and q are small integers, the sums of the three equations may be evaluated by the straightforward method using only additions, and multiplications/divisions by small integers. The corresponding complexity is $O(tN)$ (instead of $O(\sqrt{N}M(t))$). This is interesting in itself but it may also be used to obtain efficiently an approximate value of $\mathrm{erf}(x)$ or $\mathrm{erfc}(x)$ for any value x. Indeed, x can be written $x = x_0 + h$ where x_0 has the previous form and h is small. An approximate value of $\mathrm{erf}(x)$ could be obtained by considering the Taylor expansion of erf at x_0:

$$\mathrm{erf}(x) = \mathrm{erf}(x_0) + \sum_{i=1}^{+\infty} a_i h^i \quad \text{where } a_i = \frac{\mathrm{erf}^{(i)}(x_0)}{i!}.$$

Since h is small, only a few terms are necessary to obtain a good approximation. The coefficients a_i are of the form $p_i(x_0)e^{-x_0^2}$ where p_i is a polynomial of degree $2i-2$ satisfying a simple recurrence. The value $e^{-x_0^2}$ is computed only once and the coefficients are then quite easy to obtain (remark that when using Equation (2) or (3), the value $e^{-x_0^2}$ has already been computed when evaluating $\mathrm{erf}(x_0)$). We did not implement this technique yet but it seems to be a promising future work.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions.* Dover, 1965.

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

[3] R. P. Brent. A fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software (TOMS)*, 4(1):57–70, March 1978.

[4] R. P. Brent. Unrestricted algorithms for elementary and special functions. In S. Lavington, editor, *Information Processing 80: Proceedings of IFIP Congress 80*, pages 613–619. North-Holland, October 1980.

[5] G. E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *American Mathematical Monthly*, 77(9):931–956, 1970.

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

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

[8] N. N. Lebedev. *Special Functions & Their Applications*. Prentice-Hall, 1965.

[9] D. M. Smith. Efficient multiple-precision evaluation of elementary functions. *Mathematics of Computation*, 52(185):131–134, 1989.

[10] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software (TOMS)*, 17(3):410–423, 1991.