ELEMENTARY FUNCTIONS BASED UPON IEEE ARITHMETIC

David Hough
Apple Computer
PO Box 561
Cupertino, CA 95015

ABSTRACT

Adoption of a standard for binary floating point arithmetic provides an occasion for software implementers of elementary transcendental functions either to remedy past errors or to heap additional novel indignities upon users.

CHALLENGE

Historically, mini and micro manufacturers [3], following the mainframe pattern [2], usually provided carelessly written, needlessly inaccurate software for the "elementary" transcendental exponential, logarithmic, and trigonometric functions. This software often reflected the attitudes and errors existing in the machines' basic arithmetic operations for addition, subtraction, multiplication, and division [8]. This software was cheap to write but made debugging users' programs more expensive.

In 1982, however, IEEE Working Group P754 finished a proposal for a standard for binary floating point arithmetic [1]. This standard was designed for new micro implementations, but it has also been adopted for a new high performance mini [9]. The standard specifies representable number sets, operations upon them, modes which affect the results, and handling of exceptions which may arise during computation. Except for a few extreme cases, the standard categorically specifies results and exceptions, allowing no variation among conforming implementations.

The challenge now is to create elementary function codes that are as good as the basic arithmetic operations defined in the IEEE standard. Ideally the result of a call on an elementary function would

> return a correctly rounded result,
> observing the current rounding modes,
> generating only relevant exceptions,
> without being much slower or larger than
> a careless implementation.

Integrated circuits now in production or design incorporate complete [11] or partial [10] elementary functions. As their cost declines these circuits will be incorporated into increasing numbers of micro systems. In the interim, software implementations will predominate; these are the subject of this paper.

CONSTRAINTS

Codes that guarantee correctly rounded transcendental functions will generally lack acceptable efficiency. For algebraic functions such as square root, a simple test determines whether any partial result is exact. But for transcendental functions, no theory fixes in advance the number of extra bits that must be computed in order to guarantee that the final answer be correctly rounded to n bits. This "table maker's dilemma" antedates even mechanical computing devices. But to meet function code speed goals, the amount of extra precision, if any, is usually fixed in advance.

The IEEE standard recognizes a less severe form of this dilemma when it authorizes base conversion to be slightly less than perfectly rounded if the exponents involved are extremely large or small. The known algorithms for correctly rounded base conversion require progressively larger buffers and execution times as exponents increase. These algorithms are too costly within current integrated circuit technology, so the standard was relaxed to what is known to be economically achievable. The standard also allows some latitude in the settings of the inexact and underflow exceptions, so that the exceptions are allowed to be set even in some cases that are really unexceptional.

Besides the usual binary floating point numbers, the IEEE standard also defines tiny denormalized numbers, signed zeros and infinities, and not-a-number symbols (NaNs). In many cases the standard specifies special treatment for these cases. Elementary functions should handle these cases correctly, but to conserve code size, algorithms should be chosen which minimize the number of tests for unusual operands and results.

Another aspect of the challenge is to create codes that act like arithmetic operations in generating exceptions. That is, the exceptions describe the computed result and not intermediate results that may be artifacts of a particular algorithm. Most of the problems arise from intermediate underflow, overflow, and divide by zero exceptions.

11/93

In fact, the very definition of "divide by zero" needs generalization. Log(0) and tan(pi/2) are like division by zero since they produce an infinite result from a finite operand. This result is exact in the sense that no overflow occurred and the result would be infinity even with unbounded precision and range. So the general definition of "divide by zero" should be enlarged to "exact infinite result from finite operand(s)."

Typical transcendental functions have rational values for only a few rational arguments. For all other rational arguments, the values are transcendental and can not be represented exactly in floating point arithmetic. So the inexact flag exception should usually be signalled. This is seldom a problem; rather the difficulty is not signalling inexact in the few cases when the result is exact.

An optional portion of the IEEE standard specifies traps that may be defined to occur in user programs when an exception occurs. In full generality these traps allow user defined programs to determine the operation, operands, default result, and exceptions generated. Encoding and transmitting all this information to a user of higher level language is a formidable problem on many systems, to the extent that such trapping is seldom available except to machine language programmers. Sometimes halts are provided instead [12]. When an exception occurs whose halt is enabled, control passes to the operating system which interrogates the user whether to halt or to continue execution with the default result. Such a system with halts does not allow user programmed response to an exception and can be cheaper to implement than IEEE trapping.

If either traps or halts are provided, the challenge to the implementer is to make exceptions generated in elementary functions like those generated in the arithmetic operations. This is often complicated when the arithmetic is in hardware or machine language and the elementary functions are coded in a higher level language.

METHODS

The intent of the IEEE standard is that data be stored in a basic format and that computations be performed in an extended format if available. But codes may be independently characterized as "operand precision" or "extra precision." Operand precision codes must compute using arithmetic of the same precision and range as the arguments and result. Extra precision codes have the luxury of additional precision and range for intermediate computations and consequently rarely need to make provision for intermediate underflow and overflow. Note that if the arguments and results are in IEEE extended format, a code that must compute in the same IEEE extended format is "operand precision."

Thus it would appear desirable to limit arguments of elementary functions to basic formats and computations within them to extended formats. But this conflicts with another desirable property, that computations such as

z := exp(x^2-y^2)

be evaluated in extended format even when x, y, and z are in basic format. Such evaluation minimizes the effect of rounding error and allows computation of valid z even in many cases where intermediate results x^2 and y^2 might overflow or underflow.

Elementary function codes require argument reduction followed by function evaluation on the reduced argument [4]. Most implementers would do well to follow Cody [5] even though the approximations there are not optimized for IEEE arithmetic; for most of the functions, Cody does not list approximations accurate to more than 60 bits.

Argument reduction requires extra precise values of pi/2 and loge(2) for the computations of

r := remainder(x, pi/2)

or

r := remainder(x, loge(2)).

One approach is to generate these constants to any arbitrary precision as needed [13]; another is to use the same approximation for pi/2 or loge(2) regardless of the size of the argument. The first approach is essential to obtain correctly rounded approximations to the exact functions but suffers because the computed functions are not quite periodic. The second approach precludes correctly rounded computed functions but insures their periodicity, even though the period differs slightly from the period of the exact function. If the second approach is used, the constants pi/2 and loge(2) should be stored in the widest precision available and the remainder operation should take place in that precision. Thus Fortran SIN(X) and SNGL(DSIN(DBLE(X))) should rarely differ, and then only in the least significant bit of the result, even for large single precision arguments X. The two results will sometimes be totally different if SIN uses a single precision approximation to pi/2 and DSIN uses a double precision approximation to pi/2.

What about cos(pi/2)=0 or tan(pi/2)=+infinity? The user's pi/2 may differ from that internal to the function code. If the internal pi/2 is stored to a certain fixed extended precision, and the code is extra precision, then an argument which is as close to pi/2 as possible will be less accurate than the internal pi/2. After remainder, the reduced argument will not be exactly zero, so cos will be not zero but a small number, and tan will be

not infinity but a large number. But if the code is operand precision, then the argument may be the exact same value as the internal pi/2, and so will be reduced to exactly zero after remainder, with result cos exactly zero and tan exactly infinity, with never a rounding error. Either way is bound to surprise someone. Naive users expect COS(PI/2) to be zero, especially in languages like Basic that provide a name for pi; experienced programmers know that pi/2 can not be represented exactly so COS(PI/2) should not be exactly zero.

After argument reduction comes function evaluation; Table 1 below suggests forms for approximating the key functions on reduced arguments. The approximations to g(x) should be about as accurate as the precision in which they are evaluated.

As an example, over the limited x interval, the essential financial function exp(x)-1 can be calculated as

$$x + x^2 * g(x) ,$$

while the familiar exp(x) can be calculated accurately as

$$1 + x + x^2 * g(x) .$$

Note that these forms of approximation work well for denormalized x without any special consideration. For an arbitrary argument x, which could be represented as n * loge(2) + r for integer n, exp(x) can be computed from the expression

$$2^n * (1 + r + r^2 * g(r))$$

while representing x as m + f for integer m, allows 2^x to be computed from

$$s := f * \mathrm{loge}(2),$$
$$2^x := 2^m * (1 + s + s^2 * g(s)).$$

In what form are these approximating functions g(x) expressed? Most often as ratios of polynomials, or rational functions; less frequently as continued fractions. [4] and [5] contain coefficients found by using versions of the Remes algorithm [6]. The Remes algorithm can be adapted to produce new approximations providing it is executed in greater precision than the intended working precision of the elementary function code. In particular, for IEEE double extended format with 64 bit significands, Remes algorithms have been executed in 96 bits of significance using hardware like CDC 6600 double precision or software like Brent's [7].

Such investigations usually reveal several possible approximations. To determine the best it is necessary to compare the approximate function value, computed in the intended working precision, with much more accurate function values computed with greater precision, such as by the Remes algorithm. Errors should be measured in units in the last place of the function desired - f(x) in Table 1 - computed for many points over the x interval. Cody [5] explains many other essential testing details.

Although extra precision codes can produce results that are almost always correctly rounded, at least in a limited interval, it is a rare and significant accomplishment to be able to make the same claim for operand precision codes. Consequently it makes little sense to attempt to observe the rounding modes when computing the function approximation, which might as well be computed in the default mode of rounding to nearest. Then the very last step of an extra precision code can be to restore the original rounding mode, and store the extended format approximate result in the basic format in which it is to be delivered.

When coding several elementary function routines it is helpful to prepare standard procedures for entry and exit protocol. The entry protocol consists of saving the current floating point environment, consisting of rounding modes, exceptions, and traps or halts, then clearing the exceptions and traps and restoring the default modes. During the subsequent computation, some exceptions may arise, but traps are deferred so they can be handled in an orderly manner by the exit protocol. This protocol consists of noting which new exceptions arose within the function code, restoring the environment that existed previously, then ORing in the new exceptions and ANDing the new exceptions with the trap bits to determine whether a trap should occur. As indicated above, the storing of the result in a narrower format should be performed in the original rounding mode.

By following some of the previous suggestions and coding with care it is possible to create extra precision elementary functions of a single argument that almost always attain the desirable results listed above under CHALLENGE for reasonable arguments. It is much more difficult to come close to the same standard when functions of two arguments are considered such as x^i, x^y, compound interest factor, present value of annuity, conversion between rectangular and polar coordinates, and complex elementary functions of a complex variable. Even the definitions of these functions are not yet universally accepted in cases like NaN^0 or (-Inf)^(+Inf). One approach is to build the functions of one argument to a high standard, then build the functions of two arguments in a straightforward way without concern for extreme cases and exceptions. This may run afoul of many users' expectations that x^y, for instance, is as elemental an operation as x*y. Getting nearly correct rounding for these functions, not to mention meaningful exceptions, remains an interesting challenge.

To compute: f(x) =	Find approximation to: g(x) =	Over x Interval:
exp(x)-1	(exp(x)-1-x)/x^2	abs(x) <= 0.5*loge(2)
loge(1+x)	(loge(1+x)-x)/x^2	2^-0.5 - 1 <= x <= 2^0.5 - 1
sin(x)	(sin(x)-x)/x^3	0 <= x <= pi/4
cos(x)	(cos(x)-1)/x^2	0 <= x <= pi/4
atan(x)	(atan(x)-x)/x^3	0 <= x <= tan(pi/4)

Table 1. Approximation Forms for Elementary Transcendental Functions

REFERENCES

[1] "A Proposed Standard for Binary Floating Point Arithmetic," Draft 10.0 of IEEE Task P754, December 2, 1982. Inquiries concerning the progress of this draft standard toward official adoption should be addressed to the IEEE Standards Office, 345 E 47th St, New York 10017.

[2] W. Cody, "Software for the Elementary Functions," in J. Rice, ed., Mathematical Software, Academic Press, 1971.

[3] E. Battiste, "Scientific Computations Using Micro-Computers", ACM SIGNUM Newsletter, 16 (1) 1981.

[4] J. Hart, Computer Approximations, Wiley, 1968.

[5] W. Cody and W. Waite, Software Manual for the Elementary Functions, Prentice Hall, 1980.

[6] W. Cody, W. Fraser, and J. Hart, "Rational Chebyshev Approximation using Linear Equations," Numerische Mathematik, 12 (4) 1968.

[7] R. Brent, "MP, A Fortran Multiple-Precision Arithmetic Package," ACM TOMS, 4 (1) 1978.

[8] W. Kahan, Implementation of Algorithms, NTIS Document DDC AD 769 124, 1973.

[9] G. Taylor, "Arithmetic on the ELXSI System 6400," in T. Rao and P. Kornerup, eds., Proceedings 6th Symposium on Computer Arithmetic, IEEE Computer Society, 1983.

[10] The 8086 Family User's Manual Numerics Supplement, Intel Corporation, number 121586-001 Rev A, 1980.

[11] J. Boney and V. Shahan, "Floating Point Power for the M68000 Family," in Mini/Micro Northeast Conference Record, Electronic Conventions, 1983.

[12] Numerics Manual: A Guide to Using the Apple /// Pascal SANE and Elems Units, part number 030-0660-A, Apple Computer, 1983.

[13] M. Payne and R. Hanek, "Radian Reduction for Trigonometric Functions," ACM SIGNUM Newsletter, 18 (1) 1983.