ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 3, pp. 154–173. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © V.V. Kuliamin, 2007, published in Programmirovanie, 2007, Vol. 33, No. 3.

Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers

V. V. Kuliamin

Institute for System Programming, Russian Academy of Sciences,
ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia
e-mail: kuliamin@ispras.ru

Received October 4, 2006

Abstract—Requirements and designing test suites for implementations of mathematical functions in floating point arithmetic in the framework of the IEEE 754 standard are considered. A method based on this standard is proposed for designing requirements for such functions. This method can be used for the standardization of implementations of such functions; this kind of standardization extends IEEE 754. A method for designing test suites for the verification of those requirements is presented. The proposed methods are based on specific properties of the representation of floating point numbers and on some features of the functions under examination.

DOI: 10.1134/S036176880703005X

1. INTRODUCTION

One of the fields of human activities based on the use of complex software is mathematical modeling of complex phenomena. For example, it can be modeling of the evolution of the universe as a whole, of galaxies and stars, of physical processes under extreme conditions, of biochemical, climate, and social processes. In many cases, computer simulation provides important information about such phenomena; however, it is difficult to obtain an independent assessment of the validity of the simulation results, which could make it possible to verify the validity of the software. This fact causes considerable difficulties in the development of reliable systems for mathematical modeling.

The reliability of such systems can be improved using a formal validation of library components and, in particular, of implementations of mathematical functions included in those libraries. The confidence in the components underlying such systems improves the quality of the development and makes it possible to focus on detecting and correcting errors in other components of the software.

Many practical problems concern continuous systems in which the space of states is a real or complex variety. The points of such a variety (the states of the system) can be represented in a certain system of coordinates as sets of real numbers; then, the evolution of the system is described by curves in the variety. The use of computers for modeling such systems almost always assumes that the continuous space of states is represented in a discrete form, which gives rise to additional difficulties in ensuring that the software is correct.

The real numbers are most often represented by binary floating point numbers described by the IEEE 754 [1] (also called IEC 60559 [2]) standards. This standard was introduced in the middle of the 1980s; earlier, there was no common portable format for computer representation of real numbers. The aim of introducing such a standard was the development of portable libraries of computational algorithms so that the same computations performed on various platforms produced the same results. Although many hardware and software manufactures had to considerably revise their products, currently this standard is supported by a vast majority of computation platforms. Gradually, we are leaving behind the variety of architectures that existed in the early 1980s and the strong dependence of high-quality implementations of computational algorithms on a particular platform.

However, the aim has not been completely achieved. IEEE 754 regulates only the machine representation of floating point numbers and the basic operations on them, i.e., addition, subtraction, multiplication, division, comparison, and type conversion. Among the mathematical functions, only the square root is standardized. Therefore, libraries of mathematical functions can be implemented that formally conform to the standard but produce quite different results (see the examples below), which results in nonportability and unreliable operation of the applications that use the functions that are not standardized in IEEE 754. As a result, the users that need accurate computations employ other, nonstandard, libraries. Hence, the desire to standardize the implementations of a great number of mathematical functions that are often used in applications. Standardization always assumes the verification that an implementation conforms to the standard.

In this paper, we present a possible approach to the standardization of implementations of mathematical functions operating on floating point numbers in the

IEEE 754 formats. Since the standard needs a support in the form of test suites for verifying the conformity to the standard, we also propose a method for designing such test suites. This method uses formal requirements for the behavior of mathematical functions and the features of the structure of floating point numbers.

The majority of the ideas underlying the proposed approach to the standardization of implementations of mathematical functions are borrowed from various sources—the standardization group of ISO/IEC 10967 [3–5] and the studies of the team in INRIA (France) performed in the framework of the Arenaire project [6–8]. However, to the author's knowledge, the proposed procedure for determining the requirements for implementations of mathematical functions has never been presented in a unified form. The method for designing test suites is a novel one, although some of its elements can be found in the literature or in implemented projects.

2. PROBLEMS IN COMPUTING MATHEMATICAL FUNCTIONS

First, we consider the problems concerning the correct computation of mathematical functions in floating point arithmetic.

These problems are mainly caused by the discrete nature of the floating point representation of real numbers, which makes possible an efficient manipulation on such numbers on computers but leads to the fact that the exact results of most computations are not representable. The format of floating point numbers and the rules of operating on them are described in IEEE 754 [1] (IEC 60559 [2]) and IEEE 854 [9].

IEEE 754 defines the representation of binary floating point numbers. IEEE 854 extends this definition to decimal floating point numbers. In practice, binary numbers are by far the most widely used. Hence, we will consider only the binary numbers in this paper, although all the reasoning also applies to the decimal numbers.

2.1. Floating Point Numbers

Every binary floating point number has the following structure [1, 9, 10].

• A number is represented by a collection of n bits. The first one is the *sign bit*, the next k bits represent its *exponent* E, and the remaining $(n - k - 1)$ bits represent its *significand* M.

• The sign bit S, the exponent E, and the significand M of the number x determine its value by the rule

$$x = (-1)^S \cdot 2^e \cdot m,$$

where

◦ S is zero for positive numbers and is equal to unity for the negative numbers.

◦ If $0 < E < 2^k - 1$, then $e = E - 2^{(k-1)} + 1$; otherwise, if $E = 0$, then $e = E - 2^{(k-1)} + 2$. The number $b = (2^{(k-1)} - 1)$ is called the *exponent bias*.

◦ If $0 < E < 2^k - 1$, then $m = 1 + M/2^{n-k-1}$. In other words, the binary representation of m is $1.M$; i.e., the integral part of m is 1, and the sequence of digits in the fractional part coincides with the sequence of bits M. If $E = 0$, then $m = M/2^{n-k-1}$. In other words, the binary representation of m is $0.M$. The numbers with the zero exponent are called *denormalized*, and all the other numbers are called *normalized*.

• The greatest possible value of the exponent $E = 2^k - 1$ is reserved for representing the special quantities: the positive and the negative infinities ($+\infty$ and $-\infty$) and the special quantity *NaN* (not a number). *NaN* is used if the result of an operation cannot be correctly represented by a common number or by an infinity (for example, the results of the operations 0/ 0 and $(-\infty) + (+\infty)$).

$+\infty$ has the zero sign bit, the greatest possible exponent, and the zero significand; $-\infty$ differs only in the sign bit.

Any number that has the greatest possible exponent and a nonzero significand is considered as a representation of NaN.

• IEEE 754 and IEEE 854 define several types of floating point numbers. Among them, the most widely used are *single precision, double precision*, and *double-extended precision* numbers.

For single precision numbers, $n = 32$ and $k = 8$. Respectively, the significand uses 23 bits, and the exponent bias is 127.

For double precision numbers, $n = 64$ and $k = 11$. The significand uses 52 bits, and the exponent bias is 1023.

For double-extended precision numbers, no specific values of k and n are fixed in the standards; only the bounds $128 \geq n \geq 80$ and $k \geq 15$ are specified. In Intel 32-bit processors, $n = 80$ and $k = 15$. The significand uses 64 bits, and the exponent bias is 16383.

In addition *quadruple precision* numbers are sometimes used for which $n = 128$ and $k = 15$. In this case, the significand uses 112 bits, and the exponent bias is 16383. These numbers are not specified in the standard but they have a similar structure.

By way of example, consider the representation of the number -17_{10} in single and double precision. Since $-17_{10} = (-1)^1 \cdot 2^{131-127} \cdot 1.0001_2 = (-1)^1 \cdot 2^{1027-1023} \cdot 1.0001^2$, it is represented in single and in double precision as 1 10000011 00010000000000000000000 and 1 10000000011 0001000000000000000000000000000000000000000000000000, respectively.

Below, we will use the hexadecimal rather than the binary notation.

Some extreme values for floating point numbers in various formats are shown in Table 1. For the double-extended precision, the parameters of the 32-bit Intel architecture are used.

Table 1. Extreme values of floating point numbers

	General form	Single precision	Double precision	Double-extended precision	Quadruple precision
The smallest denormalized positive number	$2^{-b-n+k+2}$	2^{-149}	2^{-1074}	2^{-16446}	2^{-16494}
The smallest normalized positive number	2^{-b+1}	2^{-126}	2^{-1022}	2^{-16382}	2^{-16382}
The greatest positive number	$2^{b-n+k+1} \times (2^{n-k}-1)$	$2^{104} \times (2^{24}-1)$	$2^{971} \times (2^{53}-1)$	$2^{16319} \times (2^{65}-1)$	$2^{16721} \times (2^{113}-1)$
The greatest number less than 1	$1-2^{-n+k}$	$1-2^{-24}$	$1-2^{-53}$	$1-2^{-65}$	$1-2^{-113}$
The smallest number greater than 1	$1+{}^{-n+k+1}$	$1+2^{-23}$	$1+2^{-52}$	$1+2^{-64}$	$1+2^{-112}$

The real numbers that can be represented by a floating point number will be called *representable* (if it is clear from the context which precision is assumed or if the precision is of no importance). It is clear that not every real number is representable. For example, π is irrational while all the representable numbers are rational; more precisely, all the representable numbers are binary rational; i.e., they have the form $p/2^m$, where p and m are integers and m is nonnegative. Thus, 1/3 is not representable because it is not binary rational.

The numbers $2^{10000000}$ and $2^{-10000000}$ are not representable although they are binary rational: the first one is too big and the second one is too small in absolute value even for quadruple precision. To represent these numbers, an exponent of at least 25 bits long is required. The numbers that are too close to representable ones are not representable as well. For example, to represent $1 + 2^{-1000}$, a significand consisting of 999 zeros and a single unit is required.

Note that there is the floating point number –0, which differs from 0. IEEE 754 requires that they be considered equal. Moreover, no operation on floating point numbers described in this standard can produce –0 except for the subtraction of x from x with rounding toward $-\infty$ (see below) and, for some uncertain reason, square root of –0. In all the other cases, 0 is returned as the zero result.

It is seen that the represented numbers are distributed among the real numbers in a peculiar way. An approximate pattern of this distribution is shown in Fig. 1.

2.2. Difficulties in Performing Correct Computations

Not all real numbers are representable. Therefore, most computations with floating point numbers are approximate. However, the inaccuracy of the computation results is not the main problem. More important is the accumulation of errors in complicated computations when a great number of small errors in intermediate computations can completely distort the final result.

Moreover, since the relative error of approximations of real numbers by representable numbers is approximately constant, the approximation errors strongly depend on the absolute value of the numbers involved in the computations. Therefore, subtraction or division of two large close quantities in the process of computations can produce a considerable error. This effect is called *catastrophic cancellation*.

Let us consider some examples of large errors in floating point computations.

Harmonic sums. The harmonic sum H_n is defined as the sum of reciprocals of the integer numbers from 1 to n: $\sum_{k=1}^{n} 1/k$. Such sums often appear in combinato-

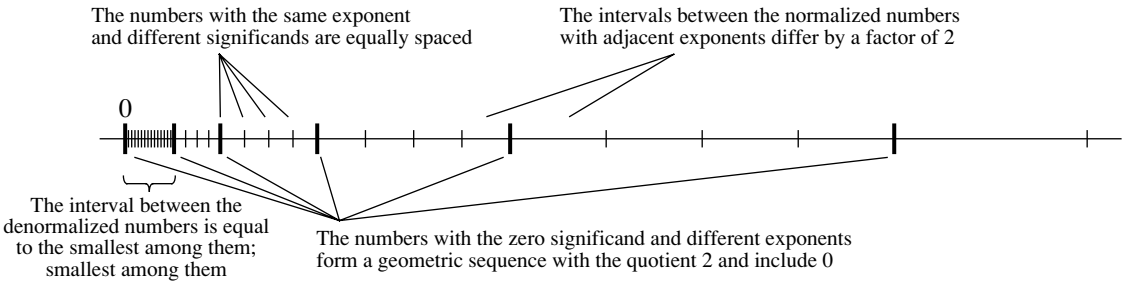

Fig.1. An approximate pattern of the distribution of floating point numbers.

rial problems or as elements of series for some special functions.

If the harmonic sum is computed straightforwardly by summing 1, 1/2, 1/3, and so on, then the terms gradually diminish and become less than the difference between the current sum and the next representable number. Hence, beginning from a certain n (which depends on the precision), the harmonic sum stops to increase.

H_n thus calculated stops growing when $1/n$ becomes less than the half of the last bit of the significand of the current value of H_n. For single precision, this happens when $n = 2^{21} = 2097152$ and for double precision, when $n = 2^{48} = 281474976710656$. The corresponding harmonic sums are H_n – 15.133306695… and 33.8482803317789…. In fact, due to accumulation of the rounding errors, we obtain $H_{2097152} = 15.403683$ for single precision (here, only two digits are correct).

The simplest way to improve the accuracy is to compute the sum beginning with the smallest terms. Then, $H_{2097152}$ for single precision is 15.132898 (here, four digits are correct). It is still better to use the asymptotic series

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \frac{1}{252n^6} + \frac{1}{240n^8} - \ldots - \frac{B_{2k}}{2k \cdot n^{2k}} - \ldots,$$

where $\gamma = 0.5772156649\ldots$ is the Euler–Mascheroni constant and B_{2k} is the Bernoulli number with the index $2k$. The use of only three terms of this series gives 15.1333065 for $H_{2097152}$, where only the last digit is incorrect. For $H_{281474976710656}$, all the digits are correct in this case.

The area of needle-like triangles. A triangle is said to be needle-like if two of its sides are approximately equal in length and both are much longer than the third one. To compute its area given the lengths of the sides, Heron's formula

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

may be used, where a, b, and c are the lengths of the sides and $p = (a + b + c)/2$ is the semiperimeter.

Let $a = b = 10^7$ and $c = 1$. Then, this formula yields 0.0 in single precision and 4999999.9999999935 in double precision (here, all the digits except for the last one are correct). The zero area in single precision is obtained because the computed semiperimeter is exactly equal to a and b due to the fact that its exact value 10000000.5 is not representable in single precision. This is an example of catastrophic cancellation, which makes the result completely senseless in this case.

To make such results impossible, the algorithm for performing computations must be stable with respect to errors. The formulas used in the computations must be rearranged so as to exclude the operations that can cause catastrophic cancellation. For the area of a triangle, such an algorithm is as follows [11].

The lengths of the sides should be arranged so that $a \geq b \geq c$. Then, the area is found by the rule

$$S = \frac{1}{4}\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}.$$

The order of the computations must be exactly as specified by the parentheses. When this rule is used for single precision computations in the example above, the result is 5000000.0, which is much closer to the correct value.

Unstable sequence. In both examples considered above, an algorithm that is stable with respect to errors can be found. Unfortunately, such algorithms are known not for all practically important problems. Moreover, for some problems no such algorithm exists because the problem itself is unstable. This situation is characteristic of many problems in population dynamics and hydrodynamics. Here, we cite an example proposed Muller (see [12]). It is of no practical importance, but clearly demonstrates the relevant effects.

Define the sequence x_n as follows: $x_0 = 2$, $x_1 = -4$, and

$$x_{n+2} = 111 - \frac{1130}{x_{n+1}} + \frac{3000}{x_{n+1}x_n}.$$

It can be proved that it is convergent as $n \longrightarrow \infty$, and the limit is 6. However, floating point computations in any precision (any number of bits can be assigned to the significand and to the exponent) produce the sequence x_n that tends to 100. The higher the precision, the later the computed values deviate from the correct ones, but this always happens.

Thus, this sequence shows that sometimes computations to any fixed accuracy can produce completely wrong results.

Incidents related to floating point computations. In addition to purely mathematical and model effects, an inaccurate account for errors and their accumulation in computational algorithms can cause severe consequences in real life.

One sad incident related to computational errors occurred in 1991 during the Desert Storm operation (see [13]). A Patriot battery failed to track and intercept an Iraqi Scud missile, which hit an army barracks, killed 28 Americans, and wounded about a hundred. An investigation revealed that the cause of the failure was

the accumulation of errors in time calculation in the battery control system.

The time was calculated in tenths of seconds starting from the beginning of the system operation, and all the computations were performed on a 24-bit processor. By the time of the incident, the battery had been operating without reloading for about four days. As a result of multiplying the original error in the representation of 1/10 of a second in binary form by a large number the tenths of seconds elapsed from the beginning of the system operation, a considerable error in time calculation (about 1/3) had accumulated. Scud travels at a speed of about 1700 m/s. Hence, an error in the time calculation exceeding several hundredth of a second makes the interception impossible.

One of the most expensive software errors in history, which caused the failure of the maiden flight of Ariane 5 launcher, was also caused by inaccurate computations. More precisely, the failure was caused by the fact that a single precision value was used to take into account the horizontal component of the launcher's velocity. This was quite adequate for Ariane 4, for which the software was initially designed. However, the velocity of Ariane 5 was much higher, which caused an overflow. The system tried to handle the exception by recalculating the same value on a backup processor, which caused another exception and an attempt to use out-of-date data for trajectory calculations. As a result, the launcher veered abruptly, began to wobble, and was destroyed due to the impossibility to continue a controlled flight.

2.3. Requirements of the Standards for Implementations of Mathematical Functions

The examples considered above show that it is important to have standards defining requirements for the components of mathematical libraries, which makes the software based on such libraries portable. However, the particular content of those requirements is also important because it can significantly affect the accuracy of computations. Let is inspect the requirements of the current standards concerning implementations of mathematical functions.

IEEE 754 and IEEE 854 standards. IEEE 754 defines four *round modes* for representing the results of floating point computations: rounding toward nearest, rounding toward positive infinity, rounding toward negative infinity, and rounding toward zero. In the first mode, the returned result is the representable number that is the nearest to the exact result; if the exact result is strictly in the middle between two representable numbers, then the representable number in which the last bit of the significand is zero must be returned. In the second mode, the least representable number that exceeds the exact result is returned. In the third mode, the greatest representable number that is less than or equal to the exact result is returned. In the fourth mode, if the exact result is positive, then the rounding is performed as in the third mode; otherwise, it is performed as in the second mode.

When the exact result is beyond the bounds of the representable numbers, different results are returned in different round modes. In the round to nearest mode, if the exact value differs from the greatest or smallest representable number more than by half of the *unit in the last place* (ulp) (see [10]), then $+\infty$ (respectively, $-\infty$) is returned. In the case of overflow in the round to zero mode, the greatest or the smallest representable number is returned, depending on the sign of the exact result. In the round to $+\infty$ (to $-\infty$) mode, $+\infty$ (respectively, the greatest representable number) is returned if the exact result is positive; if it is negative, the smallest representable number (respectively, $-\infty$) is returned. In addition, the overflow flag must be set. If the exact result is infinite, then the infinity with the corresponding sign is returned and the divide by zero flag is set.

When a nonzero result whose absolute value is less than the minimal normalized positive number is obtained, the underflow flag must be set. In all the cases, when the exact result is not representable, the inexact flag must be set. If the result of an operation cannot be interpreted (for example, the square root of –1, or 0/0, or $(-\infty) + (+\infty)$), *NaN* must be returned and the invalid flag must be set.

However, the IEEE 754 and IEEE 854 standards impose the requirements discussed above only for the arithmetic operations (addition, subtraction, multiplication, and division), type conversion, remainder in integer division, and square root. The other functions are not covered by these standards.

Standards for C libraries. The standard ISO/IEC 9899 [15] for C and the portable operating system interface (POSIX) IEEE 1003.1 standard [16], which describe the library of mathematical functions for C, do not cover the other functions either.

The C standard refers to IEEE 754 and only places additional requirements upon the results of certain functions for some specific values of the arguments (for example, $exp(0) = 1$ and $sin(0) = 0$; these requirements are in Appendix F of the ISO C standard). POSIX also refers to IEEE 754 and adds a description of the behavior of the implementations of mathematical functions in the case of overflow, underflow, and for the values of the arguments for which the corresponding function is not defined.

For example, POSIX (see [17]) requires that $sin(x) = x$ when $x = 0, -0$, and for the denormalized values of x. In addition, the value of the sine for *NaN*, $-\infty$, and $+\infty$ must be NaN. For the denormalized arguments, the range error flag must be set; if the argument is $-\infty$ or $+\infty$, the domain error flag must be set. No other constraints on the implementation of the sine are imposed.

Such requirements resemble an amazing occurrence concerning one of the first Soviet satellites. After the successful completion of the flight, it was discovered

that the routine for the sine function in the software returned the value of 1/2 for every positive argument: the developers failed to replace a debug version of this routine with the release version.

The absence of clear requirements for the majority of mathematical functions in the standards can only be explained by the desire to give time for an accurate implementation of the requirements formulated in IEEE 754, which was not a simple task in the middle of the 1980s. However, much time has elapsed since then. The absence of standards for mathematical functions sometimes results in severe errors in the mathematical models implemented in software; often, such applications produce quite different results on different platforms.

ISO/IEC 10967 standards. In the last 5–8 years, proposals for the standardization of the requirements for implementations of mathematical functions appeared [7, 8]. Many pioneers in this field work on the Arenaire project [6] carried out in France by INRIA, CNRS, and the Lyon Higher Normal School. As a result of their activities, the set of ISO/IEC 10967 standards [3–5] was issued (the third part is still under discussion), which formulate natural and independent of programming languages constraints on the implementations of elementary functions: roots, exponential function, logarithms to various bases, hyperbolic and trigonometric functions, and their inverses. These constraints concern several aspects.

• Possible errors in the computation of functions are expressed in terms of units in the last place (ulp) [10]. Rounding toward the nearest representable number produces an error not greater than 0.5 ulp; i.e., the computed result differs from the exact value not greater than by a half of the unit of the last bit in the significand of the result.

However, due to the nature of mathematical functions, such an accuracy is not always justified from the practical point of view. Any floating point number is an approximate representation of each real number for which it is the nearest representable number. Thus, the value of the function arguments can have an error that cannot be corrected by the function computation. In ISO/IEC 10967, more practical requirements for the accuracy of computations are formulated that restrict the error of the result by a value from 0/5 ulp to 2 ulp depending on the function [4].

• The ISO/IEC 10967 standards require that the implementations of mathematical functions preserve the sign of its exact value for the given value of the argument. Moreover, the implementations must be monotone on the intervals of monotonicity of the function; i.e., if a function decreases or increases on a certain interval, its implementation must decrease or, respectively, increase on the same interval.

Exceptions to this rule are trigonometric functions in the domain of large values of the argument for which the lengths of the interval of the sign change and the interval of monotonicity are comparable with the unit in the last place of the argument.

• The ISO/IEC 10967 standards require that some specific properties in a neighborhood of the points where the corresponding function has known representable values be fulfilled.

For example, the exponential function must return the exact unity for the arguments that are sufficiently close to zero.

This requirement is caused by the fact that the density of the representable numbers is much higher in the neighborhood of 0 than in the neighborhood of 1. Indeed, the distance to the nearest double precision representable number is 2^{-53} for 1 and 2^{-1074} for 0.

2.4. The Table Maker's Dilemma

The requirements for the accuracy of computations of mathematical functions lead to the so-called *table maker's dilemma* [18–20].

The problem is that in order to correctly choose the rounded nearest floating point number in approximate computations, one sometimes has to compute many extra bits of the result's significand (actually, much more than the floating point numbers used for the computations can store).

The name of the problem originates in the times when the tables of mathematical functions (logarithms, exponentials, sines, and cosines) were made up manually. To perform the rounding correctly, the table maker computed one or two significant digits more that the resulting table was supposed to hold. However, sometimes the additional digits are insufficient. In such cases, the table makers often chose one of two possible results at random.

For example, suppose that we are computing the sine for binary floating point numbers that have a significand of six bits long. The sine of the number (the bits of the significand are printed in bold) $\mathbf{11.1010}_2 = 3.625_{10}$ is $0.0\mathbf{111011011}1110\ldots_2 = 0.063225984913\ldots_{10}$. An approximate computation of the six bits of the significand can produce 0.0111011_2 or 0.0111100_2 because the exact value is very close to their arithmetic mean. We can reliably choose the first value as the nearest one to the exact result only if we know the 14th exact bit.

In the round toward nearest mode, the table maker's dilemma appears when the value of a function is not representable but is very close to the arithmetic mean of two representable numbers. If the rounding is toward zero, toward $+\infty$, or toward $-\infty$, this problem appears when the exact value is not representable but is so close to a representable number that only considerable additional computations can decide whether the exact result exceeds this representable number or not.

Consider both cases in which the table maker's dilemma manifests itself in double precision computations. Computing the natural logarithm of the number

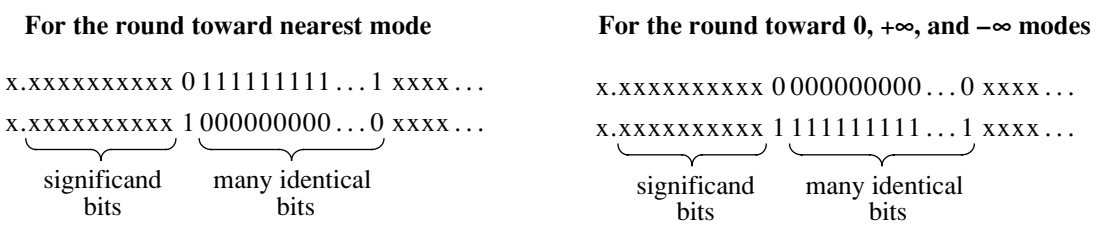

Fig. 2. The general form of values of functions for which the table maker's dilemma occurs.

$1.613955DC802F8_{16} \cdot 2^{-35}$, we obtain $-17.F02F9BAF60357F^{14}9\ldots_{16}$ (here F^{14} denotes that the digit F repeats 14 times which, taking into account the two adjacent digits, yields 60 repeating unities). This number lies almost exactly in the middle between $-17.F02F9BAF6035_{16}$ and $-17.F02F9BAF6036_{16}$ but is slightly closer to the first of them.

Thus, in order to make the correct rounding in the round mode toward nearest, the logarithm at the point under examination must be computed with a relative error not exceeding 2^{-114}. A greater error does not make it possible to determine which of the two indicated representable number is closer to the exact value of the logarithm.

The natural logarithm of the number $1AC.50B409C8AEE_{16}$ is $1.83D4BCDEBB3F3F^{15}A\ldots_{16} \cdot 2^2$. When the logarithm at this point is computed in the round modes toward zero, $+\infty$ or $-\infty$, one must find out whether or not the exact value of the logarithm exceeds $1.83D4BCDEBB3F3F4_{16} \cdot 2^2$. To this end, it must be computed with a relative error not exceeding 2^{-114}.

The general form of the numbers that present difficulties for rounding is showed in Fig. 2.

The examples above show that, in order to obtain a correctly rounded result, the computations must sometimes be much more accurate than can be achieved within the floating point numbers with a fixed accuracy.

For a greater part of widespread functions, their values for general (not specific, for example, distinct from 0 and 1) binary rational arguments are irrational; therefore, they cannot be representable and cannot be arithmetic means of two representable numbers. For the exponential function, for the hyperbolic, trigonometric functions and for their inverses, this follows from the fact that the number e raised to a nonzero algebraic power (even a complex one) is transcendental (see, e.g., [21]).

Since the set of representable numbers is finite, there is an $\varepsilon > 0$ for every function such that the computation of this function with an error not exceeding ε allows one to determine the correct rounded representable number for each round mode. However, it is inefficient to compute the function to such an accuracy for all values of the argument. For example, for the natural logarithm in double precision computations, ε can be set to 2^{-118} for any round mode (see [20]). However, such a high accuracy is required only for a single value of the argument; for all other arguments, a lower accuracy is sufficient. For the overwhelming majority of representable double precision numbers, the correct rounding of the logarithm can be obtained if the computations are performed with an error not exceeding 2^{-54}.

The table maker's dilemma requires that the computations be performed much more accurately than is possible with standard types of floating point numbers; this concerns both the correct implementations of mathematical functions and the verification of the correctness of those implementations. However, it would be inefficient to perform so accurate computations for all values of the argument. Therefore, in order to improve efficiency, one should know how to choose the accuracy depending on the current value of the function argument.

3. A SURVEY OF STUDIES DEVOTED TO TESTING IMPLEMENTATIONS OF MATHEMATICAL FUNCTIONS

Methods for computing mathematical functions are considered in many studies. One classical work on this topic is [22], although it was published long ago and is now partly obsolete. A more modern presentation of the methods for computing elementary functions (only a subset of the functions considered in [22]) can be found in [23].

Studies devoted to a systematic testing of implementations of mathematical functions in floating point numbers are scarce. On the Internet, one can find a huge number of various computer programs for testing such functions (see, e.g., [24]); however, the overwhelming majority of them consider only one aspect (sometimes, two or three) of the computations.

It was indicated in [24] (see also [25, 26]) that although the standardization of floating point computations began over 20 years ago, many manufactures of hardware and software still do not strictly follow those standards. Hence, tests for the correct behavior of implementations of mathematical functions are still required.

Among the most systematic works on testing floating point computations, we distinguish the following ones.

Studies devoted to testing the conformity to IEEE 754.

• The first systematic test suite for verifying the correctness of implementations of operations on floating

point numbers is described in [27]. It appeared even before IEEE 754 was adopted. This test suite is organized as a set of Fortran programs, and it is designed for testing only the addition, subtraction, multiplication, and division.

• In [28, 29], a test suite designed specifically for verifying the conformity to IEEE 754 is presented. It is available at the site [30]. This test suite verifies all the requirements of the standard for the arithmetic operations, square roods, remainder in integer division, and conversions between floating point and integer numbers.

• The PARANOIA program [31, 32] was developed by W. Kahan—one of the authors of IEEE 754—and it remains a popular tool for testing the conformity to this standard, although the verification is less thorough than in the test suite [28, 29]. Only the basic arithmetic operations and the square root are tested.

• Another approach to designing test suites for verifying the conformity to IEEE 754 is used in FPgen [33, 34]. Here, the test data include, in addition to special values, the numbers satisfying certain patterns, for example, in which the zero and the unit bits in the significand alternate or in which the significand contains exactly seven unit bits.

Studies devoted to testing a wide set of mathematical functions.

• The test suite ELEFUNT [35] includes tests for several mathematical functions organized in the form of C and Java programs. These tests verify the correctness of the returned values for the specific values of the argument (0, 1, $+\infty$, $-\infty$, and *NaN*); in addition, certain identities (for example, $exp(x) \cdot exp(-x) = 1$) are verified for some randomly generated arguments.

• The test suite UCBTEST [36] is designed for verifying the basic arithmetic operations and a fairly large set of mathematical functions (more functions than in ELEFUNT). This suite is organized in the form of Fortran and C programs and a prescribed set of argument values for different functions. Each test case verifies that a certain function, for a given set of the argument values, returns the correct value or a value very close to it taking into account the round mode. No procedure for choosing the test data is specified. Judging from the analysis of the test suite, one can conclude that several different ideas were used.

◦ Special floating point numbers such as 0, –0, *NaN*, $+\infty$, $-\infty$, the smallest positive, the greatest positive, the smallest and the greatest positive denormalized numbers, and the like were used as test data.

◦ The arguments for which the function under examination can be exactly represented by a floating point number (for example, $sin(0) = 0$, $cos(0) = 1$, etc.) were verified.

◦ Some values of the arguments were chosen taking into account the structure of the algorithms used to compute elementary functions. For example, some algorithms used to compute the logarithm first reduce the argument to the interval (0.5, 1] by multiplying or dividing it by 2. In this case, the test data included the boundaries of this interval and the boundaries of several adjacent intervals, as well as some numbers that are close to those boundaries.

• Similar approaches—the use of special values, boundaries of the intervals that are often used by various algorithms for computing the function under examination, and the numbers following a certain pattern or randomly generated numbers—are used to generate extensive test data in the Berkley test suite [37].

Studies in the framework of Arenaire project. The Arenaire project includes, in addition to the studies devoted to testing, studies devoted to efficient implementations of mathematical functions and to the analytical verification of the implementations.

• In [18–20, 38], the table maker's dilemma is examined, and the arguments for which the correct computation of functions with a prescribed accuracy is most difficult are found. These numbers can be used as "inconvenient" test data for practically any implementation of the corresponding function.

• The MPCheck tool [39] is under development in the framework of this project. It is designed for testing the correctness of implementations of mathematical functions from the viewpoint of preserving their monotonicity and the correctness of rounding. The test data include numbers that match certain patterns.

On the whole, the available studies devoted to the formulation of requirements for implementations of mathematical functions and to testing the conformability to those requirements do not provide a systematic unified approach to all the relevant problems. Nowhere, except for the Arenaire project, the table maker's dilemma and the arguments for which the computation of correct results is most difficult are considered. However, the Arenaire project does not consider the test data based on the boundaries of the intervals that are specific for a given function.

The most systematic requirements for the behavior of mathematical functions are presented in the ISO/IEC 10967 standard. However, the procedure used to design this standard is not explicitly described, which prevents extending it to a larger set of functions, for example, those included in the standard C library (in addition to the elementary functions, it includes, for example, the gamma and the Bessel functions).

Moreover, no test suites that verify the conformity to ISO/IEC 10967 are currently available, and the author does not have any information about the development of such test suites.

Thus, the development of systematic methods for formulating requirements for the implementations of mathematical functions and the development of procedures for the corresponding test generation are of current interest.

4. THE PROPOSED APPROACH

In the framework of the proposed approach, a method for determining requirements for the implementations of a particular mathematical function and a method for choosing test data for verifying if a particular function conforms to those requirements.

4.1. Method for Determining Requirements for Implementations of Mathematical Functions

The proposed method for determining requirements for the implementations of mathematical functions borrows the major part of its ideas from ISO 10967 [3–5] and from papers [7, 8] devoted to the development of standards with high requirements for the correctness of mathematical function computations. Some elements of the proposed method are novel—they are not described in the available literature but generalize some popular techniques. Each item below refers to the source where similar ideas are described or to common practice if this item is a generalization of the common approach to resolving similar problems.

In this paper, we consider requirements for the result of the function computation; however, we do not consider setting flags or raising exceptions. The latter requirements should be formulated in accordance with the general rules of IEEE 754, but a large number of details that must be taken into account makes the formulation of the requirements for this aspect of the function behavior a subject of a separate study.

The requirements for the results produced by implementations of mathematical functions can be divided into several parts, which will be considered separately.

• Domain of a function and singular points.

◦ *Domains* (common practice). For all the values of the argument for which the function is defined, its implementation must return a certain result that can be equal to +∞ or –∞ if the value of the function is beyond the range of the floating point numbers and if the round mode requires such a result; however, the result cannot be *NaN*.

◦ *Limiting values* [8]. For all arguments where the function is not defined (including singular points) but has a unique limit maybe equal to +∞ or –∞, the implementation must return the value of this limit.

◦ *One-sided limits at* 0. (Common practice for particular functions). If a function has a singularity at zero but has no infinite limit at this point, one-sided limits must be considered. At 0, the implementation must return the limit of the function as $x \longrightarrow +0$, if it exists; at –0, the returned value must be equal to the limit as $x \longrightarrow -0$, if such a limit exists. An example of such a function is the cotangent: it is usually assumed that $cot(0) = +\infty$ and $cot(-0) = -\infty$ (see the discussion of odd functions below).

◦ *Complete indefiniteness* [8]. In all the other cases, NaN must be returned. Here are some examples: $sqrt(-1.0) = ln(-1.0) = sin(+\infty) = NaN$. A specific consideration should be given to the arguments for which there is no common opinion on the possible continuous extension of the function to this point. An example is 0^0, which is sometimes interpreted as 1 and sometimes as *NaN*.

◦ *Neighborhood of poles* (common practice and [4]). For the poles at which the value of the function tends to infinity, the neighborhood in which the function value is no longer representable must be determined. If this neighborhood contains representable numbers, the function implementation must return for them +∞, –∞, the greatest or the smallest representable number depending on the sign of the exact value and on the round mode.

For example, the cotangent has a pole at 0, and its behavior in the neighborhood of 0 is approximately as that of $1/x$. For many denormalized numbers, the reciprocal is not representable; therefore, the implementations of the cotangent must return +∞ or –∞ for many numbers in the neighborhood of zero, depending on the sign of the argument. The value of the cotangent for 2^{-1024} is $1.\text{FFFFFFFFFFFFF}^{499}5\ldots_{16} \cdot 2^{1023}$. Therefore, in the round modes toward 0 or toward –∞, the cotangent must return the greatest positive representable (double precision) number for all the positive numbers that do not exceed 2^{-1024}. In the round mode toward nearest or toward +∞, the cotangent must return +∞ for this interval. In these round modes, its value falls into the interval of representable numbers only beginning with $1.0000000000001_{16} \cdot 2^{-1024}$.

◦ *Neighborhoods of infinities for infinite limits* (common practice for particular functions and [4]). For the functions that tend to infinity as $x \longrightarrow +\infty$ or –∞, the bounds on their representable values must be exactly defined. Beyond these bounds, the implementation must return +∞, –∞, the greatest or the smallest representable number depending on the sign of the exact value and on the round mode.

For example, $exp(x)$ tends to +∞ as $x \longrightarrow +\infty$. The greatest double precision number for which $exp(x)$ is also representable in double precision is $1.62\text{E}42\text{FEFA}39\text{EF}_{16} \cdot 2^9 = 709.7827\ldots_{10}$. For all greater arguments, $exp(x)$ must return +∞ in the round mode toward nearest and toward +∞; for the other round modes, it must return the greatest representable number.

For single precision, the greatest number for which $exp(x)$ is also representable is $1.62\text{E}42\text{E}_{16} \cdot 2^6 = 88.7228\ldots_{10}$.

• Special values, values at 0, tangents, and asymptotes.

◦ *Infinities and* –0 (common practice). For the special values of the argument (–0, +∞, and –∞), the value of the functions should be defined in the most natural way. Usually, it is sufficient to define them as the limits if those limits exist; otherwise, they are defined as *NaN*.

Many library functions are even or odd. For even functions, it is quite natural to have $f(-0) = 0$; for odd functions, it would be desirable to have $f(-0) = -0$.

Note that this rule contradicts the strange requirement of IEEE 754 that $sqrt(-0) = -0$. It would be more reasonable to set this value to 0 (another reasoning recommends *NaN*, but this value results in the situation $x = y$ and $sqrt(x) \neq sqrt(y)$ because it is assumed that $0 = -0$).

◦ *The value at NaN* (follows from the requirements of IEEE 754). The value of any function at *NaN* must be *NaN*.

◦ *Exact values* (common practice for particular functions and [4, 16]). For certain arguments, the values of a function are known exactly. If both values are representable, then the function implementation must return the exact value at such points. For example, $exp(0) = cos(0) = cosh(0) = 1$, $sin(0) = sinh(0) = tan(0) = arcsin(0) = 0$, $ln(1) = 0$, etc.

◦ *Neighborhoods of extremal values that are exact values* (common practice for particular functions and [4]). In addition, if the derivative of a function at such a point is equal to zero, then the implementation must return the same value for all arguments in a certain neighborhood of this point.

For example, the cosine has a zero derivative at 0; therefore, the dependence of the distance between its value and 1 is quadratic. The bounds of the neighborhood of 0 for which the implementation of cosine must return 1 depend on the precision and on the round mode. For double precision and rounding toward nearest the bound is between $1.6A09E667F3BCC_{16} \cdot 2^{-27}$ and the next representable number because $cos(1.6A09E667F3BCC_{16} \cdot 2^{-27}) = 1.FFFFFFFFFFFF80^{12}6\ldots_{16} \cdot 2^{-1}$ and $cos(1.6A09E667F3BCD_{16} \cdot 2^{-27}) = 1.FFFFFFFFFFFF7F^{12}B\ldots_{16} \cdot 2^{-1}$. In the case of rounding toward $+\infty$, the corresponding bound is 2^{-26} because $cos(1.0_{16} \cdot 2^{-26}) = 1.FFFFFFFFFFFF0^{13}1\ldots_{16} \cdot 2^{-1}$ and $cos(1.0000000000001_{16} \cdot 2^{-26}) = 1.FFFFFFFFFFFEF^{12}C\ldots_{16} \cdot 2^{-1}$.

For the round modes toward 0 or toward $-\infty$, the cosine is equal to 1 nowhere, except for 1. The same bounds for single precision and for the round modes toward nearest and toward $+\infty$ are 2^{-12} and $1.6A09E6_{16} \cdot 2^{-12}$.

◦ *Neighborhood of* 0 (common practice for particular functions and [4]). If the value of a function at 0 is representable but distinct from 0 even if the derivative of the function is nonzero at 0, the same rule must be used: the implementation must return the same value for all floating point numbers in a certain neighborhood of 0. This requirement follows from the fact that the density of floating point numbers near 0 is greater than near any other number.

For example, for $exp(x)$, 1 must be returned in double precision and in the round mode toward $+\infty$ beginning from $-1.0_2 \cdot 2^{-53}$ up to 0. In the round mode toward nearest, this is true for the interval from $-1.0 \cdot 2^{-54}$ to $1.FFFFFFFFFFFF_{16} \cdot 2^{-53}$.

◦ *Horizontal asymptotes* (common practice for particular functions and [4]). If a function has horizontal asymptotes, the boundary must be determined beyond which the value must be constant for a particular round mode.

For example, for $exp(x)$ in double precision and the round mode toward nearest, the implementation must return 0 for up to $-1.74910D52D3051_{16} \cdot 2^9$ inclusive. In the round mode toward 0 and toward $-\infty$, the same result must be returned for all negative numbers less than or equal to $-1.74385446D71C4_{16} \cdot 2^9$. In the round mode toward 0 and toward $+\infty$, $exp(x)$ must return 0 only for $x = -\infty$; for the numbers not exceeding $-1.74385446D71C4_{16} \cdot 2^9$, the result must be 2^{-1074}.

◦ *Asymptotics* (common practice for particular functions and [4, 16]). It seems natural to place the same requirements upon the functions that have nonhorizontal asymptotes or the functions that are asymptotically close to other functions. For example $e^x \sim 1 + x$ for $x \sim 0$, $cosh(x) = (e^x + e^{-x})/2 \sim e^x$ for $x \sim +\infty$, $cosh(x) = (e^x + e^{-x})/2 \sim e^{-x}$ for $x \sim -\infty$, and $sin(x) \sim x$ for $x \sim 0$.

However, in many cases, such requirements cannot be formulated accurately for different round modes. The point is that even a very small difference between two asymptotically close expressions can lead to a difference in the significant bits of the significand. The cause of this phenomenon is similar to the cause of the occurrence of the table maker's dilemma: some values of certain functions are too close to representable numbers.

For example, consider the asymptotics $e^x \sim 1 + x$ for $x \sim 0$ for double precision floating point numbers. For $|x| < 2^{-28}$, the difference between e^x and $1 + x$ is less than 0.5 ulp; however, there are floating point numbers that are much closer to 0 for which the significands of e^x and $1 + x$ are different in the round modes toward $-\infty$ and $+\infty$.

For example, for $x = -1.8000000000001_{16} \cdot 2^{-52}$, $exp(x) = 1.FFFFFFFFFFFD0^{13}4\ldots_{16} \cdot 2^{-1}$ and $1 + x = 1.FFFFFFFFFFFCF^{12}E_{16} \cdot 2^{-1}$.

It seems that accurate requirements can be formulated only for the asymptotics $f(x) \sim kx$ or $f(x) \sim -kx$, where k is a certain representable constant (for example, $sin(x) \sim x$ and $tan(x) \sim x$ for $x \sim 0$) because no such effects occur in such situations. The boundaries within which such requirements can be formulated should be accurately calculated for various round modes.

For example, for the implementation of the sine in double precision and the round modes toward nearest, the result must coincide with the argument on the interval $[-x_0, x_0]$, where $x_0 = 1.7137449123EF6_{16} \cdot 2^{-26}$. In the round mode toward $+\infty$, this requirement must be fulfilled on the interval from 0 to $x_1 = 1.D12ED0AF1A27F_2 \cdot 2^{-26}$. In the round mode toward $-\infty$, the same must be true for the interval $[-x_1, 0]$. In the round mode toward 0, $sin(x) = x$ only at $x = 0$. Note that this is different from the POSIX requirements (see above).

• **Range of a function**

◦ *Choosing a branch of a function* (common practice). If a mathematical function is multivalued (in the strong sense), its branch returned by the implementation must be thoroughly defined. The boundary values, which often can be chosen in several ways, must be defined so as to take into account the most widely used variant.

For example, for the inverse trigonometric functions and the inverse hyperbolic cosine, $arctan(x)$ and $arcsin(x)$ usually take the values between $-\pi/2$ and $\pi/2$, and $arccot(x)$ and $arccos(x)$ usually take the values between 0 and π. For $Arccosh(x)$ and roots of even degrees, the branch for which these functions have nonnegative values is usually used.

◦ *Preserving the range of functions* [8]. Upper and lower bounds on the function range for each connected component of its domain must be preserved in the implementation. Otherwise, various troublesome effects can occur.

For example, if an implementation of $arctan(x)$ returns, for large positive arguments, the nearest floating point number to $\pi/2$, it may happen that this number is greater than $\pi/2$. For single precision floating point numbers, this is exactly the case: the nearest single precision floating point number to $\pi/2$ is $1.921FB6_{16}$ = 13176795/8388608 > $\pi/2$. In this case, we have $tan(arctan((2^{30}))) = -2.2877\ldots \cdot 10^7$, which contradicts the main property of the inverse functions.

◦ *Denormalized values* (used for designing test suites for some functions in UCBTEST [36]). Since the denormalized numbers are specific, the intervals on which the values of a function are denormalized must be thoroughly defined.

For example, in double precision, the value of $exp(x)$ remains denormalized for all negative x up to $-1.6232BDD7ABCD3_{16} \cdot 2^9$ (this number depends on the round mode). For all greater arguments, $exp(x)$ is normalized for all round modes.

• Monotonicity and Sign Preservation

◦ *Preservation of monotonicity* [4, 8]. On all the intervals for which a mathematical function is monotone, its implementation must be monotone as well. This is required to adequately represent essential properties of mathematical models in their numerical implementation.

However, this rule becomes senseless in the domains where the function often changes the direction of monotonicity, i.e., where the length of the monotonicity interval is equal to or less than the unit in the last place of the argument's significand. Examples of such functions are the trigonometric functions (an exception from this rule is discussed in [4]) and the Bessel functions (not mentioned in [4]) for large values of the argument.

◦ *Preservation of sign* [4, 8]. The sign of the correct implementations must coincide with the sign of the exact value of the function. If the value of a function is equal to 0, then the implementation must return 0.

It is mentioned in [4] that this requirement may not be fulfilled when the unit in the last place of the argument is greater than the interval on which the function preserves its sign. This is the case for the trigonometric functions for large values of the argument. However, the use of correctly rounded values for the function result makes such stipulations unnecessary.

• Symmetries and Periodicity

◦ *Evenness and oddness* [4, 8]. If a mathematical function is even or odd, the implementation must possess the same property.

◦ *Horizontal symmetries* (author's remark). If a function is symmetric about some other representable values of the argument (for example, if $f(1-x) = -f(x)$), the similar property for the implementation cannot be always retained. Indeed, $1 - x$ can be not representable when x is representable. Such cases must be considered separately; the requirement can be imposed only if both values of the argument are representable. If a function is symmetric about a not representable value of the argument (for example, $sin(\pi - x) = sin(x)$), the symmetry for the implementation can be fulfilled only approximately; therefore, it cannot be strictly required.

◦ *Other symmetries* (author's remark and a generalization of the preceding rules). In general, all the important functional equations satisfied by the function must be analyzed, and it should be decided if it is reasonable that these equation must be satisfied in the implementation. The values of the argument for which the equation is satisfied must be explicitly specified. For example, the relations $\Gamma(1 + x) = x \cdot \Gamma(x)$ and $\Gamma(1 - x) = -x \cdot \Gamma(-x)$ must be fulfilled for positive integer arguments; otherwise, the fundamental relation $\Gamma(n) = (n - 1)!$ can be violated for positive integer n.

For the periodicity property, the same reasoning applies. If the period is representable, the periodicity must be fulfilled only for the numbers that are representable together with their shift by the period or by a multiple of the period. If the period is not representable, the implementation can be periodic only approximately.

For the arguments for which the unit in the last place is greater than the period, the periodicity requirement is senseless.

• Correct Rounding (extension of the rules of IEEE 754 [8]).

In addition to the constraints listed above, one should require that the result returned by any implementation could be obtained from the exact value of the function by applying the rounding procedure adopted in the current configuration.

The monotonicity and sign preservation requirements are fulfilled if the rounding procedure is correctly organized. However, the results obtained in the round modes toward 0, toward $+\infty$, or toward $-\infty$ some-

times contradict the requirement of preserving the constraints on the range of functions. In this case, a decision may be sometimes made depending on the function; hence, some of its important properties can be violated. More often, it is convenient to assume that the support of the round mode has a higher priority because the users that employ such a mode should be aware of its consequences. It is sometimes assumed that it is useless to require the accuracy of 0.5–1 ulp, especially in the domains where the interval of a significant function variation is shorter than the unit in the last place. However, my opinion is that standard implementations of general-purpose mathematical functions must always produce the most accurate results independently of whether these results have sense when considered as approximate. This requirement is caused by the fact that most users employ library implementations without performing the analysis of function properties. Hence, they cannot notice that the result is inaccurate simply because the function varies too quickly in the neighborhood of the argument value under examination. Thus, the user should obtain the most accurate result with respect to the given argument. The reasoning concerning the best interpretation of this result is better to be left to the user; indeed, there is some chance that the user really needs the correct value of the function for a particular argument.

The implementations in which the decrease in accuracy is justified by improved efficiency cannot be used as standard ones and cannot be applied in the general case. Such implementations may (and should) be developed for specific problems; in such cases, constraints on admissible errors in different domains of the argument and deviations from the general-purpose standards should be explicitly determined.

4.2. A Method for Conformity Test Generation

The method (presented in the preceding section) for determining requirements for implementations of a mathematical function provides a subdivision of its domain (the set of floating point numbers) into a set of intervals. Each interval is characterized by a unique set of function properties.

In each of these intervals, the value returned by the function implementation is either fixed, or is determined by an asymptotics, or is determined by the simple rule of rounding the exact value of the function in accordance with the current round mode.

To generate a test suite, one must know how to find the correctly rounded value of the function. To this end, one must know how to calculate the function much more accurately than it is possible within the set of floating point numbers under consideration. This can be done in several ways.

• Using symbolic calculations and computations to an arbitrary accuracy (for example, Maple [40], Mathematica [41], or MATLAB [42]).

• Using libraries of correctly rounded functions, such as those based on the studies by Ziv [43], IBM accurate Portable MathLib [44], GNU MPFR [45], libmcr [46] developed by Sun, or the SCSLib [47] and CRlibm [48] libraries developed in the framework of the Arenaire project [6, 49–52].

• One can develop a special implementation of a function based on the methods presented in books [22, 23] or based on the interval computation methods [53–56]. Interval computations have an additional advantage of guaranteeing the validity of the bounds that contain the result. In particular, interval computations make it possible to obtain correct results even if the problem itself is unstable. With interval computations, no unstable sequence effect described in Section 2.2 can occur.

The main ideas of the method for designing test suites presented below are fairly simple and can be summarized as follows.

• The test data include the boundaries of the intervals corresponding to the unique properties of the function being tested and the boundaries of the intervals of the floating point numbers that have a special structure (for example, boundaries of the denormalized positive numbers).

• Some points within these intervals selected using certain rules are also included in the test suite.

• The points for which the computation of the correctly rounded value of the function is most computationally costly due to the table maker's dilemma [20] are included in the test suite. This is because the majority of simple implementations produce incorrect results for such arguments.

• Finally, we include the points that are good approximations of the arguments for which the value of the function is known exactly if such values are important in practice. For example, $cos(x) = 1/2$ for $x = \pm\pi/3 + 2n\pi$. Although none of such x is representable, one can find many floating point numbers that are so very close to them that the correct result in the round mode toward nearest is equal to 1/2. Such arguments provide a good test for the correctness and accuracy of the computations.

The floating point numbers that are most close to the roots of the function are an important particular case of this rule. The use of such arguments in test suites helps verify the sign preservation.

In more detail, the main steps of the proposed method are as follows.

1. First, the set of the boundaries of the intervals of specific behavior of the function and the floating point numbers possessing a specific structure are added to the *initial set* of the test data.

◦ To this end, we add to the initial set 0, $+\infty$, $-\infty$, the greatest and the smallest representable numbers, and the greatest and the smallest positive and negative denormalized numbers.

These numbers subdivide the set of floating point numbers into intervals of the numbers possessing a specific structure.

◦ When determining the requirements for the implementations of a function, the floating point numbers are also subdivided into intervals following the procedure described above. Within each of these intervals, a specific set of constraints on the possible results of the function is in effect.

For example, these are the intervals of monotonicity and constant sign of the function, the intervals on which the function has a constant value or is described by a simple asymtotics, or the intervals on which the function is not defined.

In addition, one must consider the preimages of the intervals selected at the preceding step. The preimages of the positive and negative numbers give the constant sign domains; therefore, here we additionally obtain only the subdivision of these domains into the preimages of normalized and denormalized values. Each such preimage is decomposed into a union of finite sets of intervals, which also must be included in the set of intervals.

The number of the intervals selected as described above can be very large for some functions; the complete set can be practically infinite. For example, this is the case for the trigonometric functions: each interval $[n\pi/2, (n + 1)\pi/2]$ is the intersection of the monotonicity and sign preservation intervals; hence, it should be included in the set of intervals. In such situations, one should find some rules for selecting intervals; these rules should be based on specific properties of the function behavior, so that they reduce the set of intervals to a reasonable size. Depending on the computational resources assigned for testing, the number of intervals may be from several dozens to several thousands. For example, for the trigonometric functions, the rules might be as follows. The behavior of the function on the selected intervals should be specific in a certain sense. This can be achieved by considering the best possible approximations of the floating point numbers to the multiples of π and $\pi/2$. On the intervals containing such numbers, the values of the trigonometric functions are most close to zero and to their extreme values. The best approximations to the multiples of π and $\pi/2$ can be found by an expansion of π into a continued fraction (see [57, 50]); this yields about 2000 numbers. Choosing the intervals that contain the selected numbers, we obtain a set of intervals of a reasonable size. It is a good idea to add to this set about two or three dozens of intervals of the form $[n\pi/2, (n + 1)\pi/2]$ that are most close to zero.

2. The resulting initial set subdivides the floating point numbers into a set of intervals.

To determine the test values within these intervals, we choose two integer parameters n and k. The first parameter denotes the "frequency" of the selected points, and the second parameter determines the size of the continuous regions covered by those points.

Each of the intervals is subdivided into n shorter intervals that contain the same number of floating point numbers. This yields $(n + 1)$ points—$(n - 1)$ internal and two boundary points. Then, we choose all floating point numbers in the interval under consideration that are at a distance not greater than k floating point numbers from the points obtained at the preceding step. The resulting set is called the *trial set*.

For each of the initial intervals, individual n and k may be used; thus, some regions can be covered more densely than others.

3. The trial set should be completed with some points for which the correctly rounded result can be obtained only by performing much more accurate computations than is stipulated by the precision of the floating point numbers (see the discussion of the table maker's dilemma above and [20, 38]).

Moreover, not only the values that require the most accurate computations should be used, but also those values that require not less than $m > 0$ additional bits in the significand compared to the precision of the floating point numbers.

The available practical results and a reasoning based on probability theory [19] show that, for the floating point numbers that have K bits in the significand and P bits in the exponent, such values exist for $m < K + P$; for larger m, such values are very rare. It makes no sense to take very small m because it may happen that the corresponding values are too numerous. Moreover, modern processors often perform actual computations using double-extended precision numbers; hence, the double precision arguments with $m < 12$ do not provide practically useful tests. Experiments (see [19]) show that, for double precision, $m = 40$ is a good choice: usually, there are several thousand test values of the argument for this m.

4. If the function under examination has some practically important properties related to its exact values for certain not representable values of the argument, the floating point numbers that provide the best approximation of such arguments should be found.

First of all, the best approximations of the function roots should be considered.

Some important examples are provided by the trigonometric functions. These are the relations $sin(x) = 1/2$ for $x = \pi/6 + 2n\pi$ and $5\pi/6 + 2n\pi$, $sin(x) = -1/2$ for $x = -\pi/6 + 2n\pi$ and $-5\pi/6 + 2n\pi$, $cos(x) = 1/2$ for $x = \pm\pi/3 + 2n\pi$, $cos(x) = -1/2$ for $x = \pm 2\pi/3 + 2n\pi$, $tan(x) = cot(x) = 1$ for $x = \pi/4 + n\pi$, and $tan(x) = cot(x) = -1$ for $x = -\pi/4 + n\pi$. The best approximations of such arguments can be found by expanding π and $\pi/3$ into continues fractions [58].

5. Sometimes, hypotheses concerning possible errors in the computation of the function for certain arguments having a special structure (for example, for the arguments that have exactly one unity in their significand or exponent, or in which the unities alternate

with zeros in the significand or exponent) may be used. Then, some numbers having such a structure should be added to the trial set. The hypotheses in question can be often formulated in the form of a set of templates for the bit representation of the argument.

6. Finally, the trial set should be completed with several *NaN* values. They can be chosen using some rules; for example, the values with the maximal and the minimal significand, or various templates for the significand may be used.

The trial set thus obtained can be used for testing the implementations of the function under examination.

In the framework of the proposed method, the number of the test data, the thoroughness of testing, and the computation time for performing the tests can be controlled by varying the parameters n and k at step 2, m at step 3, and the accuracy of the approximations at step 4. The rules used to select the intervals at step 1 and the templates at step 5 may be also varied.

4.3. An Example of Using the Proposed Method

In this section, we consider the results of a practical implementation of the proposed method for formulating the requirements and generating a test suite for the exponential function $exp(x)$ in double precision. In what follows, we use the term *exponential* to denote the exponential function.

An analysis of the properties of the exponential performed in accordance with the method presented in Section 4.1 and the selection of the denormalized and exceptional numbers subdivide the real line into the points and intervals presented in Table 2. The values of the boundary points are shown in Table 3. The expressions $x++$ and $x--$ for the floating point number x denote the next and the preceding floating point number, respectively.

The test data included the boundaries of the intervals and some points within them chosen according to item 2 of the method described above.

The test data also included about 10000 numbers for which the table maker's dilemma for the exponential manifests itself. These values were found in the course of the large-scale search of such values for the elementary functions in the framework of the Arenaire project.

For large negative values of the argument where the exponential is already positive but its values are small denormalized numbers, it increases very slowly. Therefore, the preimage of each such number is a long interval of negative numbers; the exponential of all the numbers within this interval is equal to the given denormalized number provided that the given round mode is used. About a dozen of boundaries of such intervals were added to the test suite; and, for each interval, one number within it was also added.

Furthermore, the test data included several floating point numbers such that the values of the exponential for the adjacent numbers differ exactly in the last bit of the significand.

We also added to the test suite the numbers 1.0, –1.0, their neighboring numbers, and, finally, five different representations of *NaN*.

As a result, we obtained the test suite consisting of 15804 values of which 2604 are designed for testing the exponential in the round mode toward nearest, and 4400 values are used for testing in each of the three other round modes. The testing results obtained on various platforms are presented in Tables 4 and 5.

Table 4 shows the testing results for platforms A–L that support different round modes. Platforms M–Q support only rounding toward nearest; for this reason, the results for these platforms are placed in Table 5 along with the results for platforms A–L obtained for this round mode. Below, we present an analysis of the errors, discuss them, and describe the platforms used in the testing.

ANALYSIS OF THE ERRORS

For the convenience of the analysis, the floating point numbers were subdivided into the following classes: positive normalized numbers, negative normalized numbers, positive denormalized numbers, negative denormalized numbers, 0, –0, $+\infty$, $-\infty$, and *NaN*.

We assume that the incorrect class of the result is a more serious error than an incorrect result within the correct class. The second kind of errors in Tables 4 and 5 is called computational errors. Both types of errors—those that change the class of the result and the ones that do not—were classified into serious and small errors depending on the distance between the correct and the actual result (in terms of the last bits of the significand).

An error in the class of the result is said to be serious if the correct or the actual result is *NaN* and they are different, or if the correct and the actual result belong to the classes that are not adjacent to each other, or if the classes are adjacent but the distance between the correct and the actual results is greater than 2^{30} floating point numbers. Two classes are said to be adjacent if there are at least two numbers in them at a distance not greater than 2^{30} floating point numbers.

A computational error is said to be serious if the correct and the actual results are at a distance exceeding 2^{30} floating point numbers.

For small errors of both types, we calculated the sum of the distances between the correct and the actual result in terms of the intervals between the adjacent floating point numbers. This makes it possible to assess the general structure of such errors; in particular, one can find out how many errors are greater than 1 bit.

Table 2. Intervals of the specific behavior of the exponential function

Left endpoint	Right endpoint	Comments
$-\infty$		$exp(-\infty) = 0$
$-max$	x_1	The value of the exponential in the round toward nearest mode is 0.
$x_1 + +$	x_2	Rounding of the exponential toward $-\infty$ gives 0 and rounding toward the nearest gives a nonzero value.
$x_2 + +$	x_3	The values of the exponential are denormalized.
$x_3 + +$	$x_4 - -$	The values of the exponential are positive, normalized, and less than unity.
x_4	$x_5 - -$	The value of the exponential rounded toward $+\infty$ is equal to unity and is distinct from unity in the other round modes.
x_5	$-x_6$	The value of the exponential rounded toward $+\infty$ or toward nearest is equal to unity; the values of the argument are negative and normalized.
$(-x_6) + +$	$-min$	The values of the argument are negative and denormalized.
-0		$exp(-0) = 1$.
0		$exp(0) = 1$.
min	$x_6 - -$	The values of the argument are positive and denormalized.
x_6	$x_7 - -$	The value of the exponential rounded toward $-\infty$ or toward nearest is equal to unity; the values of the argument are positive and normalized.
x_7	$x_8 - -$	Rounding of the exponential toward $-\infty$ gives 1 and rounding toward nearest gives a different value.
x_8	x_9	The value of the exponential is greater than unity and does not exceed max.
$x_9 + +$	max	The value of the exponential is $+\infty$ in the round modes toward nearest and toward $+\infty$
$+\infty$		$exp(+\infty) = +\infty$.
NaN		$exp(NaN) = NaN$.

TESTING RESULTS

Below, we describe the platforms used in the testing and provide additional information concerning the testing results. A description of various types of errors discovered for different platforms is presented in Table 6. Altogether, nine types of errors and inaccuracies were distinguished (the errors of types 2', 3', and 3'' are of the same nature as the errors of types 2 and 3).

Table 3. The notation used in Table 2

Notation	Value
min	$1.0_{16} \times 2^{-1074}$
max	$1.FFFFFFFFFFFFF_{16} \times 2^{1023}$
x_1	$-1.74910D52D3052_{16} \times 2^{9}$
x_2	$-1.74385446D71C4_{16} \times 2^{9}$
x_3	$-1.6232BDD7ABCD3_{16} \times 2^{9}$
x_4	$-1.0_{16} \times 2^{-53}$
x_5	$-1.0_{16} \times 2^{-54}$
x_6	$1.0_{16} \times 2^{-1022}$
x_7	$1.0_{16} \times 2^{-53}$
x_8	$1.0_{16} \times 2^{-52}$
x_9	$1.62E42FEFA39EF \times 2^{9}$

The right bottom cell of Table 6 presents the results describing the computational inaccuracies. In parentheses after the platform symbol, we give the maximal distance between the correct and the actual result in terms of the intervals between the adjacent double precision floating point numbers, If this distance does not exceed two, the superscript indicates the number of situations in which this distance is equal to two. For platforms F and G, no such results are presented because the inaccuracies found in our tests are too numerous.

• Platform A is a dedicated POSIX compatible operating system (OS) for Intel Pentium II.

Platform B corresponds to various versions from (2.1.3 to 2.3.2) of the glibc library under RedHat Linux of different versions on Intel Pentium II, Pentium 4, and Xeon. All the results for this platform are completely identical; i.e., the number of errors, the values of the arguments on which these errors occur, and the computed values of the function are identical.

Table 4. The results of testing the implementations of the exponential in all round modes

Platform	Number of tests	Detected errors and inaccuracies					
		Incorrect class of the result			Computational errors		
		Serious	Small		Serious	Small	
			Number	Sum of distances		Number	Sum of distances
A	15607	8	0	0	0	6080	6080
B	15607	0	48	48	0	6080	6080
C	15607	0	48	48	0	6197	6197
D	15607	32	16	16	32	7226	7226
E	15607	0	42	42	0	7373	7373
F	15607	301	91	143386	5572	1739	$\sim 3 \times 10^8$
G	15607	230	21	22	5937	1805	$\sim 3 \times 10^8$
H	15607	0	54	54	0	8031	8336
I	15607	0	54	54	0	7682	7685
J	15607	0	54	54	0	7500	8146
K	15607	8	54	54	0	9373	302308
L	15607	0	54	54	0	10598	77920

Platform C corresponds to various versions (from 2.3.2 to 2.3.5) of the glibc library under RedHat Linux of different versions on AMD Athlon XP. All the results for this platform are also identical.

Platform D corresponds to OS Solaris 10 (SunOS 5.10) on the Sun UltraSpark IIIi processor.

Platform E corresponds to FreeBSD 5.4 on Intel Pentium 4.

On the whole, the testing results are similar on all these platforms. Serious errors were detected only on platforms A and D; small errors in the class of the result are almost identical on platforms B, C, D, and E (see Table 6). The computational inaccuracies are small and do not exceed 1 bit.

The computational inaccuracies on platforms A and B are identical, which is probably explained by the use of the same procedure for computing the exponential that was recommended for the Intel processors as early as in the times of i386: $exp(x) = 2^{(x log_2 e)}$. This procedure produces incorrect results for $+\infty$ and $-\infty$ (error of type 1) on platform A. In glibc, these cases are processed separately (see the comments to glibc in [59]).

On platform D an error of type 2 (and also of type 2') was detected that is characteristic only of this platform. Due to this error, $exp(x) < x$ for large positive x.

• Platform F corresponds to OS SUSE 10.0 (glibc 2.3.5) on the IBM PowerPC 750 processor.

Platform G corresponds to OS SUSE 10.0 (glibc 2.2.5) on the IBM s390 processor.

The testing results for these platforms are amazing. In the round mode toward nearest, all the results are absolutely correct: not a single error and even not a single computational inaccuracy were discovered. This fact was also verified using a wider set of data consisting of more than 38000 values. However, in the other round modes, arbitrarily large errors can occur. It seems that such a behavior is explained by the specific implementation of the exponential for the IBM processors, which operates completely correctly in the round mode toward nearest but is poorly debugged for the other round modes.

On platform F, the most serious errors occur in the round modes toward nearest, and toward $-\infty$. In the round mode toward $+\infty$, considerable computational error were detected that do not change the class of the result. On platform G, the situation is reverse. For example, on platform F in the round modes toward 0 and toward $-\infty$, many values of the exponential are negative; moreover, fairly large (in absolute value) results are encountered. For example, $exp(1.8440884407690585)$ in the round mode toward $-\infty$ is evaluated as $-1.3766469207992498 \cdot 10^6$. Other amazing results for platform F are as follows: $exp(-0.75067340262097160)$ is $1.11268633387184180 \cdot 10^{15}$ in the round mode toward $-\infty$; $exp(1)$ is 4.00921547900944878 in the round mode toward $-\infty$.

• Platform H is OS Microsoft Windows XP (the library Microsoft Visual Studio 6, linked in the debug mode) on Intel Pentium M.

Platform I is OS Microsoft Windows 2000 (the library Microsoft Visual Studio.NET 2003, linking in the debug mode) on Intel Pentium 4.

Table 5. The results of testing the implementations of the exponential in the round mode toward nearest

Platform	Number of tests	Detected errors and inaccuracies					
		Incorrect class of the result			Computational errors		
		Serious	Small		Serious	Small	
			Number	Sum of distances		Number	Sum of distances
A	2554	2	0	0	0	1128	1128
B	2554	0	0	0	0	1128	1128
C	2554	0	0	0	0	1130	1130
D	2554	16	0	0	0	1152	1152
E	2554	0	0	0	0	1128	1128
F	2554	0	0	0	0	0	0
G	2554	0	0	0	0	0	0
H	2554	0	19	19	0	1267	1267
I	2554	0	5	5	0	1197	1197
J	2554	0	3	3	0	1296	1296
K	2554	2	18	18	0	1825	73709
L	2554	0	19	19	0	1788	9389
M	2554	2	1	214	0	1609	37316
N	2554	0	1	214	0	1609	37316
O	2554	0	0	0	0	1153	1153
P	2554	0	0	0	0	1117	1117
Q	2554	0	0	0	0	1147	1147

Platform J is OS Microsoft Windows XP-64 (the 64-bit library Microsoft Visual Studio.NET 2005) on AMD Athlon-64 X2.

Platform K. Here, the results of performing the tests under various versions of Microsoft Windows (XP and 2000) on Intel processors (Pentium 4 and Pentium M) for the libraries Microsoft Visual Studio 6 and 7 (.NET 2003) linked in the release mode are presented. These results are completely identical.

Platform L. Here, the results of performing the tests under Microsoft Windows 2000 on Pentium 4 for the 32-bit libraries Microsoft Visual Studio.NET 2005 and Intel C++ Compiler 9.1 are presented. The results are identical.

When a project is linked in Microsoft Visual Studio, one of the two link modes may be chosen: debug or release. In versions 6 and 7 (.NET 2003), different libraries of mathematical functions are used in these two modes. They produce different results for the same functions and for the sane arguments. In the release mode, there are more errors. There is a flag in the advanced linking options in Visual Studio that regulates the correctness of floating point computations (Floating-Point Consistency). In the release mode, this flag is on by default, which is the cause of differences in the results.

On the whole, the results on these platforms are similar and demonstrate two tendencies: the results of the debug versions of Visual Studio improve from version 6 to version 7; and they improve slightly more in 64-bit version 8 (although the number of 2-bit errors in the last case is greater).

The release versions of the libraries in versions 6 and 7 are identical. They contain one serious error of type 1 and many inaccuracies. Errors up to 680 bits are encountered, which implies the incorrectness of the last five decimal digits. It seems that the 32 bit libraries in version 8 were developed on the basis of the libraries used in versions 6 and 7; some errors were corrected, ad the errors became less significant although slightly more numerous.

Platform M is an implementation of Sun Java version 1.1.8 under Microsoft Windows 2000.

Platform N is Microsoft .Net 1.1 under Microsoft Windows 2000.

Platform O. Here, the results of performing the tests for Java of various versions beginning with 1.2 (1.2, 1.4, and 1.5) under various operating systems (Microsoft Windows 2000, SUSE 10.0, and RedHat 9.3) of different manufacturers (Sun, IBM, Bea) are presented. All these results are completely identical.

Table 6. The detected types of errors and inaccuracies on various platforms

Type of error	Error or inaccuracy	Platforms
Serious change in the class of the result	1. $exp(\pm\infty)$ returns *NaN*	A, K, M
	2. $exp(x > x_9)$ returns the greatest single precision number ($3.4028234663852885 \times 10^{38}$) in the round modes toward nearest and toward $+\infty$	D
	3. Incorrect results for many arguments with a considerable change in the class of the result in the round modes toward $+\infty$, 0, and $-\infty$	F, G
Small change in the class of the result	4. $exp(x > x_9)$ returns $+\infty$ in the round modes toward 0 and toward $-\infty$	B, C, D (for large numbers), I (for large numbers), J
	5. $exp(x > x_9)$ returns the greatest double precision number in the round modes toward nearest and toward $+\infty$	H, I (sometimes), K, L
	6. $exp(x \le x_1)$ returns 0 in the round mode toward $+\infty$	B, C, D, E (sometimes), F, G, H, I, J, K, L
	7. for $x_1 < x \le x_2$, $exp(x)$ returns a nonzero result in the round mode toward nearest	H, I, J, K (sometimes), L
	8. for $x_1 < x \le x_2$, $exp(x)$ sometimes returns a nonzero value in the round modes toward 0 and toward $-\infty$	K
	9. $exp(x_9)$ returns $+\infty$	M, N
	3'. Incorrect results for many arguments (the result may fall in an adjacent class) in the round modes toward $+\infty$, toward 0, and toward $-\infty$ (see 3)	F, G
Serious computational error	2'. $exp(x > x_9)$ returns the greatest single precision number ($3.4028234663852885 \times 10^{38}$) in the round modes toward 0 and toward $-\infty$	D
	3". Incorrect results for many arguments (the class of the result is preserved) in the round modes toward $+\infty$, toward 0, and toward $-\infty$ (see 3)	F, G
Small computational error		K(680), M(333), N(333), L(17), J(2^{646}), H(2^{306}), I(2^3), A(1), B(1), C(1), D(1), E(1), O(1), P(1), Q(1)

Platforms P and Q are, respectively, Microsoft .NET 2.0 (32-bit version under Windows 2000) and .NET 2.0 (64-bit version under Windows XP).

Java and .NET perform computations only in the round mode toward nearest. Therefore, the tests for these platforms were performed only for this round mode.

Among them, only platform M contains a serious error of type 1, which was corrected in the next version of Java.

Platforms M and N contain the same error of type 9, and all the computational inaccuracies in them are completely identical. It seems that Microsoft developed the first version of the mathematical functions for .NET on the basis of Java version 1.

The identical results for Java beginning with version 1.2 are explained by the stricter standard for the implementations of mathematical functions introduced by Sun in this version of Java.

On all the other platforms, there are only computational errors; all of them are 1-bit errors; and they are not numerous.

The results obtained show that all the serious errors (except for the very strange results for platforms F and G) were detected using the tests constructed on the basis of the analysis of intervals of the homogeneous behavior of the exponential function. The additional 10000 tests for the accuracy of the computations make it possible to estimate the general form of the computational errors and their distribution on he whole. Thus, both components of the test suite are a good complement of each other and make it possible to test all the aspects of the behavior of implementations of mathematical functions.

5. CONCLUSIONS

The diversity of applications of mathematical modeling, the expanding requirements for their reliability and accuracy, and the necessity to get the same results on various platforms make the problem of standardizing the evaluation of mathematical functions very urgent. Despite thorough studies in this field (see, e.g., [7, 8, 18–20, 38, 49–52]) in the framework of the Arenaire project [6] and the recent adoption of ISO 10967 [3–5], which determines the requirements for the implementations of mathematical functions that are independent of software and hardware platforms and of programming languages, many issues remain unre-

solved. First, the attempts to formulate a systematic procedure for determining the requirements for the implementations of mathematical functions (see, e.g., [7, 8]) fail to take into account some details (one-sided limits at zero, neighborhoods of poles, infinities, and exact values, asymptotics, symmetries (except for evenness and oddness)). Second, there are no studies concerning the procedure of constructing test suites that systematically test all the aspects of the behavior of implementations of such functions.

In this paper, we proposed a solution of both problems. The presented method for determining the requirements for the implementations of mathematical functions is based on the ideas that were formulated in many studies; actually, it is an extension of the IEEE 745 standard for arithmetic operations to other mathematical functions. The only contribution of the author of this paper is a systematic presentation of all those ideas.

The method for test suite construction for verifying the fulfillment of those requirements described in Section 4.2 is also based on the results of some studies concerning the exact calculation of mathematical functions [20] and on the clear idea of decomposing the domain of the function to be tested into intervals on which the function behavior is homogeneous. However, I am not aware of the works describing the construction of test suites for mathematical functions that take into account all the aspects discussed in this paper.

The results of applying the proposed method for the fairly simple exponential function (they are described in Section 4.3) show that there is still a far way to the practical implementation of strict standards for the implementations of mathematical functions. For the round toward nearest mode, most implementations produce good results (there are only minor computational inaccuracies). However, other round modes present severe difficulties. The practical importance of the support of other round modes is explained by their usefulness for systems of reliable interval computations, which are very helpful in mathematical modeling of unstable problems.

The methods proposed in this paper make it possible to gradually overcome the difficulties discussed above by stating practical problems of standardizing and test suite development.

REFERENCES

1. *IEEE 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*, New York: IEEE, 1985.
2. *IEC 60559:1989. Binary Floating-Point Arithmetic for Microprocessor Systems*, Geneve: ISO, 1989.
3. *ISO-IEC 10967-1:1994. Information Technology—Language Independent Arithmetic—Part 1: Integer and Floating Point Arithmetic*, Geneve: ISO, 1994.
4. *ISO-IEC 10967-2:2002. Information Technology—Language Independent Arithmetic—Part 2: Elementary Numerical Functions*, Geneve: ISO, 2002.
5. *ISO-IEC 10967-3:2002. Information Technology—Language Independent Arithmetic—Part 3: Complex Integer and Floating Point Arithmetic and Complex Elementary Numerical Functions, Draft* Geneve: ISO, 2002.
6. http://www.inria.fr/recherche/equipes/arenaire.en.html.
7. Hanrot, G., Lefevre, V., Muller, J.-M., Revol, N, and Zimmermann, P., Some Notes for a Proposal for Elementary Function Implementation in Floating-Point Arithmetic, in *Proc. of Workshop IEEE 754R and Arithmetic Standardization, ARITH-15*, 2001.
8. Defour, D., Hanrot, G., Lefevre, V., Muller, J.-M., Revol, N., and Zimmermann, P., Proposal for a Standardization of Mathematical Function Implementation in Floating-Point Arithmetic, *Numerical Algorithms,* 2004, vol. 37, nos. 1–4, pp. 367–375.
9. *IEEE 854-1987: IEEE Standard for Radix-Independent Floating-Point Arithmetic,* New York: IEEE, 1987.
10. Goldberg, D., What Every Computer Scientist Should Know about Floating-Point Arithmetic, *ACM Comput. Surveys*, 1991, vol.23, no. 1, pp. 5–48.
11. Sternbenz, P., *Floating-Point*, Englewood Cliffs: Prentice-Hall, 1974.
12. Camlet, J. and Lefevre, V., Toward the Integration of Numerical Computations into the OMSCS Framework, in *Proc. 7th Int. Workshop on Computer Algebra in Scientific Computing (CASC 2004)*, St. Petersburg, 2004, pp. 71–79.
13. http://www.fas.org/spp/starwars/gao/im92026.htm.
14. http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.
15. *ISO-IEC 9899:1999. Programming Languages—C*, Geneve: ISO, 1999.
16. *IEEE 1003.1-2004.: Information Technology—Portable Operating System Interface (POSIX)*, New York: IEEE, 2004.
17. http://www.opengroup.org/onlinepubs/009695399/functions/sin.html.
18. Lefevre, V., Muller, J.-M., and Tisserand, A., Toward Correctly Rounded Transcendentals, *IEEE Trans. Comput.*, 1987, vol. 47, no. 11, pp. 1235–1243.
19. Lefevre, V., Muller, J.-M., and Tisserand, A., The Table Maker's Dilemma, *INRIA Research Report*, 1998, no. 98-12.
20. Lefevre, V. and Muller, J.-M., Worst Cases for Correct Rounding of the Elementary Functions in Double Precision, in *Proc. 15th Symp. on Computer Arithmetic*, Vail (Colorado), 2001.
21. Lang, S., *Algebra*, Reading: Addison–Wesley, 1965. Translated under the title *Algebra*, Moscow: Mir, 1968.
22. Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, New York: Dover, 1965. Translated under the title *Spravochnik po spetsiak'nym funktsiyam s formulami, grafikami i matemeticheskimi tablitsami*, Moscow: Nauka, 1979.
23. Muller, J.-M., *Elementary Functions: Algorithms and Implementation*, Boston: Birkhauser, 2006, 2nd ed.
24. http://www.math.utah.edu/~beebe/software/ieee/.
25. http://www.redhat.com/drepper/libm/.
26. Kahan, W., What Can You Learn about Floating-Point Arithmetic in One Hour? http://http.cs.berkeley.edu/~wkahan/ieee754status, 1996.

27. Schryer, N.L., A Test of Computer's Floating-Point Arithmetic Unit, *Computer Science Technical Report, AT&T Bell Labs*, 1981.
28. Verdonk, B., Cuyt, A., and Verschaeren, D.A., A Precision- and Range-Independent Tool for Testing Floating-Point Arithmetic I: Basic Operations, Square Roots and Remainder, *ACM TOMS*, 2001, vol. 27, no. 1, pp. 92–118.
29. Verdonk, B., Cuyt, A., and Verschaeren, D.A., A Precision- and Range-Independent Tool for Testing Floating-Point Arithmetic II: Conversions, *ACM TOMS*, 2001, vol. 27, no. 1, pp. 119–140.
30. http://www.cant.ua.ac.be/ieeecc754.html.
31. Karpinski, R. PARANOIA: A Floating-Point Benchmark, *Byte Magazine*, 1985, vol. 10, no. 2, pp. 223–235.
32. http://www.netlib.org/paranoia/.
33. Ziv, A., Aharoni, M., and Asaf, S., Solving Range Constraints for Binary Floating-Point Instructions, in *Proc. 16th Symp. on Computer Arithmetic (ARITH-16'03)*, 2003, pp. 158–163.
34. Aharoni, M., Asaf, S., Fournier, L., Koifman, A., and Nagel, R., FPgen—A Test Generation Framework for Datapath Floating-Point Verification, in *Proc. IEEE Int. High Level Design Validation and Test Workshop (HLDVT'03)*, 2003, pp. 17–22.
35. http://www.math.utah.edu/pub/elefunt/.
36. http://www.netlib.org/fp/ucbtest.tgz.
37. Liu, Z.A., Berkley Elementary Function Test Suite, *M.S. thesis*, Berkley: Computer Science Division, Department of Electrical Engineering and Computer Science, Univ. of California at Berkley, 1987.
38. Stehele, D., Lefevre, V., and Zimmermann, P., Searching Worst Cases of a One-Variable Function Using Lattice Reduction, *IEEE Trans. Comput.*, 2005, vol. 54, no. 3, pp. 340–346.
39. http://www.loria.fr/zimmerma/mpcheck/.
40. http://www.maplesoft.com/.
41. http://www.wolfram.com/products/mathematica/index.html.
42. http://www.mathworks.com/products/matlab/.
43. Ziv, A., Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit, *ACM Trans. Math. Software*, 1991, vol. 17, no. 3, pp. 410–423.
44. IBM Accurate Portable MathLib, http://rpmfind.net/linux/rpm2html/search.php?query=lib-ultim.so.2.
45. http://www.mpfr.org/.
46. http://www.sun.com/download/products.xml?id=41797765.
47. http://www.ens-lyon.fr/LIP/Arenaire/Ware/SCSLib/.
48. http://lipforge.ens-lyon.fr/projects/crlibm/.
49. de Dinechin, F., Ershov, A., and Gast, N., Towards the Post-Ultimate libm, in *Proc. 17th Symp. on Computer Arithmetic,* IEEE Computer Society, 2005.
50. Defour, D., de Dinechin, F., and Muller, J.-M., Correctly Rounded Exponential Function in Double Precision Arithmetic, *INRIA Research Report*, RR-2001-26, 2001.
51. de Dinechin, F., Lauter, C., and Muller, J.-M., Fast and Correctly Rounded Logarithms in Double Precision, *INRIA Research Report*, RR-2005-37, 2005.
52. Chevillard, S. and Revol, N., Computation of the Error Functions erf and erfc in Arbitrary Precision with Correct Rounding, in *Proc. 17th IMACS Conf. on Scientific Computation, Applied Mathematics, and Simulation*, Paris, 2005.
53. Kramer, W., Multiple-Precision Computations with Result Verification, in *Scientific Computing with Automatic Result Verification*, Adams, E. and Kulisch, U., Eds., Boston: Academic, 1993, pp. 325–356.
54. Schulte, M.J. and Swartzlander, E.E., Software and Hardware Techniques for Accurate, Self-Validating Arithmetic, in *Applications of Interval Computations*, 1996, pp. 381–404.
55. Revol, N. and Rouillier, F., Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library, *Reliable Comput.*, 2005, vol. 11, no. 4, pp. 275–290.
56. MPFI Library, http://perdo.ens-lyon.fr/nathalie.revol/mpfi_toc.html.
57. Kahan, W., Minimizing $q * m - n$, available at http://http.cs.berkley.edu/~wkahan/testpi/nearpi.c.
58. Kuliamin, V.V., Formal Approaches to Testing Mathematical Functions, in *Trudy ISP RAN*, 2006, vol. 10.
59. http://sourceware.org/cgi-bin/cvsweb.cgi/libc/sysdeps/i386/fpu/e_expl.c?cvsroot=glibc.

SPELL: 1. semiperimeter