Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité de recherche associée au CNRS n° 1398

The Table Maker's Dilemma

Vincent Lefèvre
Jean-Michel Muller
Arnaud Tisserand

Feb. 1998

Research Report N° 98-12

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.00
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

The Table Maker's Dilemma

Vincent Lefèvre
Jean-Michel Muller
Arnaud Tisserand

Feb. 1998

Abstract

The Table Maker's Dilemma is the problem of always getting correctly rounded results when computing the elementary functions. After a brief presentation of this problem, we present new developments that have helped us to solve this problem for the double-precision exponential function in a small domain. These new results show that this problem can be solved, at least for the double-precision format, for the most usual functions.

Keywords: Table Maker's Dilemma, Elementary Functions, Correct Rounding, Floating-Point Arithmetic.

Résumé

Le dilemme du concepteur de tables est le problème de toujours fournir des résultats arrondis correctement lors du calcul de fonctions élémentaires. Après une brève présentation du problème, nous présentons de nouveaux résultats qui permettent de résoudre ce problème pour l'exponentielle en double précision dans un petit domaine. Ces résultats montrent que le problème peut être résolu, au moins pour le format double précision, pour la plupart des fonctions usuelles.

Mots-clés: Dilemme du constructeur de tables, fonctions élémentaires, arrondi correct, arithmétique virgule flottante.

1 Introduction

The IEEE-754 standard for floating-point arithmetic [2, 11] requires that the results of the arithmetic operations should always be correctly rounded. That is, once a rounding mode is chosen among the four possible ones, the system must behave as if the result were first computed *exactly*, with infinite precision, then rounded. There is no similar requirement for the elementary functions. The possible rounding modes are:

- round towards $-\infty$: $\nabla(x)$ is the largest machine number less than or equal to x;
- round towards $+\infty$: $\Delta(x)$ is the smallest machine number greater than or equal to x;
- round towards 0: $\mathcal{Z}(x)$ is equal to $\nabla(x)$ if $x \geq 0$, and to $\Delta(x)$ if $x < 0$;
- round to the nearest: $\mathcal{N}(x)$ is the machine number which is the closest to x (with a special convention if x is exactly between two consecutive machine numbers).

Throughout this paper, "machine number" means a number that is exactly representable in the floating-point format being considered. The first 3 modes are called "directed modes". A very interesting discussion on the current status of the IEEE-754 standard is given by Kahan [13].

Our ultimate goal is to provide correctly rounded elementary functions at a "reasonable" cost. Requiring correctly rounded results not only improves the accuracy of computations: it is the best way to make numerical software portable. Moreover, as noticed by Agarwal et al. [1], correct rounding facilitates the preservation of useful mathematical properties such as monotonicity, symmetry[1], and some identities.

Throughout the paper, we want to implement a function f (f being sine, cosine, exponential, natural logarithm or arctangent) in a radix-2 floating-point number system, with n mantissa bits. We assume that from any real number x and any integer m (with $m > n$), we are able to compute an approximation of $f(x)$ with an error on its mantissa y less than or equal to 2^{-m+1}. This can be achieved with the presently known methods, using polynomial or rational approximations or Cordic-like algorithms, provided that a very careful range reduction is performed [19, 12, 22, 7, 17]. The intermediate computations can be carried out using a larger fixed-point or floating-point format.

Therefore the problem is to get an n-bit mantissa floating-point correctly rounded result from the mantissa y of an approximation of $f(x)$, with error $\pm 2^{-m+1}$. This is not possible if y has the form:

- in rounding to the nearest mode,

$$\overbrace{\underbrace{1.xxxxx\ldots xxx}_{n \text{ bits}}1000000\ldots000000}^{m \text{ bits}}xxx\ldots$$

[1] Only when the rounding mode itself is symmetric, i.e., for rounding to the nearest and rounding toward zero.

or

$$\overbrace{\underbrace{1.xxxxx\ldots xxx}_{n \text{ bits}}0111111\ldots111111}^{m \text{ bits}}xxx\ldots;$$

- in rounding towards 0, $+\infty$ or $-\infty$ modes,

$$\overbrace{\underbrace{1.xxxxx\ldots xxx}_{n \text{ bits}}0000000\ldots000000}^{m \text{ bits}}xxx\ldots$$

or

$$\overbrace{\underbrace{1.xxxxx\ldots xxx}_{n \text{ bits}}1111111\ldots111111}^{m \text{ bits}}xxx\ldots.$$

This problem is known as the *Table Maker's Dilemma* (TMD) [11]. For example, assuming a floating-point arithmetic with 6-bit mantissa,

$$\sin(11.1010) = 0.0\boxed{111011}01111110\ldots,$$

a problem may occur with rounding to the nearest if the sine function is not computed accurately enough.

Although we deal with radix-2 floating-point systems, the same problem obviously occurs with other radices[2]. For instance, in a radix 10 floating-point number system, with 4 digit mantissas,

$$\sin(73.47) = -0.\boxed{9368}0000058\ldots,$$

therefore a problem may occur with a directed rounding mode. Similar problems with radix 10, directed rounding modes, and the cosine function are given in Table 1.

n	x	$\cos(x)$
4	1.149	$0.409400000428\ldots$
5	1.6406	$-0.69747000000219\ldots$
6	1.41162	$0.15850500000180\ldots$
7	1.225910	$0.33808970000005907\ldots$
8	1.9509449	$-0.37105844000000012\ldots$

Table 1: Worst cases for n-digit mantissa, radix-10, floating-point numbers between 1 and 2, directed rounding modes and the cosine function.

Our problem is to know if there is a maximum value for m, and to estimate it. If this value is not too large, then computing correctly rounded results will become possible.

It is worth noticing that even if the maximum value of m is quite large, it is not necessary to always evaluate $f(x)$ with large accuracy. The key point is

[2] Other radices are actually used. For example, most pocket computers use radix 10.

that if the intermediate approximation y is not accurate enough, we are in one of the four cases listed above, so we can be aware of the problem. This leads to a solution first suggested by Ziv [23]: we first start to approximate $f(x)$ with a value of m slightly larger than n (say, $n+10$). In most cases this will suffice to get a correctly rounded result. If this does not suffice, we continue with a larger value of m, and so on. We increase m until we have enough accuracy to be able to correctly round $f(x)$. In a practical implementation of this strategy, the first step may be hardwired, and the other ones are written in software. It is very important that the first steps (say, the first two steps) be fast. The other ones are unlikely to occur, so they may be slower without significantly changing the average computational delay.

In 1882, Lindemann showed that the exponential of a nonzero (possibly complex) algebraic number is not algebraic [3]. From this we easily deduce that the sine, cosine, exponential, or arctangent of a machine number different from zero cannot be a machine number (and cannot be exactly between two consecutive machine numbers), and the logarithm of a machine number different from 1 cannot be a machine number (and cannot be exactly between two consecutive machine numbers). Therefore, for any non-trivial machine number x (the cases $x=0$ and $x=1$ are obviously handled), there exists m such that the TMD cannot occur. This is not always true for functions such as 2^x, $\log_2 x$, or x^y. This is why we do not consider them in this paper[3]. Since there is a finite number of machine numbers x, there exists a value of m such that for any x the TMD cannot occur. Schulte and Swartzlander [20, 21] proposed algorithms for producing correctly rounded results for the functions $1/x$, $\sqrt{x}$, 2^x and $\log_2 x$ in single-precision. Those functions are not discussed here, but Schulte and Swartzlander's result helped us to start our study. To find the correct value of m, they performed an exhaustive search for $n=16$ and $n=24$. For $n=16$, they found $m=35$ for $\log_2 x$ and $m=29$ for 2^x, and for $n=24$, they found $m=51$ for $\log_2 x$ and $m=48$ for 2^x. We performed similar exhaustive searches for very small values of n and a few different exponents only, and always found m close to twice n. For instance, the largest value of m for cosines of single-precision numbers between 1 and 2 is obtained for 3 numbers, including

$$x = 1.00001100010011010100101_2 = \frac{8,791,717}{8,388,608},$$

whose cosine equals

$$0.0\underbrace{111111111001111010111000}_{24}0\underbrace{111111111111111111111111}_{24}000011101\ldots$$

For this number $m=49$. One would like to extrapolate those figures and find $m \approx 2n$.

In [16], two of us showed that if the bits of $f(x)$ after the n-th position can be viewed as if they were random sequences of zeroes and ones, with probability $\frac{1}{2}$ for 0 as well as for 1, then for n-bit mantissa normalized floating-point input numbers, assuming that n_e different exponents are considered, m is close to

[3] However, these functions satisfy some properties that might probably be used. For instance, if x is a machine number and if x is not an integer, then 2^x cannot be a rational number.

$2n + \log_2(n_e)$ with very high probability[4]. Similar probabilistic studies have been done previously by Dunham [9], and by Gal and Bachelis [10].

Of course, probabilistic arguments do not constitute a proof: they can only give an estimate of the accuracy required during the intermediate computation to get a correctly rounded result. There are a few results from number theory that can be used to get an upper bound on the maximum value of m. Unfortunately, such bounds are very large. In practice, they appear to be much larger than the actual maximum value of m. For instance, using a theorem due to Nesterenko and Waldschmidt [18], we could show [16] that getting correctly rounded results in double-precision could be done with $m \approx 1\,000\,000$. Although computing functions with 1 000 000 digits is feasible (on current machines, this would require less than half an hour using Brent's algorithms [5] for the functions and Zuras' algorithms [24] for multiplication), this cannot be viewed as a satisfactory solution. Moreover, after the probabilistic arguments, the actual bound is likely to be around 110. Therefore, we decided to study how could an exhaustive search of the worst cases be possible.

2 Exhaustive tests

For a given elementary function f, a given floating-point format (in practice, IEEE-754 double precision), a given rounding mode, and a given range (ideally, the range of all machine numbers for which the function is mathematically defined, but this may be difficult in some cases), we want to find "worst cases". A "worst case" is a machine number x, belonging to the considered range, for which the distance between $f(x)$ and a machine number (for directed rounding modes) or the distance between $f(x)$ and the middle point of two consecutive machine numbers (for rounding to the nearest) is minimal. Here, "distance" means "mantissa distance", that is, the distance between $f(x)$ and y is

$$\frac{|f(x) - y|}{2^{\lfloor \log_2 |f(x)| \rfloor}}.$$

To find such worst cases within reasonable time, we need to workout efficient filters. A "filter" is an algorithm that allows to eliminate most cases, that is, that quickly eliminates any number x for which the distance between $f(x)$ and a machine number or the middle point of two consecutive machine numbers is far from being minimal. If the filter is adequately designed, a very small fraction of the set of the considered machine numbers will not have been filtered out, so that we will be able to deal with these numbers later on, during a second step, using more accurate yet much slower techniques.

We consider here two different filters. Both are based on very low degree (consequently, valid on very small domains only) polynomial approximations of the functions being considered.

[4] If all possible exponents are considered, this formula can be simplified: with a p-bit floating-point number system, it becomes $n+p$. And yet, we prefer the formula $2n+\log_2(n_e)$ because in practice, we seldom have to deal with all possible exponents: when an argument is very small (i.e., its exponents are negative and have a rather large absolute value), simple Taylor expansions suffice to find a correctly rounded result. When an argument is very large, the value of the function may be too large to be representable (for instance with the exponential function), or so close to a limit value that correctly rounding it becomes obvious (for instance with the arctangent function).

The exhaustive search is restricted to a given interval; outside this interval other methods can be chosen. Indeed, if x is small enough (less than 2^{-53} for double precision), an order-1 Taylor expansion can be used, and if x is large enough, the exponential gives an overflow, and the values of trigonometric functions do not make much sense any longer for most applications (although we prefer to *always* return correctly rounded results).

A similar work for the single-precision floating-point numbers has already been done by various authors [20, 16]. Our methods have first been applied to the exponential function with double-precision arguments in the interval $[\frac{1}{2}, 1)$. We are extending them to other intervals and other functions. Results concerning other intervals cannot be deduced from the results in $[\frac{1}{2}, 1)$, thus these intervals will have to be considered later.

Concerning the double-precision normalized floating-point numbers ($n = 53$), there are 2^{52} possible mantissas (the first bit is always 1), which is a large number. Thus one of the most important concerns is that the test program should be as fast as possible, even though it is no longer portable (we only used SparcStations). Indeed, one cycle per test corresponds to approximately two years of computation on a 100 MHz workstation: saving any cycle is important!

2.1 First filtering strategy

2.1.1 Algorithm (general idea)

The TMD occurs for an argument x if the binary expansion of $f(x)$ has a sequence of consecutive 0's or 1's starting from a given position, where the position and the length of the sequence depend on the final and the intermediate precisions. The problem consists in checking for all x whether these bits are all 0's or all 1's, and in this case, finding the maximal value of m for which the TMD occurs.

As explained previously, to get fast tests, we apply a two-step strategy similar to Ziv's [23]. The first step (that is, the filter) must be very fast and eliminate most arguments; the second one, that may be much slower, consists in testing again the arguments that have failed at the first step, by approximating $f(x)$ with higher precision.

The first step consists in testing a given subsequence of the binary expansion (bits of weight 2^{-54} to $2^{-(M-1)}$) of an approximation of each $f(x)$ with error 2^{-M}. In our case, we have chosen $M = 86$ from the algorithm and the processor characteristics. The test fails if and only if these bits are the same, and the argument will have to be checked during the second step. In the opposite case, one can easily show that the bits 54 to M of the *exact* result cannot all be equal. In the following, we assume $f(x) = e^x$.

2.1.2 First step: overall presentation

To perform the first step, the exponential function is approximated by a polynomial. The chosen polynomial must have a low degree for the following reasons: on the one hand, to reduce the computation time, on the other hand, to limit possible rounding errors. As a consequence, the approximation is valid on a small interval only.

Let us consider an interval $[-2^{r-53}, 2^{r-53})$, where r is a positive integer (it will be about 15), in which we know a polynomial approximation. We can use the formula

$$e^{t+x} = e^t . e^x$$

where x is in this interval and t has the form $(2\ell+1)2^{r-53}$, to test every argument in the range 1/2 to 1.

The main idea consists in computing a polynomial approximation of the exponential in the intervals $[t-2^{r-53}, t+2^{r-53})$, then evaluating the obtained polynomial at consecutive values by the finite difference method [14], briefly recalled in Section 2.1.4. This method is attractive, for it only requires additions (two for each argument in the case of a polynomial of degree two, which was chosen) and the computations can be performed modulo 2^{-53} (the first tested bit having weight 2^{-54}). Thus the algorithm consists of two parts:

- the computation of the e^t's with error 2^{-s}, where t will have the form $(2\ell+1)2^{r-53}$ (and will be between 1/2 and 1),
- the computation of the e^{t+x}'s with error 2^{-M}, where x is in $[-2^{r-53}, 2^{r-53})$, knowing e^t with error 2^{-s}.

For our tests, we chose $r = 16$, $M = 86$ and $s = 88$. These parameters have been determined from an error analysis (giving relations between r, M and s) and hardware parameters (register width and disk size).

2.1.3 Computing the e^t's

We seek to compute $u_\ell = e^{y+\ell z}$ with error 2^{-s}, where $y = 1/2+2^{r-53}$, $z = 2^{r-52}$ and ℓ is an integer such that $0 \leq \ell < 2^{51-r}$.

The following method allows to compute a new term with only one multiplication (using the formula $e^{a+b} = e^a.e^b$), with a balanced computation tree to avoid losing too much precision, and without needing to store too many values u_ℓ, the disk storage being limited.

We write ℓ in base 2: $\ell = \ell_{50-r}\ell_{49-r} \ldots \ell_0$. We have

$$e^{y+\ell z} = e^y . e_0^{\ell_0} . e_1^{\ell_1} \ldots e_{50-r}^{\ell_{50-r}}$$

where $e_i = (e^z)^{2^i}$; to simplify, e^y and the e_i's are precomputed.

The problem now consists in computing for all $I \subseteq \{0, 1, \ldots, h\}$ containing h:

$$P_I = \prod_{i \in I} e_i$$

where the e_i's are the above precomputed values ($e_h = e^y$). For that, we partition $\{0, 1, \ldots, h\}$ into two subsets that have the same size (or almost) to balance, and we apply this method recursively on each subset (we stop the recursion when the set has only one element); then we calculate all the products xy, where x is in the first subset and y is in the second subset. The last products xy are computed later, just before testing the corresponding interval.

These computations can be performed sequentially on one machine: they only need several minutes.

x	$P(x)$	$\Delta P(x)$ $= P(x+1)$ $-P(x)$	$\Delta^2 P(x)$ $= \Delta P(x+1)$ $-\Delta P(x)$	$\Delta^3 P(x)$ $= \Delta^2 P(x+1)$ $-\Delta^2 P(x)$
0	1	0	4	6
1	1	4	10	6
2	5	14	16	6
3	19	30	22	6
4	49	52	28	6
5	101	80	34	6
6	181	114	40	6
7	295	154	46	6
8	449	200	52	6
9	649	252	58	
10	901	310		
11	1211			

Table 2: Illustration of the finite difference method, with $P(x) = x^3 - x^2 + 1$, $x_0 = 0$ and $h = 1$. Once 4 consecutive values of P are evaluated, we compute 3 values of ΔP, 2 values of $\Delta^2 P$ and one value of $\Delta^3 P$ using subtractions. After this, evaluating a new value $P(x_i)$ requires 3 additions only.

2.1.4 Computing and testing the e^{t+x}'s

The exponential e^x is approximated by $1+x+\frac{1}{2}x^2$ on the interval $[-2^{r-53}, 2^{r-53}]$. The computation of the e^{t+x}'s using that polynomial approximation is performed using the *finite difference method* [14]. This method allows to calculate consecutive values of a polynomial of degree d with only d additions for each value. It is probably known by most readers, so we just briefly recall it, to make the paper self-contained. Assume we wish to evaluate a degree-d polynomial P at regularly-spaced values x_0, $x_1 = x_0 + h$, $x_2 = x_0 + 2h$, $\ldots$, $x_i = x_0 + ih$, $\ldots$. Define the values:

- $\Delta P(x_i) = P(x_{i+1}) - P(x_i)$,
- $\Delta^2 P(x_i) = \Delta P(x_{i+1}) - \Delta P(x_i)$,
- $\Delta^3 P(x_i) = \Delta^2 P(x_{i+1}) - \Delta^2 P(x_i)$,
- $\ldots$
- $\Delta^d P(x_i) = \Delta^{d-1} P(x_{i+1}) - \Delta^{d-1} P(x_i)$.

It is quite easy to show that for any i, $\Delta^d P(x_i)$ is a constant that only depends on P. This is illustrated in Table. 2 in the case $P(x) = x^3 - x^2 + 1$, $x_0 = 0$ and $h = 1$. This shows that once the first $d+1$ values of P, then the first d values of ΔP, then the first $d-1$ values of $\Delta^2 P$, $\ldots$ then the (constant) value of $\Delta^d P$ are computed, it suffices to perform d additions to get a new value of P.

In our problem (with $h = 1$ and $x_0 = 0$), the polynomial is not given by its coefficients nor by its first values $P(0)$, $P(1)$, $\ldots$, $P(d)$, but by the elements $P(0)$, $\Delta P(0)$, $\Delta^2 P(0)$, $\ldots$. These values are more suited to our computations.

They are the coefficients of the polynomial in the base

$$\left\{1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{3!}, \dots\right\}.$$

The test of the bits cannot be performed with only one instruction on a Sparc processor without modifications. This test consists in testing whether a number is in a given interval centered on 0 modulo 2^{-53}. With the finite difference method, we can translate the interval so that its lower bound is 0, and we can use an unsigned comparison to test whether the number is in the interval; thus we finally need only one instruction.

Thus the first step takes 5 cycles per argument on average (two 64-bit additions and one 32-bit comparison), the time required by the other computations – branch instructions – being negligible.

2.1.5 Parallelizing on a computer network

The total amount of CPU time required for our computations is several years. We wanted to quickly get the results of the tests (within a few months). Therefore we had to parallelize the computations. We used the network of 120 workstations of our department. These workstations often have a small load. We sought to use each machine at its maximum without disturbing its user; in particular, the process uses very little memory and there are very few communications (e.g. file system accesses).

2.1.6 Second step

The second step consists in a more precise test for the arguments that failed during the first step. The exponential is computed with a higher precision, chosen so that the probability that the test fails for an argument is very low.

We chose a variation of De Lugish's algorithm [8] for computing the exponential (since it contains no multiplication). The computations were performed on 128-bit integers (four 32-bit integers). The algorithm was implemented in assembly language (which is, for the present purpose, simpler than in C language).

2.1.7 Improvements

The algorithm given above (first filtering strategy) has been used during the summer of 1996. We present the results in Section 3. Since 1996, we have developed another filtering strategy, presented in Section 2.2.1. Before examining that strategy, let us notice some remarks that may help to accelerate the tests.

First, we can test both a function and its inverse at the same time. As we would need to test twice as many numbers as before, it seems that we would save nothing, but this is no longer true in combination with the following methods.

Since testing a function and testing its inverse are equivalent (the exponential of a machine number a is close to a machine number b if and only if the logarithm of b is close to the machine number a), we can choose the function that is the fastest to test, i.e. the function for which the number of points to test is the smallest in the given domain; of course, this choice may depend on the considered domain.

We can approximate a degree-2 polynomial by several degree-1 polynomials in subintervals (which is not equivalent to directly approximating the function by degree-1 polynomials). By doing this, the tests would require 3 cycles per argument in average instead of 5 cycles. But we may do better: now we have a function that may be simple enough (a translation of a linear function) to find an attractive algorithm based on advanced mathematical properties. This is done in the next section.

2.2 Second filtering strategy: towards faster tests

2.2.1 Faster tests

As previously, we wish to know how close can $f(x)$ be to a machine number (or the middle point of two consecutive machine numbers), where x is a machine number. Our second approach is based on the 3-*distance theorem* [4]. Let us start from a circle and an angle α. We build a sequence of points x_1, x_2,... on the circle (see Fig. 1) as follows:

- x_0 is chosen anywhere on the circle ;
- x_{i+1} is obtained by rotating x_i of angle α.

The 3-distance theorem says that, at any time during this process (that is, for any given n, when the points x_0, x_1, $x_2, \ldots$, x_n are built), the distance between two points that are neighbors on the circle can take at most 3 possible values. This is illustrated on Fig. 2. Moreover, there are infinitely many values of n (number of points) for which the distance between two points that are neighbors on the circle can take two values.

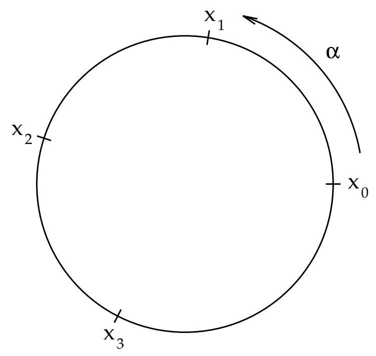

Figure 1: The 3-distance theorem. Construction of the first points.

To use this result, we will do the following:

- the initial domain is cut into subdomains small enough to make sure that, within an acceptable error, function f can be approximated by a linear approximation on each sub-domain. We also assume that all numbers in a sub-domain I, as well as in $\exp(I)$, have the same exponent;

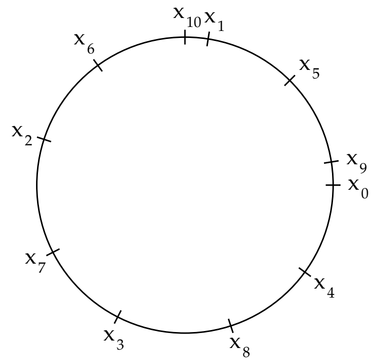

Figure 2: The 3-distance theorem. Construction of more points. There are at most 3 possible distances between consecutive points.

- now, let us focus on a given sub-domain. We scale the problem (input floating-point numbers, linear approximation) so that floating-point numbers now correspond to integers. Therefore our problem becomes the following: given a rectangular grid (whose points have integral coordinates belonging to a given range), a straight line, and a (small) real number ϵ, are there points of the grid that are within a distance ϵ from the line ?
- we slightly modify the problem again: we translate the line down by a distance ϵ. Our initial problem is equivalent to: are there points of the grid that are *above* the (translated) line, and at a distance less than 2ϵ from it ?

After these modifications, assuming that the translated straight line is of equation $y = ax - b$, our problem becomes: is there an integer x, belonging to the considered domain, and another integer i such that $0 \leq i - ax + b \leq 2\epsilon$? Now, we compute **modulo 1**, and we look for an integer x such that $b - ax$ modulo 1 is less than 2ϵ. Now we can understand in what this problem is related to the 3-distance theorem. We compute modulo 1: the reals modulo one can be viewed as the points of a circle, a given point being arbitrarily chosen for representing 0. Adding a value a modulo 1 to some number z is equivalent to rotating the point that represents z of an angle $2\pi a$. Therefore, by choosing $x_0 = b$ and $\alpha = -2\pi a$, building the values $b - ax$ (where x is an integer) is equivalent to building the values x_1, x_2, x_3... of the 3-distance theorem.

Now, we give a fast algorithm that computes the lower bound on the distance d between a segment and the points of a regular, rectangular grid. More details are given in [15]. This algorithm is related to the 3-distance theorem and is an extension to Euclid's algorithm, which is used to compute the development of a real number (the slope of the segment) into a continued fraction. We have seen that, in the 3-distance theorem,that there are infinitely many values of n (number of points) for which the distance between two points that are neighbors on the circle can take two values. Let us call γ_i and δ_i these distances the i-th time there are two distances. The algorithm given below directly computes the sequences γ_i and δ_i, without having to build the points x_i.

In the following, $\{x\}$ denotes the *positive fractional part* of the real number x, i.e., $\{x\} = x - \lfloor x \rfloor$.

As previously , let $y = ax - b$ be the equation of the segment, where x is restricted to the interval $[0, N-1]$. The following algorithm gives a lower bound on $\{b - ax\}$, where x is an integer in $[0, N-1]$.

```
Initialization:   γ = {a} ; δ = 1 − {a} ; d = {b} ; u = v = 1;
Infinite loop:
    if (d < γ)
        while (γ < δ)
            if (u + v ⩾ N) exit
            δ = δ − γ;  u = u + v;
        if (u + v ⩾ N) exit
        γ = γ − δ;  v = v + u;
    else
        d = d − γ;
        while (δ < γ)
            if (u + v ⩾ N) exit
            γ = γ − δ;  v = v + u;
        if (u + v ⩾ N) exit
        δ = δ − γ;  u = u + v;
Returned value:  d
```

γ and δ contain the lengths that appear in the 3-distance theorem; u and v contain the number of intervals (arcs) of respective lengths γ and δ ($u+v$ is the total number of intervals, i.e., the number of points). During the computations, d is the temporary lower bound (for the current number of points $u+v$), which becomes the final lower bound when the algorithm stops.

3 Results

In 1996, we tested the exponential function with double-precision arguments in the interval $[\frac{1}{2}, 1)$, using the first filtering strategy (see Section 2.1). We needed up to 121 machines during three months for the first step. The second step was carried out in less than one hour on one machine.

The very same results have been obtained in January 1997, with the second filtering strategy (see Section 2.2.1) . Thanks to this better strategy, we needed a few machines only. The computation was approximately 150 times faster than with the first strategy (we tested 30 arguments per cycle on average).

Among all the $2^{20} = 1\,048\,576$ intervals, each containing 2^{32} values, $2\,097\,626$ exceptions have been found. From the probabilistic approach, the estimated number of exceptions was to be $2^{21} = 2\,097\,152$. This shows that, in this case, the probabilistic estimate is excellent. Our experiments allowed us to perform an in-depth check of the probabilistic hypotheses. For each double-precision number x, let us define an integer k_x such that the mantissa of e^x has the following form:

$$\underbrace{b_0 b_1 b_2 \ldots b_{52}}_{53 \text{ bits}} \underbrace{b_{53} 000 \ldots 00}_{k \text{ bits}} 1 \ldots$$

or

$$\underbrace{b_0 b_1 b_2 \ldots b_{52}}_{53 \text{ bits}} \underbrace{b_{53} 111 \ldots 11}_{k \text{ bits}} 0 \ldots .$$

From the probabilistic hypotheses, $k_x \geq k_0$ for given numbers x and k_0 with a probability of 2^{2-k_0}.

In the following table, we give the following numbers as a function of k_0 (with $k_0 \geq 40$) for $\frac{1}{2} \leq x < 1$:

- the actual number of arguments for which $k_x = k_0$;
- the actual number of arguments for which $k_x \geq k_0$;
- the estimated (after the probabilistic hypotheses) number of arguments for which $k_x \geq k_0$: $2^{52} \times 2^{2-k_0}$, i.e. 2^{54-k_0}.

k_0	$k_x = k_0$	$k_x \geq k_0$	estimate
40	8185	16427	16384.0
41	4071	8242	8192.0
42	2113	4171	4096.0
43	999	2058	2048.0
44	541	1059	1024.0
45	258	518	512.0
46	123	260	256.0
47	63	137	128.0
48	43	74	64.0
49	14	31	32.0
50	5	17	16.0
51	7	12	8.0
52	1	5	4.0
53	2	4	2.0
54	1	2	1.0
55	1	1	0.5

We see that the probabilistic estimate is still very good.

Now, let us focus on the worst case, for the exponential of double precision numbers between 1/2 and 1. This worst case is constituted by the exception for $k_x = 55$, namely

$$x = 0.11010110011001111110111111001000110101101001110111110000_2$$
$$= \frac{471,483,227,223,279}{562,949,953,421,312}$$

with

$$\begin{aligned} \exp x \quad = \quad & 10.0100111110000101110010010111110 \\ & 00001111011100111000 0\ \underbrace{11111 \ldots 1111}_{54}\ 01 \ldots . \end{aligned}$$

Therefore the value of m for the exponential function in $[\frac{1}{2}, 1)$ in double-precision is $109 = 53+55+1$. We are computing the worst cases for the logarithm function. For instance, one of the worst cases for logarithms of double precision numbers between $1+2^{-29}$ and $1+351040\times 2^{-29}$, and directed rounding, is

$$\begin{aligned} x &= 1.0000000000100000100111000000011101101111011010000101_2 \\ &= \frac{4,505,840,534,353,541}{4,503,599,627,370,496} \end{aligned}$$

with

$$\begin{aligned} \ln x \quad = \quad & 2^{-11}\times 1.00000100110011111001111101100000 \\ & 10010011111111\ \underbrace{000000000000\ldots000000000}_{42}\ 10110\ldots. \end{aligned}$$

4 Conclusion

We have shown, using as an example the case of the exponential function in $[\frac{1}{2},1)$, that correctly rounding the double-precision elementary functions is an achievable goal. Moreover, if all the values of m have the same order of magnitude as the value we obtained for the exponential function (this is likely to be true), always computing correctly rounded functions will not be too expensive. As said in the introduction, correct rounding will facilitate the preservation of useful mathematical properties. And yet, to be honest, we must say that there are a few examples for which correct rounding may *prevent* from preserving a useful property. Let us consider the following example [17]. Assume that we use an IEEE-754 single-precision arithmetic. The machine number which is closest to $\arctan(2^{30})$ is

$$\frac{13,176,795}{8,388,608} = 1.57079637050628662109375 > \frac{\pi}{2}.$$

Therefore, if the arctangent function is correctly rounded in round-to-nearest mode, we get an arctangent larger[5] than $\pi/2$. A consequence of this is that in our system,

$$\tan\left(\arctan\left(2^{30}\right)\right) = -2.2877\cdots\times 10^{7}.$$

In this case, the *range preservation property* "$|\arctan x|$ is less than $\pi/2$ for any x" is not satisfied due to the use of correct rounding in round-to-nearest mode. This is a very peculiar case. One can easily show the following property

Theorem 1 *If a function f is such that:*

- *$f(x)$ is defined for any $x\in[a,b]$,*
- *$c=\min_{x\in[a,b]} f(x)$ is a machine number,*
- *$d=\max_{x\in[a,b]} f(x)$ is a machine number,*

then if $f(x)$ is computed with correct rounding[6] for a machine number $x\in[a,b]$, the computed value will belong to $[c,d]$.

[5] But *equal* to the machine representation of $\pi/2$.

[6] Assuming this is possible

This property shows that the problem that occurred in the previous example will not appear frequently with the usual functions. For example, if they are correctly rounded (in any rounding mode), sines, cosines and hyperbolic tangents will always be between -1 and $+1$.

Our programs were written for Sparc-based machines, but a small portion of the code only is written in assembly code, so that getting programs for other machines would be fairly easy. Of course, it is very unlikely that somebody will be able to perform exhaustive tests for quadruple-precision in the near future, but all the results of our experiments shows that the estimates obtained from the probabilistic hypotheses are very good, so that adding a few more digits to $2n + log_2(n_e)$ for the sake of safety will most likely ensure correct rounding (it is important to notice that if correct rounding is impossible for one argument, we can be aware of that, so a flag can be raised). Therefore we really think that in the next ten years, libraries and / or circuits providing correctly rounded double-precision elementary functions will be available. Now, it is time to think about what should appear in a floating-point standard including the elementary functions. Among the various issues that should be discussed, one can cite:

- should we provide correctly rounded results in all the range of the function being evaluated (that is, for all machine numbers belonging to the domain where the function is mathematically defined), or in a somewhat limited range only ? Although we would prefer the first solution, it might lead to more complex (and therefore time-consuming) calculations when computing trigonometric functions of huge arguments;

- even if we provide correctly rounded results, should we allow a "faster mode", for which faithful rounding [6] only is provided ? Faithful rounding means that the returned result is always one of the two machine numbers that are nearest to the exact result. This is not a rounding mode in the usual sense (for instance, it is not monotonic). Should we allow roundings that are intermediate between faithful rounding and correct rounding, for instance, should we allow instead of rounding to the nearest, results that are within $1/2\text{ulp} + \epsilon$ from the exact result, for some very small ϵ?

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.

[3] G. A. Baker. *Essentials of Padé approximants.* Academic Press, New York, 1975.

[4] V. Berthé. Three distance theorems and combinatorics on words. Technical report, Institut de Mathématiques de Luminy, Marseille, France, 1997.

[5] R. P. Brent. Fast multiple precision evaluation of elementary functions. *Journal of the ACM*, 23:242–251, 1976.

[6] M. Daumas and D. W. Matula. Rounding of floating-point intervals. In *Proceedings of SCAN-93*, Wien, Austria, June 1993.

[7] M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3):162–175, March 1995.

[8] B. DeLugish. *A class of algorithms for automatic evaluation of functions and computations in a digital computer.* PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, 1970.

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

[10] 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.

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

[12] 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.

[13] W. Kahan. Lecture notes on the status of IEEE-754. Postscript file accessible electronically through the Internet at the address http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps, 1996.

[14] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[15] V. Lefèvre. An algorithm that computes a lower bound on the distance between a segment and z^2. Research report RR97-18, LIP, Ecole Normale Supérieure de Lyon, 1997. Available at `http://www.ens-lyon.fr/LIP/research_reports.us.html`.

[16] J. M. Muller and A. Tisserand. Towards exact rounding of the elementary functions. In Alefeld, Frommer, and Lang, editors, *Scientific Computing and Validated Numerics (Proceedings of SCAN'95)*, Wuppertal, Germany, 1996. Akademie Verlag.

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

[18] Y. V. Nesterenko and M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). *Mat. Zapiski*, 2:23–42, 1996.

[19] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[20] 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.

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

[22] R. A. Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995.

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

[24] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.