Semi-automatic implementation of the complementary error function

Anastasia Volkova, Jean-Michel Muller

Semi-automatic implementation of the complementary error function

Anastasia Volkova
Univ Lyon, Inria,
CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP

Jean-Michel Muller
Univ Lyon, CNRS,
ENS de Lyon, Inria, Université Claude Bernard Lyon 1, LIP

January 31, 2019

Abstract

The normal and complementary error functions are ubiquitous special functions for any mathematical library. They have a wide range of applications in probability and statistics. Practical applications call for customized implementations that have strict accuracy requirements. Accurate numerical implementation of these functions is, however, non-trivial. In particular, the complementary error function erfc for large positive arguments heavily suffers from cancellation, which is largely due to its asymptotic behavior. In this paper we provide a semi-automatic code generator for the erfc function which is parameterized by the user-given bound on the relative error. Our solution exploits erfc's asymptotic expression and leverages the automatic code generator Metalibm that provides accurate polynomial approximations. A fine-grained a priori error analysis provides a libm developer with the required accuracy for each step of the erfc evaluation. In critical parts, we exploit double-word (also called "double-double" in the literature) arithmetic to achieve implementations that are fast, yet accurate up to 50 bits, even for large input arguments. We demonstrate that for high required accuracies the automatically generated code has performance comparable to that of the standard libm and for lower ones our code demonstrated roughly 25% speedup.

Keywords. Error function, floating-point arithmetic, error analysis, semi-automated code generation.

1 Introduction

Erf and erfc are important functions, they have applications in statistics and finance [1], Gaussian sampling in cryptography [2], partial differential diffusion

equations, etc. They are significantly more complex to implement than exponentials, logarithms and trigonometric functions. The main reasons for that is that there is no obvious relation that allow some range reduction, and that the asymptotic behaviour of these functions make them difficult to approximate by polynomials or rational functions for large arguments. This is in particular the case for erfc(x) with large x: roughly speaking the function is close enough to its asymptotical expression $e^{-x^2}/(x\sqrt{\pi})$ to be very "flat" — which is not a behaviour easily approximable by polynomials — but not close enough to be just replaced by the asymptotical expression.

Much work on the implementation of these functions is due to Cody [3–6]. A recent presentation is given by Beebe [1, Chapter 19]. Beebe summarizes that existing approaches for the approximations to the erfc work on segment-by-segment basis. A classical approach is to decompose the implementation domain into several subdomains and use rational approximations. However, achieving close-to-1/2ulp error is non-trivial even for small $|x|$: for example the current GNU libm has a 4ulp error for $x = 2797326291814245/2^{51} = 1.24\cdots$ and the current Apple libm has 7ulp error for $x = 7812247216079717/2^{51} = 3.47\cdots$. Since the field of application of the erfc function is very vast, different flavors of the functions are required, with different accuracy/speed tradeoffs.

With this work we aim at satisfying needs for different flavors of the erfc function by an automatic implementation. In particular, we rely on the automatic code generator for mathematical functions Metalibm[1] [7]. This tool provides implementations using polynomial approximations and ensures that a user-given target error bound for both approximation and evaluation is satisfied. Metalibm can be used in a "naive way" when the function is relatively regular and easy to approximate by polynomials, for example this is the case for the erfc function in the domain $[0; 5]$. However, in more complex cases, such as erfc(x) for large x, some expert-knowledge is still required.

We will propose a solution that is parametrized by the user given error bound and consider in details the case when the underlying arithmetic is binary64 (a.k.a. "double precision"). First, we present some useful properties of the erfc function and our approximation technique. The center-point of our solution is the ability to control accuracy of certain polynomial approximations via Metalibm. Our goal is, given a certain error budget, to deduce the required accuracy of different steps of the evaluation of erfc. For this we first provide a completely generic step-by-step error analysis with a straightforward error budget distribution. Then, we assume binary64 data formats and show how, by using double-word arithmetic in critical places, one can significantly improve the accuracy of computations. Finally, we demonstrate our solution on several flavors of the erfc function.

[1] `http://www.metalibm.org/ANRMetaLibm/`

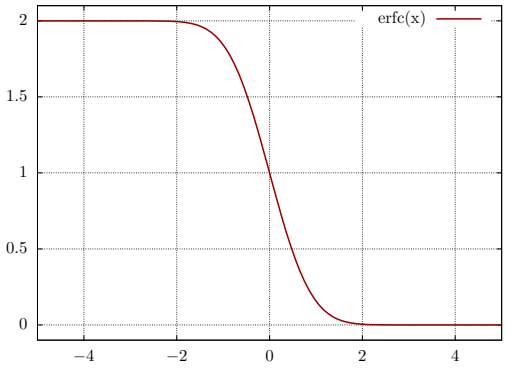

Figure 1: Function erfc for input arguments in $[-5; 5]$.

2 Background

In the following, we assume that the underlying arithmetic is the binary64 arithmetic of the IEEE 754 Floating-Point (FP) Standard [8]. For expressing errors, we will frequently use the "rounding unit" $u = 2^{-53}$. RN is the round-to-nearest function.

Functions erf and erfc are defined as follows:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{1}$$

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \tag{2}$$

Definitions (1) and (2) imply the following properties (see [9] for more)

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) \tag{3}$$

$$\mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x) \tag{4}$$

For large x, $\mathrm{erfc}(x)$ has the following asymptotic behaviour

$$\mathrm{erfc}(x) \sim_{x \to +\infty} \frac{e^{-x^2}}{x\sqrt{\pi}}. \tag{5}$$

A plot of function erfc is given Fig. 1.
We have the following properties:

- if $x \leq -6601809522387275/2^{50} \approx -5.863584748755$ then the binary64 number nearest $\mathrm{erfc}(x)$ is 2;
- if $x \geq x_{\mathrm{LARGE}} = 3735631527617609/2^{47} \approx 26.543258$ then $\mathrm{erfc}(x)$ is in the subnormal domain;

- if $x > x_{\text{BIG}} = 1915860633068287/2^{46} \approx 27.22601711$ then the binary64 number nearest $\text{erfc}(x)$ is 0;
- if $x_{\text{MID}} = 4295860071587845/2^{53}$ is the binary64 number nearest the solution of the equation $\text{erfc}(x) = 1/2$ then $\text{RN}(\text{erfc}(x_{\text{MID}})) = 1/2$.

These properties facilitate the implementation and testing of the erfc function, as described below.

3 Approximation technique

The basic technique consists in dividing the input domain into several subdomains. In the "internal" subdomains (i.e., those that do not contain the extremal input values), conventional polynomial or rational approximation techniques hold and Metalibm automatically gives convenient approximations that satisfy a user-given target error bound. In the other, "external" subdomains, one has to cleverly use the asymptotic behaviour of the function. For erfc, the external subdomain of the negative side is easily handled. Hence, in this paper, we will mainly focus on the external subdomain of the positive side (say, the domain $[5, x_{\text{BIG}}]$). That subdomain is much more difficult to tackle: roughly speaking, function erfc becomes too "flat" to be easily approximable by a polynomial or a rational function, but not close enough to its asymptotic equivalent $e^{-x^2}/(x\sqrt{\pi})$ to be just replaced by it.[2] In that domain, we will use the asymptotic expression *and* a correction. More precisely, defining

$$g(x) = \frac{1}{xe^{x^2}\,\text{erfc}(x)} - 2, \tag{6}$$

we will design, using Metalibm, an approximation to this easier to approximate function g (see Fig. 2). Function g is a decreasing function, negative in $[5, x_{\text{BIG}}]$. Hence $|g(x)|$ is less than its limit at $+\infty$, namely $2-\sqrt{\pi} \leq 0.228$. The evaluation of erfc in $[5, x_{\text{BIG}}]$ will be reduced to the evaluation of the approximation to $g(x)$, followed by the use of

$$\text{erfc}(x) = \frac{e^{-x^2}}{2x + xg(x)}. \tag{7}$$

Eq. (7), however, cannot be used in a straightforward manner. The main reason is that the computation of e^{-x^2} may underflow. Since $x \in [5; x_{\text{BIG}}]$, we have $-x^2 \in [-741.256, -25]$, while if we want it to use all the available precision (i.e., the output is not in subnormal) the exp function accepts arguments between $-708.396\cdots$ and $670.96\cdots$. The solution is to scale the computation by some number e^s. To facilitate rescaling back by e^{-s}, we propose to set $s = k\ln 2$, where k is an integer, in a domain that we are going to explicit, chosen such that s is as close as possible to a FP number. Hence multiplication by $e^{\pm s}$ is

[2] For instance, for $x = 26$, the ratio between erfc and the asymptotic equivalent is $0.99926\cdots$, i.e., still very far from 1.

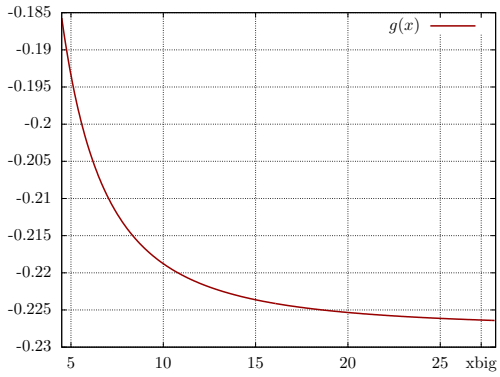

Figure 2: Function $g(x)$ for $x \in [5; x_{\text{BIG}}]$.

replaced by multiplication by $2^{\pm k}$. Depending on the desired final accuracy, and denoting $\Omega = 2^{1024} - 2^{971}$ the largest positive FP number, we can have two different strategies for choosing k:

1. We wish to represent $-x^2 + k \ln 2$ as a full precision FP number. This gives the constraints

$$-741.256 + k \ln 2 > -1022 \ln 2 \tag{8}$$
$$-25 + k \ln 2 < \ln \Omega. \tag{9}$$

2. We wish to represent $-x^2 + k \ln 2$ as a "double word" (also called "double-double") number, i.e., the unevaluated sum (t_h, t_ℓ) of two floating-point numbers. This gives the constraints

$$-741.256 + k \ln 2 > (-1022 + 54) \ln 2 \tag{10}$$
$$-25 + k \ln 2 < \ln \Omega. \tag{11}$$

The first set of constraints gives $48 \leq k \leq 1060$, and the second one gives $102 \leq k \leq 1060$. In these domains, we choose k that minimizes $|k \ln 2 - \text{RN}(k \ln 2)|$. We obtain $k = 61$ in the first case and $k = 1021$ in the second one.

Hence, what we finally compute is

$$\text{erfc}(x) = 2^{-k} e^{-x^2 + \hat{s}} \cdot \frac{1}{2x + xg(x)}, \tag{12}$$

where $\hat{s} = \text{RN}(k \ln 2)$, $k = 61$ for a binary64 representation with an absolute error 0.2583u; and $\hat{s} = s_h + s_\ell$, $s_h = \text{RN}(k \ln 2)$, $s_\ell = \text{RN}(k \ln 2 - s_h)$, $k = 1021$ for a double-word representation with an absolute error $0.0289u^2$.

Note that for $x \geq x_{\text{LARGE}}$, $\text{erfc}(x)$ is in the subnormal domain, so that trying to guarantee a good relative error no longer makes sense: for such values, one

needs to focus on the absolute error (and the best possible absolute error bound is $\frac{1}{2}\text{ulp}(\text{subnormal}) = 2^{-1075}$).

The general problem is to evaluate erfc via (12) with a user-required accuracy. The outline of our technique can be summarized as:

- On the interval $[0; x_{\text{MID}}]$ compute an approximation to the erf function, using Metalibm, and exploit Property (3);
- On the interval $(x_{\text{MID}}; 5)$ compute a polynomial approximation to the erfc function, using Metalibm;
- On the interval $[5; x_{\text{BIG}}]$, use (12), and for that, determine the required accuracy for each step of the evaluation of (12) and for the approximations to $g(x)$ and exp.
- For negative arguments, exploit Property (4).

To guarantee a user-given relative error bound on the evaluation of erfc, we provide a fine-grained error analysis and a repartition of the error budget among the various steps, where the accuracy requirements for each step are modeled as functions of the target bound ε. We first give a generic step-by-step error analysis which assumes arbitrary error bounds even for the basic arithmetic operations. This approach makes it possible to assume different underlying hardware formats. If the arithmetic operations are performed in binary64 (and we do not use double-word arithmetic), the relative accuracy of the basic arithmetic operations strongly restricts the maximum possible accuracy of the evaluation of erfc. In section 5 we show how to use double-word arithmetic for critical parts of the evaluation and significantly improve the range of supported accuracies. The central point here is that we can control the accuracy of the approximations to g and exp using Metalibm.

4 Generic error analysis

As said above, we focus on the "difficult case" $x \in [5, x_{\text{BIG}}]$.

The computations involved in (12) are performed in several steps. Let us use the following notation for them:

$$\begin{aligned} y(x) &= 2^{-k} a(x)/d(x) \\ a(x) &= e^{t(x)} \\ t(x) &= -x^2 + k \ln 2, \\ d(x) &= 2x + r(x) \\ r(x) &= x g(x) \end{aligned}$$

In the following we analyze these various steps and assume a generic a priori error bound for each of them, chaining the bounds in the end to obtain a table of necessary accuracies. We use a naive equal budget distribution for the generic

Table 1: Summary of the generic error analysis based on equal error budget distribution on each step.

Computation step	Error terms	Examples of error requirements			
	$\lvert\varepsilon_y\rvert$	δ	2^{-32}	2^{-46}	2^{-53}
$y(x) = 2^{-k}a(x)/d(x)$	$\lvert\varepsilon_{\text{DIV}}\rvert$	$\delta/4$	2^{-34}	2^{-48}	2^{-55}
$a(x) = e^{t(x)}$	$\lvert\varepsilon_{\text{EXP}}\rvert$	$\delta/16$	2^{-36}	2^{-50}	2^{-57}
$t(x) = -x^2 + k\ln 2$	$\lvert\Delta_t\rvert$	$\ln(1+\delta/16)$	$1.99 \cdot 2^{-37}$	$1.99 \cdot 2^{-51}$	$1.99 \cdot 2^{-58}$
$d(x) = 2x + r(x)$	$\lvert\varepsilon_{\text{ADD}}\rvert$	$\delta/8$	2^{-35}	2^{-49}	2^{-56}
$r(x) = xg(x)$	$\lvert\varepsilon_{\text{MUL}}\rvert$	$\frac{\delta}{4\overline{\alpha}(8+\delta)}$	$1.94 \cdot 2^{-35}$	$1.94 \cdot 2^{-49}$	$1.94 \cdot 2^{-56}$
$g(x)$	$\lvert\varepsilon_g\rvert$	$\frac{\delta}{4\overline{\alpha}(8+\delta)}$	$1.94 \cdot 2^{-35}$	$1.94 \cdot 2^{-49}$	$1.94 \cdot 2^{-56}$

analysis, and slightly refine it in section 5. To distinguish the "exact" values defined above and the "computed" ones, we will add a "hat" to the computed ones: $\hat{a}(x)$ is the computed value that approximates $a(x)$. In the following, relative errors of arithmetic operations such as addition or division are parameters (e.g., variable ε_{ADD} below). This is because we want the analysis to be as generic as possible. Of course, if we perform the operations in binary64 arithmetic (i.e., we do not use double-word operations), the relative error bound on the arithmetic operations is u, provided that no underflow or overflow occurs.

4.1 Computation of $y(x) = 2^{-k}a(x)/d(x)$

The left shift 2^{-k} is an exact operation. However, both $a(x)$ and $d(x)$ are computed with some relative errors:

$$\hat{a}(x) = a(x)(1+\varepsilon_a), \tag{13}$$

$$\hat{d}(x) = d(x)/(1+\varepsilon_d). \tag{14}$$

Hence, we obtain an approximation $\hat{y}$ to $\text{erfc}(x)$ as:

$$\begin{aligned}\hat{y} &= 2^{-k}\text{RN}\left(\hat{a}(x)/\hat{d}(x)\right)\\ &= 2^{-k}\left(a(x)/d(x)\right)(1+\varepsilon_{\text{DIV}})(1+\varepsilon_a)(1+\varepsilon_d)\\ &= 2^{-k}\left(a(x)/d(x)\right)(1+\varepsilon_y),\end{aligned} \tag{15}$$

where

$$\begin{aligned}\varepsilon_y =& \varepsilon_a + \varepsilon_d + \varepsilon_{\text{DIV}}\\ &+ \varepsilon_a\varepsilon_d + \varepsilon_{\text{DIV}}\varepsilon_a + \varepsilon_{\text{DIV}}\varepsilon_d + \varepsilon_{\text{DIV}}\varepsilon_a\varepsilon_d.\end{aligned} \tag{16}$$

If it must be ensured that $|\varepsilon_y| \le \varepsilon$, $\varepsilon > 0$, then different strategies for the error budget repartition are possible. When distributing the error budget equally, one requires

$$|\varepsilon_a| \le \varepsilon/4, \quad |\varepsilon_d| \le \varepsilon/4, \quad |\varepsilon_{\text{DIV}}| \le \varepsilon/4. \tag{17}$$

Indeed, if the above conditions hold, we obtain $|\varepsilon_y| \leq \frac{61}{64}\varepsilon$.

4.2 Computation of $a(x) = e^{t(x)}$

We assume that the floating-point function EXP implements the real function exp with relative error ε_{EXP}. To guarantee a *relative* error bound on the exponential of $t(x)$, we need to guarantee an *absolute* error bound on $t(x)$. Thus, we assume

$$\hat{t}(x) = t(x) + \Delta_t. \tag{18}$$

We have

$$\begin{aligned} \hat{a}(x) &= \text{EXP}(t(x) + \Delta_t) = e^{t(x)+\Delta_t}\,(1+\varepsilon_{\text{EXP}}) \\ &= e^{t(x)}(1 + e^{\Delta_t} - 1)\,(1+\varepsilon_{\text{EXP}}) \\ &= e^{t(x)}\,(1+\varepsilon_a)\,, \end{aligned} \tag{19}$$

where, when denoting $\tilde{\varepsilon_t} = e^{\Delta_t} - 1$,

$$\varepsilon_a = \tilde{\varepsilon}_t + \varepsilon_{\text{EXP}} + \tilde{\varepsilon}_t\varepsilon_{\text{EXP}}. \tag{20}$$

To ensure a relative error bound $\varepsilon > 0$ for the evaluation of $a(x)$, it suffices to guarantee $|\varepsilon_{\text{EXP}}| \leq \varepsilon/4$ and $|\tilde{\varepsilon}_t| \leq \varepsilon/4$, or in terms of absolute error, $|\Delta_t| \leq \ln(1+\varepsilon/4)$. Indeed, if these bounds hold, then $|\varepsilon_a| \leq \frac{9}{16}\varepsilon$.

4.3 Computation of $t(x) = -x^2 + k\ln 2$

With our approach, the approximation of $k\ln 2$ introduces an absolute error equal to $0.2583u$ (or $0.0289u^2$ if double-word arithmetic is used), which restricts the range of possible precision requirements. Indeed, computing $t(x)$ accurately needs the use of double-word (or even, possibly, multiple-word) arithmetic. This will be detailed in Section 5. Let us analyze here what happens if we just perform a straightforward binary64 evaluation of $t(x)$:

$$x_h = \text{RN}(x^2), \qquad \hat{t} = \text{RN}(-x_h + \hat{s}), \tag{21}$$

where $\hat{s} = \text{RN}(61\ln 2)$. The overall approximation error $\left|\hat{t} - t\right|$ can be decomposed in the following way:

$$\left|\hat{t} - t\right| \leq |\hat{s} - k\ln 2| + \left|x_h - x^2\right| + \left|\hat{t} - (-x_h + \hat{s})\right|. \tag{22}$$

We remind the reader that $|\hat{s} - k\ln 2| \leq 0.2583u$. Since $x \in [5; x_{\text{BIG}}]$, we have $x^2 \leq 742$ and the error due to squaring is

$$\left|x_h - x^2\right| \leq \frac{1}{2}\text{ulp}(742) \leq 2^{-44}. \tag{23}$$

The error due to addition is

$$\left|\hat{t} - (-x_h + \hat{s})\right| \leq \frac{1}{2}\mathrm{ulp}(-742 + \hat{s}) \leq 2^{-44}. \tag{24}$$

Hence, the overall error is bounded by

$$\left|\hat{t} - t\right| \leq 2^{-44} + 2^{-44} + 0.2583u \leq 1024.2583u. \tag{25}$$

4.4 Computation of $d(x) = 2x + r(x)$

The multiplication $2x$ is exact. The term $r(x)$ is computed with a certain relative error ε_r. Thus, we actually compute

$$\begin{aligned} \hat{d}(x) &= \mathrm{RN}\left(2x + r(x)(1+\varepsilon_r)\right) \\ &= \left(2x + r(x) + r(x)\varepsilon_r\right)(1+\varepsilon_{\mathrm{ADD}}) \\ &= \left(2x + r(x)\right)(1+\varepsilon_d) \end{aligned} \tag{26}$$

where

$$\varepsilon_d = \varepsilon_{\mathrm{ADD}} + \alpha(x)\varepsilon_r + \alpha(x)\varepsilon_r\varepsilon_{\mathrm{ADD}} \tag{27}$$

with $\alpha(x) = \frac{r(x)}{2x+r(x)} = 1 - 2xe^{x^2}\,\mathrm{erfc}(x)$. Function α is a decreasing function, negative in $[5, x_{\mathrm{BIG}}]$. Hence we can bound $|\alpha(x)|$ by its limit at $+\infty$, namely $\overline{\alpha} = 2\sqrt{\pi} - 1 \leq 0.129$.

Given a generic error bound $\varepsilon > 0$, different error budget repartition strategies are possible for (27). For example, one can choose to dedicate half of the error budget to the rounding error due to addition and then adapt the requirement for $r(x)$ accordingly (which consequently influences the bound on the approximation error for $g(x)$). If $|\varepsilon_{\mathrm{ADD}}| \leq \varepsilon/2$, then

$$|\varepsilon_d| \leq \varepsilon/2 + |\alpha(x)| \cdot |\varepsilon_r|\,(1 + \varepsilon/2). \tag{28}$$

Consequently, if we require

$$|\varepsilon_r| \leq \frac{\varepsilon}{\overline{\alpha}(2+\varepsilon)}, \tag{29}$$

then $|\varepsilon_d| \leq \varepsilon$.

4.5 Computation of $r(x) = xg(x)$

Rounding errors in the computation of $r(x)$ can be handled analogously. Function $g(x)$ is approximated, using Metalibm, with some accuracy ε_g:

$$\hat{g}(x) = g(x)(1+\varepsilon_g). \tag{30}$$

Thus, the approximation $\hat{r}(x)$ to $r(x)$ satisfies

$$\begin{aligned} \hat{r}(x) &= xg(x)(1+\varepsilon_g)(1+\varepsilon_{\mathrm{MUL}}) \\ &= xg(x)\left(1+\varepsilon_r\right), \end{aligned} \tag{31}$$

where

$$\varepsilon_r = \varepsilon_g + \varepsilon_{\text{MUL}} + \varepsilon_g \varepsilon_{\text{MUL}}. \tag{32}$$

If the error budget is repartitioned equally, i.e.

$$|\varepsilon_{\text{MUL}}| \le \varepsilon/4, \quad |\varepsilon_g| \le \varepsilon/4 \tag{33}$$

then $|\varepsilon_r| \le \varepsilon$.

4.6 Summary of the error bounds

In the above analysis we considered generic error bounds for each evaluation. Now, given a relative error bound $\delta > 0$ that must be satisfied by the implementation of erfc, we can express bounds on each step of the evaluation in terms of δ.

First, from (17) we obtain: $|\varepsilon_a| \le \delta/4$, $|\varepsilon_d| \le \delta/4$, and $|\varepsilon_{\text{DIV}}| \le \delta/4$. Then, we use the bound for ε_a as the requirement in section 4.2 and obtain that the exponential function must have a relative error bounded by $\delta/16$ and $t(x)$ must be computed with absolute error $\ln(1 + \delta/16)$.

We continue analogously and summarize the obtained accuracy requirements in Table 1. There we also illustrate the obtained bounds for several values of δ: 2^{-32}, 2^{-46} and 2^{-53}.

Naturally, the step requiring the highest accuracy is the computation of $t(x)$. With a binary64 approximation to $k \ln 2$ and a straightforward evaluation, the absolute error Δ_t is bounded by $1024.2583u \le 0.5002 \cdot 2^{-42}$. Hence, in this case, the erfc function can be approximated only up to relative error $0.5002 \cdot 2^{-38}$. As we show below, using double-word arithmetic can significantly improve the accuracy of $t(x)$.

5 Use of double-word arithmetic

As said above, if all arithmetic operations are performed in binary64 arithmetic, the relative errors due to these operations in the above analysis are bounded by u. Obviously, the error budget must be large enough to account for the arithmetic operations and leave some room for the approximations to exp and $g(x)$, whose accuracy we can control using Metalibm. We have seen in section 4.6 that straightforward binary64 arithmetic operations yield extremely high errors and, hence, restrict the minimum size of the error budget.

In this section we still assume binary64 arithmetic but at some critical places, such as the computation of $t(x)$, we use double-word arithmetic [10–13]. We need two well-known algorithms of the floating-point literature. Algorithm Fast2Sum, that takes two FP numbers a and b as input and returns two FP numbers s and t such that $s = \text{RN}(a + b)$ and $t = a + b - s$ (that is, t is the error of the floating-point addition of a and b), and Algorithm Fast2Mul, that requires the availability of an FMA instruction and takes two FP numbers a and b as input

and returns two FP numbers π and ρ such that $\pi = \mathrm{RN}(ab)$ and $\rho = ab - \pi$. Fast2Sum(a, b) consists in first computing $s = \mathrm{RN}(a + b)$, then $z = \mathrm{RN}(s - a)$ and $t = \mathrm{RN}(b - z)$. Fast2Mul(a, b) consists in computing $\pi = \mathrm{RN}(ab)$ and, assuming FMA is available, $\rho = \mathrm{RN}(ab - \pi)$.

To return a correct result, Fast2Sum requires the floating-point exponent of a to be larger than or equal to the floating-point exponent of b. See [14] fort a recent presentation.

Our goal now is to take into account the accuracy of the straightforward binary64 operations, possibly using double-word arithmetic for certain computations, and to deduce the requirements on the accuracy of the functions exp and g.

In Section 5.6 we analyze the restrictions due to double-word arithmetic and deduce the range of accuracies for the erfc function that can be supported with the proposed approach.

5.1 Computation of $y = 2^{-k}a(x)/d(x)$

Assuming binary64 division, we have $|\varepsilon_{\mathrm{DIV}}| \leq u$. Then, bound (16) becomes

$$|\varepsilon_y| \leq u + (1 + u)(\varepsilon_a + \varepsilon_d + \varepsilon_a\varepsilon_d) \tag{34}$$

Hence, to ensure a generic error bound $\varepsilon > u$, it suffices to guarantee $|\varepsilon_a| \leq \frac{\varepsilon - u}{4(1+u)}$ and $|\varepsilon_d| \leq \frac{\varepsilon - u}{4(1+u)}$.

5.2 Ensuring a smaller absolute error bound on $t(x)$

To improve the accuracy of the evaluation of $t(x)$, we compute it as a double-word number, i.e., as an unevaluated sum $t_h + t_\ell$ of two FP numbers. In the following we propose two methods for the evaluation of $t(x)$. The first one guarantees an absolute error bounded by $1.009 \cdot 2^{-48}$ at the cost of 6 FP operations, and the second, more accurate one guarantees an error bound $1.034 \cdot 2^{-55}$ at the cost of 10 FP operations.

Method 1 (gives $\left|\hat{t} - t\right| \leq 32.259u \leq 1.009 \cdot 2^{-48}$). As previously, we use $\hat{s} = \mathrm{RN}(61\ln(2))$.

1. Compute x^2 exactly using Fast2Mul

   $$(x_h, x_\ell) = \text{Fast2Mul}(x, x)$$

   Since $x \leq x_{\mathrm{BIG}}$, we have $x^2 \leq 742$, hence $|x_\ell|$ is less than $\frac{1}{2}\mathrm{ulp}(742) = 2^{-44}$.

2. Subtract the lower part:

   $$e = \mathrm{RN}(\hat{s} - x_\ell)$$

   From the value of $\hat{s}$ and the bound on x_ℓ we have $|\hat{s} - x_\ell| \leq 42.29$. Therefore $|e - (\hat{s} - x_\ell)| \leq \frac{1}{2}\mathrm{ulp}(42.29) = 2^{-48}$.

3. Subtract the higher part:

$$(t_h, t_\ell) = \text{Fast2Sum}(-x_h, e)$$

where the usage of Fast2Sum is possible, since $x_h \geq 25$ and $e \leq 42.29$, hence exponent$(x_h) \geq$ exponent(e).

Overall, adding the error with which $\hat{s}$ approximates $61\ln(2)$, we deduce that $t_h + t_\ell$ approximates $t(x)$ with an absolute error $|t_h + t_\ell - t| \leq 2^{-48} + 0.2583u \leq 32.259u \leq 1.009 \cdot 2^{-48}$.

Method 2 (gives $\left|\hat{t} - t\right| \leq 0.2584u \leq 1.034 \cdot 2^{-55}$)

1. As in Method 1, compute x^2 exactly using Fast2Mul

$$(x_h, x_\ell) = \text{Fast2Mul}(x, x)$$

2. Subtract the lower part x_ℓ using Fast2Sum:

$$(r_h, r_\ell) = \text{Fast2Sum}(\hat{s}, -x_\ell),$$

which gives $|r_h| \approx 42.281$ and $|r_\ell| \leq 2^{-48}$.

3. Compute the higher part of the result using Fast2Sum:

$$(z_h, z_\ell) = \text{Fast2Sum}(-x_h, r_h).$$

Here we can use Fast2Sum since $x_h \geq 25$ and $r_h \approx 42.281$, i.e. exponent(x_h) $\geq$ exponent(r_h). We have $|-x_h + r_h| \leq |-742 + 42.281| \leq 700$. Hence, $|z_\ell| \leq \frac{1}{2}\text{ulp}(700) = 2^{-44}$.

4. Add together the lower parts:

$$e = \text{RN}(z_\ell + r_\ell),$$

for which we have $|z_\ell + r_\ell| \leq 2^{-44} + 2^{-48}$ so that the error of that FP addition is bounded by $\frac{1}{2}\text{ulp}(2^{-44}) = 2^{-97}$.

$z_h + e$ approximates $t(x)$ with an absolute error less than $0.2583u + 2^{-97} \leq 0.2584u \leq 1.034 \cdot 2^{-55}$. However, the pair (z_h, e) is not necessarily a double-word, since $|e|$ may be significantly larger than $\frac{1}{2}\text{ulp}(z_h)$. Hence we need to "nomalize" that pair as follows:

$$(t_h, t_\ell) = \text{Fast2Sum}(z_h, e).$$

The Fast2Sum algorithm can be used here: $|z_h| < |e|$ would mean that the subtraction $r_h - x_h$ is a "catastrophic cancellation". A consequence of this would be that $z_\ell = 0$, so that $|e| = |r_\ell| \leq 2^{-48}$, and that r_h, x_h (and therefore z_h) would be multiple of $\text{ulp}(r_h) = 2^{-47}$.

5.3 Computation of $a(x) = e^{t(x)}$

Now, since $t(x)$ is approximated by a double-word number (t_h, t_ℓ), the evaluation of its exponential must be adjusted accordingly.

If Method 1 is used for the evaluation of $t(x)$, then $t_h + t_\ell$ approximates $t(x)$ with an absolute error $\Delta_t \leq 32.259u \leq 1.009 \cdot 2^{-48}$. Hence, we have

$$e^t = e^{t_h} e^{t_\ell} e^{\Delta_t}. \tag{35}$$

The lower part t_ℓ is small in magnitude (less than 2^{-44}), thus we can replace its exponential by $E_\ell = 1 + t_\ell$ (which is not necessarily a FP number, we just memorize t_ℓ). The overall absolute error of that substitution is bounded as follows:

$$|E_\ell - e^{t_\ell}| = |(1 + t_\ell) - e^{t_\ell}| \leq \tfrac{t_\ell^2}{2!} e^{t_\ell},$$

therefore the relative error

$$\varepsilon_{E_\ell} = \frac{|E_\ell - e^{t_\ell}|}{e^{t_\ell}}$$

is bounded by $t_\ell^2/2 \leq 2^{-89} = 2^{-36}u$. The relative error ε_t resulting from the absolute error Δ_t is bounded by $|e^{\Delta_t} - 1|$, which gives

$$|\varepsilon_t| \leq 32.26u.$$

The product $e^{t_h} \cdot E_\ell$ is therefore computed (with an FMA instruction) as $e^{t_h} + e^{t_h} t_\ell$. The relative error of this operation is bounded by ε_{FMA}. Assuming that the exponential e^{t_h} is computed with a certain relative error ε_{E_h}, we have

$$\hat{a}(x) = e^{t_h} E_\ell \, (1 + \varepsilon_{E_h}) \, (1 + \varepsilon_{E_\ell}) \, (1 + \varepsilon_t) \, (1 + \varepsilon_{\text{FMA}})$$

Assuming $\varepsilon_{\text{FMA}} \leq u$ and using the bounds on ε_t and ε_{E_ℓ} we can express the relative error ε_a as a function of the unit roundoff and accuracy of exp:

$$|\varepsilon_a| \leq (1 + \varepsilon_{E_h}) \, (1 + 33.261u) - 1. \tag{36}$$

Hence, to guarantee a generic relative error $\varepsilon > 33.261u$ for ε_a, it is sufficient to require the exponential function with an error bounded by

$$|\varepsilon_{E_h}| \leq \frac{\varepsilon - 33.261u}{1 + 33.261u}. \tag{37}$$

If Method 2 is used to evaluate $t(x)$, the error study is similar to previously, with now $\varepsilon_t \leq 0.2585u$, and $|t_\ell| \leq 2^{-43} + 2^{-48}$, which gives $\varepsilon_{E_\ell} \leq (1089/2^{44}) \cdot u$, so that

$$|\varepsilon_a| \leq (1 + \varepsilon_{E_h}) \, (1 + 1.259u) - 1. \tag{38}$$

Hence, to guarantee a generic relative error $\varepsilon > 1.259u$ for ε_a, it is sufficient to require the exponential function with an error bounded by

$$|\varepsilon_{E_h}| \leq \frac{\varepsilon - 1.259u}{1 + 1.259u}. \tag{39}$$

It is possible to improve the bound $1.259u$ as follows. One can evaluate e^{t_h} as a double-word number and then compute the product $e^{t_h} \cdot e^{t_\ell}$ using the algorithm DWTimesFP1 of [12], for which $|\varepsilon_{\text{MUL}}| \leq (3/2)u^2 + 4u^3$. By doing this, a will be a double-word number (this of course will need to be taken into account in the final step of computation of $y(x)$). We then obtain

$$|\varepsilon_a| \leq (1 + \varepsilon_{E_h})\,(1 + 0.259u) - 1. \tag{40}$$

5.4 Computation of $d(x) = 2x + r(x)$

In the definition (27) for ε_d we substitute $|\varepsilon_{\text{ADD}}| \leq u$ and obtain

$$|\varepsilon_d| \leq u + |\alpha(x)| \cdot |\varepsilon_r| + |\alpha(x)| \cdot |\varepsilon_r| \cdot u \tag{41}$$

If we require the relative error $|\varepsilon_d|$ to be bounded by a certain $\varepsilon > u$, the requirement on ε_r can be expressed as

$$|\varepsilon_r| \leq \frac{\varepsilon - u}{\overline{\alpha}(1+u)}. \tag{42}$$

Since $\overline{\alpha} \leq 0.129$, the above bound can be simplified to $|\varepsilon_r| \leq 7.76(\varepsilon - u)$.

5.5 Computation of $r(x) = xg(x)$

In binary64 arithmetic, $|\varepsilon_{\text{MUL}}| \leq u$. In order to satisfy a generic error bound $\varepsilon > u$ for the relative error ε_r, it is sufficient to satisfy

$$|\varepsilon_g| \leq \frac{\varepsilon - u}{1+u}. \tag{43}$$

However, in practice Metalibm often reports $\hat{g}(x)$ to be represented as a double-word number (g_h, g_ℓ) such that $|g_\ell| \leq u \cdot |\hat{g}|$. This leads to modifications in the computation of $r(x)$ and, in particular, a new error bound for the multiplication must be deduced. We propose to evaluate $x(g_h + g_\ell)$ as follows:

1. Multiply x by the lower part:

$$\hat{\xi} = \text{RN}(xg_\ell), \tag{44}$$

   which gives $\hat{\xi} = xg_\ell \cdot (1 + \varepsilon_\xi)$, with $|\varepsilon_\xi| \leq u$.

2. Multiply x by the higher part g_h and add $\hat{\xi}$ using an FMA instruction:

$$\hat{\phi} = \text{RN}(xg_h + \hat{\xi}), \tag{45}$$

   which gives $\hat{\phi} = (xg_h + \hat{\xi}) \cdot (1 + \varepsilon_{\text{FMA}})$, with $|\varepsilon_{\text{FMA}}| \leq u$.

   We have

$$\hat{\phi} = x\hat{g} + xg_\ell\varepsilon_\xi + x\hat{g}\varepsilon_{\text{FMA}} + xg_\ell\varepsilon_\xi\varepsilon_{\text{FMA}},$$

Table 2: Error bounds for approximations to functions exp and g that work for erfc accurate up to $0.76 \cdot 2^{-50}$.

Error terms	Examples of error requirements		
$\lvert\varepsilon_y\rvert$	δ	2^{-32}	2^{-46}
$\lvert\varepsilon_{E_h}\rvert$	Method 1 for $\delta > 0.52 \cdot 2^{-45}$ $\frac{\frac{\delta-u}{4(1+u)}-33.261u}{1+33.261u}$	$0.49 \cdot 2^{-33}$	-
	Method 2 for $\delta > 0.76 \cdot 2^{-50}$ $\frac{\frac{\delta-u}{4(1+u)}-1.259u}{1+1.259u}$	$0.49 \cdot 2^{-33}$	$0.47 \cdot 2^{-47}$
$\lvert\varepsilon_g\rvert$	$\frac{7.76\left(\frac{\delta-u}{4(1+u)}-u\right)-u-u^2-u^3}{1+u+u^2+u^3}$	$0.96 \cdot 2^{-31}$	$0.92 \cdot 2^{-45}$

from which we deduce

$$|\hat{\phi} - x\hat{g}| \leq x\hat{g} \cdot (u + u^2 + u^3).$$

Hence, the overall relative error of the multiplication satisfies $\varepsilon_{\text{MUL}} \leq u + u^2 + u^3$.

Consequently, in order to satisfy a generic bound $\varepsilon > u + u^2 + u^3$ for the approximation of $r(x)$, it suffices to approximate function $g(x)$ with a relative error

$$|\varepsilon_g| \leq \frac{\varepsilon - u - u^2 - u^3}{1 + u + u^2 + u^3}. \tag{46}$$

5.6 Summary of the error bounds

Given an error bound δ that must be satisfied by the implementation, we now deduce accuracy requirements for the approximations to functions exp and $g(x)$. Analogously to what was done in Section 4.6, we propagate the error requirement δ through the computations (see Table 2 for a summary). However, since the accuracy of the arithmetic operations is fixed, we must also deduce lower bounds on the feasible "error budget" δ.

The computation of $a(x)$ imposes $\delta \geq 0.52 \cdot 2^{-45} > 133.1u$ if Method 1 is used to evaluate $t(x)$ and $\delta \geq 0.76 \cdot 2^{-50} = 6.04u$ if Method 2 is used. The addition in the computation of $d(x)$ requires the error budget δ to be at least $0.626 \cdot 2^{-50} > 5u + 4u^2$. Consequently, the multiplication in $r(x)$ requires $\delta > 0.69 \cdot 2^{-50} > 5.51u$.

Hence, by combining binary64 arithmetic with the double-word arithmetic in critical places, we can implement the erfc function up to relative error $0.76 \cdot 2^{-50}$. Compared to the generic case when $t(x)$ is evaluated in straightforward binary64 arithmetic, this lower bound on δ is a significant improvement. This bound can be improved further if, instead of the binary64 approximation $\hat{s}$ we use a

double-word approximation $\hat{s} = s_h + s_\ell$, $s_h = \mathrm{RN}(k \ln 2)$, $s_\ell = \mathrm{RN}(k \ln 2 - s_h)$, $k = 1021$.

Table 2 shows that taking into account the accuracy of the basic arithmetic operations and tuning them up using double-word arithmetic, allowed us to slightly relax the requirements on the accuracy of the approximations to exp and g (compare to Table 1).

6 Numerical experiments

Matching the performance of the hand-tuned standard libm by an automatic code generation tool is a difficult challenge. The advantages of our approach are the numerical guarantees and the possibility of trying numerous different flavors for function implementation. As we shall see in this section, accuracy can be traded for performance while automatically providing numerical guarantees.

Our approach was implemented as a semi-automatic code generation tool, written in the C and Sollya[3] languages, that generates C source files. The automation is not yet full: the tool still requires a human intervention for the generation of approximations with Metalibm. We provide numerical results for implementations with several user-given relative accuracy requirements: 2^{-32}, 2^{-46} and $0.76 \cdot 2^{-50}$. Naturally, we compare our code to the standard GNU libm library provided with the GNU C compiler, in our case we use version 6.3.0.

Approximation choice Using Table 2, for any user-given error-bound $\delta \geq 0.76 \cdot 2^{-50}$, one can deduce the target errors for approximations to functions exp and g. We use Metalibm to generate approximations that satisfy the a priori error bounds. Apart from the target error, Metalibm supports a variety of parameters that influence the performance of the generated code. In particular, the degree of the approximation polynomials heavily affects the evaluation delay and the number of subdomains (restricting possibilities for the vectorization). In this work we semi-automatized the design space exploration, and chose the degrees that, empirically, suited best the chosen platform. However, one might wish to fully automatize this step of the implementation. In [15], an example of a possible methodology is presented.

Experimental settings Experiments were done on a machine with an Intel Xeon Gold 6136 CPU with 12 cores running at 3.00 GHz. The performance is reported in number of clock cycles. The absolute and relative errors (measured by comparison with GNU MPFR, version 4.0.0) are given in terms of ulps and multiples of u, respectively.

We measured performance and rounding errors on test inputs in three subdomains: the "easy" domain $[0; 5]$ where the Metalibm-generated polynomial approximations to erf and erfc are directly used; the "difficult" domain $[5; x_{\mathrm{LARGE}}]$ where our double-word based approach is used; the domain $[x_{\mathrm{LARGE}}; x_{\mathrm{BIG}}]$, where

[3] Freely available at `http://sollya.gforge.inria.fr`.

Table 3: Maximum absolute and relative errors of the libm and our implementations according to random tests

	$[0;5]$		$[5;x_{\text{LARGE}}]$		$[x_{\text{LARGE}},x_{\text{BIG}}]$
	abs	rel	abs	rel	abs
GNU libm	4 ulp	6.34 u	3 ulp	3.98 u	1.5 ulp
$\delta = 0.76 \cdot 2^{-50}$	2 ulp	3.84 u	4 ulp	4.02 u	1.5 ulp
$\delta = 2^{-46}$	18 ulp	21.07 u	15 ulp	16.6 u	1.5 ulp

RN(erfc(x)) is subnormal. We tested the code on both random and equally-distributed inputs in each domain, taking sets of 10^7 points.

Results Table 3 illustrates the maximum relative and absolute errors for the GNU libm and our implementations[4], while Figures 3 and 4 compare the performance with and without compiler optimizations, respectively.

Our first observation is that the GNU libm is far from providing an evaluation of erfc with error around 1/2ulp, having errors as large as 4ulp. We encountered even larger errors (up to 7ulp), with the standard Apple libm.

In terms of evaluation time, standard libm's performance remains unchanged when enabling compiler optimizations. Our implementations, in the meantime, experience $\approx 2x$ speedup with the $-O3$ optimization level.

Figure 3 illustrates the "staircase" effect in the reported timings for our implementations. This is due to the domain decomposition and possibly non-uniform approximation degrees. Overall, for the "easy" domain our implementations, even with the highest accuracy, have better or comparable performance. Moreover, our implementation with $\delta = 0.76 \cdot 2^{-50}$ has absolute error twice as small as the error of the libm.

In the domain $[5; x_{\text{LARGE}}]$ our automatically-generated implementation matches the performance of the libm while guaranteeing the relative error to be bounded by 2^{-46}. For applications where accuracy 2^{-32} is enough, our implementation permits to win roughly 25% of the evaluation time. Finally, our most accurate implementation, with accuracy $0.76 \cdot 2^{-50}$ has only 15% overhead in the average number of clock cycles, compared to the libm.

Metalibm is somewhat conservative in its implementation technique (actual rounding errors are usually smaller than the target error bound), and in practice our implementations are, as well, more accurate than the user-given error bound. For instance, the maximum relative error that we encountered during tests of our implementation with $\delta = 0.76 \cdot 2^{-50} = 6.08u$ is only $4.02u$.

We observed that in the region $[x_{\text{LARGE}}; x_{\text{BIG}}]$ the libm heavily looses in performance, reaching as much as 962 clock cycles. Our implementations experience a similar behavior but keeping up with around 500 cycles. It should

[4]The low-accuracy implementation with $\delta = 2^{-32}$ has, as expected, very large absolute and relative errors and is left out in the Table 3.

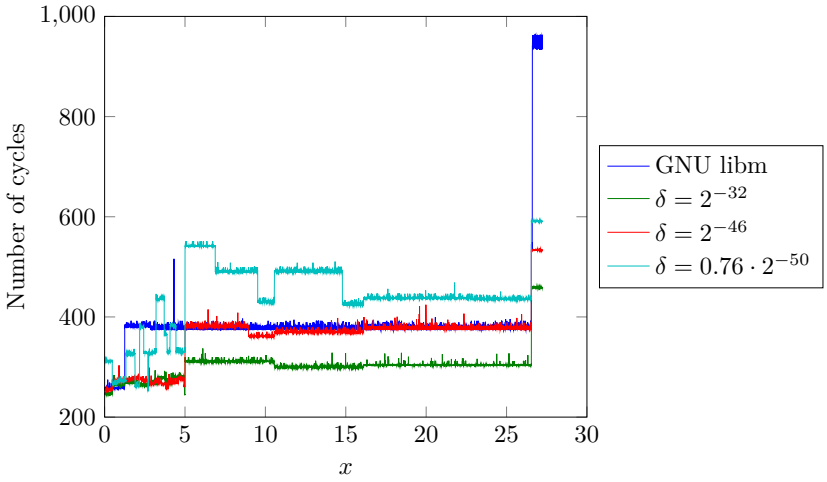

Figure 3: Performance comparison with maximum compiler optimization (-O3).

be noted that all implementations, including the GNU libm one, fail to ensure close-to-1/2ulp error for the subnormal outputs.

7 Conclusion and Perspectives

We have presented a partly-automated implementation of the complementary function. The advantages of this approach are i) easy adaptation to accuracy requirements, ii) guaranteed error bounds, and iii) the possibility of exploring a large design space. Our analysis could be adapted for other special functions, in particular those with a similar asymptotic behavior. For instance, we can easily extend our approach to the $e^{-x^2/2\sigma^2}$ Gaussian function, which is needed for Gaussian sampling for Lattice-based cryptography.

In general, the decisions on the "error budget" allocation for each step (done straightforwardly in this brief presentation) could be optimized for given criteria, for example towards improvement of the final error, or toward avoiding as much as possible costly multiple-word arithmetic. A future work will be to formalize such an optimization problem. Another future goal is to extend our approach to the inverse error function and design codes that support efficient vectorization.

References

[1] N. Beebe, *The Mathematical Function Handbook*. Springer, 2017.

[2] M. O. Saarinen, "Gaussian sampling precision and information leakage in lattice cryptography," *IACR Cryptology ePrint Archive*, 2015.

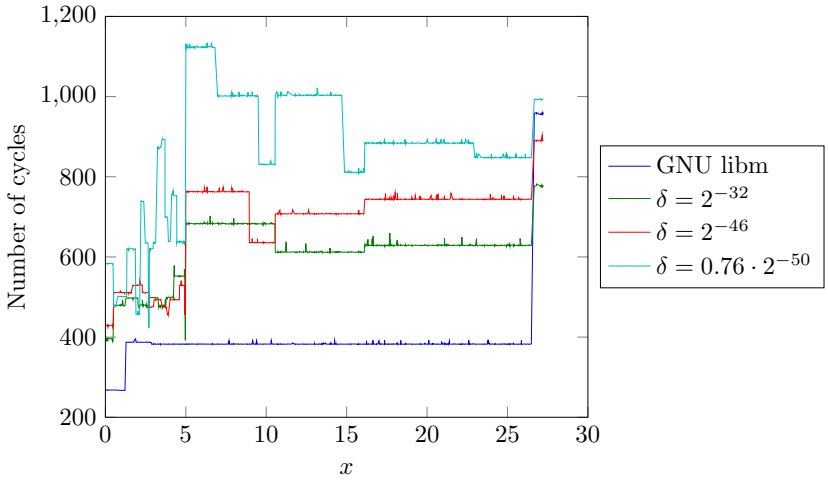

Figure 4: Performance comparison without compiler optimization (-O0).

[3] W. J. Cody, "The FUNPACK package of special function subroutines," *ACM Transactions on Mathematical Software*, vol. 1, no. 1, 1975.

[4] ——, "Rational Chebyshev approximations for the error function," *Mathematics of Computation*, vol. 23, no. 107, 1969.

[5] ——, "Performance evaluation of programs for the error and complementary error functions," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 29–37, Mar. 1990.

[6] ——, "Algorithm 715: SPECFUN – a portable FORTRAN package for special function routines and test drivers," *ACM Transactions on Mathematical Software*, vol. 19, no. 1, pp. 22–32, Mar. 1993.

[7] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Q. Lauter, "Code generators for mathematical functions," in *22nd IEEE Symposium on Computer Arithmetic*, 2015, pp. 66–73.

[8] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[9] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions with formulas, graphs and mathematical tables*, ser. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964.

[10] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision BLAS," Lawrence Berkeley National Laboratory, Tech. Rep. 45991, 2000.

[11] ——, "Design, implementation and testing of extended and mixed precision BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 152–205, 2002.

[12] M. Joldeş, J.-M. Muller, and V. Popescu, "Tight and rigourous error bounds for basic building blocks of double-word arithmetic," *ACM Transactions on Mathematical Software*, vol. 44, no. 2, Oct. 2017.

[13] Y. Hida, X. S. Li, and D. H. Bailey, "C++/fortran-90 double-double and quad-double package, release 2.3.17," Mar. 2012.

[14] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2018.

[15] E. Darulova and A. Volkova, "Sound approximation of programs with elementary functions," Tech. Rep., 2018. [Online]. Available: http://arxiv.org/abs/1811.10274