Worst Cases for Correct Rounding of the Elementary Functions in Double Precision.

Vincent Lefevre, Jean-Michel Muller

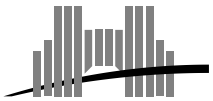
Laboratoire de l'Informatique du Parallélisme

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

Worst Cases for Correct Rounding of the Elementary Functions in Double Precision

Vincent Lefèvre, Jean-Michel Muller November 2000

Research Report N° 2000-35



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

Worst Cases for Correct Rounding of the Elementary Functions in Double Precision

Vincent Lefèvre, Jean-Michel Muller

November 2000

Abstract

We give here the results of a four-year search for the worst cases for correct rounding of the major elementary functions in double precision. These results allow the design of reasonably fast routines that will compute these functions with correct rounding, at least in some interval, for any of the four rounding modes specified by the IEEE-754 standard. They will also allow to easily test libraries that are claimed to provide correctly rounded functions.

Keywords: Elementary functions, Computer arithmetic, Table Maker's Dilemma, Correct rounding, Floating-Point Arithmetic.

Résumé

Nous donnons dans ce rapport les résultats de quatre ans de recherche des pires cas pour l'arrondi correct des principales fonctions élémentaires en double précision. Ces résultats permettent de construire des programmes raisonnablement rapides calculant ces fonctions avec arrondi correct – au moins dans un domaine donné – pour chacun des quatre modes d'arrondi spécifiés par la norme IEEE-754. Ils permettent également de tester des bibliothèques censées fournir l'arrondi correct de ces fonctions.

Mots-clés: Fonctions élémentaires, arithmétique des ordinateurs, Dilemme du fabricant de tables, Arrondi correct, Virgule flottante.

1 Introduction

In general, the result of an arithmetic operation on two floating-point numbers is not exactly representable in the same floating-point format: it must be *rounded*. In a floating-point system that follows the IEEE 754 standard [2, 3, 6], the user can choose an *active rounding mode* from: rounding towards $-\infty$, rounding towards $+\infty$, rounding towards 0 and rounding to the nearest (with a special convention if x is exactly between two machine numbers). The IEEE-754 standard requires that the system should behave as if the result of an arithmetic operation ($+$, $-$, $\div$, $\times$) were first computed *exactly*, with "infinite precision", and then rounded accordingly to the active rounding mode. Operations that satisfy this property are called "exactly rounded" or "correctly rounded". There is a similar requirement for the square root.

Unfortunately, there is no such requirement for the elementary functions[1], probably because it has been believed for many years that exact rounding of the elementary functions would be much too expensive for double precision (for single precision, since checking 2^{32} input numbers requires a few days only, there already exist libraries that provide correct rounding. See for instance [13, 14]).

Requiring correctly rounded results (that is, the "best possible" results) in some standard would not only improve the accuracy of computations: it would help to make numerical software more portable. Moreover, as noticed by Agarwal et al. [1], correct rounding facilitates the preservation of useful mathematical properties such as monotonicity, symmetry and important identities. See [12] for more details.

Before going further, let us start with definitions. We call **Infinite mantissa** of a nonzero real number x the number

$$\frac{x}{2^{\lfloor \log_2 |x| \rfloor}}.$$

In other words, the infinite mantissa of x is the real number x' such that $1 \leq x' < 2$ and $x = x' \times 2^k$, where k is an integer. If x is a floating-point number, then its infinite mantissa coincides with its floating-point mantissa. If a and b belong to the same "binade" (they have the same sign and satisfy $2^p \leq |a|, |b| \leq 2^{p+1}$, where p is an integer), we call their **Mantissa distance** the distance between their infinite mantissas (that is, $|a-b|/2^p$).

Let f be an elementary function and x a floating-point number. Unless x is a very special case – e.g., $\log(1)$ or $\sin(0)$ –, $y = f(x)$ cannot be exactly represented. The only thing we can do is to compute an *approximation* y^* to y. If we wish to provide correctly rounded functions, the problem is to know what the accuracy of this approximation should be to make sure that rounding y^* is equivalent to rounding y.

In other words, from y^* and the known bounds on the approximation, the only information we have is that y belongs to some interval Y. Let us call $\diamond$ the rounding function. Let us call a "breakpoint" a value z where the rounding changes (that is, if t_1 and t_2 are real numbers satisfying $t_1 < z < t_2$ then $\diamond(t_1) < \diamond(t_2)$). For "directed" rounding modes (i.e., rounding upwards, downwards, or towards 0), the breakpoints are the floating-point numbers. For rounding to the nearest mode, the breakpoints are the exact middle of two consecutive floating-point numbers.

[1] By *elementary functions* we mean the radix 2, e and 10 logarithms and exponentials, and the trigonometric and hyperbolic functions.

If Y contains a breakpoint, then we cannot provide $\diamond(y)$: the computation must be carried again with a larger accuracy. There are two ways of solving that problem:

- iteratively increase the accuracy of the approximation, until interval Y no longer contains a breakpoint[2]. The problem is that it is difficult to predict how many iterations will be necessary;
- compute, in advance and once for all, the smallest relative nonzero distance[3] between the image of a floating-point number and breakpoint. This will allow to deduce the relative accuracy with which $f(x)$ must be approximated to make sure that rounding the approximation is equivalent to rounding the exact result.

The first solution was suggested by Ziv [15]. It has been implemented in a library, called ml4j, available through the internet[4]. As a matter of fact, the last iteration uses 768 bits of precision. There is no formal proof that this suffices (the results presented in this paper actually give the proof for the functions and domains considered here!), but probabilistic arguments[4, 5, 12] may be used to show that requiring a larger precision is extremely unlikely.

We decided to implement the second solution, since the only way to implement the first one is to overestimate the accuracy that is needed in the worst cases. The basic principle of our algorithm for searching the worst cases was outlined in [10, 11]. We now present properties that have allowed to fasten the search, as well as the results obtained after having run our algorithms for 4 years on several workstations, and interesting properties that can be obtained from our results.

The results we have obtained are "worst cases for the Table Maker's Dilemma", that is, floating point numbers whose image is closest (for the "mantissa distance") to a floating-point number (i.e., a breakpoint for a directed rounding mode) or to the exact middle of two consecutive floating-point numbers (i.e., a breakpoint for rounding to the nearest).

For instance, the worst case for the natural logarithm in the full double precision range is reached for

$$\begin{aligned} x &= 1.01100010101010001000011000010011011000101 0 \\ &\qquad 0110110110 \times 2^{678} \\ &\approx 1.737429606443346566788426 \times 10^{204} \text{ in decimal} \end{aligned}$$

whose logarithm is

$$\log x = \overbrace{111010110.01000111100111101\cdots110001}^{53 \text{ bits}} \underbrace{000000000000000000\cdots000000000000000}_{65 \text{ zeroes}} 1110\ldots$$

[2] This is not possible if $f(x)$ is equal to a breakpoint. However one can show that $x = 0$ is the only input value for which $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arctan(x)$ and e^x have a finite radix-2 representation – and the breakpoints do have finite representations –, $x = 1$ is the only input value for which $\ln(x)$ has a finite representation. Concerning 2^x and 10^x, they have a finite representation if and only if x is an integer. Also, $\log_2(x)$ (resp. $\log_{10}(x)$) have a finite representation if and only if x is an integer power of 2 (resp. 10). All these cases are straightforwardly handled separately, so we do not discuss them in the sequel of the paper.

[3] In fact, the mantissa distance.

[4] http://www.alphaWorks.ibm.com/tech/mathlibrary4java.

this worst case is a "difficult case" in a directed rounding mode, since it is very near a double precision floating-point number. The two worst cases for radix-2 exponentials in the full IEEE-754 double precision range are reached for

$$\begin{aligned} x_1 &= 1.10111111101110111101111001000100111011011 \\ &\quad 11111000101 \times 2^{-25} \\ &\approx 5.21230814407693756691262 5 \times 10^{-8} \text{ in decimal} \end{aligned}$$

and

$$\begin{aligned} x_2 &= 1.11100100010110010110010100100110101111 1 \\ &\quad 100101001101 \times 2^{-10} \\ &\approx 0.00184764556769286274369446 0 \text{ in decimal} \end{aligned}$$

whose radix-2 exponentials are

$$2^{x_1} = \overbrace{1.00000000000000000000000100110110\cdots101}^{53 \text{ bits}} \underbrace{0000000000000000000000000\cdots0000000000000}_{60 \text{ zeroes}}1011\ldots$$

$$2^{x_2} = \overbrace{1.00000000010100111111110000 10111\cdots0011}^{53 \text{ bits}} 0\underbrace{1111111111111111\cdots1111111111111111}_{59 \text{ ones}}0100\ldots$$

The former is a difficult case for directed rounding modes, whereas the latter is a difficult case for rounding to the nearest mode (since it is very close to the exact middle of two consecutive floating-point numbers).

2 Our algorithms for finding the worst cases

2.1 Basic principles

The basic principles have been given in [11], so we only quickly describe them and focus on new aspects. Assume we wish to look for the worst cases for function f in double precision. Let us call **test number** a number that is representable in floating-point with **54** bits of mantissa (a test number is either a double precision number or a number that lies exactly halfway between two consecutive double precision numbers). The test numbers are the values that are breakpoints for one of the rounding modes. Our problem of finding worst cases now reduces to the problem of finding double precision numbers x such that $f(x)$ is closest (for the mantissa distance) to a test number. We proceed in two steps: we first use a fast "filtering" method that eliminates all points whose distance to the closest breakpoint is above a given threshold. The value of the threshold is chosen so that this filtering method does not require highly accurate computations, and so that the number of values that remain to be checked after the filtering is so small that an accurate computation of the value of the function at each remaining value is possible. Details on the choice of parameters are given in [9].

In [11], we suggested to perform the filtering as follows:

- first, the domain where we look for worst cases is split into "large subdomains" where all input values have the same exponent;

- each large subdomain is split into "small subdomains" that are small enough so that in each of these subdomains, within the accuracy of the filtering, the function can be approximated by a linear function. Hence in each small subdomain, our problem now is to find a point on a grid that is closest to a straight line. We solve a slightly different problem: given a "threshold" ϵ we just try to know if there can be a point of the grid that is at distance less than ϵ from the straight line. The value of ϵ is chosen so that for one given small subdomain this event is very unlikely.

- using a variant to the Euclidean algorithm suggested by V. Lefèvre [8], we solve that problem. If we find that there can be a point of the grid at distance less than ϵ from the straight line, we check all points of the small subdomain.

2.2 Optimization: f and f^{-1} simultaneously

Now, let us present a useful optimization of that method. Instead of finding double precision numbers x such that $f(x)$ is closest (for the mantissa distance) to a test number, we solve a slightly different problem: we look for **test numbers** x such that $f(x)$ is closest to a test number. This allows to compute worst cases for f and for its inverse f^{-1} in one pass only (since the image $f(a)$ of a breakpoint a is very near a breakpoint b if and only if $f^{-1}(b)$ is very near a). One could object that by checking the images of test numbers[5] instead of checking the double precision numbers only, we double the number of points that are examined. So getting in one pass the results for two functions (f and f^{-1}) seems to be a no-win no-loss operation. This is not quite true, since there are sometimes *much less* values to check for one of the two functions than for the other one.

Consider as an example the radix-2 exponential and logarithm, with input domain $I = [-1, 1]$ for 2^x (which corresponds to input domain $J = [1/2, 2]$ for $\log_2(y)$.). The two following strategies would lead to the same final result: they would give the worst cases for 2^x in I and for $\log_2(y)$ in J.

1. check 2^x for every test number x in I;

2. check $\log_2(y)$ for every test number y in J.

If we use the first strategy, we need to check all test numbers of exponent between[6] -53 and -1. Hence we have to check 106×2^{53} numbers. If we use the second strategy, we need to check all positive test numbers of exponent equal to -1 or 0, that is, 2×2^{53} numbers. The second strategy is approximately 53 times faster than the first one.

If we separately check all double precision numbers in I and all double precision numbers in J, we check $106 \times 2^{52} + 2 \times 2^{52}$. The second strategy is approximately 27 times faster than this last method.

Hence, in the considered domain, it is much better to check $\log_2(y)$ for every test number y in $[1/2, 2]$. In other domains, the converse holds: since the exponential of a large number is an

[5] At the end of the test, we will suppress the values for which the input number is not a double precision number and the output number is close to a test number that is not a double precision number. These values (statistically, $1/4$ of the obtained values) do not correspond to a worst case for any of the rounding modes and any function (f or f^{-1}).

[6] For smaller numbers, there is no longer any problem of implementation: their radix-2 exponential is 1 or $1^- = 1 - ulp(1/2)$ or $1^+ = 1 + ulp(1)$ depending on their sign and the rounding mode.

Table 1: *Some results for small values in double precision, assuming rounding to the nearest. These results make finding worst cases useless for negative exponents of large absolute value.*

This function	can be replaced by	when
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-53}$
$\exp(\epsilon), \epsilon \leq 0$	1	$\lvert\epsilon\rvert \leq 2^{-54}$
$\sin(\epsilon), \arcsin(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq 2^{-26}$
$\cos(\epsilon)$	1	$\lvert\epsilon\rvert \leq \sqrt{2} \times 2^{-27}$
$\tan(\epsilon), \arctan(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq 2^{-27}$

overflow, when we want to check both functions in the domain defined by $x > 1$ (for 2^x) or $y > 2$ (for $\log_2(y)$), we only have to consider 10 values of the exponent (hence 10×2^{53} test numbers) if we check 2^x for every test number in the domain, whereas we would have to consider 1022 values of the exponent (hence 1022×2^{53} test numbers) if we decided to check $\log_2(x)$ for the test numbers in the corresponding domain.

The decision whether it is better to base our search for worst cases on the examination of f in a given domain I or f^{-1} in the corresponding domain $J = f(I)$ can be helped by examining the value of $T_f(x) = |x \times f'(x)/f(x)|$ in the considered domain. If $T_f(x)$ is much larger than 1, then I will contain less test numbers than J, so it will be preferable to check f in I. If it is much less than 1, it will be preferable to check f^{-1} in J. When $T_f(x)$ is close to 1, a more thorough examination is necessary. In all cases, another important point is which of the two functions is simpler to approximate.

2.3 Optimization: special input values

For most functions, it is not necessary to perform tests for the very small arguments (i.e., arguments whose exponent is negative and has large absolute value). As an example, consider the exponential of a very small positive number ϵ, on a floating-point format with p-bit mantissas, assuming rounding to nearest.

If $\epsilon < 2^{-p}$ then (since ϵ is a p-bit number), $\epsilon \leq 2^{-p} - 2^{-2p}$. Hence,

$$e^{\epsilon} \leq 1 + (2^{-p} - 2^{-2p}) + \frac{1}{2}(2^{-p} - 2^{-2p})^2 \ldots < 1 + 2^{-p}.$$

therefore $\exp(\epsilon)$ is less than $1 + (1/2)ulp(1)$. Thus, the correctly rounded value of $\exp(\epsilon)$ is 1. A similar reasoning can be done for other functions and rounding modes. Some results are given in Table 1.

2.4 Normal and denormal numbers

Our algorithm for finding worst cases assumes that input and output numbers are normalized floating-point values. Hence, we have to separately handle the case of input denormal numbers,

and to check whether there exist normalized floating-point numbers x such that $f(x)$ is so small that it should be represented by a denormal number.

2.4.1 Can the output value be a denormal number?

A method based on the continued fraction theory, and originally designed for finding the worst cases for range reduction was suggested by Kahan [7]. Using this method, one can find the normalized floating point number that is closest to an integer multiple of $\pi/2$ different from zero. This number is:

- $\alpha = 16367173 \times 2^{72} = 7729178919452901966118440 1408$ in single precision;
- $\beta = 6381956970095103 \times 2^{797}$ in double precision.

Therefore, the numbers $A = |\cos(\alpha)| \approx 1.614769798 \times 10^{-9}$ and $B = |\cos(\beta)| \approx 4.687165924 \times 10^{-19}$ are lower bounds on the absolute value of the sine, cosine and tangent of normalized single precision (for A) and double precision (for B) floating-point numbers. These values are much larger than the smallest normalized floating-point numbers. Therefore when the input arguments to sines, cosine and tangent are not so small that the results of Table 1 could be used, then their values are representable as normalized floating-point numbers.

3 Implementation of the method

3.1 Overview of the implementation

The tests are implemented in three steps:

1. As said previously, the first step, which is by far the most time-consuming one, is a *filter*. It consists in eliminating most of the tested arguments. This step amounts to testing if 32 (in general) consecutive bits are all zeroes[7] thus keeping one argument out of 2^{32}, in average. This step is very slow and needs to be parallelized.
2. The second step consists in reducing the number of worst cases obtained from the first step and grouping all the results together in a same file. This is done with a slower but more accurate test than in the first step. As the number of arguments has been reduced, this step is fast enough to be performed on a single machine.
3. The third step is run by the user to restrict the number of worst cases when he needs them. Results on the inverse function are also obtained. As the number of arguments is small, this step is very fast.

Most programs are written in Perl (text data handling, process control...). Concerning the calculations, the tests of the first step are currently written in Sparc assembly language, as they need to be as fast as possible; and for the other calculations, we currently use Maple with an interval arithmetic package.

[7] These are the bits following the first 54 bits of the mantissa, unless the exponent of the output values changes in the tested domain.

3.2 The First Step

Let us give more details about the first step. The user chooses a function f, an exponent, a mantissa size (usually 53) and some other parameters, and the first step starts as follows:[8]

- First, the tested interval I is split into 2^{13} subintervals J_i containing 2^{40} test numbers and the function f is approximated by polynomials P_i of degree d_i (~ 4 to 20) on J_i. We have chosen to use Taylor expansions, as the error can easily be bounded and this approximation suffices for us. For each i, we start with $d_i = 1$, and successively increase d_i until the error due to the approximation is small enough. P_i is expressed modulo the distance between two consecutive test numbers, as we only need to know information about the bits following the mantissa.

- Then, each interval J_i is split into subintervals $K_{i,j}$ containing 2^{15} arguments and P_i is approximated by degree-2 polynomials $Q_{i,j}$ on $K_{i,j}$, with 64-bit precision.

- On $K_{i,j}$: $Q_{i,j}$ is approximated by a degree-1 polynomial (by ignoring the degree-2 coefficient) and the variant of the Euclidean algorithm is used. If it fails, that is, if the obtained distance is too small, one has to perform more tests:

  - $K_{i,j}$ is split into 4 subintervals $L_{i,j,k}$.
  - For each k: the Euclidean algorithm is used on $L_{i,j,k}$, and if it fails, the arguments are tested the one after the other, using two 64-bit additions for each argument.

The program performs the first point (thanks to Maple and the interval arithmetic package), generates a C/assembly source for the following points, then compiles and executes it.

The first step requires much more time than the other steps, thus it is run on several machines (we have used around one hundred machines, in background). As the calculations in different intervals are totally independent, there is no need for communications between different machines. We only have a server on a particular machine to distribute intervals to each client (the program that performs the tests).

It is more interesting to run the program on a network of workstations than on a dedicated machines, and these workstations belong to users, who work on them. We must not disturb them. So, the programs were written so that they can

- run with a low priority (*nice*),

- automatically stop after a given time,

- automatically detect when a machine is used (in particular the keyboard and the mouse) and stop if this is the case.

[8] The numbers that are given here are just those that are generally chosen; the user or the program may choose other values for particular cases.

4 Results: natural (radix-e) exponentials and logarithms

These functions have been among the most time-consuming during our searches, since there is no known way of deducing the worst cases in a domain from the worst cases in another domain. And yet, we have obtained the worst cases for all possible double precision floating-point inputs. They are given in Tables 2 and 3. From these results we can deduce the following properties.

Property 1 (Computation of exponentials) *Let y be the exponential of a double-precision number x. Let y^* be an approximation to y such that the mantissa distance[9] between y and y^* is bounded by ϵ.*

- *for $|x| \geq 2^{-30}$, if $\epsilon \leq 2^{-53-59} = 2^{-112}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y;*
- *for $|x| < 2^{-30}$, if $\epsilon \leq 2^{-53-104} = 2^{-157}$ then rounding y^* is equivalent to rounding y.*

Property 2 (Computation of logarithms) *Let y be the natural (radix-e) logarithm of a double-precision number x. Let y^* be an approximation to y such that the mantissa distance between y and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-64} = 2^{-117}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y.*

Table 2: *Worst cases for the exponential function in the full range. Exponentials of numbers less than $\ln(2^{-1074})$ are underflows (a routine should return 0 or the smallest non zero positive representable number, depending on the rounding mode). Exponentials of numbers larger than $\ln(2^{1024})$ are overflows.*

Interval	worst case (binary)
$[\ln(2^{-1074}), \frac{-1}{2^{30}}]$	$\exp(-1.11101101001100011000111011111011011000100111111101010 2^{-27})$ $= 0.11111111111111111111111111000010010110011100111000100 \quad 1 \quad 1^{59}00...$
$[\frac{-1}{2^{30}}, 0)$	$\exp(-1.00000000000000000000000000000000000000000000000000001 \times 2^{-51})$ $= 0.11111111111111111111111111111111111111111111111111100 \quad 0 \quad 0^{100}10...$
$(0, +2^{-30}]$	$\exp(1.11111111111111111111111111111111111111111111111111111 \times 2^{-53})$ $= 1.00000000000000000000000000000000000000000000000000000 \quad 1 \quad 1^{104}01...$
$[2^{-30}, \ln(2^{1024})]$	$\exp(1.01111111111111100111111111111111011100000000000100100 \times 2^{-32})$ $= 1.00000000000000000000000000000010111111111111101000 \quad 0 \quad 0^{57}11...$ $\exp(1.10000000000000010111111111111111011011111111111011100 \times 2^{-32})$ $= 1.00000000000000000000000000000011000000000000010111 \quad 1 \quad 1^{57}00...$ $\exp(1.10011110100111001011101111111101011000001000000001011 \times 2^{-31})$ $= 1.00000000000000000000000000000011001111010011100101111 \quad 1 \quad 0^{57}10...$ $\exp(110.00001111010100101111001101111010111011001111110100)$ $= 110101100.01010000101101000000100111001000101011101110 \quad 0 \quad 0^{57}10...$

[9] If one prefers to think in terms of relative error, one can use the following well-known results: if the mantissa distance between y and y^* is less than ϵ then their relative distance $|y - y^*|/|y|$ is less than ϵ. If the relative distance between y and y^* is less than ϵ_r then their mantissa distance is less than $2\epsilon_r$.

Table 3: *Worst cases for the natural (radix e) logarithm in the full range.*

Interval	worst case (binary)
$[2^{-1074}, \frac{1}{8}]$	$\log(1.10010100011101101110001100000100110011010111111000111 \times 2^{-384})$ $= -100001001.10110110000011001010111101000111101100110101 \quad 1 \quad 0^{60}10...$ $\log(1.11101010011100011101100001011100111011100000000100000 \times 2^{-509})$ $= -101100000.00101001011010100110011010110100001011111111 \quad 1 \quad 1^{60}00...$ $\log(1.00100110111010011100010011010011001001111001011000000 \times 2^{-232})$ $= -10100000.10101011001011000010010111100110100001000010 \quad 0 \quad 0^{60}10...$
$[\frac{1}{8}, 1)$	$\log(1.11001000111111010100001101010100000000010010000110011 \times 2^{-3})$ $= -1.01111111111111010100111111111000101010010001111011111 \quad 1 \quad 0^{54}11...$ $\log(0.11011101011011110110100110000110010111111000111000100)$ $= -1.00101001000011101010000010011110001101100100001111000 \quad 1 \quad 1^{54}01... \times 2^{-3}$
$(1, 8]$	$\log(110.01000110101000110111111111010101000000111111110111)$ $= 1.11010110001100110110101010001000000001110111101010011 \quad 1 \quad 1^{54}00...$
$[8, 2^{1024}]$	$\log(1.01100010101010001000011000010011011000101001101101100 \times 2^{678})$ $= 111010110.01000111100111101011101001111100100101110001 \quad 0 \quad 0^{64}11...$

5 Results: radix-2 exponentials and logarithms

5.1 Radix-2 exponentials

Using the identity $2^{n+x} = 2^n 2^x$ allows to efficiently fasten the search. First, getting the worst cases for $x \in [1, 2)$ allows to derive all worst cases for $x < -1$ and $x > 1$, since an n-mantissa-bit number belonging to these domains is the sum of an integer and a number with at most n mantissa bits. The worst cases for $|x| < 1$ were obtained through the radix-2 logarithm in $(1/2, 2)$.

These results, given in Table 4, allow to deduce the following property.

Property 3 (Computation of radix-2 exponentials) *Let y be the radix-2 exponential 2^x of a double-precision number x. Let y^* be an approximation to y such that the mantissa distance between y and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-59} = 2^{-112}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y.*

5.2 Radix-2 logarithms

Concerning radix-2 logarithms, let us show that it suffices to test the input numbers greater than 1, and whose exponent is a power of 2:

First, let us show that it suffices to test the input numbers whose exponent is a positive power of 2 to get the worst cases for all input values greater than 1. Consider $x = m \times 2^p$, with $p \geq 1$. The radix-2 logarithm of x is $y = p + \log_2(m)$, so that the infinite mantissa of y begins with $\ell_p = \lfloor \log_2(p) \rfloor$ bits that represent p, followed by the binary representation of $\log_2(m)$. Let p' be the largest power of 2 that is less than or equal to p. Since $\lfloor \log_2(k) \rfloor = \lfloor \log_2(p) \rfloor = \ell_p$, we deduce that the infinite mantissa of the radix-2 logarithm y' of $x' = m \times 2^{p'}$ has the same bits as the infinite mantissa of y after position ℓ_p. Hence, there is a chain of k consecutive ones (or zeroes) after bit 54 of the infinite mantissa of y if and only if there is a chain of k consecutive

ones (or zeroes) after bit 54 of the infinite mantissa of y'. Therefore, from the worst cases for an exponent equal to 2^ℓ we easily deduce the worst cases for exponents between $2^\ell + 1$ and $2^{\ell+1} - 1$. For instance, in Table 5, we only give one of the worst cases: the input value has exponent 512. All worst cases are easily deduced: they have the same mantissa, and exponents between 512 and 1023.

Now, let us show how to deduce the worst cases for numbers less than 1 from the worst cases for numbers greater than 1. This was done as follows: consider a floating-point number $x = m \times 2^{-p}$, with $1 \leq m < 2$, and $p \geq 1$. x is less than 1. Its radix-2 logarithm is $y = -p + \log_2(m)$. The integer part of the *absolute value* of y is $p - 1$ and its fractional part is $1 - \log_2(m)$. So the infinite mantissa of y begins with $\lfloor \log_2(p-1) \rfloor$ bits that represent $p - 1$, followed by the bits that represent $1 - \log_2(m)$. Now, consider the floating point number $x' = m \times 2^{p-1}$. x' is greater than 1. Its radix-2 logarithm is $y' = (p-1) + \log_2(m)$. So, the infinite mantissa of y' begins with $\lfloor \log_2(p-1) \rfloor$ bits that represent $p - 1$ (that is, the same as for y), followed by the bits that represent $\log_2(m)$. But the bits that represent $1 - \log_2(m)$ are obtained by complementation[10] of the bits that represent $\log_2(m)$. Hence, there is a chain of k consecutive ones (or zeroes) after bit 54 of the infinite mantissa of y if and only if there is a chain of k consecutive zeroes (or ones) after bit 54 of the infinite mantissa of y'. Therefore, x is the worst case for input values less than 1 if and only if x' is the worst case for input values greater than 1. This is illustrated in Table 5: the infinite mantissa of the worst case for input values greater than 1 starts with the same bit chain (1000000000 as the mantissa of the worst case for $x < 1$, then the bits that follow are complemented ($1000100011111101001011111100001011001000110 \quad 0 \quad 0^{55}1100...$ for the case $x < 1$ and $0111011100000010110100000011110100110111001 \quad 1 \quad 1^{55}0011...$ for the case $x > 1$).

Using these properties, we rather quickly obtained the worst cases for the radix-2 logarithm of all possible double precision input values: it sufficed to run our algorithm for the input numbers of exponents 0, 1, 2, 4, 8, 16, $\ldots 512$.

These results, given in Table 5, allow to deduce the following property.

Property 4 (Computation of radix-2 logarithms) *Let y be the radix-2 logarithm $\log_2(x)$ of a double-precision number x. Let y^* be an approximation to y such that the mantissa distance between y and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-55} = 2^{-108}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y.*

6 Results: trigonometric functions

The results given in Tables 6 to 11 give the worst cases for functions $\sin$, $\arcsin$, $\cos$, $\arccos$, $\tan$ and $\arctan$. For these functions, we have worst cases in some bounded domain only, because trigonometric functions are more difficult to handle than the other functions.

And yet, it is sometimes possible to prune the search. Let us consider the arc-tangent function of large values. The double precision number that is closest to $\pi/2$ is

$$\alpha = \frac{884279719003555}{562949953421312}.$$

[10] 1 is replaced by 0 and 0 is replaced by 1.

Table 4: *Worst cases for the radix-2 exponential function 2^x in the full range. Integer values of x are omitted.*

Interval	worst case (binary)
$[-1074, 0)$	$2 * *(-1.0010100001100011101010111010111010101111011110110010 \times 2^{-15})$ $= 0.11111111111111100110010100011111010001100000111101111 \quad 0 \quad 0^{57}1110...$
	$2 * *(-1.0100000101101111011011000110010001000101101011001111 \times 2^{-20})$ $= 0.11111111111111111111001000010011001010111010011001110 \quad 1 \quad 1^{57}0000...$
	$2 * *(-1.0000010101010110000000011100100010101011001111110001 \times 2^{-32})$ $= 0.11111111111111111111111111111111101001010110110110000 1 \quad 1 \quad 1^{57}0000...$
	$2 * *(-1.0001100001011011100011011011011011010101100000011101 \times 2^{-33})$ $= 0.11111111111111111111111111111111100111101101010111100 \quad 0 \quad 0^{57}1100...$
$(0, 1024]$	$2 * *(1.1011111110111011110111001000100111011011111111000101 \times 2^{-25})$ $= 1.00000000000000000000000010011011001011000011100000101 \quad 0 \quad 0^{59}1011...$
	$2 * *(1.1110010001011001011001010010011010111111100101001101 \times 2^{-10})$ $= 1.00000000010100111111110000101110110000101011010100 11 \quad 0 \quad 1^{59}0100...$

Table 5: *Worst cases for the radix-2 logarithm function $\log_2(x)$ in the full range. Values of x that are integer powers of 2 are omitted.*

Interval	worst case (binary)
$(0, 1/2)$	$\log_2(1.0110000101010101010111110111010110001000010110110100 \times 2^{-513})$ $= -1000000000.10001000111111010010111111000001011001000110 \quad 0 \quad 0^{55}1100...$
$(1/2, 2^{1024})$	$\log_2(1.0110000101010101010111110111010110001000010110110100 \times 2^{512})$ $= 1000000000.01110111000000101101000000111101001101110 01 \quad 1 \quad 1^{55}0011...$

Assuming rounding to the nearest, the breakpoint that is immediately below α is

$$\beta = \frac{14148475504056879}{9007199254740992}.$$

For any real number x, if $\arctan(x)$ is larger than β then the correctly rounded (to the nearest) value that should be returned when evaluating $\arctan(x)$ in double precision is α. We deduce from this that for $x \geq 5805358775541311$, we should return α. A similar calculation shows that for x between 2536144836019042 and 5805358775541310, we should return $\alpha - ulp(\alpha)$.

For rounded to nearest arc-tangent, the worst case for input numbers larger than 2.25×10^{12} is

$$\frac{4621447055448553}{2048} = 2256565945043.23876953125$$

whose arc-tangent is

$$\overbrace{1.10010010000111111011010101000100010000100101010011\overset{53 \text{ bits}}{00}}$$
$$1 \quad 0^{45}111011\ldots$$

Property 5 (Computation of sines) *Let y be the sine of a double-precision number x satisfying $1/32 \leq |x| \leq 2$. Let y^* be an approximation to y such that the mantissa distance between y*

and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-65} = 2^{-118}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y.

Property 6 (Computation of arc-sines) *Let y be the arc-sine of a double-precision number x satisfying $\sin(1/32) \leq |x| \leq 1$. Let y^* be an approximation to y such that the mantissa distance between y and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-64} = 2^{-117}$ then for any rounding mode of the IEEE-754 standard, rounding y^* is equivalent to rounding y.*

Similar properties are deduced for the other trigonometric functions: $\epsilon \leq 2^{-108}$ for cosines between $1/64$ and $12867/8192 = 1.5706\ldots$, $\epsilon \leq 2^{-115}$ for arc-cosine between $\cos(12867/8192) \approx 0.0001176$ and $\cos(1) \approx 0.54$; $\epsilon \leq 2^{-110}$ for tangent between $1/32$ and $\arctan(2)$; and $\epsilon \leq 2^{-108}$ for arc-tangent between $\tan(1/32)$ and 2.

Table 6: *Worst cases for the sine function in the range* $[1/32, 2]$.

Interval	worst case (binary)
$[\frac{1}{32}, 1]$	$\sin 0.011111111001110110011101110011100111010000111101101101$ $= 0.011110100110010101000001110011000011000100011010010101 \quad 1 \quad 1^{65}00\ldots$
$[1, 2]$	$\sin 1.100100100001111111011010101000100010000101101000011000$ $= 0.111111111111111111111111111111111111111111111111111111 \quad 1 \quad 1^{54}01\ldots$

Table 7: *Worst cases for the arc-sine function in the range* $[\sin(1/32) = 0.0312449\ldots, 1]$.

Interval	worst case (binary)
$[\sin\frac{1}{32}, 1]$	$\arcsin 0.011110100110010101000001110011000011000100011010010110$ $= 0.011111111001110110011101110011100111010000111101101101 \quad 0 \quad 0^{64}10\ldots$

Table 8: *Worst cases for the cosine function in the range* $[1/64, 12867/8192]$. $12867/8192$ *is slightly less than* $\pi/2$.

Interval	worst case (binary)
$[\frac{1}{64}, 1]$	$\cos 1.100101111100110011010011110100101100010000111000 1111 \times 2^{-6}$ $= 0.111111111110101110110011010111010100001110000101010000 \quad 1 \quad 1^{55}01\ldots$
$[1, \frac{12867}{8192}]$	$\cos 1.011010111000101001100010011100111101011111000010000 1$ $= 1.001100110111111111000101101100000111001010110001010010 \quad 1 \quad 0^{54}10\ldots \times 2^{-3}$

Conclusion

The worst cases we have obtained will allow the design of reasonably fast routines for evaluating most common mathematical functions with correct rounding (at least in some intervals) in the four rounding modes specified by the IEEE-754 standard. We are extending the domains for the

Table 9: *Worst cases for the arc-cosine function in the range* $[\cos(12867/8192), \cos(1)] \approx [0.0001176, 0.540]$.

Interval	worst case (binary)
$[\cos\left(\frac{12867}{8192}\right), \cos(1)]$	$\arccos 1.11111101011100110111101111101001000101000010101111000 \times 2^{-11}$ $= 1.10010001111000000000011011010100000111011000110110000 \quad 1 \quad 1^{62}00...$

Table 10: *Worst cases for the tangent function in the range* $[1/32, \arctan(2)]$, *with* $\arctan(2) \approx 1.107148$.

Interval	worst case (binary)
$[\frac{1}{32}, \frac{1}{16}]$	$\tan(1.010100000100100001101011001011111000011100000000010100 \times 2^{-5})$ $= 1.010100000111100011001110101111111111111001110001110010 \quad 1 \quad 0^{57}10... \times 2^{-5}$
$[\frac{1}{16}, \tan^{-1}\frac{1}{2}]$	$\tan(1.101000110011111110011001010101100010111001110101101101 \times 2^{-3})$ $= 1.101010010011001111111111000010111011010110011011101101 \quad 0 \quad 0^{55}10... \times 2^{-3}$
$[\tan^{-1}\frac{1}{2}, \tan^{-1} 2]$	$\tan(0.101000110101011000011011100100010010000110101001101110)$ $= 0.101111011101001001001111101110011100110000010100111110 \quad 1 \quad 1^{54}00...$

functions for which we have not yet obtained the worst cases in the full range. More examples can be obtained through the URL `http://www.ens-lyon.fr/~jmmuller/TMD.html`. These "worst cases" will also be good test cases for checking whether a library provides correct rounding or not.

References

[1] R. C. Agarwal, J. C. Cooley, F. G. Gustavson, J. B. Shearer, G. Slishman, and B. Tuckerman. New scalar and vector elementary functions for the IBM system/370. *IBM Journal of Research and Development*, 30(2):126–144, March 1986.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, New York, 1985.

Table 11: *Worst cases for the arc-tangent function in the range* $[\tan(1/32), 2]$, *with* $\tan(1/32) \approx 0.0312601$.

Interval	worst case (binary)
$[\tan\left(\frac{1}{32}\right), \frac{1}{2}]$	$\arctan(1.101010010011001111111111000010111011010110011011101101 \times 2^{-3})$ $= 1.101000110011111110011001010101100010111001110101101100 \quad 1 \quad 1^{55}01... \times 2^{-3}$
$[\frac{1}{2}, 2]$	$\arctan(0.101111011101001001001111110111001110011000001010011111)$ $= 0.101000110101011000011011100100010010000110101001101110 \quad 0 \quad 0^{55}11...$

[3] J. T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, January 1980.

[4] C. B. Dunham. Feasibility of "perfect" function evaluation. *SIGNUM Newsletter*, 25(4):25–26, October 1990.

[5] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[6] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[7] W. Kahan. Minimizing q*m-n, text accessible electronically at http://http.cs.berkeley.edu/∼wkahan/. At the beginning of the file "nearpi.c", 1983.

[8] V. Lefèvre. *Developments in Reliable Computing*, chapter An Algorithm That Computes a Lower Bound on the Distance Between a Segment and $\mathbb{Z}^2$, pages 203–212. Kluwer, Dordrecht, Netherlands, 1999.

[9] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[10] V. Lefèvre, J. M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, Asilomar, USA, 1997. IEEE Computer Society Press, Los Alamitos, CA.

[11] V. Lefèvre, J.M. Muller, and A. Tisserand. Toward correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[12] J.M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[13] M. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In E. E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 138–145, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[14] M. J. Schulte and E. E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, August 1994.

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