Comments on Proposed ANSI C Standard

David Hough - dhough@sun.com

ABSTRACT

The proposed C standard falls short in three aspects critical to providers of portable mathematical software. And there are several lesser problems. I recommend:

Comment #1, Section 3.9:	encourage sound practices
Comment #2, Section 3.9:	disparage hazardous practices
Comment #3, Section 1.1:	emphasize surprises in rationale
Comment #4, Section 1.1:	anticipate supplemental standards
Comment #5, Section 2.2.4.2:	use "significand"
Comment #6, Section 2.2.4.2:	**postpone <float.h>**
Comment #7, Section 3.2.1.4:	round conversions between floating types
Comment #8, Section 3.5.4.2:	**allow array parameters to have variable dimensions**
Comment #9, Section 3.7.1:	remove incorrect rationale comment
Comment #10, Section 4.5:	**don't overspecify exceptions in mathematical functions**
Comment #11, Section 4.5:	tell more in the rationale
Comment #12, Section 4.5:	standardize hypot
Comment #13, Section 4.5.4.6:	delete modf
Comment #14, Section 4.7:	specify which signals can arise

Preface

The following comments are based upon *Draft Proposed American National Standard for Information Systems - Programming Language C*, document number X3J11/88-090, dated 13 May 1988, and its accompanying Rationale. The comments are personal opinions of the author, and should neither be construed as wholly original nor as representing the position of any organization or other person. A number of individuals helped formulate and clarify them; some of their names are listed here and at the end. The following, while not necessarily agreeing in every detail, have expressed agreement with the main points:

T. Andrews	gatech.edu!uflorida!ki4pv!cdis-1!tanner
W. J. Cody	cody@anl-mcs.arpa
Stuart Feldman	sif@bellcore.com
David Gay	research!dmg
Eric Grosse	research!ehg
W. Kahan	University of California, Berkeley
Zhishun Alex Liu	zliu%hobbes@berkeley.edu
Doug McIlroy	research!doug
Jim Meyering	meyering@cs.utexas.edu
Tom Rowan	rowan@cs.utexas.edu
Dan Schlitt	phri!ccnysci!dan
Steve Sommars	att.arpa!iwtsf!sesv
Gene Spafford	spaf@purdue.edu
Zdenko Tomasic	zdenko@csd4.milw.wisc.edu
Jim Valerio	verdix!radix!jimv

Introduction

C was not particularly designed to facilitate numerical computation. In recent years it has come to be increasingly used for implementing portable systems and applications, including numerical ones. This is more a tribute to the good judgment embodied in some other aspects of C's design than to its numerical facilities: it is easier to get a usable C compiler and library working than a comparable Fortran compiler and library. Examples of such applications are the SPICE 3B1 circuit simulation package, which is far more flexible and maintainable than its Fortran predecessors, and Alex Liu's elementary function test programs, which would have been far more difficult to implement in Fortran.

In its drafts the ANSI committee has removed some of traditional C's numerical weaknesses, such as requiring double-precision expression evaluation and parameter passing, and overspecifying error response in the elementary transcendental function library. With a little more effort the remaining numerical stumbling blocks could be removed and C would be as convenient for the numerical parts of applications as for the other parts.

A more extensive version of the following commentary, dated 29 March 1988, was submitted to X3J11 in conjunction with the second public review period. X3J11 published formal replies to all such comments on April 22, 1988, as document X3J11/88-083.

The principles guiding the scope of the following commentary for the third public review are as follows:

* According to X3J11/88-083, X3J11 will not consider adding any substantial new features to the language, regardless of merit. Of the three major recommendations that follow, two reduce the size of the language and one extends it very slightly.
* According to X3J11/88-083, X3J11 will not consider any change that renders non-conforming large numbers of existing C programs. Where appropriate, therefore, I suggest directions for future evolution rather than requirements to be effective immediately.
* Changes in the Draft's Appendices and Rationale are far more likely to be considered than changes in the Standard itself.
* Changes introduced in the third public review draft are especially susceptible to modification.
* Erroneous assertions in X3J11/88-083 should be corrected.

Comments follow with specific **Recommendations** containing additional or revised wording. For brevity, I assume that readers have access to my 29 March commentary.

In general, "infinity" refers to a floating-point representation that is intended to act like mathematical ∞, such as is found on CDC and IEEE implementations. Similarly "NaN" refers to any kind of non-numeric floating-point representation, including CDC Indefinite, VAX Reserved Operand, and IEEE NaN.

Comment #1, Section 3.9, 4.13, and A.6:encourage sound practices

Certain features and practices, not accepted for the current Draft, should be listed as Future Directions and Common Extensions to encourage implementations and programmers to do the right thing and to indicate that they might be incorporated into future revisions of the Standard.

Recommendation:

2.2.4.2: long double's exponent range should satisfy:

-LDBL_MIN_10_EXP >= 99
LDBL_MAX_10_EXP >= 99

3.3: There is a clear need for some method for specifying non-aliasing. The previous draft's **noalias** was not the answer. This is a good area for local experimentation. What appears to be required is a method of specifying "within this scope, the compiler may assume that the pointers {x, y, z, ... } do not point to the same storage." This will be essential to attain Fortran-like performance on simple linear algebra problems, even on personal computers, within the lifetime of the first ANSI C Standard. The exact syntax and semantics are not clear yet, but should become clear before ANSI C undergoes its next major revision some years hence.

3.4: Constant expressions which would have side effects at run time should be evaluated at run time rather than compile time if the side effects would not otherwise occur at run time. This includes inexact floating-point expressions in systems implementing IEEE arithmetic. A method of forcing compile-time expression evaluation should also be supplied.

3.5.6: C needs a syntax for initializing large arrays to the same value corresponding in intent, but not necessarily in syntax, to Fortran constructs like:

```
data x/ 100 * 0.0 /
```

With such a feature automatic initialization of static data to zero is longer required or desirable in C. Implementations should provide means for detecting assumptions of implicit zero initialization of storage; an example is a loader option to initialize memory bytes to 0xFF (to catch uninitialized floating-point variables by setting them to NaNs) or to 0xAA or 0xCC rather than 0 to catch uninitialized integer variables. Such initializations also help to identify mistaken references outside of array bounds, references through badly constructed pointers, and many other forms of software error often masked by zero-initialized memory. Initialization to 0 should also be provided as an option. The default should be undefined so that portable code will not depend on it. Empirical and theoretical reasons can be given for not using 0 as an initializer if at all possible; see, for instance,

%A Eugene H. Spafford
%T Initializing Uninitialized Memory
%R Technical Report GIT-SERC-87/02
%I Software Engineering Research Center, Georgia Institute of Technology
%C Atlanta, GA
%D 1987

4.9.6.1: Implementations in which the sign of zero is significant should always display that sign when the + modifier is included in a printf specification. Implementations in which the zero has no sign or zero's sign bit has no significance should always display a + sign.

4.9.6.1: If the argument is exactly zero and the # alternate form is not specified, then printf %f uses blanks in lieu of the decimal-point character and its trailing zeros.

4.10.1.4: If an implementation includes infinities or NaNs, printf %e, %f, or %g should format them with appropriate character strings that distinguish them from finite numbers. Those character strings should also be readable by strtod and by scanf in the usual numeric format input fields and converted into the appropriate internal representation of infinity or NaN respectively. *Nothing in the foregoing requires every implementation to support infinities and NaNs.*

End of recommendation.

Comment #2, Section 3.9 and 4.13: disparage hazardous practices

Certain historical features and practices, accepted for the current Draft primarily to accommodate existing code, should be disparaged by listing as Common Warnings, with their removal contemplated as Future Directions, to encourage implementations to issue warnings for them and to indicate that they might be dropped from future revisions of the Standard.

3.2.1.5: Use of implicit narrowing type conversions is a hazardous coding practice. Such conversions should be accomplished by explicit casts. Compilers should provide means to detect such conversions.

3.5.6: Reliance on automatic implicit initialization of variables to zero is a hazardous coding practice. Initializations should be explicit. Compilers should provide means to detect implicit initialization.

4.5: Portable code must not rely on any aspect of **errno** behavior.

4.7: Implementations should not generate SIGFPE for exceptions other than floating-point.

4.10.1.1: Use of atof and atol is hazardous due to lack of reliable error detection; the hazards can be avoided with strtod and strtol.

Comment #3, Section 1.1: emphasize surprises in rationale

Certain X3J11 decisions, revealed in its formal public response to the second public review, may well surprise some implementers who haven't been following the deliberations extremely closely. These surprises, together with the already-documented Quiet Changes, should be collected together in one place in the Rationale.

Recommendation:

2.2.4.2: The sign-magnitude floating-point model intentionally excludes one's and two's-complement floating-point representations and logarithmic floating-point representations.

3.1.3.1: Conversion of decimal strings to floating-point values at compile time (and hence at run time as well) is required to be correctly rounded to a nearest representable value, either up or down, and exactly representable values are required to be converted exactly. For large exponents this requirement is quite a bit stricter than what IEEE 854 requires. Many existing C implementations fail to satisfy this requirement; it is impossible to satisfy unless the implementation uses greater than double precision during computation - integer or floating-point arithmetic may be used. For instance, the portable atof() supplied with System V Release 3 attempts to convert strings to double using only double arithmetic. For the input string 1.57772181044202309e-30, to be converted to IEEE double precision, this atof() returns 39BF FFFFFFFFFFC instead of the correctly-rounded 39BF FFFFFFFFFFD. The correct answer would be expressed in the same format as 39BF FFFFFFFFFFD.06+, and thus must be rounded to either 39BF FFFFFFFFFFD or 39BF FFFFFFFFFFE according to the Draft's specification.

Furthermore, according to X3J11's formal response, the compiler must convert constants identically to strtod or scanf.

3.2.1.3 and 3.2.1.4: Conversion of integral or floating types to less-precise floating types also is required to be correctly rounded in the sense described above. Some existing C implementations may not meet these requirements.

3.3: X3J11 has made a major advance by requiring that parentheses be observed when evaluating expressions. This requirement doesn't leap out of the Draft (although it's mentioned in the Rationale) and might surprise an implementer more familiar with C's traditions than with the history of X3J11's deliberations.

3.3.4: Casts such as (float) of a double expression must cause conversion to occur.

4.5: Several of the mathematical function specifications are so rigorous that efficient implementations are impossible on machines lacking guard digits, such as Cray, CDC, and ETA.

4.5.4.3: ldexp and frexp contain a bias against non-binary floating-point implementations. In those implementations, such as IBM 370, ldexp and frexp are not free of roundoff.

4.8: Conforming implementations require a prototype in scope for printf. Implementations allowing printf without a prototype in scope may suffer reduced performance.

4.8: Unlike 4.3 BSD or System V, X3J11 C permits **errno** to be set only by the 4.5 math functions and the 4.10 functions ato{f,l} and strto{d,l,ul}.

4.9.6.2: When scanf() processes a *correct* format conversion and reads beyond it, that first character beyond the formatted field is guaranteed to have been pushed back to the input stream. When scanf() processes an *incorrect* format conversion, at most one input character is guaranteed to have been pushed back by scanf() - so that any previous characters read and subsequently rejected by scanf may have been lost, as in the case of attempting to scan "-.x" with %f. In either case, after a return from scanf(), ungetc() must be able to accept at least one more character of pushback. Thus if the ungetc() implementation is minimal then scanf() must be implemented with some more powerful pushback mechanism.

Note that the foregoing conclusions, apparently endorsed by the Committee in its formal response, are difficult to reconcile with the Rationale's section 4.9.6.2.

4.10: Unlike most other X3J11 identifiers, "strtoul" doesn't fit in six characters.

End of recommendation.

Comment #4, Section 1.1: anticipate supplemental standards

Certain aspects of the definition of C should be deferred until X3J3 completes its work on Fortran 8x, such as the environmental inquiries currently defined in 2.2.4.2 for <float.h>.

Since it's desirable in most cases for C to follow rather than lead Fortran in these areas, C standardization should be deferred until the Fortran work has been tested in practice.

Many C implementations for personal computers, workstations, and superminicomputers are built upon the IEEE 754 standard for binary floating-point arithmetic. Certain aspects of the C environment could well be standardized among such implementations, without impacting the many non-IEEE C implementations:

* Definition of functions for IEEE modes and status;
* Definition of IEEE 754/854 Appendix functions.
* Definition of IEEE trapping.
* Definition of syntax for infinities and NaNs.
* A corrected and improved version of the current Draft's <float.h> based upon Fortran-8x facilities after the latter stabilize.

The X3J11 Draft should avoid prejudicing such later work. Many of the subsequent recommendations aim to eliminate such prejudice.

Most X3J11 members are weary of the arduous standardization process, so it would be appropriate for another body to develop an informal standard a couple of years after Fortran 8x is finalized; that informal standard might be incorporated into a subsequent revision of C. An IEEE/CS committee will probably develop an IEEE Recommended Practice for implementations intending to conform to ANSI C and IEEE 754 or 854. This may well occur within a year after ANSI C is formalized.

Comment #5, Section 2.2.4.2 and 3.1.3.1: use "significand"

The term *significand* was adopted by the IEEE floating-point committees to designate the part of a floating-point number that contains its significant digits. Significand is a better term than either "mantissa", which refers to the fractional part of a logarithm, or "value part", which implies that the exponent doesn't contribute to the value of a number.

Recommendation:

Replace "mantissa" and "value part" with "significand" throughout the Draft and Rationale.

End of recommendation.

The foregoing is consistent with ANSI/IEEE Std 1084-1986, *IEEE Standard Glossary of Mathematics of Computing Terminology;* in contrast X3J11's formal response states "we believe that the use of *mantissa* is synonymous with *significand.*"

Comment #6, Section 2.2.4.2: postpone <float.h>

The Rationale states that the constants in <float.h> were derived from a similar section of the Fortran 8x proposal. X3J11 may not have been aware that those corresponding Fortran proposals are not universally accepted. Parts of those proposals are inherently ambiguous because existing computer arithmetic is so diverse.

If the intent is to define a <float.h> that is truly useful and unambiguous then considerable additional effort is required; I indicated previously the directions that effort should take. From this point of view <float.h> is not ripe for standardization. There is certainly no widespread prior art in C.

If the intent is to define a <float.h> that is as close as possible to the spirit of whatever Fortran-8x adopts, then again the effort should be deferred until Fortran-8x stabilizes; additional major revisions appear forthcoming, some of which may affect the numerical environment and inquiries facilities. Although facilitating translation of Fortran-8x to C was listed as a goal for <float.h>, the dynamic precision capabilities in the current Fortran-8x draft have no analog in C, so attainment of this goal appears doubtful.

Recommendation:

Remove the <float.h> specification from 2.2.4.2 and mention, as a Common Extension and Future Direction, a C analog of the final Fortran-8x floating-point characterization.

End of recommendation.

That X3J11 was willing to accept on faith a major invention not yet widely implemented in any language is in remarkable contrast to its expressed attitude on many less extensive proposals. Here's an example of the problems. **FLT_EPSILON** is defined in 2.2.4.2 as the "minimum positive floating-point number x such that 1.0+x != 1.0, b**(1–p)". These two expressions are intended to be equivalent. Consider these 32-bit examples:

Architecture	b	p	min pos x (1+x) != 1	b**(1–p)	comment
IBM 370	16	6	16**–5	16**–5	formulas equivalent since 370 usually chops
DEC VAX	2	24	2**–24	2**–23	(1+2**–24) rounds up to (1+2**–23)
IEEE	2	24	2**–24+2**–47	2**–23	(1+2**–24) rounds down to 1

Note that the IEEE FLT_EPSILON value mentioned in the Draft's example, 1.19209290e–7, is b**(1–p), not the minimum positive x, 5.96046519e–8.

If the underlying question is actually "is x negligible compared to y", then it is best asked directly: "if ((float)(x+y) != y)". The (float) is only necessary on systems that might hold x+y in a higher-precision temporary.

If the underlying question is "what's the next representable number from x in the direction y" then that's rather painful to determine except on IEEE systems that provide the nextafter() function.

If an error analysis is desired so that the underlying question is "what's the least ρ>0 bounding |δ| in the equation

$$\text{computed}(x \text{ op } y) = \text{correct}(x \text{ op } y) * (1 + \delta) \quad "$$

then the answer ρ is b**(1–p) on IBM 370, b**–p on VAX, and on IEEE, b**–p or b**(1–p) depending on rounding mode. On Cray, the answer is probably 1.5 * b**(1–p) for multiplication, 6 * b**(1–p) for division, and is 1.0 for addition and subtraction, due to lack of a guard digit, a lack shared by CDC, ETA, and older Univac machines. Thus, the current structure does not aid in developing portable software as was intended, but may in fact lead to additional confusion.

Comment #7, Section 3.2.1.4: round conversions between floating types

The C Draft requires that conversion of a floating-point value to fit in a floating-point format of less precision or exponent range be accomplished by an acceptable rounding operation; acceptable roundings include rounding toward zero. This is all consistent with IEEE arithmetic. But the Rationale still asserts that "The new IEEE floating point processor chips control floating to integral conversion with the same mode bits as for double-precision to single-precision conversion; since truncation-toward-zero is the appropriate setting for C in the former case, it would be expensive to require such implementations to round to float."

This generalization is quite wrong in general; while possibly true for the first IEEE chip, the Intel 8087, it is not true for most "new"er implementations such as the MC 68881/2, SPARC, Intel 80960, HP Precision, most Weitek chips, and many others, all of whom have specific instructions for converting floating-point values to integer format by rounding toward zero without regard to the current rounding direction mode governing most floating-point arithmetic. This was of course the intent of the IEEE 754 committee from its earliest drafts although implementations may choose to be inefficient. Jim Valerio relates:

> When the 80387 was being defined, one issue addressed was the request for a FIST (convert to integer) instruction that always truncated rather than rounded according to the current rounding mode. After careful consideration, the designers concluded that there were no substantive performance improvements or code savings to be obtained with such a new instruction, and rejected its inclusion. When discussing this issue with several of the major suppliers of compilers for the 80387, the representatives of the compiler companies generally agreed that the perceived efficiencies of a new instruction were illusory, and that their compilers would generate the mode saving and restoring operations.

Recommendation:

Eliminate the offending sentence from the Rationale.

End of recommendation.

Note that the Draft Standard itself is unobjectionable; only the sentence in the Rationale.

Comment #8, Section 3.5.4.2: allow array parameters to have variable dimensions

I know of no C translation that's as clear as the following Fortran code:

```
      SUBROUTINE MATMUL(X,LX,Y,LY,Z,LZ,NX,NY,NZ)
      REAL X(LX,*),Y(LY,*),Z(LZ,*)
      DO 1 I=1,NX
            DO 2 J=1,NZ
            SUM=0
                  DO 3 K=1,NY
 3                SUM=SUM+X(I,K)*Y(K,J)
 2          Z(I,J)=SUM
 1    CONTINUE
      END
```

Code like this is at the heart of most of the major portable Fortran libraries of mathematical software developed over the last twenty years. The declared leading dimensions of X, Y, and Z are not known until runtime.

The Draft, like traditional C, disallows the equivalent

```
void matmul( int lx, int cx, double x[lx][cx] )
```

unless lx and cx are known at compile time. GNU CC, however, allows variably-dimensioned arrays to be passed as parameters and, even more generally, to be declared as local variables; the only requirement appears to be that the array bounds can be evaluated on function entry.

The goal is not to duplicate Fortran's features exactly, however, but rather to insure that portable linear algebra libraries are as easy to create in C as in Fortran. The ability to declare local arrays with variable dimensions is not as important as the ability to declare array arguments with variable dimensions; the former may present implementation problems in some systems while the latter is simply a way of changing the interpretation of a pointer. Thus the following is the minimum necessary:

Recommendation:

Permit arrays that are function arguments to be declared with dimensions that are integer expressions which can be evaluated at run time on function entry; that is, the integer expressions can contain constants, other arguments of that function, and globals. Change the section "Constraints" to read:

> The expression that specifies the size of an array shall be an integral *conditional-expression* that evaluates to greater than zero. Except in the case of an array parameter to a function, the size shall be a constant expression. The expression that specifies the size of an array parameter to a function may contain globals and other parameters to that function, as well as constants, as long as the size evaluates to an integral value when the function is entered.

End of recommendation.

The following formal response by X3J11 appears to be directed to a much more sweeping proposal for conformant arrays, which goes far beyond the need outlined above (and affords corresponding additional safety):

> The Committee discussed this proposal but decided against it. This invention would have far-reaching implications such as creating pointers to conformant arrays and pointer arithmetic with those pointers. Also, variable argument list processing is more complicated with variably dimensioned arrays.

None of these objections apply to the simpler proposal above.

Comment #9, Section 3.7.1: remove incorrect rationale comment

The Rationale states:

> Not many implementations can subset the bytes of a double to get a float. (Even those that apparently permit simple truncation often get the wrong answer on certain negative numbers.)

The second sentence may be omitted. Because the Draft excludes all but sign-magnitude floating-point representations, implementations that can convert a positive double to a float, rounding toward zero, by picking up the first half (VAX, IBM 370, maybe Cray) can equally legitimately convert negative numbers that way.

Comment #10, Section 4.5: don't overspecify exceptions in mathematical functions

A number of specific criticisms in my second-review comments dealt with overspecified exceptions in the mathematical library. The trend of the X3J11 drafts has been to reduce the number of constraints on exception handling, which is in general a good idea. The job should be finished as follows:

Recommendation:

Revise section 4.5.1 as follows:

4.5.1 Treatment of exceptions for mathematical functions

The following describes all the 4.5 functions, as well as the 4.10.1.4 function **strtod**.

The behavior of each of these functions is defined below for all unexceptional finite input arguments producing **double** results that are unexceptional except possibly for normal roundoff. When only normal roundoff affects the result, each function shall execute as if it were a single operation, without generating any externally visible exceptions due to intermediate steps in its implementation. When some exception renders questionable a computed result, then an implementation will return an appropriate implementation-defined value and may optionally store in **errno** the value of the macro **EDOM** or **ERANGE**. A domain error is appropriate if an input argument is outside the domain over which the corresponding mathematical function is defined. A range error is appropriate if the mathematical function result is well-defined but too large or too small in magnitude to be represented as a **double** value suffering only normal roundoff. The macro **HUGE_VAL** should be defined to be the largest positive **double** value, finite or infinite, that can result from range errors corresponding to floating-point overflow. Whether **errno** is assigned any value, and which of **EDOM** or **ERANGE** is assigned, is implementation-defined and should not be relied upon in portable code. Some implementations may compute excessively inaccurate results for certain well-defined functions because of limitations in the algorithms chosen; they should so indicate by treating arguments outside the reliable domain of those algorithms as domain or range errors.

End of recommendation.

There are several points to note about the foregoing:

* In one of the IEEE rounding modes intended to support interval arithmetic, the result of positive floating-point overflow is the largest finite positive number, while the result of negative floating-point overflow is negative infinity. Thus defining ±**HUGE_VAL** to be the result of overflow is not really appropriate.
* In its formal response X3J11 says "The description applies to model numbers only." However the Draft's 4.5.1 states "The behavior of these functions is defined for all representable values of its input arguments." To avoid distinguishing "model" from "representable" numbers here, use "unexceptional finite".
* strtod() is included as a mathematical function, as it should be. It is neither less desirable, nor more difficult, to conceal internal exceptions in strtod() than in pow().
* The current 4.5 has to provide for sensible results, for both IEEE and non-IEEE implementations, in several special cases such as atan(0,0), log(0), pow(0,0), and fmod(x,0). That's no longer necessary if the **double** result in exceptional cases is implementation-defined.
* Implementations aren't required to set **errno** since that costs everybody something but portable software can't rely on it.
* There can't be a sensible consistent rule for remembering whether **EDOM** or **ERANGE** is returned, in general, as long as log(0) is specified as **ERANGE** - defying mathematical usage and all existing implementations! Better then to avoid all further argument by leaving each case up to each implementation. If log(0) could be construed to be a range error on a system lacking a representation for infinity, so might log(−1) be construed to be a range error on a system lacking automatic extension to complex arithmetic.

* In view of the foregoing, the requirement in 4.1.3 that **EDOM** and **ERANGE** evaluate distinctly seems pointless.
* My previous proposal **EIMPL** is now encompassed for all functions by allowing **EDOM** or **ERANGE** to be signaled for e.g. large radian arguments to trigonometric functions and fmod. X3J11's interpretation "an implementation that truncates to int [at a critical point in trigonometric argument reduction] is not Standard conforming" excludes all implementations directly derived from 4.2 BSD or System V.
* The net effect is that a maximally portable application may rely neither on **errno** (since it's subject to asynchronous modification) nor on the return value of the function (since appropriate infinity or NaN return values may not exist in an implementation) and therefore must check function *arguments* for suitability.

Recommendation:

Remove or revise specific references to domain or range errors:

atan2: "A domain or range error occurs if both y and x are zero."

cos, sin, tan: Remove statement about large arguments and significance loss. The significance loss may occur for rather small arguments too. As indicated in my previous commentary, the entire issue is too complex to encapsulate usefully in a sentence. How much significance an implementation preserves is a "quality of implementation issue" which X3J11 usually declines to address. The existing wording doesn't constrain an implementation in any way, and so should be part of the Rationale if retained.

log, log10: "A domain or range error occurs if the argument x <= 0." In the Rationale, remove the sentence "The choice in the Standard, *range error*, is for compatibility with IEEE P854." Whatever the merits of range error for log(0), it is definitely not in the spirit of IEEE 854.

pow: "A domain or range error occurs if i) x < 0 and y not an integral value, or ii) x == 0 and y <= 0. A range error may occur anyway."

fmod: "A domain or range error occurs if y == 0."

strtod: "A domain or range error occurs if no conversion could be performed. A range error may occur anyway."

End of recommendation.

Comment #11, Section 4.5: tell more in the rationale

Certain aspects of implementation of mathematical functions have proven to be recurrent sites of blunders. In a number of instances X3J11 maintains that the proper interpretation of the Draft implies certain desirable conclusions. The Rationale should mention these explicitly rather than rely upon implementers' discernment.

The Rationale's wording "the implementation of this function is properly by scaled subtraction rather than division" may shed more darkness than light, being open to easy misinterpretation.

Recommendation:

In the Rationale, add

z = fmod(x,y) for finite x and finite non-zero y is uniquely and exactly defined as that z of minimum magnitude which has the same sign as x and differs from x by an integral multiple of y. The phrase "integral multiple" emphasizes that that multiple need not fit in an int, a long, or any other storage format, for it need not be explicitly computed. Implementations that choose to compute fmod(x,y) by a *formula* like

```
x – y * (int)(x/y)
```

are not conforming. Instead, fmod could be computed in principle by subtracting ldexp(y,n) from x, for appropriately chosen decreasing n until the remainder is between 0 and x,

although efficiency considerations lead to better implementations.

End of recommendation.

Recommendation:

Add a statement in the Rationale:

Trigonometric argument reduction should be performed by a method that causes no catastrophic discontinuities in the error of the computed result. In particular methods based solely on naive application of a *formula* like

$$r = x - (2 * \pi) * (int)(x/(2 * \pi))$$

are not conforming.

End of recommendation.

Note that conforming fmod and trigonometric functions can't be efficiently implemented on certain mainframes (Cray, CDC, ETA, ...) since lacking a guard digit, they can't even perform correct subtraction efficiently.

Recommendation:

Add to the Rationale:

The ceil function returns the smallest integral value in double format greater than or equal to x, even though that integral value might not fit in a variable of type int or long. Thus ceil(x) == x for all x sufficiently large in magnitude; for IEEE 754 double precision, that includes all x such that fabs(x) >= 2**52. An implementation that implements ceil(x) as (double)(int)x for negative x, for instance, is not conforming.

End of recommendation.

Comment #12, Section 4.5: standardize hypot

Recommendation:

Standardize hypot():

4.5.6.5 The hypot function

```
#include <math.h>
double hypot (double x, double y);
```

Description

The hypot function is intended to compute sqrt(x*x+y*y).

In the Rationale, add "some implementations of hypot are especially susceptible to gratuitous intermediate underflow and overflow. This must be suppressed for hypot just as for the other mathematical functions."

End of recommendation.

X3J11's previous response on this issue was:

This was considered to be an invention of limited utility. Although this function has uses, the Committee felt because this function did not exist in the base document and because of the limited scope in which it is useful that there was insufficient reason to add it to the Standard.

This response is difficult to understand since hypot has been defined and used for many years in 4.2 BSD and System V. Unlike atan2, which despite another X3J11 assertion is principally associated with transformations equivalent to rectangular to polar coordinate conversion, hypot appears to be of greater interest in graphics, robotics, and matrix computations, so much so that several papers have

been published explaining ways to compute hypot without spurious overflow and underflow (Blue) and without an explicit sqrt (Moler, Dubrulle).

Comment #13, Section 4.5.4.6: delete modf

X3J11 claims that important portability issues mandate retention of frexp. However no such claim can be made for modf, since it has been defined and implemented in various incompatible ways. It provides no essential function.

Recommendation:

Delete modf from the Draft.

End of recommendation.

Comment #14, Section 4.7: specify which signals can arise

Some implementations will never detect any exceptions of particular classes (such as floating point) under any circumstances. This can be exploited if <signal.h> contains #define can_SIG... for each X3J11-standardized SIG that an implementation can generate by means other than raise():

Recommendation:

Add to 4.7:

For each standardized exception SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM, an implementation's <signal.h> also defines macros

```
#define can_SIGABRT
#define can_SIGFPE
  ...
```

for each signal that can arise by means other than raise().

End of recommendation.

This recommendation is repeated because of the perplexing formal response received from X3J11:

A specific proposal is needed before action can be taken. No prior art exists for this feature. Many aspects of the signal function are implementation defined. This is due to the wide variety of implementations in existence, each of which has unique requirements.

Conclusion

The X3J11 Rationale acknowledges many difficult issues that arose from conflict between the descriptive and prescriptive approaches to standardization, and like many language standards the result often tends toward the descriptive. Since X3J11 is unlikely to want to transcend previous language standards by closely specifying the entire C numerical environment, it should avoid prejudicing the issue in any of the ways mentioned above. Then another body, consisting of those knowledgeable about numerical issues and those affected by decisions about numerical issues, may properly build upon the ANSI C standard by adding to it, rather than subtracting from it, in order to define a suitable environment for scientific computation.

Acknowledgements

Although they have not expressed agreement with the foregoing recommendations, the following persons generously contributed detailed criticism of earlier versions of these comments:

Greg Astfalk	convex!c1east!astfalk
Nelson Beebe	Beebe@SCIENCE.UTAH.EDU
Larry Breed	ucbvax!ibmpa!lmb
John Gilmore	hoptoad!gnu
Karl Heuer	karl@haddock.isc.com
Blair Houghton	bph@buengc.bu.edu
Iain Johnstone	iainj@playfair.stanford.edu
Earl Killian	ames.arpa!mips!earl
Tom MacDonald	Cray Research
David Mendel	mendel@playfair.stanford.edu
K-C Ng	kcng@sun.com
Richard O'Keefe	quintus!ok
Philippe Toint	phtoint%bnandp10.bitnet
Scott Turner	husc6.harvard.edu!panda!genrad!mrst!sdti!turner
David Wolverton	att.arpa!houxs!daw