

Worst Cases for the Exponential Function in the IEEE 754r decimal64 Format

Vincent Lefèvre, Damien Stehlé, Paul Zimmermann

▶ **To cite this version:**

Vincent Lefèvre, Damien Stehlé, Paul Zimmermann. Worst Cases for the Exponential Function in the IEEE 754r decimal64 Format. P. Hertling; C.M. Hoffmann; W. Luther; N. Revol. Reliable Implementation of Real Number Algorithms: Theory and Practice, 5045, Springer, pp.114-126, 2008, Lecture Notes in Computer Science, 978-3-540-85521-7. 10.1007/978-3-540-85521-7_7 . inria-00068731v2

HAL Id: inria-00068731

https://inria.hal.science/inria-00068731v2

Submitted on 3 Sep 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution| 4.0 International License

Worst Cases for the Exponential Function in the IEEE 754r decimal64 Format

Vincent Lefèvre[1] and Damien Stehlé[2⋆] and Paul Zimmermann[3]

[1] INRIA/ÉNS Lyon/Université de Lyon/LIP/Projet Arénaire,
46 allée d'Italie, F-69364 Lyon Cedex 07, France
Vincent.Lefevre@inria.fr

[2] CNRS/ÉNS Lyon/Université de Lyon/LIP/INRIA Arénaire,
46 allée d'Italie, F-69364 Lyon Cedex 07, France
damien.stehle@gmail.com

[3] Centre de Recherche INRIA Nancy Grand Est, Projet CACAO - Bâtiment A,
615 rue du Jardin Botanique, F-54600 Villers-lès-Nancy Cedex, France
Paul.Zimmermann@loria.fr

Abstract. We searched for the worst cases for correct rounding of the exponential function in the IEEE 754r decimal64 format, and computed all the bad cases whose distance from a breakpoint (for all rounding modes) is less than 10^{-15} ulp, and we give the worst ones. In particular, the worst case for $|x| \geq 3 \times 10^{-11}$ is $\exp(9.407822313572878 \times 10^{-2}) = 1.098645682066338\,5\,0000000000000000\,278\ldots$. This work can be extended to other elementary functions in the decimal64 format and allows the design of reasonably fast routines that will evaluate these functions with correct rounding, at least in some domains.

1 Introduction

Most computers nowadays support the IEEE 754-1985 standard for binary floating-point arithmetic [1], which requires that all four arithmetic operations ($+$, $-$, $\times$, $\div$) and the square root are *correctly rounded*. However radix 10 is more suited to some applications, such as financial and commercial ones, and there have been propositions to normalize it as well and also design hardware implementations. The IEEE 854-1987 standard for radix-independent floating-point arithmetic [2] has been a first step in this direction, but this standard just gives some constraints on the value sets and is not even specific to radix 10. The article [3] describes a first specification of a decimal floating-point arithmetic; it has been improved and the specification included in the current working draft of the revision of the IEEE 754 standard (754r) is described in [4].

One also seeks to extend the IEEE 754 standard to elementary functions, such as the exponential, logarithm and trigonometric functions, by requiring correct rounding on these functions too. Unfortunately fulfilling this requirement

⋆ Hosted and partially funded by the MAGMA group (University of Sydney) during the completion of this work.

is much more complicated than with the basic operations. Indeed, while efficient algorithms to guarantee the correct rounding are known for the basic operations, the only known way to evaluate $f(x)$, where f is an elementary function and x is a machine number[1], is to compute an approximation to $f(x)$ without any useful knowledge except an error bound; and the exact result $f(x)$ may be very close to a machine number or to the middle of two consecutive machine numbers (which are the discontinuity points of the rounding functions), in which case correct rounding can be guaranteed only if the error on the approximation is small enough. This problem is known as the *Table Maker's Dilemma* (TMD). Some cases can be decided easily, but the only known way to obtain a bound on the acceptable error for any input value is to perform an exhaustive search (with a 64-bit format, as considered below, there are at most 2^{64} possible input values). The arguments x for which the values $f(x)$ are the hardest to round are called *worst cases*.

Systematic work on the TMD in radix 2 was first done by Lefèvre and Muller [5], who published worst cases for many elementary functions in double precision, over the full IEEE 754 range for some functions. And correct rounding requirements for some functions in some domains have been added to the 754r working draft. Improved algorithms to deal with higher precisions are given in [6], and in the present paper, the practical feasibility of the method for decimal formats is demonstrated. Indeed the worst cases depend on the representation (radix and precision) and the mathematical function.

Section 2 describes the decimal formats, how worst cases are expressed and briefly recalls the algorithms (in the decimal context) to search for these worst cases. Section 3 gives all worst cases of the exponential function in the 64-bit decimal format. These results allow us to give Theorem 1.

Theorem 1. *In the IEEE 754r decimal64 format, among all the finite values* $|x| \geq 3 \times 10^{-11}$ *such that* $\exp(x)$ *does not yield an exception, the input* x *such that* $\exp(x)$ *is nearest from a breakpoint, both for rounding-to-nearest and directed rounding modes, is* $9.407822313572878 \times 10^{-2}$*, and for this input, the exact value of* $\exp(x)$ *is:*

$$\underbrace{1.098645682066338}_{16\ digits}\underbrace{50000000000000000}_{17\ digits}278\ldots .$$

Among all finite values x *such that* $\exp(x)$ *does not yield an exception and* $\exp(x) \notin [1 - 10^{-16}/2, 1 + 10^{-15}/2]$*, the input* x *such that* $\exp(x)$ *is nearest from a breakpoint is* $9.999999999999995 \times 10^{-16}$*, and for this input, the exact value of* $\exp(x)$ *is:*

$$\underbrace{1.000000000000000}_{16\ digits}\underbrace{99999\ldots99999}_{30\ digits}666\ldots .$$

[1] A number that is exactly representable in the floating-point system.

2 The Table Maker's Dilemma in Decimal

In this section, the decimal formats are described in Section 2.1, then the general form of worst cases is given, along with a few illustrating examples (Section 2.2). Finally, the algorithms to search for these worst cases are briefly recalled and applied to radix 10 (Section 2.3).

2.1 The Decimal Formats

As specified by the IEEE 854 standard [2], a non-special[2] decimal floating-point number x in precision n has the form:

$$x = (-1)^s \, 10^E \, d_0.d_1 d_2 \dots d_{n-1}$$

where $s \in \{0, 1\}$, the exponent E is an integer between two given integers $E_{\min}$ and $E_{\max}$, and the mantissa $d_0.d_1 d_2 \dots d_{n-1}$ is a fixed-point number written in radix 10; i.e., for i between 0 and $n-1$, one has: $0 \leq d_i \leq 9$.

As d_0 may be equal to 0, some numbers have several representations and the standard does not distinguish them. Without changing the value set, one can require that if $E \neq E_{\min}$, then $d_0 \neq 0$, and this will be done in the following for the sake of simplicity. A number such that $d_0 \neq 0$ is called a *normal number*, and a number such that $d_0 = 0$ (in which case $E = E_{\min}$) is called a *subnormal number*. In this way, the representation of a floating-point number is uniquely defined.

Below $\mathrm{ulp}(x)$ denotes the weight of the digit d_{n-1} in this unique representation; i.e., $\mathrm{ulp}(x) = 10^{E-n+1}$.

The document [4], based on the IEEE 854 standard, defines three decimal formats, whose parameters are given in Table 1: decimal32, decimal64 and decimal128, with an encoding on 32, 64 and 128 bits respectively. This specification has been included in the 754r working draft.

Table 1. The parameters of the three 754r decimal formats.

Format	decimal32	decimal64	decimal128
Precision n (digits)	7	16	34
$E_{\min}$	-95	-383	-6143
$E_{\max}$	96	384	6144

[2] The special floating-point numbers are not-a-number (NaN), the positive and negative infinities, and the positive and negative zeros.

2.2 The Bad and Worst Cases

Given a floating-point format, let us call a *breakpoint* a value where the rounding changes in one of the rounding modes, i.e., a discontinuity point of the rounding functions. A breakpoint is either a machine number (for the directed rounding modes) or the middle of two consecutive machine numbers (for the rounding-to-nearest mode).

For a given function f and a "small" positive number ε, a machine number x is a *bad case* when the distance between the exact value of $f(x)$ and the nearest breakpoint(s) is less than $\varepsilon \cdot \mathrm{ulp}(f(x))$. For instance, if f is the exponential function in the decimal64 format ($n = 16$ digits), then the machine numbers 0.5091077534282133 and 0.7906867968553504 are bad cases for $\varepsilon = 10^{-16}$, since for these values, $\exp(x)$ is close enough to the middle of two consecutive machine numbers:

$$\exp(0.5091077534282133) = \underbrace{1.663806007261509}_{16\text{ digits}}\underbrace{5000000000000000}_{16\text{ digits}}49\ldots$$

and

$$\exp(0.7906867968553504) = \underbrace{2.204910231771509}_{16\text{ digits}}\underbrace{4999999999999999}_{16\text{ digits}}16\ldots,$$

i.e., rounding $\exp(x)$ in the rounding-to-nearest mode requires to evaluate $\exp(x)$ in a precision significantly higher than the target precision. Similarly, with the following bad cases, $\exp(x)$ is very close to a machine number, so that rounding it in directed rounding modes also requires to evaluate it in a precision significantly higher than the target precision:

$$\exp(0.001548443067391468) = \underbrace{1.001549642524374}_{16\text{ digits}}\underbrace{9999999999999999}_{16\text{ digits}}26\ldots$$

and

$$\exp(0.2953379504777270) = \underbrace{1.343580345589067}_{16\text{ digits}}\underbrace{0000000000000000}_{16\text{ digits}}86\ldots.$$

2.3 Searching for Bad and Worst Cases

Searching for bad cases in decimal is very similar to the search in binary. First the domain of the tested function is selected: arguments that give an underflow or an overflow are not tested, and some other arguments do not need to be tested either when a simple reasoning can be carried out (see Section 3.1 as an example). And like in binary [7–9], probabilistic hypotheses allow us to guess that the smallest distance amongst all the arguments to be tested is of the order of 10^{-n} ulp (divided by the number of exponents E), so that we can choose

$\varepsilon \sim 10^{-n}$ to get only a few bad cases[3]; i.e., we search for bad cases with at least n (or $n-1$) identical digits 0 or 9 (possibly except the first one, which may be respectively 5 or 4) after the n-digit mantissa.

In the decimal32 format, the number of arguments to be tested is small enough for a naive algorithm to be sufficient: for each argument x, one computes $f(x)$ in a higher precision to eliminate the values x for which the distance between $f(x)$ and the nearest breakpoint(s) is larger than $\varepsilon \cdot \mathrm{ulp}(f(x))$. Since finding bad cases is rather easy for the decimal32 format, this paper will not focus on this format; the reader may find some results for the exponential function at `http://www.loria.fr/~zimmerma/wc/decimal32.html`.

In the decimal64 format, the number of remaining arguments after reducing the domain is still very large (say, around 10^{17} to 10^{19}, depending on the function), and a naive algorithm would require several centuries of computations. Like in the binary double precision, one needs specific algorithms, and since the decimal arithmetic has the same important properties as the binary one (the machine numbers are in arithmetic progression except at exponent changes, the breakpoints have a similar form...), the same methods can be applied.

In radix 2, bad cases for precision n and any rounding mode are the same as bad cases for precision $n+1$ and directed rounding modes[4], so that the problem was restricted to directed rounding modes in [6]. This property is no longer true in radix 10, but the breakpoints are still in an arithmetic progression (except when the exponent changes, just like in radix 2), which is the only important property used by our algorithms. Indeed in each domain where the exponent of $f(x)$ does not change, one needs to search for the solutions of:

$$|f(x) \bmod (u/2)| < \varepsilon u\,,$$

where $u = \mathrm{ulp}(f(x))$, which is a constant in the considered domain.

To solve this problem, one splits the domain into subintervals, and in each subinterval, one approximates the function f by a polynomial P of small degree and scales/translates the input and output values to reduce the problem to the following (as in the binary case [6]):

Real Small Value Problem (Real SValP). Given positive integers M and T, and a polynomial P with real coefficients, find all integers $|t| < T$ such that:

$$|P(t) \bmod 1| < \frac{1}{M}\,. \tag{1}$$

The coefficients of the polynomial are computed using the MPFR library [10] in order to obtain guaranteed error bounds.

[3] This may not be true in some domains, for instance when the function can be approximated accurately by a simple degree-2 polynomial, such as $\exp(x) \simeq 1 + x + x^2/2$ for x sufficiently close to 0; in this case, one can get bad cases which are much closer to breakpoints and more numerous than what can be estimated with the probabilistic hypotheses. This is not a problem in practice: A simple reasoning is usually sufficient instead of an exhaustive search in this domain.

[4] Said otherwise, in radix 2, the breakpoints for precision n and all rounding modes are the machine numbers in precision $n+1$.

Then several fast algorithms can be used to solve the Real SValP. Lefèvre's algorithm needs degree-1 polynomial approximations; as these approximations are valid on very small intervals, one also needs a way to determine these approximations very quickly [11]. The Stehlé-Lefèvre-Zimmermann (SLZ) algorithm allows to have polynomials of higher degrees and has a smaller asymptotic complexity [6], but with a high constant factor. It is based on Coppersmith's technique to find the small roots of multivariate polynomials modulo an integer: informally, in our situation, we look for small roots of $P(x) + y$ modulo 1. Coppersmith's technique was first introduced in a cryptographic context [12], and heavily relies on the LLL algorithm for reducing Euclidean lattice bases [13]. Heuristically, LLL takes as input a basis derived from the multivariate polynomial and its powers: this basis contains the information we are interested in (the roots of the initial polynomial), but in an inconvenient way (there is no known way to efficiently compute roots modulo an arbitrary integer). LLL outputs a basis made of shorter vectors. In particular, if all the various parameters are chosen adequately, the first output vectors will be short enough to ensure that the corresponding polynomials contain among their roots (over the integers, without the modulus) the roots of the initial polynomial.

In order to make the implementation of the SLZ algorithm as efficient as possible, it is crucial to use an efficient LLL code. For instance, one should avoid using the text-book LLL algorithm making use of a rational arithmetic. In the implementation of the SLZ algorithm, it is better to use variants of the LLL algorithm relying on floating-point arithmetic rather than rational arithmetic within the Gram-Schmidt computations (central in LLL).

In his PhD thesis [14], Stehlé describes three floating-point variants of LLL, respectively called "fast", "heuristic" and "proved". The corresponding codes are available at `http://perso.ens-lyon.fr/damien.stehle`. The proved variant implements the algorithm described in [15], whereas the other two can fail[5] but are usually more efficient.

Remark 1. The above methods may no longer work well for the smallest subnormals, due to the loss of precision for these numbers. For instance, a low-degree polynomial approximation may be valid on an interval that contains only very few machine numbers. Nevertheless these few values may be tested separately with a naive algorithm, if need be.

3 The Exponential Function

We now show the feasibility of our method on the exponential function, denoted exp, in the decimal64 format. This is just an example: a similar work can be carried out for other functions. After a simple analysis of the function (Section 3.1), we search for bad cases (Section 3.2).

[5] In practice, when they fail, they loop forever; they may also return a badly-reduced basis. But in both situations, no bad case will be missed.

3.1 Correctly Rounding the Exponential Function

Let us first recall the parameters of the decimal64 format, with a few more details. A non-special floating-point number x has the form:

$$x = (-1)^s \, 10^E \, d_0.d_1 d_2 \dots d_{15}$$

where $s \in \{0, 1\}$ and $-383 \leq E \leq 384$. So, the largest finite machine number is $10^{385} - 10^{369}$, the smallest positive normal machine number is 10^{-383} and the smallest positive machine number is 10^{-398}.

Now let us briefly analyze the exponential function, assuming that the argument is a finite number, to eliminate the special cases. The exponential function is mathematically defined on the whole domain of real numbers, so that the value will never be a NaN. It is increasing, with $\exp(x) \to +\infty$ when $x \to +\infty$, and $\exp(x) \to 0$ when $x \to -\infty$. And the mathematical properties of the exponential function are such that there will be an overflow when x is larger than some value and an underflow when x is smaller than some value. Moreover, $\exp(0) = 1$, meaning that for values of x close to 0, the rounding of $\exp(x)$ is determined only by the rounding mode and the sign of x.

So, there are four couples of consecutive machine numbers (a^-, a^+), (b^-, b^+), (c^-, c^+) and (d^-, d^+) that determine the following five intervals:

$$\underbrace{-\infty \ \dots \ a^-}_{+0} \ \underbrace{a^+ \ \dots \ b^-}_{\text{search}} \ \underbrace{b^+ \ \dots \ c^-}_{1} \ \underbrace{c^+ \ \dots \ d^-}_{\text{search}} \ \underbrace{d^+ \ \dots \ +\infty}_{+\infty}$$

where in intervals 1, 3 and 5, the rounded values in the rounding-to-nearest mode are respectively $+0$, 1 and $+\infty$ (the rounded values in the directed rounding modes can also be determined, keeping the same interval bounds for the sake of simplicity), and in intervals 2 and 4, a search for bad cases is needed. These interval bounds are determined below.

An argument x generates an overflow when the *rounded* result obtained assuming an unbounded exponent range exceeds the largest finite machine number $10^{385} - 10^{369}$. One has:

$$\log(10^{385} - 10^{369}/2) = \underbrace{886.4952608027075}_{\text{16 digits}}\, 882469 \dots ,$$

so that one gets an overflow if and only if $x \geq d^+$, with $d^+ = 886.4952608027076$ (x being a machine number).

Concerning a^-, one has:

$$\log(10^{-398}/2) = -\underbrace{917.1220141921901}_{\text{16 digits}}\, 2 \dots ,$$

so that in any rounding mode, $\exp(x)$ is rounded to the same value for any $x \leq a^-$, with $a^- = -917.1220141921902$: It is rounded to 10^{-398} in the rounding to $+\infty$ mode, and $+0$ in the other rounding modes.

Concerning b^+ and c^-, one has:

$$\log(1 - 10^{-16}/2) = -\underbrace{5.000000000000000}_{\text{16 digits}}125\ldots \times 10^{-17}$$

and

$$\log(1 + 10^{-15}/2) = \underbrace{4.999999999999998}_{\text{16 digits}}750\ldots \times 10^{-16}\,,$$

so that one chooses $b^+ = -5 \times 10^{-17}$ and $c^- = 4.999999999999998 \times 10^{-16}$.

Finally, in the other domains, that is for x in

$$[a^+, b^-] = [-917.1220141921901, -5.000000000000001 \times 10^{-17}]$$

and in

$$[c^+, d^-] = [4.999999999999999 \times 10^{-16}, 886.4952608027075]\,,$$

a search for bad cases needs to be done to be able to round $\exp(x)$ correctly in any rounding mode.

Remark 2. When x is close enough to 0, one could use the approximation $\exp(x) \simeq 1 + x + x^2/2$ to find bad cases with much less computing time in this domain. But globally, one would gain very little since this is an easy domain (as the error on a polynomial approximation is very small compared to higher values of x, and the algorithms work much better).

3.2 Searching for Bad and Worst Cases of the Exponential Function

To search for bad cases, one first splits the tested domain into intervals in which both the argument x and the result $\exp(x)$ have a constant (possibly different) exponent. This has been done with a small Maple program.

As said in [11] and [5], one could test the inverse function, i.e., the logarithm, instead of the exponential when x is small enough (say, $|x| < 1$). The reason is that there are fewer machine numbers to test in this domain for the inverse function. However this domain requires very little computation time compared to those with high values of x.

The search for bad cases was performed with BaCSeL[6], running on a few machines. The chosen parameters were: a working precision of 200 bits, $m = 14.6$ (the quality of the bad cases, i.e., $-\log_{10}(2\varepsilon)$, to get all bad cases for $\varepsilon = 10^{-15}$), $t = 5.5$ (a parameter that fixes the size of the sub-intervals), $d = 3$ (the degree of the polynomials) and $\alpha = 2$ (a parameter for Coppersmith's technique). For values of x close enough to 0, the fast LLL variant fails, so that the proved variant is used in this domain.

Tables 2 and 3 present all the bad cases for $x \geq 10^{-9}$ and for $x \leq -10^{-10}$ respectively, whose distance from a breakpoint is less than 5×10^{-17} ulp.

[6] Available on `http://perso.ens-lyon.fr/damien.stehle`.

Table 2. All worst cases of the decimal64 exponential function for $x \geq 10^{-9}$, whose distance from a breakpoint is less than 5×10^{-17} ulp. The notation d^k means that the digit d is repeated k times.

x	$\exp(x)$
$6.581539478341669 \times 10^{-9}$	$1.000000006581539\,5\,0^{15}\,177\ldots$
$2.662858264545929 \times 10^{-8}$	$1.000000026628583\,0\,0^{15}\,318\ldots$
$3.639588333766983 \times 10^{-8}$	$1.000000036395884\,0\,0^{15}\,240\ldots$
$6.036998017773271 \times 10^{-8}$	$1.000000060369982\,0\,0^{15}\,379\ldots$
$6.638670361402304 \times 10^{-7}$	$1.000000663867256\,4\,9^{15}\,569\ldots$
$9.366572213364879 \times 10^{-7}$	$1.000000936657659\,9\,9^{15}\,883\ldots$
$7.970613003079781 \times 10^{-6}$	$1.000007970644768\,5\,0^{15}\,362\ldots$
$3.089765552852523 \times 10^{-5}$	$1.000030898132866\,0\,0^{15}\,241\ldots$
$1.302531956641873 \times 10^{-4}$	$1.000130261678980\,0\,0^{16}\,798\ldots$
$2.241856702421245 \times 10^{-4}$	$1.000224210801727\,5\,0^{15}\,118\ldots$
$7.230293679121590 \times 10^{-4}$	$1.000723290816653\,4\,9^{16}\,127\ldots$
$5.259640428979129 \times 10^{-3}$	$1.005273496619909\,4\,9^{15}\,739\ldots$
$9.407822313572878 \times 10^{-2}$	$1.098645682066338\,5\,0^{16}\,278\ldots$
$1.267914924960933 \times 10^{-1}$	$1.135180299492843\,0\,0^{16}\,706\ldots$
$5.091077534282133 \times 10^{-1}$	$1.663806007261509\,5\,0^{15}\,492\ldots$
3.359104074009002	$28.76340944572687\,5\,0^{16}\,904\ldots$
19.10511686234796	$1.9826535384149819\,9^{15}\,735\ldots \times 10^{8}$
294.9551257293143	$1.251363586659789\,5\,0^{15}\,108\ldots \times 10^{128}$
587.9131381356093	$2.125356221825522\,4\,9^{15}\,594\ldots \times 10^{255}$

Table 3. All worst cases of the decimal64 exponential function for $x \leq -10^{-10}$, whose distance from a breakpoint is less than 5×10^{-17} ulp. The notation d^k means that the digit d is repeated k times.

x	$\exp(x)$
$-2.090862502185853 \times 10^{-9}$	$0.9999999979091375\,0\,0^{15}\,371\ldots$
$-3.803619857233762 \times 10^{-9}$	$0.9999999961963801\,4\,9^{15}\,841\ldots$
$-7.170496225708008 \times 10^{-9}$	$0.9999999928295038\,0\,0^{15}\,252\ldots$
$-9.362256793825926 \times 10^{-9}$	$0.9999999906377432\,4\,9^{15}\,580\ldots$
$-4.024416580979643 \times 10^{-8}$	$0.9999995975583500\,0^{15}\,308\ldots$
$-6.306378165019860 \times 10^{-7}$	$0.9999936936238235\,0^{15}\,301\ldots$
$-7.720146779532548 \times 10^{-7}$	$0.9999922798562004\,9^{15}\,612\ldots$
$-9.753167969712726 \times 10^{-7}$	$0.9999990246836786\,4\,9^{16}\,120\ldots$
$-5.911964024384330 \times 10^{-5}$	$0.9994088210728765\,0^{15}\,384\ldots$
$-8.232272117182855 \times 10^{-5}$	$0.9999176806672504\,0\,0^{15}\,312\ldots$
$-8.232461306131942 \times 10^{-5}$	$0.9999176787755166\,4\,9^{15}\,555\ldots$
$-8.496743395712491 \times 10^{-2}$	$0.9185421971989605\,4\,9^{15}\,843\ldots$
$-9.250971335383380 \times 10^{-2}$	$0.9116403558361098\,9\,9^{15}\,563\ldots$
$-9.337621398029658 \times 10^{-2}$	$0.9108507610382665\,0\,0^{15}\,400\ldots$
$-9.341228128742237 \times 10^{-2}$	$0.9108179096965556\,4\,9^{16}\,587\ldots$
$-9.998733949173545 \times 10^{-2}$	$0.9048488738100865\,0\,0^{15}\,330\ldots$
-1.452866822458144	$0.2338987797314129\,0\,0^{15}\,413\ldots$
-5.085363904672046	$6.186635335115975\,4\,9^{15}\,774\ldots \times 10^{-3}$
-5.815903811599861	$2.9797859449458045\,0^{15}\,173\ldots \times 10^{-3}$
-11.93382527979436	$6.5645586526114569\,9^{15}\,658\ldots \times 10^{-6}$
-46.84177248885496	$4.5381274182205359\,9^{15}\,769\ldots \times 10^{-21}$
-84.88822783213444	$1.3599128388934695\,0^{15}\,266\ldots \times 10^{-37}$
-495.9839910528425	$3.9526610430311695\,0^{15}\,371\ldots \times 10^{-216}$
-524.2585830842744	$2.0767789638678450\,0^{15}\,287\ldots \times 10^{-228}$

For $-10^{-9} < x < 10^{-8}$ (and in particular for the smaller domain $-10^{-10} < x < 10^{-9}$), many bad cases have some patterns in their mantissa. For instance, one has the following bad cases with $\varepsilon = 3 \times 10^{-15}$ (look at the 8th, 9th and 10th digits):

$$3.897940992403028 \times 10^{-9}\,,$$
$$4.230932991049603 \times 10^{-9}\,,$$
$$4.291382990792016 \times 10^{-9}\,,$$
$$4.581289989505891 \times 10^{-9}\,.$$

This comes from the fact that $\exp(x)$ can be approximated by $1+x+x^2/2+x^3/6$ in these domains, and even by $1+x+x^2/2$ for smaller values of x. Tables 4 and 5 give some other bad cases for $c^+ \le x < 10^{-9}$ and $-10^{-10} < x \le b^-$ respectively.

Table 4. Some bad cases of the exponential function in the decimal64 format, for $c^+ = 4.999999999999999 \times 10^{-16} \le x < 10^{-9}$. At most two bad cases (the worst ones) are given per exponent.

x	$\exp(x)$
$6.000119998199928 \times 10^{-10}$	$1.000000000600011\,9\,9^{16}\,567\ldots$
$5.999879998200072 \times 10^{-10}$	$1.000000000599988\,0\,0^{16}\,431\ldots$
$1.039999999994592 \times 10^{-11}$	$1.000000000010399\,9\,9^{17}\,625\ldots$
$1.019999999994798 \times 10^{-11}$	$1.000000000010199\,9\,9^{17}\,646\ldots$
$1.199999999999280 \times 10^{-12}$	$1.000000000001199\,9\,9^{20}\,423\ldots$
$1.099999999999395 \times 10^{-12}$	$1.000000000001099\,9\,9^{20}\,556\ldots$
$1.399999999999902 \times 10^{-13}$	$1.000000000000139\,9\,9^{23}\,085\ldots$
$1.199999999999928 \times 10^{-13}$	$1.000000000000119\,9\,9^{23}\,423\ldots$
$2.999999999999955 \times 10^{-14}$	$1.000000000000029\,9\,9^{25}\,099\ldots$
$1.999999999999980 \times 10^{-14}$	$1.000000000000019\,9\,9^{25}\,733\ldots$
$3.999999999999992 \times 10^{-15}$	$1.000000000000003\,9\,9^{27}\,786\ldots$
$1.999999999999998 \times 10^{-15}$	$1.000000000000001\,9\,9^{28}\,733\ldots$
$9.999999999999995 \times 10^{-16}$	$1.000000000000000\,9\,9^{29}\,666\ldots$

The complete list of all worst cases which are at a distance less than 10^{-15} ulp from a breakpoint is available at `http://www.loria.fr/~zimmerma/wc/decimal64.html`.

4 Conclusion

Like in binary arithmetic, correct rounding can be guaranteed in decimal arithmetic at a reasonable cost if the upper bound on the necessary precision for the intermediate computations is determined. This requires exhaustive tests on

Table 5. Some bad cases of the exponential function in the decimal64 format, for $-10^{-10} < x \leq b^- = -5.000000000000001 \times 10^{-17}$. At most two bad cases (the worst ones) are given per exponent.

x	$\exp(x)$
$-1.020000000005202 \times 10^{-11}$	$0.9999999999898000\,0\,0^{16}\,353\ldots$
$-1.000000000005000 \times 10^{-11}$	$0.9999999999900000\,0\,0^{16}\,333\ldots$
$-1.100000000000605 \times 10^{-12}$	$0.9999999999890000\,0\,0^{19}\,443\ldots$
$-1.000000000000500 \times 10^{-12}$	$0.9999999999900000\,0\,0^{19}\,333\ldots$
$-1.200000000000072 \times 10^{-13}$	$0.9999999999998800\,0\,0^{22}\,575\ldots$
$-1.000000000000050 \times 10^{-13}$	$0.9999999999999000\,0\,0^{22}\,333\ldots$
$-2.000000000000020 \times 10^{-14}$	$0.9999999999999800\,0\,0^{24}\,266\ldots$
$-1.000000000000005 \times 10^{-14}$	$0.9999999999999900\,0\,0^{25}\,333\ldots$
$-4.000000000000008 \times 10^{-15}$	$0.9999999999999960\,0\,0^{26}\,213\ldots$
$-2.000000000000002 \times 10^{-15}$	$0.9999999999999980\,0\,0^{27}\,266\ldots$

the whole input domain. While some subdomains can easily be handled, a large number of input values need to be tested.

For the 754r decimal32 format, the tests can be carried out with naive algorithms. However, for the 754r decimal64 format, specific algorithms needed to be designed and implemented. The complete results for the exponential function have been given in this paper. The worst case for $|x| \geq 3 \times 10^{-11}$ (i.e., if we disregard very small values) is

$$\exp(9.407822313572878 \times 10^{-2}) = \underbrace{1.098645682066338}_{16\text{ digits}}\underbrace{50000000000000000}_{17\text{ digits}}278\ldots,$$

meaning that a faithful approximation to 34 digits, which corresponds to the decimal128 format, would be enough to guarantee correct rounding for the exponential in the decimal64 format in this domain. For the smaller values of x, the worst case is

$$\exp(9.999999999999995 \times 10^{-16}) = \underbrace{1.000000000000000}_{16\text{ digits}}\underbrace{999\ldots999}_{30\text{ digits}}666\ldots,$$

so that a faithful approximation to $\exp(x) - 1$, also known as $\mathrm{expm1}(x)$, in the decimal128 format would be enough to guarantee correct rounding for the exponential in the decimal64 format in this domain.

Ziv's strategy [16] can be used to evaluate the decimal64 exponential function; it consists in carrying out the computations in a small precision (e.g., 22 digits) first, and increasing the precision only in the very unlikely case where the correct rounding cannot be decided. The results presented in this paper can be used

to implement Ziv's strategy in an efficient way and prove that the algorithm terminates within limited time and memory.

Other elementary functions could be tested as well, with the same algorithms. As a consequence, standards could recommend (or even require) correct rounding for these functions in these formats.

Acknowledgements

The writing of this paper was completed while the second author was visiting the University of Sydney, whose hospitality is gratefully acknowledged. In particular, part of the computations described in the present article was performed on the machines of the MAGMA team.

The computations were also partly performed on machines of the Laboratoire de l'Informatique du Parallélisme (at the École Normale Supérieure de Lyon, France).

The third author acknowledges the support from the Schloss Dagstuhl International Conference and Research Center for Computer Science, in particular the Dagstuhl Seminar 06021 *Reliable Implementation of Real Number Algorithms: Theory and Practice*, which stimulated the writing of this article.

The authors also thank the anonymous reviewers for their helpful comments.

References

1. IEEE: IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Institute of Electrical and Electronics Engineers, New York (1985)
2. IEEE: IEEE Standard for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Standard 854-1987. Institute of Electrical and Electronics Engineers, New York (1987)
3. Cowlishaw, M., Schwarz, E.M., Smith, R.M., Webb, C.F.: A decimal floating-point specification. In Burgess, N., Ciminiera, L., eds.: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA, IEEE Computer Society Press, Los Alamitos, CA (2001) 147–154
4. Cowlishaw, M.: Decimal arithmetic encoding strawman 4d, draft version 0.96. Report, IBM UK Laboratories, Hursley, UK (2003)
5. Lefèvre, V., Muller, J.M.: Worst cases for correct rounding of the elementary functions in double precision. In Burgess, N., Ciminiera, L., eds.: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, IEEE Computer Society Press, Los Alamitos, CA (2001) 111–118
6. Stehlé, D., Lefèvre, V., Zimmermann, P.: Searching worst cases of a one-variable function using lattice reduction. IEEE Transactions on Computers **54**(3) (2005) 340–346
7. Dunham, C.B.: Feasibility of "perfect" function evaluation. ACM Sigum Newsletter **25**(4) (1990) 25–26
8. Gal, S., Bachelis, B.: An accurate elementary mathematical library for the IEEE floating point standard. ACM Transactions on Mathematical Software **17**(1) (1991) 26–45

9. Muller, J.M.: Elementary Functions, Algorithms and Implementation. Birkhauser, Boston (1997)
10. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. Research report RR-5753, INRIA (2005)
11. Lefèvre, V.: Moyens arithmétiques pour un calcul fiable. PhD thesis, École Normale Supérieure de Lyon, Lyon, France (2000)
12. Coppersmith, D.: Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology **10**(4) (1997) 233–260
13. Lenstra, A.K., Lenstra, Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen **261** (1982) 513–534
14. Stehlé, D.: Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques. PhD thesis, Université Henri Poincaré – Nancy 1, Nancy, France (2005)
15. Nguyen, P., Stehlé, D.: Floating-point LLL revisited. In: Proceedings of Eurocrypt 2005. Volume 3494 of Lecture Notes in Computer Science., Springer-Verlag (2005) 215–233
16. Ziv, A.: Fast evaluation of elementary mathematical functions with correctly rounded last bit. ACM Transactions on Mathematical Software **17**(3) (1991) 410–423