Towards Correctly Rounded Transcendentals

Vincent Lefèvre and Jean-Michel Muller and Arnaud Tisserand
CNRS-Laboratoire LIP, École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07
FRANCE
{Vincent.Lefevre,Jean-Michel.Muller,Arnaud.Tisserand}@ens-lyon.fr

Abstract

The Table Maker's Dilemma is the problem of always getting exactly rounded results when computing the elementary functions. After a brief presentation of this problem, we present new developments that 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.

1. Introduction

The IEEE-754 standard requires that the results of the arithmetic operations should always be exactly rounded. That is, once a rounding mode is chosen, 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.

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, and some identities.

We want to implement a function f (f being sine, cosine, exponential, 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}. This can be achieved with the presently known methods, using polynomial or rational approximations or Cordic-like algorithms, provided that a careful range reduction is performed [13, 16, 4]. 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 floating-point exactly rounded result from the mantissa y of an approximation of $f(x)$, with error $\pm 2^{-m}$. One can easily see that this is not possible if y has the form:

- in rounding to the nearest mode,

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

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) [9]. 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.

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 exactly rounded results will become possible.

In 1882, Lindemann showed that the exponential of a nonzero (possibly complex) algebraic number is not algebraic [2]. 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 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. 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 [14, 15] proposed algorithms for producing exactly rounded results for the functions $1/x$, square root, 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. One would like to extrapolate those figures and find $m \approx 2n$.

In [11], 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-mantissa bit normalized floating-point input numbers, assuming that n_e different exponents are considered, m is close to $2n + \log_2(n_e)$ with very high probability. Similar probabilistic studies have been done by Dunham [6], and by Gal and Bachelis [8].

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, such as the Nesterenko-Waldschmidt theorem [12], given below that can be applied to find upper bounds on m.

If p/q is a rational number, with $q > 0$ and $\gcd(p, q) = 1$, let us define $H(p/q)$ as $\max\{|p|, q\}$.

Theorem 1 (Yu. Nesterenko and M. Waldschmidt (1995)) *Let α, β be rational numbers, with $\beta \neq 0$. Let A, B and E be positive, real numbers with $E \geq e$ satisfying*

$$A \geq \max(H(\alpha), e), \quad B \geq H(\beta).$$

Then

$$|e^\beta - \alpha| \geq \exp\Big(-211 \times (\log B + \log\log A + 2\log(E|\beta|_+) + 10) \times (\log A + 2E|\beta| + 6\log E) \times (3.3\log(2) + \log E) \times (\log E)^{-2}\Big)$$

where $|\beta|_+ = \max(1, |\beta|)$.

In [11], this theorem was used to show that getting exactly 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 [3] for the functions and Zuras' algorithms [18] 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

2.1. Introduction

An approach to ensure the exact rounding of an elementary function f at a reasonable cost is to search, by means of exhaustive tests, for the arguments x for which the *Table Maker's Dilemma* occurs if $f(x)$ is approximated with error 2^{-m_0}. Here, m_0 is chosen (using the probabilistic results) so that there are a few such arguments only. For these arguments, $f(x)$ could be stored in tables or computed with a more accurate algorithm. By the way, these elements could be used to design algorithms for computing f by the means of Gal's *Accurate Tables Method* [7].

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}), 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 would prefer to *always* give exactly rounded results).

A similar work for the single-precision floating-point numbers have already been done by various authors [14, 11]. The following study has been done for the exponential with double-precision arguments in the interval $[\frac{1}{2}, 1)$, but will be extended 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; for the time being, we only wanted to show that an exhaustive search is possible for double-precision numbers.

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 two years of computation on a 71 MHz reference machine: any cycle is important!

2.2. 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.

To get fast tests, we apply a two-step strategy similar to Ziv's [17]. The first step 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 their exponential 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.

2.3. First Step

2.3.1. Chosen Method

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, 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 [10]. 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}.

We now give the values that have been chosen: $r = 16$, $M = 86$ and $s = 88$. They have been determined from an error analysis (giving relations between r, M and s) and hardware parameters (register width and disk size).

2.3.2. 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.

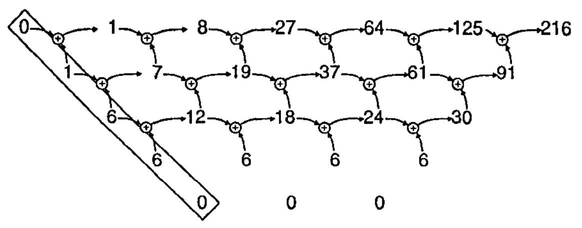
These computations can be performed sequentially on one machine: they only need several minutes.

2.3.3. Computing and Testing the e^{t+x}'s

The computation of the e^{t+x}'s are performed with the finite difference method, which allows to calculate consecutive values of a polynomial of degree d with only d additions for each value. The exponential e^x is approximated by $1 + x + \frac{1}{2}x^2$. Let us take a simple example: $P(x) = x^3$.

One first evaluates $P(0)$, $P(1)$, ..., $P(d)$ (in the example, $d = 3$) with any method. Then one calculates the finite differences: below two consecutive elements x and y, one writes $y - x$. Then one can successively calculate the other values $P(d+1)$, $P(d+2)$, etc... with additions, as shown on the figure.

In fact, the polynomial is not given by its first values $P(0), P(1), \ldots, P(d)$, but by the elements on the left in each



line (surrounded on the figure), which are the coefficients of the polynomial in the base

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

as this base is more suited to our computations.

The test of the bits cannot be performed with only one instruction on a Sparc processor. It 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 tests take 5 cycles per argument in average (two 64-bit additions and one 32-bit comparison), the time required by the other computations being negligible.

2.3.4. Faster Tests

The algorithm given above has been used during the summer of 1996. We present the results below. Before this, let us examine how it could be improved.

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 reciprocal are equivalent, 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. We are finalizing an algorithm that would require $O(\log N)$ time to test the N arguments in the subinterval. With such an algorithm, we can save a factor almost 2 by doubling the number of arguments, for instance. Thus it is better to perform computations with a lower precision in order to increase the lengths of the intervals in which the function can be approximated by a degree-1 polynomial. If the test fails, the interval will be split in subintervals and another test will be performed in each subinterval with a higher precision.

2.3.5. 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 our network of 120 workstations, which often have a null 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. NFS access).

2.4. 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 [5] 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).

2.5. Results

We tested the exponential function with double-precision arguments in the interval $[\frac{1}{2}, 1)$.

In 1996, we used 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.

In January 1997, we performed the first step again on a few machines with a new algorithm based on degree-1 polynomials as explained in section 2.3.4. We reached an average speed-up of 150: 30 arguments tested per cycle in average.

Among all the 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 2 097 152. This shows that the probabilistic estimate was excellent.

For each double-precision number x, we define an integer k such that the mantissa of $\exp x$ 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, $k \geq k_0$ 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 $k = k_0$;
- the actual number of arguments for which $k \geq k_0$;
- the estimated (with the probabilistic hypotheses) number of arguments for which $k \geq k_0$: $2^{52} \times 2^{2-k_0}$, i.e. 2^{54-k_0}.

k_0	$k = k_0$	$k \geq k_0$	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.

The exception for $k = 55$ is $x = 0.11010110011001$ $111101111100100011010110100111011110000$, with

$$\begin{aligned}\exp x \quad = \quad & 10.0100111110000101110010010111110 \\ & 0000111101110011100000\,0\,1^{54}\,01\ldots.\end{aligned}$$

Therefore the value of m for the exponential function in $[\frac{1}{2}, 1)$ in double-precision is $109 = 53 + 55 + 1$.

3. Conclusion

We have shown, using as an example the case of the exponential function in $[\frac{1}{2}, 1)$, that exactly 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 exactly rounded functions will not be too expensive. To obtain the right value of m for other functions and domains, we need some help: we would like to run our programs on several networks of workstations. 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 our experiment shows that the estimates obtained from the probabilistic hypotheses are quite 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 exactly rounded double-precision elementary functions will be available... if you help!

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, Mar. 1986.

[2] A. Baker. *Transcendental Number Theory*. Cambridge University Press, 1975.

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

[4] 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, Mar. 1995.

[5] B. de Lugish. *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.

[6] C. B. Dunham. Feasibility of "perfect" function evaluation. *ACM Sigum Newsletter*, 25(4):25–26, Oct. 1990.

[7] S. Gal. Computing elementary functions: a new approach for achieving high accuracy and good performance. In Springer-Verlag, editor, *Accurate Scientific Computations. Lecture Notes in Computer Science, No 253*, volume 235, pages 1–16, 1986.

[8] 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, Mar. 1991.

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

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

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

[12] 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.

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

[14] M. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In M. I. E.E. Swartzlander and G. Jullien, editors, *11th Symposium on Computer Artithmetic*, pages 138–145, Los Alamitos, CA, June 1993. IEEE Computer Society Press.

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

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

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

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