Test Construction for Mathematical Functions[*]

Victor Kuliamin

Institute for System Programming
Russian Academy of Sciences
109004, B. Kommunistitcheskaya, 25, Moscow, Russia
kuliamin@ispras.ru

Abstract. The article deals with problems of testing implementations of mathematical functions working with floating-point numbers. It considers current standards' requirements to such implementations and demonstrates that those requirements are not sufficient for correct operation of modern systems using sophisticated mathematical modeling. Correct rounding requirement is suggested to guarantee preservation of all important properties of implemented functions and to support high level of interoperability between different mathematical libraries and modeling software using them. Test construction method is proposed for conformance test development for current standards supplemented with correct rounding requirement. The idea of the method is to use three different sources of test data: floating-point numbers satisfying specific patterns, boundaries of intervals of uniform function behavior, and points where correct rounding requires much higher precision than in average. Some practical results obtained by using the method proposed are also presented.

1 Introduction

In modern world computers help to visualize and understand behavior of very complex systems. Confirmation of this behavior with the help of real experiments is too expensive and often even impossible. To ensure correct results of such modeling we need to have adequate models and correctly working modeling systems. Constructing adequate models is very interesting problem, which, unfortunately, cannot be considered in this article in detail. The article concerns the second part – how to ensure correct operation of modeling systems. Such systems often use very sophisticated and peculiar algorithms, but in most cases they need for work basic mathematical functions implemented in software libraries or in hardware.

So, mathematical libraries are common components of most modeling software and correct operation of the latter cannot be achieved without correct implementation of basic functions by the former. In practice software quality is controlled and assured mostly with the help of testing, but testing of mathematical libraries usually is organized using simplistic ad hoc approaches and random

[*] This work is partially supported by RFBR grants 05-01-0099a, 06-07-89261a, and by contract 02.467.11.1006 with Russian Federal Science and Innovation Agency.

test data generation. Specifics of floating-point calculations makes construction of both correct and efficient implementations of functions, along with testing that they are actually correct, a nontrivial task. The main goal of this paper is to present a systematic method of test construction for mathematical functions implemented in software or hardware on the base of floating-point arithmetic stated in IEEE 754 standard [1].

The main ideas of the method proposed are to check correct rounding requirement and to combine three different sources of test data, which are targeted to catch common errors made by implementors of mathematical libraries.

- Floating-point (FP) numbers of specific structure. They include both boundary numbers like the greatest FP number (within certain precision), the smallest normalized number, etc., and numbers, which binary representations satisfy some specific patterns. In addition the closest floating-point numbers to the values of the reverse function in such points are calculated and also used as testing points for the direct function. Roughly speaking, a function is tested on points where its argument or its value are on the boundary or satisfy one of the chosen patterns.
- Boundaries of intervals, where the function under test behaves in uniform way. Several points on each of those intervals are also added to the test suite. Detailed rules for determining such intervals are presented below.
- Floating-point numbers, for which correct rounding of the function value requires much higher precision of calculations than average. These points are rather hard to seek and some methods for their search are presented in the section on test construction method.

The contribution of this article is the systematic description of the approach proposed, much more clear and concise then it was done in the paper [2] describing the starting phase of the underlying work. In addition this paper describes methods for searching FP numbers, for which correct rounding of the function value is hard, including the new one, *the integer secants method.* This research was started during the OLVER project [3] on formalization of LSB [4] standard requirements and conformance test development.

Before presenting the test construction method itself it is useful to recall some details of FP arithmetic, which are necessary for understanding details of the method. To make tests practically useful we also should clearly understand what precise requirements should be checked. So, the next section presents review of existing standards concerning mathematical functions working with FP numbers and analysis of these standards' requirements.

2 Standards' Requirements

To be able to check implementation of a function we should know how it should behave. Practically significant requirements on the behavior of functions on FP numbers can be found in several standards.

- Standards IEEE 754 [1] (also known as IEC 60559 [5]) and IEEE 854 [6] define representation of FP numbers, rounding modes, also describe basic arithmetic operations, comparisons, type conversions, square root function, and floating-point remainder.
- Standards ISO C [7] and POSIX [8] impose additional requirements on about 40 functions of real and complex variable implemented in standard C library.
- Standard ISO/IEC 10697 [9,10,11] gives more elaborated and precise set of requirements for elementary functions.

2.1 Floating-Point Numbers

Standards IEEE 754 and IEEE 854 define FP numbers based on various radices. Further exposition concerns only binary numbers, because other radices are used in practice rarely. Nevertheless, all the techniques presented can be extended to FP numbers with different radix, if it is necessary.

Representation of binary FP numbers is defined by two main parameters – n, the number of bits in the representation, and $k < n$, the number of bits used to represent an exponent. The interpretation of different bits is presented below.

- The first bit represents the sign of a number.
- The next k bits – from the 2-nd to the $k + 1-$th – represent *the exponent* of a number.
- All the rest bits – from $k + 2-$th to $n-$th – represent *the mantissa* or *the significand* of a number.

A number X with the sign bit S, the exponent E, and the mantissa M is expressed in the following way.

1. If $E > 0$ and $E < 2^k - 1$ then X is called *normalized* and is calculated with the formula $X = (-1)^S 2^{(E-2^{k-1}+1)}(1 + M/2^{n-k-1})$. Actual exponent is shifted to make possible representation of both large and small numbers. The last part of the formula is simply 1 followed by point and mantissa bits as the binary representation of X without exponent.
2. If $E = 0$ then X is called *denormalized* and is computed according to another formula $X = (-1)^S 2^{(-2^{k-1}+2)}(M/2^{n-k-1})$. Here mantissa bits follow 0 and the point. Note also, that this gives two zero values $+0$ and -0.
3. Exponent $2^k - 1$ is used to represent special values – positive and negative infinities (using zero mantissa) and *not-a-number* NaN (using any nonzero mantissa). Infinities represent results of operations that actually give mathematically infinite result or too big result to be represented as a floating-point number. NaN represents results of operations that cannot be considered consistently as finite or infinite, e.g. $0/0 = \mathrm{NaN}$.

IEEE 754 standard defines the following FP number formats: single precision ($n = 32$ and $k = 8$), double precision ($n = 64$ and $k = 11$), and extended double precision ($128 \geq n \geq 79$ and $k \geq 15$ (Intel processors use $n = 79$ and $k = 15$). In the next version of the standard quadruple precision numbers ($n = 128$ and $k = 15$) will be added.

2.2 IEEE 754 Requirements

Along with the representation of FP numbers IEEE 754 defines requirements to basic arithmetic operations on them (addition, subtraction, multiplication, and division), comparisons, conversions between different formats, square root function, and calculation of FP remainder [12]. Since results of these operations applied to FP numbers are often not exact FP numbers, it defines rules of rounding such results. Four rounding modes are defined: to the nearest FP number, up (to the least FP number greater than the result), down (to the greatest FP number less than the result), and to 0 (up for negative results and down for positive ones). If the result is exactly in the middle between two neighbor FP numbers, its rounding to nearest get the one having 0 as the last bit of its mantissa.

To make imprecise results more visible IEEE 754 defines a set of FP exception flags that should be raised in specific circumstances.

- Invalid flag should be raised if the result is NaN, while arguments of the operation performed are not NaNs.
- Divide-by-zero flag should be raised if the result is exactly positive or negative infinity, while arguments of the operation are finite.
- Overflow flag should be raised if the results' absolute value is greater than maximum FP number.
- Underflow flag should be raised if the result is not 0, while its absolute value is less than minimum positive normalized FP number.
- Inexact flag should be raised if the precise result is not an FP number, but its absolute value is inside the interval between minimum and maximum positive FP numbers.

2.3 Requirements of ISO C and POSIX

ISO C [7] and POSIX [8] standards provide description of mathematical functions of standard C library, including most important elementary functions (square and cubic roots, power, exponential and logarithm with bases $e, 2$ and 10, most commonly used trigonometric, hyperbolic functions and their reverse functions) of real or complex variables. Also some special functions are described – error function, complementary error function, gamma function, and logarithmic gamma function.

ISO C standard defines a set of points where the specified functions have exact well-known values, e.g. $\log 1 = 0, \cos 0 = 1, \sinh 0 = 0$. It also specifies situations where invalid and divide-by-zero flags should be raised, the first one – if a function is calculated outside of its domain, the second one – if the value of a function is precisely positive or negative infinity. These requirements are specified as normative for real functions and only as informative for complex functions.

POSIX slightly extends the set of described functions; it requires implementing Bessel functions of the first and the second kind of orders $0, 1$, and of an arbitrary integer order given as the second parameter. It also extends ISO C by specifying situations when overflow and underflow flags should be raised for functions in real variables. Additional POSIX requirement is that real functions having asymptotic $f(x) \sim x$ near 0 should return x for each denormalized argument value x. Note, that this would be in contradiction with IEEE 754 rounding requirements if they were applied to such functions.

Both standards do not say anything on precision of function calculation in general situation.

2.4 Requirements of ISO 10697

The only standard specifying some calculation precision for rich set of mathematical functions is ISO 10697 [9,10,11], standard on language independent arithmetic. It describes the following requirements to implementations of elementary functions in real and complex variables.

– Preservation of sign and preservation of monotonicity of ideal mathematical function where no frequent oscillation occurs. Frequent oscillation occurs where difference between two neighbor FP numbers is comparable with length of intervals of monotonicity or sign preservation. Trigonometric functions are the only elementary functions that oscillate frequently on some intervals. So, the standard defines *the big angle* – the least positive value, for which the last unit of mantissa or *ulp,* unit on the last place, becomes greater than $\pi/1000$. This value is about $2.8 \cdot 10^{13}$ for double precision. For arguments greater than the big angle preservation of sign and monotonicity does not required from implementations of trigonometric functions.
– The standard requires that rounding errors should not be greater than $0.5 - 2.0$ ulp, depending on the function implemented. Again, this in not required from implementations of trigonometric functions on arguments greater than the big angle. Note that precision 0.5 ulp is equivalent to the correct rounding to the nearest FP number.
– ISO 10697 requires to preserve evenness or oddity of implemented functions, and for this reason it does not support directed rounding modes – up and down. Only symmetric modes – to nearest and to zero – are considered as correct.
– The standards also specifies well-known exact values for all functions, extending ISO C requirements. In addition it requires from an implementation to preserve asymptotic of the implemented function in 0. FP numbers are distributed with different densities along the real axis, and their density increases while approaching 0 – the double precision number closest to 0 has value 2^{-1074}, while the one closest to 1 differs from it by 2^{-53}. For that reason, for example, an implementation of exp should return exactly 1 in some neighborhood of 0, and an implementation of sin should return x also in some neighborhood of 0.
– The last set of requirements imposed by ISO 10697 is concerned with natural inequalities between some functions, e.g. $\cosh(x) \geq \sinh(x)$, which should be preserved by their implementations.

So, ISO 10697 provides the most detailed set of requirements including requirements on calculation precision. Unfortunately, it has not yet recognized by practitioners and no widely-used library has declared correspondence with this standard. Maybe this situation will improve in future.

3 Analysis of Requirements

Analysis of existing standards shows that they are not fully consistent with each other and are usually restricted to some specific set of functions. Trying to construct some systematic description of general requirements based on significant properties of mathematical functions concerned with their computation one can get the following list.

- Exact values and asymptotic near them.
- Preservation of sign and monotonicity.
- Preservation of inequalities with other functions.
- Symmetries – evenness, oddity, periodicity, or more complex properties like $\Gamma(x+1) = x\Gamma(x)$.
- NaN results outside of function's domain, infinity results in function's poles, correct overflow and underflow detection, raising correct exception flags (extension of IEEE 754 and POSIX requirements).
- Preservation of bounds of function range, e.g. $-\pi/2 \leq \arctan(x) \leq \pi/2$, $-1 \leq \tanh(x) \leq 1$.
- Correct rounding according to natural extension of IEEE 754 rules and raising inexact flag on imprecise results.

In this list correct rounding requirement is of particular significance. It has the following important advantages.

- If we provide correct rounding, we immediately have almost all other properties in this list [13]. But if we want to preserve these properties without correct rounding, it requires much harder work, peculiar errors become possible, and thorough testing of such an implementation becomes very nontrivial and much harder task.
- Correct rounding provides results closest to the precise ones. If we have no correct rounding, it is necessary to specify how the results may differ from the precise ones, which is very rarely done in practice. It is supposed usually that correct rounding for sine function on large arguments is too expensive, but none of sine implementations used in practice explicitly declares its error bounds on various intervals. Most users usually don't analyze the results obtained from standard mathematical libraries, and are not competent enough to see the boundaries between areas where their results are relevant and the ones where they become irrelevant due to (not stated explicitly) calculating errors in standard functions. Correct rounding moves most of the problems of error analysis to the algorithms used by the applications, standard libraries become as precise as it is possible.
- Correct rounding requirement also implies almost perfect compatibility of different mathematical libraries and precise repeatability of calculation results of modeling applications on different platforms, which means very good

portability of such applications. This goal is also rather hard to achieve without such a requirement – one needs to standardize specific algorithms as it was made by Sun in mathematical library of Java 2. Also, strict precision specification is much more flexible requirement than standardization of algorithms.

High effort required to develop a function implementation and its resulting ineffectiveness are always mentioned as drawbacks of correct rounding requirement. However, good algorithms and techniques that help to resolve these issues are already known for a long time (e.g. see [14,15] for correct argument reduction for trigonometric functions). Work of Arenaire group [16] in INRIA on **crlibm** [17,18] library demonstrates that inefficiency problems can be resolved in almost all cases. So, now these drawbacks of correct rounding can be considered as not really relevant.

More serious issue is contradiction between correct rounding requirement and some other useful properties of mathematical functions. In each case of such a contradiction we should decide how to resolve it.

– Oddity and some other symmetries using minus sign or taking reciprocal values, like $\tan(\pi/2 - x) = 1/\tan(x)$, can be broken by directed rounding modes (up and down), while symmetric modes (to nearest and to 0) preserve them. In this case it is natural to prefer correct directed rounding if it is chosen, because usually such modes are used to get correct boundaries on exact results.
– Correct rounding can sometimes contradict with natural boundaries of function range, if these boundaries are not representable as precise FP numbers. For example, $-\pi/2 \leq \arctan(x) \leq \pi/2$ is an important property. It occurs that single precision FP number closest to $\pi/2$ is greater than it, so if we round arctangent values on large arguments to the nearest FP number, we get $\arctan(x) > \pi/2$, that can radically change the results of modeling of some complex systems. In this case we prefer to give priority to the bounds preservation requirement and do not round values of arctangent (with either rounding mode) to FP numbers out of its range.

So, further we consider test construction to check correct rounding requirement with 4 rounding modes specified by IEEE 754 with the single exception – when correct rounding breaks natural boundaries of a function range, we preserve these boundaries. The reader will see that even for implementations that do not satisfy these requirements such tests can also be very useful.

3.1 Table Maker Dilemma

An important issue related with correct rounding requirement is so called *table maker dilemma* [19,20]. It occurs when we need much higher precision of calculations to get correctly rounded value of a function. An example is the value of natural logarithm of a double precision FP number $1.613955\text{DC}802\text{F}8_{16} \cdot 2^{-35}$ (mantissa is represented in hexadecimals) equal to $-17.\text{F02F9BAF6035 7F}^{14}9\ldots_{16}$. Here

F^{14} means 14 digits F, giving with neighbor digits 60 consecutive units staying after a zero just after the double precision mantissa. This value is very close to the mean of two neighbor FP numbers, and to be able to round it correctly to the nearest FP number we need calculations with relative error bound about 2^{-113} while 0.5 ulp precision is only 2^{-53}.

Simple statistical model [19] presuming that mantissa's bits of function values are distributed uniformly and independently implies that 2^{n-k-1} FP numbers with one exponent can give $2^{n-k-1} \cdot 2^{-m+1}$ values with m consecutive equal bits – *bad cases,* in which table maker dilemma occurs. So, if $m \leq n-k$ (24 for single and 53 for double precision), then there may exist points where correct rounding of a function requires precision about 2^{m+n-k}.

Experiments (see below methods for bad cases search) shows that this is true in common case, on the intervals where a function has no singularities or simple asymptotic like $\cos x \sim 1$ or $\sin x \sim x$. But extraordinary bad cases also exists for most functions – m can be greater than 60 for double precision.

4 Test Construction Method

Test construction method proposed checks difference between correctly rounded value of a function and the value returned by its implementation in a set of test points. We prefer to have a rules of test point selection based only on the properties of the function under test and structure of FP numbers, and do not consider specific algorithms of implementations. This black box approach appears to be rather effective in revealing errors in practice, and at the same time it does not require detailed analysis of numerous and growing set of possible implementation algorithms and various errors that can be made in them.

Test points are chosen by the following rules.

1. **FP numbers of special structure:**
   First, natural boundary values in the set of FP numbers are taken as test points: $0, -0, \infty, -\infty$, NaN, the least and the greatest positive and negative denormalized and normalized numbers. This boundary values usually uncover errors in non-mature implementations. For example, the procedure recommended by Intel to calculated exponential function on Intel processors older than Pentium II gives NaN in $\pm\infty$, instead of 0 and $+\infty$ (see [21])

   Second, numbers with mantissa satisfying some specific patterns are chosen. Errors in an algorithm or an implementation often lead to incorrect calculations on some patterns. The notorious Pentium division bug [22] can be detected only on divisors having units as mantissa bits from 5-th to 10-th. In out practice an algorithm of square root calculation encoded in hardware made errors on about 10% of double precision numbers, square roots for which have mantissa of a form ****FFFFFFFFF, where * means an arbitrary hexadecimal digit. Pattern use for testing FP calculations is already described, e.g. in [23].

   So, several dozens different patterns are chosen, including all zeroes, all units, 0FFFF0000AAAA, and some others with alternating groups of digits

0, F, 5 (0101), and A (1010). On each exponent where the function under test is defined and is not degenerate (that is its values are in the range of representable numbers and do not all equal to one constant value or to function's argument) we take points with mantissas satisfying these patterns.

Third, two previous rules are used to get points where reverse function is calculated and pairs of closest FP numbers to its values are taken as test arguments for direct function. So, a function is tested in points, which satisfy some patterns, and in points where its value is closest to the same patterns.

2. **Boundaries of intervals of specific function behavior:**
   All singularities of the function under test, bounds of intervals of its non-overflow behavior, of constant sign, of monotonicity or simple asymptotic determine some partitioning of FP numbers. Boundaries of these intervals and several points on each of them are chosen as test points. They also usually reveal various errors.

   In case of frequent oscillation (e.g. for trigonometric and Bessel functions) this approach faces with an enormous number of intervals to be covered. To resolve this problem we choose only intervals where extreme values of the function are closest to its actual extreme values. For example, for trigonometric functions one can found the set of FP numbers closest to integer multiples of $\pi/2$ with the help of continued fractions [15]. This method gives about 2000 points for double precision numbers. On quarter-periods containing these points trigonometric functions approach to $0, \pm 1$, or $\pm\infty$ much closer than on ordinary quatre-period. So, these quatre-periods are taken as characteristic intervals for those functions.
3. **FP numbers, for which calculation of correctly rounded function value requires higher precision:**
   Bad cases, which require more than M additional bits for correct rounding (the "badness"), are taken as test points. M is chosen equal to $n - k - 10$, which gives about 1000 test points on each exponent where the function under test is not degenerate. In addition some points with M near $(n - k)/2$ are chosen too, because some errors can be uncovered in not-very-bad cases ([24] gives an example of such an error in an implementation of square root). This rule adds test points helping to reveal calculation errors and inaccuracies of various nature.

Implementation of the method is rather straightforward. Test points are gathered into simple text files, each test point is accompanied with correctly rounded value of the function under test for each rounding mode (only two different values are required at most). Correctly rounded values are calculated with the help of multiprecision implementations of the same functions (e.g. in Maple or MPFR library [25]), taking higher precision to guarantee correct results. A test program reads test data, calls the function under test, and compares its result with the correct one. In case of discrepancy the difference in ulps is counted and reported. In addition the test program checks exception flags raising according to IEEE 754 rules extended to the function under test. So, test execution is completely automated.

The only not so easy task is to compute bad cases for a function. Methods that solve this problem are presented in the next section.

4.1 Bad Cases Computation

The following techniques can be used for bad case computation.

– Exhaustive search. In practice it can be applied only for single precision numbers (they have less than 2^{32} values). Almost 2^{64} double precision numbers and the need to calculate the function under test with high precision for each number make this technique unfeasible for higher precisions even with use of the most powerful modern computers.

– *Dyadic method* [24,26]. This method gets argument bad cases on the base of the function values in dyadic numbers in some points. It can be used to compute bad cases for algebraic functions – for square root [24,26] and for cubic root (no references).

  For example, if square root of FP number X is near the mean of two FP number, then $\sqrt{2^m N} \approx M + 1/2$, where we can consider M, N as integers between 2^{n-k-1} and 2^{n-k}, $X = 2^m N$, m is $n-k-1$ or $n-k-2$. By squaring this almost-equality we get integer equality $2^{m+2}N = (2M+1)^2 - j$, where j is a small integer number. So $(2M+1)^2 = j \bmod 2^{m+2}$ and $2M+1$ can be found as a square root of j modulo 2^{m+2}. The last task can be solved by consecutive computing square roots of j modulo $8, 16, 32, \ldots 2^{m+2}$ with the help of Henzel lifting (they give an infinite sequence tending to the dyadic square root of j).

  The same considerations can be applied to directed rounding bad cases [24] and to cubic root.

– *Reduced search* [19,27]. This method searches bad cases as FP-numbers grid nodes closest to the function under test graph. The function is approximated with high precision by linear polynomials on a set of intervals, and then its graph is substituted by straight line segments corresponding to those polynomials. For each segment the grid nodes closest to it are determined using 3-distance theorem. This theorem says that for a given sequence $x_0, x_1, \ldots$ of points on a circle, where x_0 is an arbitrary point and x_{i+1} is obtained from x_i by rotation on some angle α, not depending on i, there are always at most three different distances between neighbor points (see details in [27]). Intersections of a straight line segment with vertical lines of FP-numbers grid make up such a sequence being regarded modulo the distance between horizontal lines. 3-distance theorem thus help to perform only several simple operations for each FP point to learn whether it gives the value close to the segment or not. All suspicious points are stored and on the second phase the function is calculated in them with high precision to get really bad cases.

– *Lattice reduction*[28]. The idea is the same – to look for FP-numbers grid nodes closest to the function under test graph. In difference with the previous approach this one uses high precision approximations of the function under test by polynomials of small degree (but not linear). Then it searches the closest approximation of this polynomial values in FP points by polynomials with integer coefficients with the help of *lattice reduction* – search of the shortest vector in a lattice of the same-degree polynomials with integer coefficients. Integer roots of found integer polynomial correspond to points of FP grid near the graph of the source polynomial (see details in [28]).

– *Integer secants method*. This is a new method proposed by the author. The idea is again to look for FP-numbers grid nodes closest to the function under test graph. But instead of search-based approaches more direct calculation is provided. this direct calculation is possible in a neighborhood of a point where the tangent $ax + b$ to the function graph has integer first coefficient a (or a is a reciprocal of an integer). Straight lines-secants parallel to this tangent on the distances of integer multiples of FP grid step cover all grid nodes. So, the closest grid nodes are near graph's intersections with these secants (see Fig. 1).

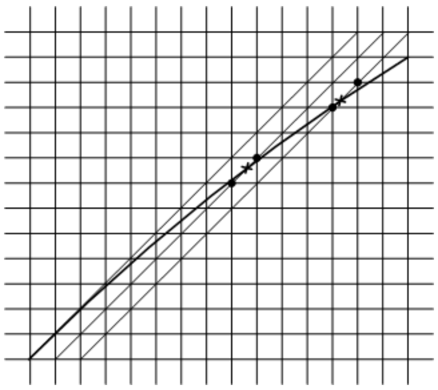

Fig. 1. Integer secants method

To compute such intersections consider the distance between the function graph and its tangent as a new function $F(x) = f(x) - ax - b$. This function has Taylor series in the point of contact starting from the square (or cubic) term, since it is the difference between Taylor series of the initial function f in the point of contact and the polynomial $ax + b$ of the contacting tangent. This series can be reversed in two steps – by computing a series that is a square (or cubic) root of this series, and then by reversing the root series (it is possible since root series starts from linear term). Having this reverse function $H(y)$ we can directly compute abscissas of intersection points – they are mere values of this function on integer multiples of grid step, since the distance between the graph and the tangent in these points is exact multiple of the distance between horizontal lines of FP grid.

This method helps to compute bad cases near integer tangents rather quickly, but with linearly growing distance from the contact point the "badness" (number of additional bits of function value to calculate for correct precision) of points decreases linearly while the number of points themselves increases exponentially. For example, for sine function near 0 and double precision numbers on exponents from -26 to -12 about 10^9 bad points can be found, but the number of points with "badness" greater than 40 is about 70000. The empiric rule for sine is that the number of intersections on each next exponent is multiplied by 8, but the number of points with certain "badness" remains almost the same.

The presented approaches gives a way to automate test point calculation. The only drawback of existing methods is that they require to write many different programs for one function under test to compute bad cases in different intervals.

5 Applications

The test construction method presented above along with test data generation methods have been applied to make test suites for **sqrt, exp, sin,** and **atan** functions in single and double precision. The tests have been executed on various platforms including Windows XP (MS Visual Studio 2005 libraries) and Linux (**glibc** library of different versions).

Surprisingly big number of various errors has been found, most of them are related with incorrect results on the boundaries of intervals where values of the function are representable as FP numbers or with big angles for sine. FP exception flags raising errors are also often uncovered in the situations of the first kind. Bad cases usually reveal small calculations errors and argument reduction errors for trigonometric functions.

Square root implementations are more mature and have only one-bit calculation errors. Arctangent implementations are different – the one of Mircosoft Visual Studio runtime libraries has only one-bit errors (for rounding to nearest only every fourth bad case reveal such an error, while for other rounding modes – every second one), while glibc implementations of arctangent have more serious errors especially for rounding up, down and to zero.

Versions of **glibc** library are partitioned into two groups – in the first group calculation of functions like square root and exponential have mostly one-bit errors in all rounding modes, but sine became highly erroneous on large arguments; in the second group calculations are precise for rounding to nearest, but can have big errors in other rounding modes, breaking even the basic properties like $\exp x >= 0$ and $-1 \leq \sin x \leq 1$. Examples of the first group are **glibc** 2.1.3, 2.3.2, 2.7 in RedHat Fedore Core distributions – sine implementation in them demonstrates big argument reduction errors, small bad cases reveal only 1-bit calculation errors in about 5% cases in double precision. Examples of the second group are **glibc** 2.3.4 in RHEL 4.0 or **glibc** 2.3.5 in SUSE 10.0 – sine implementation is almost absolutely correct in rounding to nearest mode (only flag setting errors detected in it), but is highly erroneous in other rounding modes (sometimes sine results exceeds 10^{15}!) Even single precision sine sometimes is greater than 1. Bad cases reveal small calculation errors in double precision in about 15% cases. Implementation of sine in Visual Studio.NET 2005 shows argument reduction errors and confusingly incorrect calculation of sine for negative numbers – almost all results for negative arguments are incorrect. Bad cases reveal small calculation errors in about 38% cases.

Systematic exposition of testing results for 17 different implementations of exponential function can be found in [2]. This work revealed about 10 different error kinds, some of them being specific for a single platform, but 3-4 common for most platforms tested. Usually exponential is implemented without significant errors, but bad cases still found small errors in about 12% cases. On **glibc** implementations of the second group exponential function can be highly erroneous for rounding to zero or to infinities. Several points found where its results are negative or much greater than 1 for negative arguments.

The main result is that tests based on structure of FP numbers and intervals of uniform behavior of the function under test are very good for finding various errors, while bad cases for correct rounding help to assess calculation errors in whole and general distribution of inaccuracies.

6 Conclusion

The approach presented in the paper helps to construct systematic test suites for floating-point based implementations of various mathematical functions in one real variable. Error-revealing power of such test suites is rather high – many errors were found in mature and widely used libraries. Although test suites obtained check correct rounding requirement, they also give important information about implementations that do not obey this restriction. For example, the implementations demonstrating only one-bit errors (difference with the correct result only in the last bit of mantissa) can surely be classified as more mature and correct than others.

The search for bad cases requires a lot of machine time, and now the author has explored only neighborhood of zero for all elementary functions in one variable mentioned in POSIX, except for trigonometric and hyperbolic arccosine, and neighborhood of infinity for some of these functions (exponential, arctangent, hyperbolic tangent). Some test data were taken from tests of **crlibm** library [18], which comprise a part of worst cases found by methods described in [19,20,27,28]. Only last year a method has been proposed to compute bad cases for trigonometric functions on large arguments [29]. So, to obtain the full data on bad cases a lot of work is still required.

The experiments conducted demonstrated that most of the test data can be excluded from a test suite without any loss of its error detection power. In particular, for exponential function the test suite constructed using the method described and consisting of about 3.7 million test cases, and the reduced test suite of about 10000 test cases detect all the same errors. The reduction was made by simple removing the test points that do not reveal specific errors or do not add something new to inaccuracy distribution on available 17 implementations of exponential. Now the author tries to formulate a set of rules that can help to reduce test suites without losses in errors revealed.

The interesting problem is to extend the method proposed for functions in two or more variables (and so, in complex variables). One idea is rather straightforward – it is necessary to use not intervals, but areas of uniform behavior of the function under test. But extension of rules concerning FP numbers of special structure and bad cases seem to be much more peculiar, since their straightforward generalizations gives zillions of test cases without any hope to get all the data in a reasonable time. So, some reduction rules should be introduced here from the very beginning to obtain observable test suites.

The tests developed with the presented approach can also facilitate and simplify construction of correct mathematical libraries giving more adequate and precise means for evaluation of their correctness.

References

1. IEEE 754-1985. IEEE Standard for Binary Floating-Point Arithmetic. IEEE, NY (1985)
2. Kuliamin, V.: Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers. Programming and Computer Software 33(3), 154–173 (2007)
3. `http://linuxtesting.org`
4. `http://linux-foundation.org`
5. IEC 60559. Binary Floating-Point Arithmetic for Microprocessor Systems. Geneve, ISO (1989)
6. IEEE 854-1987. IEEE Standard for Radix-Independent Floating-Point Arithmetic. IEEE, NY (1987)
7. ISO/IEC 9899. Programming Languages - C. Geneve: ISO (1999)
8. IEEE 1003.1-2004. Information Technology - Portable Operating System Interface (POSIX). IEEE, NY (2004)
9. ISO/IEC 10967-1. Information Technology - Language Independent Arithmetic - Part 1: Integer and Floating Point Arithmetic. Geneve, ISO (1994)
10. ISO/IEC 10967-2. Information Technology - Language Independent Arithmetic - Part 2: Elementary Numerical Functions. Geneve, ISO (2002)
11. ISO/IEC 10967-3. Information Technology - Language Independent Arithmetic - Part 3: Complex Integer and Floating Arithmetic and Complex Elementary Numerical Functions. Draft. Geneve, ISO (2002)
12. Goldberg, D.: What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
13. Defour, D., Hanrot, G., Lefevre, V., Muller, J.-M., Revol, N., Zimmermann, P.: Proposal for a standardization of mathematical function implementation in floating-point arithmetic. Numerical Algorithms 37(1-4), 367–375 (2004)
14. Ng, K.C.: Arguments Reduction for Huge Arguments: Good to the Last Bit (1992), `http://www.validlab.com/arg.pdf`
15. Kahan, W.: Minimizing q*m − n, Unpublished (1983), `http://cs.berkeley.edu/~wkahan/testpi/nearpi.c`
16. `http://www.inria.fr/recherche/equipes/arenaire.en.html`
17. de Dinechin, F., Ershov, A., Gast, N.: Towards the post-ultimate libm. In: Proc. of 17th Symposium on Computer Arithmetic, IEEE Computer Society Press, Los Alamitos (2005)

18. http://lipforge.ens-lyon.fr/www/crlibm/
19. Lefèvre, V., Muller, J.-M., Tisserand, A.: The Table Maker's Dilemma. INRIA Research Report 98-12 (1998)
20. Lefèvre, V., Muller, J.-M.: Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In: Proc. of 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA, June (2001)
21. http://sourceware.org/cgi-bin/cvsweb.cgi/libc/sysdeps/i386/fpu/e_expl.c?cvsroot=glibc
22. Edelman, A.: The Mathematics of the Pentium Division Bug. SIAM Review 39(1), 54–67 (1997)
23. Ziv, A., Aharoni, M., Asaf, S.: Solving Range Constraints for Binary Floating-Point Instructions. In: Proc. of 16-th IEEE Symposium on Computer Arithmetic (ARITH-16 2003), pp. 158–163 (2003)
24. Parks, M.: Number-Theoretic Test Generation for Directed Rounding. IEEE Trans. on Computers 49(7), 651–658 (2000)
25. http://www.mpfr.org
26. Kahan, W.: A Test for Correctly Rounded SQRT. Computer Science Dept, Berkeley (1994), http://www.cs.berkeley.edu/~wkahan/SQRTest.ps
27. Lefèvre, V.: An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. In: Developments in Reliable Computing, pp. 203–212. Kluwer, Dordrecht, Netherlands (1999)
28. Stehlé, D., Lefèvre, V., Zimmermann, P.: Worst cases and lattice reduction. In: Proc. of the 16th Symposium on Computer Arithmetic (ARITH'16), pp. 142–147. IEEE Computer Society Press, Los Alamitos (2003)
29. Hanrot, G., Lefèvre, V., Stehlé, D., Zimmermann, P.: Worst Cases of a Periodic Function for Large Arguments. INRIA Research Report 6106 (2007)