Standardization and Testing of Mathematical Functions

Victor Kuliamin

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

Abstract. The article concerns problems of formulating standard requirements to implementations of mathematical functions working with floating-point numbers and conformance test development for them. Inconsistency and incompleteness of available standards in the domain is demonstrated. Correct rounding requirement is suggested to guarantee preservation of all important properties of functions and to support high level of interoperability between different mathematical libraries and software using them. Conformance test construction method is proposed based on different sources of test data: numbers satisfying specific patterns, boundaries of intervals of uniform function behavior, and points where correct rounding needs much higher precision than in average. Analysis of test results obtained on various implementations of POSIX mathematical library is also presented.

1 Introduction

Computers now are widely used in physics, chemistry, biology, social sciences to model and understand behavior of very complex systems, which can hardly be examined in any other way. Confirmation of such models' correctness by experiments is too expensive and often even impossible. To ensure accurateness of this modeling we need to have adequate models and correctly working modeling systems. The article is concerned with the second problem – how to ensure correct operation of modeling systems. Such systems are often based on very sophisticated and peculiar numeric algorithms, and in any case they use mathematical functions implemented in software libraries or in hardware.

Thus mathematical libraries are common components of most simulation 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 often uses simplistic ad hoc approaches and random test data generation. Specifics of floating-point calculations make construction of both correct and efficient implementations of functions along with their testing a nontrivial task. This paper proposes an approach for standardization of floating-point calculations beyond the bounds of IEEE 754 standard [1] and presents a

systematic method of conformance test construction for mathematical functions implemented in software or hardware.

The core of the standardization proposed is correct rounding requirement. It means that an implementation of a function is always required to provide results, which are mathematically precise values correctly rounded to floating-point (FP) numbers according to the current rounding mode. Obeying such a requirement gives an easy way to preserve almost all useful properties of a function implemented, and, more important, it provides an unprecedented interoperability between various mathematical libraries and modeling software using them. Now this interoperability is far from the high level.

The test construction method presented checks correct rounding requirement and uses three different sources of test data – floating-point numbers of specific structure, boundaries of intervals where the function under test behaves in uniform way, and floating-point numbers, for which correct rounding of the function value requires much higher precision of calculations than in average. All these sources are concerned with common errors made by developers of mathematical libraries, which is confirmed both by the practical experience and by the results of tests developed according to this method also presented in the paper.

The main contribution of this article in comparison with [2] and [3] is precise formulation of requirements proposed for standardization and the presentation of considerable testing statistics, which demonstrates rather high error rate of commonly used implementations of mathematical functions and confirms the practical adequacy of the test construction approach proposed.

2 Current Standards' Requirements

Practically significant requirements on the behavior of functions on FP numbers can be found in several standards.

- IEEE 754 [1] (a.k.a IEC 60559) defines representation of FP numbers, rounding modes and describes basic arithmetic operations.
- ISO C [4] and POSIX [5] impose additional requirements on about 40 functions of real and complex variable implemented in standard C library.
- ISO/IEC 10697-2 [6] gives more elaborated requirements for elementary functions.

2.1 Floating-point Numbers

Standard IEEE 754 defines FP numbers based on various radices. Further only binary numbers are considered, since other radices are used in practice rarely. Nevertheless, all the techniques presented can be extended to FP numbers with different radices.

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 using another formula $X = (-1)^S 2^{(-2^{k-1}+2)}(M/2^{n-k-1})$. Here mantissa bits follow 0 and the point. Note that this gives two zero values $+0$ and -0.
3. Exponent $2^k - 1$ is used to represent positive and negative infinities (zero mantissa) and *not-a-number* NaN (any nonzero mantissa). Infinities represent mathematically infinite values or numbers greater or equal to $2^{2^{k-1}}$. NaN represents results of operations that cannot be considered consistently as finite or infinite, e.g. $0/0 = \text{NaN}$.

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 current (2008) version of the standard quadruple precision numbers ($n = 128$ and $k = 15$) are added.

2.2 IEEE 754 Requirements

IEEE 754 defines requirements to basic arithmetic operations on them (addition, subtraction, multiplication, and division, fused multiplication-addition $x*y+z$), comparisons, conversions between different formats, square root function, and calculation of FP remainder [7]. 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 and incorrect results more visible IEEE 754 defines a set of FP exception flags.

– Invalid flag should be raised if the result is NaN, while arguments of the operation performed are not NaNs. In addition NaNs are separated in two classes – *signaling NaNs* and *quiet NaNs.* NaN result of an operation on

not-NaN arguments is signaling one. If any of arguments of an operation is signaling NaN, then invalid flag is raised. Quiet NaNs can be used as arguments without raising invalid flag with quiet NaN as the result.
- Divide-by-zero flag should be raised if the result is exactly positive or negative infinity, while arguments are finite.
- Overflow flag should be raised if the result 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 no FP number, but no overflow or underflow occurs.

2.3 Requirements of ISO C and POSIX

ISO C [4] and POSIX [5] 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 and their reverse functions) of real or complex variables. Some special functions are added – error function, complementary error function, gamma function, and logarithmic gamma function.

ISO C standard defines points where the specified functions have exact well-known values, e.g. $\log 1 = \sinh 0 = 0, \cos 0 = 1$. 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 as informative for the complex ones.

POSIX slightly extends the set of described functions; it adds 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. POSIX specifies that value of **errno** should be set to ERANGE if overflow or underflow occurs or if the result is precise infinity, and **errno** should be set to EDOM if the arguments are out of the domain of a function, excluding arguments for which it returns signed infinite results.

POSIX requires 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 inconsistent with IEEE 754 rounding requirements if they were applied to such functions.

One more contradiction between POSIX and natural extension of IEEE 754 concerns overflow. IEEE 754 in this case requires to take rounding mode in account – e.g. to return positive infinity for to nearest and up rounding modes and the biggest positive finite FP number for to 0 or down modes. POSIX requires returning in any case one value BIG_VALUE.

Both ISO C and POSIX 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 [6], standard on language independent arithmetic. It provides the following requirements to implementations of elementary functions.

- Preservation of sign and 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.
- Rounding errors should not be greater than $0.5 - 2.0$ unit of least precision (ulp), depending on the function. Again, this is not applied to implementations of trigonometric functions on arguments greater than some big angle. Note that precision 0.5 ulp is equivalent to the correct rounding to the nearest FP number.
- Preservation of evenness or oddity of implemented functions. For this reason the standard does not support directed rounding modes – up and down. Only symmetric modes – to nearest and to zero – are considered.
- Well-known exact values for functions are specified, extending ISO C requirements. In addition it requires to preserve asymptotic of the implemented function in 0 or near infinity.
- Natural inequalities (e.g. $\cosh(x) \geq \sinh(x)$) should also be preserved.

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

3 Correct Rounding Requirement

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, monotonicity, and 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.
- Preservation of bounds of function range, e.g. $-\pi/2 \leq \arctan(x) \leq \pi/2$.
- Correct rounding according to natural extension of IEEE 754 rules and raising inexact flag on imprecise results.

Correct rounding requirement here is of particular significance.

- It immediately implies almost all other properties in this list. If we want to preserve these properties without correct rounding, much harder work is required, peculiar errors become possible, and thorough testing of such an implementation becomes much harder task.
- It provides results closest to the precise ones. Without correct rounding it is necessary to specify how the results may differ from the precise ones, which is hard and very rarely done in practice. It is supposed usually that correct rounding for sine function on large arguments is too expensive, but none of widely used sine implementations (except for Microsoft's one [9]) explicitly declares its error bounds on various intervals. 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 implies almost perfect compatibility of different mathematical libraries and precise repeatability of calculation results of modeling software on different platforms, which means very good portability of such applications. This goal is 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. Note that strict precision specification is much more flexible requirement than standardization of algorithms.

High efforts required to develop a function implementation and its resulting ineffectiveness are always mentioned as drawbacks of correct rounding. However, efficient algorithms and resolving techniques are already known for a long time (e.g. see [10, 11] for correct argument reduction for trigonometric functions). Work of Arenaire group [12] in INRIA on **crlibm** [13, 14] 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 issues are contradictions 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.

- Correct rounding can sometimes contradict with boundaries of function range, if they are not precise FP numbers. For example, $\arctan(x) \leq \pi/2$ 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.
- Oddity and some other symmetries using minus sign or taking reciprocal values 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 for different modes contradicts with two POSIX requirements – that some BIG_VALUE should be always returned in case of overflow, and that a function close to x near 0 should return the value of its argument for denormalized arguments. In both cases correct rounding seems to be more justified.

So, we propose to make correct rounding according to the current rounding mode the main requirement for standardization of any kind of functions working with FP numbers. The single exception is more privileged range preservation requirement in cases where it comes to contradiction with correct rounding. In all other cases correct rounding is sufficient to infer all the properties of an implementation.

In case of overflow a function should return the corresponding infinity for rounding to the nearest and in the direction of the overflow. For rounding in the opposite direction and to 0 maximum positive or negative FP number should be returned. On the arguments outside function's domain it should return signaling NaN, or signed infinity if the sign can be naturally determined by mathematical properties of this function. On infinite arguments a function should return the corresponding limit value, if it has any one, otherwise signaling NaN should be returned. If any of the arguments is signaling NaN, the result should also be signaling NaN. If any of the arguments is quiet NaN and there are no signaling NaNs among them, the result should be quiet NaN.

These requirements should be supplemented with IEEE 754 exception flags raising and setting **errno** to specific values in the corresponding specific situations (see above).

Further we consider test construction to check correct rounding requirement with 4 rounding modes specified by IEEE 754. We also demonstrate that such tests are useful and informative even for implementations that do not satisfy these requirements.

3.1 Table Maker Dilemma

An important issue related with correct rounding requirement is *table maker dilemma* [15, 16]. It consists in the fact that sometimes one needs much higher precision of calculations to get correctly rounded value of a function than in average. An example is the value of natural logarithm of a double precision FP number $1.613955DC802F8_{16} \cdot 2^{-35}$ (mantissa is represented in hexadecimals) equal to $-17.F02F9BAF6035\ 7F^{14}9\ldots_{16}$. 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 corresponds to only 2^{-53} bound.

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 test point selection rules based only on the properties of the function under test and structure of FP numbers, and do not consider specific implementation algorithms. 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 ever growing set of possible implementation algorithms and various errors that can be made in them.

Test points are chosen by the following rules (see more details in [3]).

1. **FP numbers of special structure.**
   First, natural boundary FP numbers are taken as test points: $0, -0, \infty, -\infty$, NaN, the least and the greatest positive and negative denormalized and normalized numbers.
   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 [17] can be detected only on divisors having units as mantissa bits from 5-th to 10-th. Pattern use for testing FP calculations in hardware is already described, e.g. in [18].
   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 satisfying 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.
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. Two values of M are used: $n-k-10$ for the worst cases and $(n-k)/2$, because some errors can be uncovered in not-very-bad cases. 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 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 taking higher precision to guarantee correct results and using both Maple and MPFR library [19] to double check the correctness. 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. Test execution is completely automated.

The hard step of the approach is to compute bad cases. Methods to do this include search based techniques, dyadic method, lattice reduction, and integer secants method (see details in [3]). They do not solve the problem completely, but help to deal with it in many particular cases.

5 Test Results Analysis

The test construction method presented above has been applied to make test suites for POSIX functions **exp, expm1, log, log10, log1b, sin, asin, cos, acos, tan, atan, sinh, asinh, cosh, acosh, tanh, atanh** in double precision. The tests have been executed on the following platforms.

- Platform A – Sun Solaris 10 on UltraSpark III processor.
- Platform B – SUSE Linux Enterprise Server (SLES) 10.0 with **glibc** 2.4 or Debian Linux 4.0 with **glibc** 2.3.6 on Intel Itanium 2 processor.
- Platform C – Red Hat Fedore Core 6 with **glibc** 2.5 or Debian Linux 4.0 with **glibc** 2.7 on Intel Pentium 4.
- Platform D – Windows XP operating system with Microsoft Visual Studio 2005 C runtime library on Intel Pentium 4 processor.
- Platform E – Red Hat Enterprise Linux 4.0 with **glibc** 2.3.4 on Intel Pentium 4 or AMD Athlon 64 processors.
- Platform F – Debian Linux 4.0 with **glibc** 2.7 on 64-bit PowerPC processor.
- Platform G – SLES 10.0 with **glibc** 2.4 on IBM s390 processor.
- Platform H – SLES 10.0 with **glibc** 2.3.5 on 32-bit PowerPC processor.

Platforms E-H being different in minor details are very similar in general picture and demonstrate almost the same numbers of similar errors, so they seem to have almost identical implementations of **glibc** math functions (and actually identical results are demonstrated by platforms F and G for **atan, asinh, atanh, expm1, log1p, log, log10** and by platforms E and H for **atan, log, log10**). Other platforms show more differences and specific errors.

Func.	Platf.	Rounding	Argument	Value
exp	E	down, to 0	-1.02338886743052249	1.12533187332226478e+307
exp	F	up	7.07297806243595915e+2	-5.62769256250533586e+306
exp	H	up	-6.59559560885092266e-1	1.00194134452638006
cosh	G	up	7.09150193027364367e+2	-2.35289304846008447e+307
sinh	E	down	6.68578051047927488e+2	-5.48845314236507489e+288
sin	H	up	3.36313335479954389e+1	7.99995094799809616e+22
cos	F	down, to 0	1.62241253252029984e+22	-1.19726021840874908e+52
sin	D	all	-1.79346314152566190e-76	9.80171403295605760e-2

Table 1. Examples of errors found.

The following errors and "features" were detected.

– The most serious bug is numerous and huge calculation errors in implementations of **sin, cos, tan, exp, sinh, cosh** on many platforms. Only platforms A and B implement trigonometric functions without serious errors (with only 1-bit or 2-bit errors). Exponential function and hyperbolic sine and cosine are implemented without big errors on platforms A-D. The salient fact is that on the platforms E-H all these functions work almost correctly for the rounding to nearest (only few 1-bit errors were detected for trigonometric functions, 2-bit ones for hyperbolic functions, and no calculation errors were found for **exp**), but for other rounding modes almost arbitrary inaccuracies are possible. Examples are given in Table 1.

  Implementations of trigonometric functions on platforms C and D, although erroneous, have imprecise argument reduction [10] as the main source of errors, so they show smooth increase of inaccuracies from 0 to infinities, independently of rounding modes.
– Sine is implementated with errors in the interval $(-0.1, -10^{-76})$ on platform D. An example is shown in the last row of Table 1. This error is hard to show due to compiler-implemented transformation $\sin(-x) = -\sin(x)$. To overcome it test data should be generated or loaded dynamically, so that the compiler cannot notice that they are negative numbers.
– The platform B is the only one which preserves oddity or evenness of all the functions tested. **cosh** is implemented as an even function on all the platforms except for D. **atan, cos, sin, tan** also preserve their symmetry on the platform C, **asin** – on the platform D. In all other cases the symmetry is somehow broken.
– Arctangent function for large arguments calculated with up rounding returns the result greater than $\pi/2$ on the platforms A, B, and C.
– Platform C shows big calculation errors in **acos** for rounding up, **tanh** for rounding down, **expm1** for rounding up and down. Also $\mathbf{acos}(-1) = -\pi$ instead of π for rounding up.
– On platform A functions **asin, acos, log, log10** return not-NaN FP number instead of NaN for arguments out of function domain.
– Finite number (actually maximum float value) is returned instead of infinity in case of overflow for **exp, cosh, sinh** and in 0 for **log, log10** on platform A. Maximum finite double value is returned instead of infinity for **exp, cosh, sinh** on platform D. These bugs may be related with POSIX requirement to return some BIG_VALUE in case of overflow independently of rounding mode. This requirement (with BIG_VALUE equal to infinity) is implemented on all platforms except for A and D for **exp,** on platforms B, E, H for **cosh,** on platforms E, H for **expm1,** and only on the platform B for **sinh.**
– On all platforms, except for B, functions that are equivalent to x near 0 "try" to return the argument value for denormal arguments for all rounding modes. That is, POSIX requirement is implemented on most platforms in most cases. However, this requirement is implemented only for positive denormals for **asin, tanh** and **atanh** on the platform C and for **tanh** on the platform G. Platform B implements correct rounding requirement.

- A lot of minor calculation errors were detected. The best accuracy is demonstrated by platform B – maximum errors are only 2-bit and such errors are rather rare, the worst results are shown by **atan,** for which 10% of test points discovered such errors. Platform A shows maximum 3-bit difference from correct results. Sometimes such errors are very often, for example, 99.8% of test points discovered 1-bit errors in **acos** implementation on platform D for rounding up. In some cases probable mistakes in table values used in implementations are uncovered, for example,
  - On platforms E-H **atan** has erroneous 7 last bits of mantissa for rounding up, down or to 0 near -6.25.
  - On platform C **sinh** value for rounding up has erroneous 15 last bits near 1.986821485e-8.
  - On platform D **exp** value has 6 erroneous last bits near -2.56e-9.
- Some errors detected concern incorrect flag (not-)raising and incorrect **errno** values. For example, for almost all implementations of functions that are equivalent to argument near 0 UNDERFLOW flag and ERANGE **errno** value are not set for denormal arguments. For **atanh** in 1 or -1 **errno** value in all implementations is set to domain error, not to range error as it is required by POSIX.

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 related with mistakes made by programmers, 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 formulate consistent requirements and construct corresponding conformance test suites for floating-point 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 are intended to check correct rounding requirement, they also give important information about implementations that do not obey this restriction.

Some further research is needed to formulate heuristics or rules that help to make test suites more compact. Now they consists of about $2 \cdot 10^6$ test points and require sometimes several hours to execute. The experiments conducted demonstrated that for exponential function the test suite constructed using the method described and consisting of about $3.7 \cdot 10^6$ test cases, and the reduced test suite of about 10^4 test cases detect all the same errors.

One idea to extend the method proposed for functions in two or more variables is rather straightforward – it is necessary to use not intervals, but areas of uniform behavior of functions. But extension of rules concerning FP numbers of special structure and bad cases seem to be much more peculiar, since their straightforward generalizations gives huge number of tests 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 manageable test suites.

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

References

1. IEEE 754-2008. IEEE Standard for Binary Floating-Point Arithmetic. NY, IEEE, 2008.
2. V. Kuliamin. Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers. Programming and Computer Software, 33(3):154-173, 2007.
3. V. Kuliamin. Test Construction for Mathematical Functions. In K. Suzuki, T. Higashino, A. Ulrich, T. Hasegawa, eds. Testing of Software and Communicating Systems, LNCS 5047:23-37, Springer, 2008.
4. ISO/IEC 9899. Programming Languages - C. Geneve: ISO, 1999.
5. IEEE 1003.1-2004. Information Technology - Portable Operating System Interface (POSIX). NY, IEEE, 2004.
6. ISO/IEC 10967-2. Information Technology - Language Independent Arithmetic - Part 2: Elementary Numerical Functions. Geneve, ISO, 2002.
7. D. Goldberg. What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Computing Surveys, 23(1):5-48, 1991.
8. D. Defour, G. Hanrot, V. Lefevre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. Numerical Algorithms, 37(1-4):367-375, December 2004.
9. http://msdn.microsoft.com/library/wkbss70y.aspx
10. K. C. Ng. Arguments Reduction for Huge Arguments: Good to the Last Bit. 1992. Available as http://www.validlab.com/arg.pdf.
11. W. Kahan. Minimizing $q * m - n$. 1983. Unpublished, available as http://http.cs.berkeley.edu/ ∼wkahan/testpi/nearpi.c.
12. http://www.inria.fr/recherche/equipes/ arenaire.en.html
13. F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. Proc. of 17-th Symposium on Computer Arithmetic. IEEE Computer Society Press, June 2005.
14. http://lipforge.ens-lyon.fr/www/crlibm/
15. V. Lefèvre, J.-M. Muller, and A. Tisserand. The Table Maker's Dilemma. INRIA Research Report 98-12, 1998.
16. V. Lefèvre, J.-M. Muller. Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. Proc. of 15-th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA, June 2001.
17. A. Edelman. The Mathematics of the Pentium Division Bug. SIAM Review, 39(1):54-67, March 1997.
18. A. Ziv, M. Aharoni, and S. Asaf. Solving Range Constraints for Binary Floating-Point Instructions. Proc. of 16-th IEEE Symposium on Computer Arithmetic (ARITH-16'03), pp. 158-163, 2003.
19. http://www.mpfr.org