Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit

ABRAHAM ZIV
IBM Israel

Methods for fast and accurate evaluation of the elementary mathematical functions (sin, cos, exp, log, etc.) are needed in order to program elementary function routines. Such routines are widely used in every floating-point computation environment. This is true for small scientific calculators as well as for the largest existing supercomputers. The art of performing this type of evaluation has already reached a point where fast computation is possible with accuracy that is very close to the theoretical limit, namely, the accuracy reached by correctly rounding to nearest the exact value of the function evaluated to fit into the data format reserved for the output. The accuracy of the results produced by these routines is quite satisfactory. There is one important aspect, though, in which they are not sufficiently perfect: Any changes, even slight, in the algorithm may produce changes in the output. For this reason it is virtually impossible to achieve full compatibility among different routines with the known fast evaluation techniques. Such compatibility is valuable for several reasons (e g., portability of software, standardization).

In this paper a general methodology is suggested that is expected to produce the "ultimate" type of elementary function routines: fast routines that produce results which are always identical to the machine numbers, obtained by correctly rounding the exact values of the function evaluated. The rounding rule need not be "round to nearest " Other rounding rules can be implemented also.

Actually, the method is expected to work for all transcendental elementary functions, for all possible floating point data formats, and for all reasonable rounding rules.

With this type of routine, full compatibility is achievable in a most natural way.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*numerical algorithms*; G.1.2 [**Numerical Analysis**]: Approximation—*elementary function approximation*; G 4 [**Mathematics of Computing**]: Mathematical Software—*efficiency*

General Terms: Algorithms, Standardization

Additional Key Words and Phrases: Compatibility, correct rounding, mathematical library

1. INTRODUCTION

The art of producing elementary mathematical function routines (sin, cos, exp, log, etc.), has already reached a point where routines can be produced which are very fast and are accurate almost to the last bit (see Gal [6] and Gal and Bachelis [7]). There is a problem, though, if one wants to achieve

Author's address: IBM Israel, Science and Technology, Technion City, Haifa, Israel.

compatibility to the last bit, between routines that compute the same function by different algorithms. Usually, even a slight change in the algorithm changes the values of one or more bits of the output for some of the possible inputs. In the past the differences in the outputs of different algorithms delayed improvement of old versions of commercial elementary function routines because of fear of the consequences of incompatibility of new versions with the old ones.

Compatibility to the last bit of one algorithm with another would be easy to achieve had there existed a method to control the value of the last bits of the output of an elementary function routine in such a way that a simple rounding rule, independent of the details of the algorithm, could be agreed upon and followed exactly by a fast routine. All versions of routines for the same function, which follow the same rounding rule, will produce exactly the same outputs.

A closely related subject is that of defining an industry standard for the outputs of elementary function routines. The need for such a standard is discussed by Black et al. [1].

No doubt sin, cos, log, etc., could fit quite naturally in the IEEE standards [10, 11], alongside the four arithmetic operations and the square root, had there existed a practical fast method to evaluate these functions with the same accuracy.

In what follows the term "ultimate" will be used to describe a routine whose output is controlled to the last bit to conform to some prespecified simple rounding rule. The four rounding rules mentioned in the IEEE standards [10, 11] are examples of what is meant by a "simple rounding rule." Fast ultimate routines seem to provide the best possible solution to the problems mentioned above. Such routines have several advantages: full compatibility is achievable, mathematics can be built around them (see computer arithmetic analog in Kulisch and Miranker [12]), they can be applied in interval arithmetic software, etc.

Some fast, commercial, ultimate elementary function routines have already existed for some time (see, e.g., the routines EXP, SQRT, and DSQRT of IBM's Elementary Math Library [9]). The problem with the methodologies which were used to produce them is that they are limited and can be applied only to some of the routines in a complete elementary function routines package.

Gal's accurate tables method [6, 7] is quite general and enables one to produce fast routines which are almost ultimate in the sense that, if applied many times, only a small fraction of their output is incorrectly rounded. In fact, Gal claims that about 99.8 percent of the output of his routines equals the correctly rounded value to the last bit (the percentage varies slightly from routine to routine).

In principle it is not difficult to produce a correctly rounded value of a real number, provided it is known how to compute it to an arbitrary precision. One must only compute the number to a precision somewhat higher than the target precision and then round the result. Sometimes, however, the approximation obtained is too close to a value where the direction of rounding

changes. For example, if the combination of bits of the approximation, beyond the required precision, is either 10000 . . . or 01111 . . . and one wants to compute the number obtained by rounding the exact value to nearest, there might remain a doubt as to the correct direction of rounding. One arrives at a basic problem to which no solution is yet known: Given input and output data formats for an elementary mathematical function routine, what precision should the routine compute in the function values, before they are rounded to the output format, so as to ensure that the routine is ultimate. The lack of a satisfactory general solution to this basic problem is perhaps the main reason for the delay in the appearance of complete ultimate routine packages.

Gal and Bachelis tried a probabilistic approach to the problem. They argued [7] that, for an output data format equivalent to the IEEE standard's "double" (see [10]), if the number of correct bits in the approximation is 137 or more, then the rounded result is most likely to be ultimate. Unfortunately nobody knows how to prove such a thing rigorously, and the task of finding a proof seems to be difficult.

If the input data format is not too long, 4-bytes long, say, then the problem can be solved by an exhaustive testing of all of the possible inputs to see what is the worst case that needs the highest precision computations. However, if the input data format is longer, 8-bytes long say, (e.g., IBM S/370 long precision), such a testing requires too much CPU time, and is therefore impractical to perform.

In addition to the question of what precision an ultimate routine should work in, internally, there is the problem of how to make the routine fast. Higher precision computations usually require more CPU time.

The abovementioned IBM ultimate EXP routine performs the computation in two stages. The first stage uses a fast "short" precision EXP algorithm, which works in Gal's accurate tables method. In this method the last arithmetic operation of the computation is usually an addition: $f_t + b$. f_t is an accurate table value, which is specially chosen to be more accurate than the number of digits in a machine number usually permitted, and b is few orders of magnitude smaller then f_t, with only a few of its last bits being inaccurate. As a result of this special situation the exact sum $f_t + b$ has significantly more accurate bits than a machine number can hold. So rounding the exact sum, the result usually equals, exactly, the correctly rounded value of the exact exponent.

Only in the small fraction of "dangerous" cases, where the configuration of bits of $f_t + b$ beyond the "short" precision is close to either 10000, . . . or 01111, . . . , is a second stage activated by the routine, after a test is made first, to identify dangerous configurations. The second stage uses a "long" precision algorithm whose result is rounded to the target short precision. The fact that it is sufficiently precise to ensure that the routine is ultimate was proved by an exhaustive testing. Since a dangerous configuration is rare, the fact that the computation is slower in cases where it is found has only small effect on the average CPU time requirement of the subroutine.

This discussion makes it clear that there are still some difficulties in producing complete packages of ultimate elementary function routines. It is

the purpose of this paper to suggest a few simple ideas that solve the technical problems that remain.

2. POSSIBLE MODELS OF FAST ULTIMATE ELEMENTARY FUNCTIONS ROUTINES

The IBM short precision EXP routine, described in the introduction, seems to provide a general two-stage model for fast ultimate routines. However, as already mentioned, there is no known sufficiently general method to prove that some accuracy, chosen for its second stage, is sufficient in order to ensure that the routine is really ultimate.

The first idea that is considered here is increasing, without a limit, the number of stages of the subroutine. According to such a model an algorithm of an ultimate routine to evaluate $f(x)$ would look as follows[1]:

1. Set $i = 1$.
2. Invoke $PROC_i$.
3. If $|Y_i - J(Y_i)| > \varepsilon_i$, set $Y = round(Y_i)$ and return.
   Note: If $|Y_i - y| < \varepsilon_i$ and $|Y_i - J(Y_i)| > \varepsilon_i$, then $round(Y_i) = round(y)$.
   Else, set $i = i + 1$ and go back to Step 2.

Such an algorithm produces $round(y)$ exactly, to the last bit in finite time, provided $J(y) \neq y \neq 0$.

Since almost all of the elementary functions are transcendental and all of the possible input machine numbers are rational, y is usually irrational. For all of the principal rounding rules, $J(.)$ assumes only rational values. Hence, with the exception of very few cases, which can be easily taken care of, the condition $J(y) \neq y \neq 0$ is satisfied. Therefore, this type of an algorithm is potentially a good general model for an ultimate elementary mathematical function routine.

This scheme permits an unbounded number of stages and, as a result, an unbounded number of bits in the numerical data formats used by the higher *PROC*s. So, in principle, although it is extremely unlikely (see Section 4), the memory requirements might grow to a very large size. A routine which is programmed by this scheme must include, therefore, a mechanism to deal with growth of the required memory size, and this complicates matters. Unboundedness also prevents the use of some convenient algorithms. For instance, one cannot use stored values of helpful constants that can be exactly expressed only by infinitely many bits. In order to simplify the implementation process, the following is suggested: permit only a bounded number of stages. If the last stage is reached and a dangerous configuration is still encountered, signal an exception.

In Section 4, a probabilistic model is used in order to predict the number of inputs that cause an exception to be signaled by an ultimate routine of this type. The resulting figures indicate that exception-causing input is practically impossible. The built-in exception-signaling mechanism is meant only to be a safeguard against the improbable possibility that such an input does

[1] Notation and its meaning can be found at the end of this paper, in Section 6.

exist for some exceptional routine, which, extremely rarely, might produce an unnoticed output whose last bit is erroneous.

The set of exception-causing inputs of a given routine of this type is most likely to be empty. In a bad case it might include at most a small number of machine numbers. Anyway, an exception is expected to be so extremely rare that it is worthwhile, upon its occurrence, to stop the program and take measures in order to find the input that gave rise to it. Knowledge of such an input can be used to give it a special treatment in later versions of the routine so that it will no longer cause an exception. (The manufacturer may offer a price—that he will most likely never have to pay—to the first discoverer of every exception-causing input.

It should be emphasized that the similarity in appearance between the random nature of the rare occurrence of the exception and the random nature of the rare occurrence of an incorrectly rounded result in Gal's type routine is misleading. In Gal's approach there is no indication as to which of the outputs is correctly rounded and which is not. There is only a general knowledge that the percentage of incorrectly rounded results is small. In the exception approach, on the other hand, outputs, which are given with no exception signaled, are guaranteed to be correctly rounded. An exception is usually impossible, and even with routines for which it is theoretically possible, it is extremely rare and can easily be detected.

The discussion above raises a few questions:

(1) Is there a simple way to program $PROC_i$ for large values of i?
(2) Will the resulting routines really be as fast as one would like them to be?
(3) What about other factors of importance: size of storage needed, number of lines of program code needed, technical difficulties in imbedding such routines in existing software (e.g., compilers), etc.

The first two questions are discussed in Section 3. The last one will probably have to wait until a sufficiently large number of routines of this type exist (it does not seem to pose serious problems, though).

3. IMPLEMENTATION PROBLEMS AND THEIR SOLUTIONS

Two issues are discussed in this section:

(1) making an ultimate subroutine fast;
(2) implementation of the infinite sequence of algorithms $PROC_i$ by a finite code of a reasonable size.

Assume that the subroutine is invoked many times, the way it is done in benchmarks. The average CPU time required per one value of x is

$$\bar{t} = \sum_{i=1}^{\infty} p_i \sum_{j=1}^{i} t_j = \sum_{j=1}^{\infty} t_j \sum_{i=j}^{\infty} p_i = \sum_{j=1}^{\infty} \left(1 - \sum_{i=1}^{j-1} p_i\right) \times t_j$$
$$= t_1 + (1 - p_1)t_2 + (1 - p_1 - p_2)t_3 + \cdots .$$

If one is able to produce a situation in which each of the terms of the last series is much smaller than its predecessor, one will have approximately $\bar{t} \simeq t_1$. The practical meaning of this observation is that $PROC_1$ must be as fast as possible and that p_1 must be as close as possible to 1. If one succeeds in achieving a value of p_1 that is sufficiently close to 1, one wins great freedom in the programming of $PROC_i$, $i = 2, 3, \ldots$: Up to quite a high limit, their CPU time requirements have almost no effect on the value of $\bar{t}$. The value of p_1 is close to 1 if ε_1 is small. This is because the smaller ε_1 is, the larger the chances are that the condition $| Y_1 - J(Y_1) | > \varepsilon_1$ is satisfied. And this is exactly the condition for $i(x)$ to equal 1.

In fact, if one achieves a value close to 1 for p_1, the mechanism of infinitely many PROCs becomes only a safeguard against the rare possibility that the algorithm will not terminate at the end of the execution of $PROC_1$. Because this mechanism is activated very rarely, the size of its CPU time requirements is of minor importance.

One might fear that, since $\bar{t}$ is the sum of an infinite series, its value might be infinite even if the sum of the few first terms is not large. This is not the case, though. The reason is that the sequence t_j rises much slower than the sequence $(1 - \sum_{i=1}^{j-1} p_i)^{-1}$. The complexity of computation of an elementary mathematical function with unbounded precision was investigated by Brent [4]. From his results it is clear that if k_j, the number of digits of the mantissa in the floating-point arithmetic used to compute the function, rises linearly with j, then t_j rises in a polynomial rate (somewhat faster then k_j^2 if the arithmetic is programmed in the simplest possible way and even slower if an effort is made to make the arithmetic faster).

On the other hand, probabilistic arguments of the type discussed in Section 4 indicate that $(1 - \sum_{i=1}^{j-1} p_i)^{-1}$ rises at an exponential rate (somewhat slower than r^{k_j}, where r is the radix of the arithmetic).

As a result, the series for $\bar{t}$ behaves much like a geometrical progression with a very small ratio. Actually the main difficulty in achieving low, average CPU time requirements lies in the production of a good $PROC_1$.

Since the CPU time requirements of $PROC_1$ are crucial for the performance of the complete algorithm, it must be implemented in an especially fast method which is principally different from the one in which $PROC_i$, with $i = 2, 3, \ldots,$ is implemented. As was previously described, Gal's method seems to be appropriate for this purpose. A preliminary, careful, and rigorous analytical error analysis of $PROC_1$, which produces an appropriate small value for ε_1, is essential to the success of the implementation (see [13, 15, 16, 18] for techniques of rigorous a priori error analysis).

$PROC_1$ may be considerably simplified if an appropriate hardware support exists. Thus, if a fast floating-point arithmetic, somewhat more precise than the target precision (15 additional bits, say), is available, then the task of designing $PROC_1$ is made much easier. The use of a carefully designed algorithm like Gal's and the relatively large tables it uses are then not necessary. One can even imagine a special elementary function coprocessor which uses somewhat higher precision arithmetic internally to produce a longer precision elementary function output for which an a priori error bound

is known. This will make the implementation of $PROC_1$ almost trivial. It seems that such hardware support is likely in the near future to exist on, at most, a limited number of systems, and even then not for all precisions of interest. Appropriate hardware support, though, seems to provide a promising solution for the future.

$PROC_i$, $i = 2, 3, \ldots$, can be implemented in the following way: A small package of programmed floating-point arithmetic is made available in which the radix as well as the number of digits of the mantissa are easily controllable variables. It should be such that bounds for the roundoff errors produced by its operators are known so that numerical algorithms which use it can be rigorously analyzed a priori for bounds of the accumulated roundoff errors. A set of routines for the basic arithmetic operations and a small number of auxiliary routines, such as format conversion and size comparison, are sufficient.

The published literature includes many examples of packages of multiple precision subroutines. Brent [2, 3], Hill [8], Wyatt et al. [17] are just a few. Such packages usually include many routines, only a small number of which are necessary here. The papers mentioned, however, as well as others, pay little attention to rigorous error bounds, which are a necessity in our case.

Applying the package, one can program $PROC_i$, $i = 2, 3, \ldots$, using some algorithm with potentially unbounded precision (Taylor expansion-based, for instance). Brent [4] describes a number of such algorithms. Unfortunately he does not discuss roundoff errors. Perhaps a better example of the necessary type of an algorithm is the one discussed by Clenshaw and Olver [5] (see also [14]). Clenshaw and Olver go into a detailed theoretical discussion of the tuning of their algorithm to make it optimally fast. For our purposes such a tuning is necessary only for the first few *PROC*s, and this can be efficiently achieved by inspection and experimentation. A significant theoretical effort is unnecessary.

It may happen, depending on the details of the implementation, that a $PROC_2$ which was programmed in this way will not be sufficiently fast to make its contribution small enough for the average CPU time requirements of the routine. For this reason an effort to find ways to reduce the contribution of $PROC_2$ to the average CPU time requirements is worthwhile.

An obvious method would be to program $PROC_2$ by an ordinary, fast algorithm of a precision sufficiently higher than the precision of the output of the routine. For example, a double-precision EXP ultimate routine may use an ordinary fast, extended precision EXP algorithm for $PROC_2$. Such a solution, however, is problematic in cases where the precision of the output of the ultimate routine is the highest for which floating-point hardware support exists.

Another way would be to reduce the percentage of cases in which the activation of $PROC_2$ is needed. $PROC_1$, from the example of an EXP routine described in Section 5.1, is designed by a new method that makes it possible to reduce the percentage of cases in which $PROC_2$ is activated to a significantly lower level than Gal's method.

To summarize the discussion in the section: It is possible to implement

ultimate elementary function routine packages which are almost as fast as routines designed by Gal's method (and Gal's is one of the fastest methods known).

4. ESTIMATIONS BASED ON A PROBABILISTIC MODEL

A basic assumption on which Gal [6] and Gal and Bachelis [7] rely is that bits of low significance of value of elementary functions behave as if they were independent, binary, random variables, taking the values 1 and 0 with equal probability. In this section a similar probabilistic model is used in order to draw conclusions regarding the probabilities of some events of interest related to the performance of routines of the type discussed here.

All of the estimates below follow from the relation:

$$\text{Probability of}\{|Y_i - J(Y_i)| \le \varepsilon_i\} = 2 \times \varepsilon_i / ulp(Y_i).$$

This relation is an immediate consequence of the *Uniformity assumption:* The ratio $(Y_i - J(Y_i))/ulp(Y_i)$ is a random variable which is uniformly distributed in the interval $[-\frac{1}{2}, \frac{1}{2}]$.

One should realize that Y_i is a discrete variable. So this uniformity assumption cannot possibly be exactly true. What is claimed, though, is that it is approximately true to the extent that the abovementioned probability estimation is sufficiently realistic for our purposes.

This uniformity assumption is not equivalent to Gal's assumption, which is more closely related to the slightly different assumption, as follows:

The ratio $(y - J(y))/ulp(y)$ is a random variable which is uniformly distributed in the interval $[-\frac{1}{2}, \frac{1}{2}]$.

Let us now consider, for example, an elementary function routine with its output being of data format "long" of an IBM S/370, like the EXP routine described below. If Y_2 is accurate to 28 hexadecimal digits (like the IBM S/370 extended precision), then the probability of $PROC_3$ being activated does not exceed

$$2 \times \varepsilon_2 / ulp(Y_2) = 2 \times 16^{e(Y_2)-28}/16^{e(Y_2)-14} \simeq 3 \times 10^{-17}$$

or, equivalently, the expected fraction of cases in which $PROC_3$ is activated is less then 3×10^{-15} percent.

Suppose, now, that the routine signals an exception in case $PROC_{\text{last}}$ does not terminate the computation and assume that the precision of Y_{last} is of 56 hexadecimal digits. The probability of an exception being signaled does not exceed

$$2 \times 16^{e(Y_{\text{last}})-56}/16^{e(Y_{\text{last}})-14} = 2 \times 16^{-42} \simeq 5 \times 10^{-51}.$$

In order to make the meaning of this extremely small probability more intuitively clear, assume that the routine is run nonstop for n years on a computer that is dedicated for this purpose and that invokes the routine repeatedly in 1 μs (10^{-6} s) time intervals. The probability that no exception

will occur, even once, during these n years is

$$\begin{aligned}&\left(1 - 2 \times 16^{-42}\right)^{365 \times 24 \times 60 \times 60 \times 10^6 \times n}\\ &\quad > 1 - 2 \times 16^{-42} \times 365 \times 24 \times 60 \times 60 \times 10^6 \times n\\ &\quad \simeq 1 - 1.7 \times 10^{-37} \times n.\end{aligned}$$

The total number of normalized floating-point "long" IBM S/370 numbers is $256 \times 15 \times 16^{13} \simeq 1.7 \times 10^{19}$. Hence the expected number of inputs which cause an exception is no more than $1.7 \times 10^{19} \times 5 \times 10^{-51} = 8.5 \times 10^{-32}$. This means, of course, that the set of exception-causing inputs is most likely to be empty, or, in bad cases, will include very few possible input numbers.

The use made in this section of the uniformity assumption is quite extreme, so the conclusions should be considered with care. There is no doubt, however, that they give a realistic indication of the situation.

Note. The validity of the uniformity assumption must be carefully examined near points where the mantissa of y is extremely insensitive to small changes in the mantissa of x (e.g., near $x = 0$, for the exponential function).

5. EXAMPLE OF AN ULTIMATE EXP ROUTINE

The description below is not meant to give full detailed structure of a working algorithm. Details are given only as long as they are necessary in order to clarify the main points.

5.1 PROC_1 for exp(x)

The exp(.) routine described below is a modification of an exp(.) routine designed by Gal and Bachelis [7], to which the test, mentioned in Step 3 of the algorithm (see Section 2), is added. The tables used by the algorithm are constructed in a new way which has two advantages: Their preparation is simple and straightforward, and they are so constructed that two consecutive range reductions (and possibly more) can be performed with them. This enables one to achieve a higher degree of range reduction using tables similar in size to Gal's. Such a higher degree of range reduction makes it easier to achieve a value of p_1 which is closer to 1 (see discussion in Section 3). The method by which the routine works is referred to as the repeated reduction method.

The algorithm is designed for IBM S/370 architecture. The data formats of both the input and output are long, that is, both are 8-byte, floating-point numbers with radix 16, a fraction of 14 hexadecimal digits, and an exponent between -64 and 63. The routine's rounding rule is "round toward 0."

The first stage of the algorithm includes three consecutive range reductions (as opposed to two in Gal and Bachelis), each of which significantly reduces the possible size of $|x|$:

1. $x^1 = x - n \ln 2$
2. $x^2 = x^1 - u_i^1$
3. $x^3 = x^2 - u_j^2$.

The integer n is so chosen that $-\frac{1}{2}\ln 2 < x^1 < \frac{1}{2}\ln 2$. During the execution of the algorithm, u_i^1, u_j^2 are extracted from fixed tables that are kept in memory: $u_i^1 = v_i^1 + w_i^1$, $u_j^2 = v_j^2 + w_j^2$ (the reason for expressing u_i^1, u_j^2 in two pieces each is the need for increased accuracy of their values).

The table $\{u_i^1\}$ "covers" the interval $(-\frac{1}{2}\ln 2, \frac{1}{2}\ln 2)$ with more or less equally spaced numbers: $u_i^1 \simeq i\Delta^1$.

Given x^1, i is chosen so that $-\frac{1}{2}\overline{\Delta}^1 < x^2 < \frac{1}{2}\overline{\Delta}^1$, where $\overline{\Delta}^1$ is slightly larger than Δ^1.

The table $\{u_j^2\}$ covers the interval $(-\frac{1}{2}\overline{\Delta}^1, \frac{1}{2}\overline{\Delta}^1)$ with more or less equally spaced numbers: $u_j^2 \simeq j\Delta^2$.

Given x^2, j is chosen so that $-\frac{1}{2}\overline{\Delta}^2 < x^3 < \frac{1}{2}\overline{\Delta}^2$, where $\overline{\Delta}^2$ is slightly larger than Δ^2.

Together with the four tables $\{v_i^1\}$, $\{w_i^1\}$, $\{v_j^2\}$, $\{w_j^2\}$, two additional tables are kept, $\{y_i^1\}$, $\{y_j^2\}$, where $y_i^1 = \exp(u_i^1)$, $y_j^2 = \exp(u_j^2)$.

The special property of the tables, in which they differ from Gal's, is that the points u_i^1, u_j^2 are so chosen that the table entries y_i^1, y_j^2 have very short mantissas.

To prepare such tables, construct triplets (v, w, y) such that $y = \exp(v + w)$, the value of $v + w$ is approximately known, and y is expressible exactly by a floating-point number whose mantissa is of a prescribed short length. This may be done as follows, starting from the known approximation of $v + w$:

> Compute the exponential of the approximation $v + w$, round the result to the required number of bits, assign the outcome to y, compute $\log y$ to a sufficiently high precision (a good, standard, extended precision routine will do), cut the mantissa of the result into two pieces, and assign the more significant piece to v and the less significant piece to w.

Obviously,

$$\exp(x) = z + z \times (\exp(x^3) - 1),$$

where $z = 2^n \times y_i^1 \times y_j^2$. Since z is the product of floating-point numbers with very short mantissa, it is a machine number and can be computed exactly, with no roundoff errors at all.

The expression $\exp(x^3) - 1$ is computed by an approximation polynomial $P(.)$ which is sufficiently accurate in the interval $(-\frac{1}{2}\overline{\Delta}^2, \frac{1}{2}\overline{\Delta}^2)$.

In addition to the six tables $\{v_i^1\}$, $\{w_i^1\}$, $\{y_i^1\}$, $\{v_j^2\}$, $\{w_j^2\}$, $\{y_j^2\}$, one should prepare and keep in memory six constants: $LN1$, $LN2$, $LN3$, δ^1, δ^2, ε. They satisfy approximately (in the sense of relative error) the following:

$$LN1 \simeq \ln 2, \quad LN2 \simeq LN1 - \ln 2, \quad LN3 \simeq 1/\ln 2, \quad \delta^1 \simeq 1/\Delta^1, \quad \delta^2 \simeq 1/\Delta^2.$$

The last three hexadecimal digits of $LN1$ must be zero in order to ensure that the product $n \times LN1$ is always an exact machine number. ε is an error estimate produced by error analysis of the algorithm. It satisfies

$$|\exp(x) - (z + c)| < \varepsilon \times ulp(Y)$$

where $\exp(x)$ is exact, c is the approximation computed for $z \times (\exp(x^3) - 1)$, $(z + c)$ denotes the exact sum of z and c, and $Y = round(z + c)$.

Other constants computed in advance include the coefficients of the minimax polynomial $P(.)$.

The values x^1, x^2, obtained from x by first and second reductions, are expressed as sums of two variables each: $x^1 = a^1 + b^1$, $x^2 = a^2 + b^2$.

We denote by $INT(A)$ the integer closest to the real number A. Here is a complete scheme of $PROC_1$ for $\exp(x)$:

1. Compute n: $n = INT(x \times LN3)$.
2. Compute $x^1 = a^1 + b^1$: $a^1 = x - n \times LN1$, $b^1 = n \times LN2$.
3. Compute i: $i = INT(a^1 \times \delta^1)$.
4. Load v_i^1, w_i^1, y_i^1 from tables.
5. Compute $x^2 = a^2 + b^2$: $a^2 = a^1 - v_i^1$, $b^2 = b^1 - w_i^1$.
6. Compute j: $j = INT(a^2 \times \delta^2)$.
7. Load v_j^2, w_j^2, y_j^2 from tables.
8. Compute x^3: $x^3 = (a^2 - v_j^2) + (b^2 - w_j^2)$.
9. Compute an approximation $P(x^3)$ to $\exp(x^3) - 1$.
10. Compute z: $z = 2^n \times y_i^1 \times y_j^2$.
11. Compute c: $c = z \times P(x^3)$.
12. Compute Y: $Y = round(z + c)$ ($round(.)$ denotes round toward 0).
13. Check for termination of the algorithm: Let $d_0 = (z + c - Y)/ulp(Y)$, $d = \min\{d_0, 1 - d_0\}$, where, $ulp(Y) = 16^{e(Y)-14}$.
    If $d > \varepsilon$, let Y be the final result and return.
    Else, transfer control to $PROC_2$.

5.1.1 *Remarks Related to Accuracy.* Some of the intermediate results of the algorithm are decomposed into sums of two terms: a major one and a minor one. An example is $x^2 = a^2 + b^2$. The major part of such a number must be computed with absolute accuracy. The computation of the minor part, on the other hand, may involve some roundoff errors.

The following two useful theoretical results specify cases where error-free subtraction is possible within a given set of floating-point numbers. This possibility is realized only if the computer arithmetic provides the exact outcome of an operation whenever this exact outcome is within the set of machine numbers.

The machine arithmetic of IBM S/370, which is relevant to the present example, as well as that of any arithmetic with a strictly less than one *ulp* rounding error rule (e.g., any of the rounding rules of the IEEE Standards [10, 11]) do provide this exact outcome, and therefore the results below are relevant for them.

(1) If α, β are two floating-point numbers of like signs that satisfy $0 < |\alpha| \leq |\beta| \leq |\alpha| + r^{e(\alpha)}$, then $\alpha \ominus \beta = \alpha - \beta$, provided no underflow occurs.

(2) If α, β are two floating-point numbers that satisfy $\frac{1}{2} \leq \alpha/\beta \leq 2$, then $\alpha \ominus \beta = \alpha - \beta$, provided no underflow occurs.

Apparently the first of these results has not been published yet, in this general form. Its proof is simple and will be published elsewhere. The second result follows from the first. Its formulation, as well as a proof for the case of $FP(r, p, clq)$, appears in Sterbenz [15, Sec. 4.3]. Both are true for any

integral radix. They might be useful in performing a few of the steps of the algorithm, including in particular the reduction steps.

5.2 The higher PROCs for exp(x)

The basis is an algorithm of Clenshaw and Olver [5] (see also [14]).

Denote

$$|x| = 2^m \times t, \quad f_0 = 1 + t/1! + t^2/2! + \cdots + t^n/n!.$$

m, n should be chosen to be large enough so that f_0 is a sufficiently good approximation of $\exp(t)$. From f_0, an approximation F to $\exp(x)$ can be produced by recursion:

$$f_{j+1} = f_j \times f_j, \qquad j = 0, 1, \ldots, m-1,$$

With $F =$ either f_m or $1/f_m$, depending on the sign of x.

With a package of programmed floating-point arithmetic at hand, such as described in Section 3, it is possible to implement Clenshaw and Olver's algorithm. First, the free parameters of the algorithm, m, n, and those of the floating-point arithmetic, radix r, and the number of digits in the mantissa, k, must be chosen. Then the implementation of the algorithm is straightforward. In addition, an error estimate ε_i based on a theoretical, rigorous analysis must be prepared and used to perform the final test.

A possible scheme for $PROC_i$, $i = 2, 3, \ldots,$ is as follows.

1. Set $i = 2$.
2. Choose $m, n, r, k, \varepsilon_i$ (depending on the value of i).
3. Compute $t = 2^{-m} \times x$.
4. Compute f_0: $a_n = 1$, $a_{j-1} = 1 + (t \times a_j)/j$, $j = n, n-1, \ldots, 1$, $f_0 = a_0$.
5. Compute f_m: $f_{j+1} = f_j \times f_j$, $j = 0, 1, \ldots, m-1$.
6. Set $Y_i = f_m$.
7. Perform the final test: If $|Y_i - J(Y_i)| > \varepsilon_i$, convert Y_i into a REAL*8 number (round toward 0, if necessary), set Y to this number, and return.
   Else, set $i = i + 1$ and go back to Step 2.

6. NOTATION

Included in this section are definitions and notation used throughout the paper. It is assumed that the real valued function $f(x)$ is to be evaluated, for some given value of x, by an ultimate routine. x may be scalar, vector, real, complex, or anything else, and of any precision.

$e(q)$, $m(q)$	The exponent and mantissa of the normalized floating-point number $q = r^{e(q)} \times m(q)$. $e(q)$ is an integer and $1/r \leq \lvert m(q) \rvert < 1$.
$i(x)$	The index i of the last $PROC$ activated in the process of evaluation of $y = f(x)$.
$J(q)$	The jump point of the step function $round(.)$ which is nearest to q.

$m(q)$	See $e(q)$.
p_i	Probability of the event $i(x) = i$.
$PROC_i (i = 1, 2, \ldots)$	An infinite sequence of procedures, of increasing precisions, to evaluate $f(x)$.
r	Radix of the floating-point data format of the output of the routine.
$round(q)$	The correctly rounded value of q. It is a t-digit, normalized, floating-point number and may be defined according to any of the principal, simple, rounding rules (round to nearest, round toward 0, etc.).
t	The number of digits in base r of the output of the routine.
t_i	CPU time required in order to execute $PROC_i$.
$ulp(q)$	Unit in the last place of q, that is, $ulp(q) = r^{e(q)-t}$.
y	The exact value of $f(x)$.
Y	Output of a routine which is meant to compute y. It is a t-digit, normalized, floating-point number.
Y_i	The approximation to y produced by $PROC_i$.
ε_i	An error bound for Y_i produced by a theoretical error analysis of $PROC_i$. It satisfies $\lvert Y_i - y \rvert < \varepsilon_i$.
$\ominus$	The approximate machine "minus." It is assumed that its operands and its output are all of the same precision.

ACKNOWLEDGMENT

The author wishes to thank Vardy Amdursky, who, for several years, promoted attempts to solve the problem addressed here and made this work possible. His contribution extends to valuable discussions which helped to shape some of the basic ideas (the unbounded model in particular). He is also the one who suggested the term "ultimate" in connection with the round to nearest rule.

REFERENCES

1. Black, Burton, and Miller. The need for an industry standard of accuracy for elementary-function programs. *ACM Trans. Math. Softw. 10*, 4 (Dec. 1984), 361–366
2. Brent, R P. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4*, 1 (1978), 57–70.
3. Brent, R. P. Algorithm 524. MP: A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw., 4*, 1 (1978), 71–81.
4. Brent, R. P. Fast multiple-precision evaluation of elementary functions. *J. ACM 23*, 2 (1976), 242–251.
5. Clenshaw, C. W., and Olver, F. W. J. An unrestricted algorithm for the exponential function. *SIAM J. Numer. Anal., 17*, 2 (Apr. 1980).

6. GAL, S. Computing elementary functions: A new approach for achieving high accuracy and good performance accurate scientific computations. In *Lecture notes in Computer Science*, no. 235, Springer-Verlag, New York 1986, pp. 1-16.
7. GAL, AND BACHELIS. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans. Math. Softw. 17,* 1 (March 1991), 26-45; also Tech. Rep. 88.223, IBM Israel Science and Technology, June 1987.
8. HILL, I. D. Procedures for the basic arithmetical operators in multiple-length working. *Comput. J. 11* (1968), 232-235.
9. IBM CORPORATION. Elementary math library: Programming RPQ (5799-BTB), Program Reference and Operations Manual, 1984.
10. *IEEE Standard for Binary Floating Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.*
11. *IEEE Standard For Radix-Independent Floating Point Arithmetic.* ANSI/IEEE Std 854-1987, 1987.
12. KULISCH, U. W., AND MIRANKER, L. M. *Computer Arithmetic in Theory and Practice.* Academic Press, Orlando, Fla., 1981.
13. OLVER, F. W. J. A new approach to error arithmetic. *SIAM J. Numer. Anal., 15*, 2 (1978), 368-393.
14. OLVER, F. W. J. Unrestricted algorithms for generating elementary functions. *Computing, Supplement* 2. Springer-Verlag, New York, 1980, pp. 131-140.
15. STERBENZ, P. H. *Floating Point Computation.* Prentice-Hall, Englewood Cliffs, N. J., 1974.
16. WILKINSON J. H. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, N. J., 1963.
17. WYATT, J. R., LOZIER, D. W., AND ORSER D. J. A portable extended precision arithmetic package and library with Fortran precompiler. *ACM Trans. Math. Softw. 2*, 3 (1976), 209-231.
18. ZIV, A. Relative distance—An error measure in round-off error analysis. *Math. Comput., 39*, 160 (1982), 563-569.

Received April 1989; revised November, 1989; accepted May 1990