# DESIRABLE FLOATING-POINT ARITHMETIC AND ELEMENTARY FUNCTIONS FOR NUMERICAL COMPUTATION

T.E. Hull
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A7

The purpose of this paper is to summarize proposed specifications for floating-point arithmetic and elementary functions. The topics considered are: the base of the number system, precision control, number representation, arithmetic operations, other basic operations, elementary functions, and exception handling. The possibility of doing without fixed-point arithmetic is also discussed.

The specifications are intended to be entirely at the level of a programming language such as Fortran. The emphasis is on convenience and simplicity from the user's point of view. The specifications are not complete in every detail, but it is intended that they be complete "in spirit" - some further details, especially syntatic details, would have to be provided, but the proposals are otherwise relatively complete.

"... *the expression 25+1/3 yields the value 5.333333 in PL/I, and .333334 in PL/C ...*"
Conway and Gries, An Introduction to Programming, 1975, p.24.

## Introduction

There has been a great deal of progress during recent years in the development of programming languages. However, almost all of this progress has been under the general banner of "structured programming" and almost no attention has been paid to those aspects, such as the semantics of floating-point operations, that are of special interest to practitioners who are interested in numerical computation.

The purpose of this paper is to propose some specifications for floating-point arithmetic and elementary functions. The main design goal is to produce a set of specifications which is most desirable from a user's point of view. There is of course no claim that the set is unique. In fact, many details, especially syntatic details, have been omitted because there are obviously so many possible variations that would be equally acceptable. However, there are some basic requirements that, in one form or another, do seem to be desirable, and perhaps even necessary. It is convenient to illustrate them in terms of concrete examples.

The specifications are described in the following sections in terms of the base of the number system, precision control, number representation, arithmetic operations, other basic operations, elementary functions, and exception handling. These sections are followed by another that is devoted to a discussion of fixed-point arithmetic. Fortunately, these different aspects are to a large extent independent of each other, so that it is usually possible to propose alternative specifications in one aspect without seriously affecting the others.

It should be emphasized that the specifications are intended to be entirely at the level of a programming language such as Fortran. For example, in discussing arithmetic operations, our concern is entirely with the syntax and semantics of the programming language expressions. It may be convenient to refer to hardware representations as a way of discussing the details of the arithmetic, but the details of the hardware implementation are irrelevant. (Of course, we must have hardware possibilities in mind, since it would be foolish to propose specifications that are not practical to implement, but the implementations themselves are not intended to be part of the specifications in this paper.)

We feel that it is important to consider such specifications for floating-point arithmetic and elementary functions. Indeed, users who are interested in numerical computation have an obligation to try to reach a consensus on such specifications, unless they are prepared to put up forever with whatever facilities the manufacturers and language designers are, perhaps grudgingly, willing to provide. If some sort of consensus became possible, it could evolve towards a standard. And with the technology changing as rapidly as it is, such a standard may not be too difficult to achieve, or at least to approach much more closely than is the case at present. In any event, with a standard agreed upon, we would at least have a basis against which we could judge the appropriateness of various trade-offs.

The usefulness of a standard in terms of portability of numerical software, and particularly in terms of portability of proofs about what the software does, is obvious. One fortunate circumstance is that some proofs can of course be portable, even under circumstances which do not meet all of the requirements of a standard. But closer adherence to standards would make the situation better, and would lead to other improvements as well, in terms of convenience, programmer efficiency, and so on.

An ideal arithmetic system should be complete, simple and flexible. Completeness means that the programmer knows what will happen under any circumstance. Simplicity leads us to conclude, for example, that the base should be 10. We will also argue that there is no need to have a second number system (such as "fixed-point" or "integer") since the floating-point system can serve all purposes. For simplicity we also argue for a particular way of determining the precision at which calculations are performed. We choose a flexible way of controlling precision, and also a flexible mechanism for coping with exceptions such as overflow and underflow.

An ideal system for elementary functions is more difficult to agree upon. Completeness, in the sense of always producing the same results whenever the precisions are the same, is probably desirable here too. But what is more to the point at this stage is that we emphasize simplicity, and this leads us to require only a single simply-stated accuracy requirement for all elementary functions. In particular, we argue against insisting that a long list of additional special properties be required to hold. However, we do also need flexibility in precision control for the elementary functions, to match the precision control of the arithmetic.

## Base

Much has been written about different number bases, and their relative merits with respect to efficiency of storage, round-off behavior, and so on. We believe that simplicity and convenience for the user should be the primary consideration and this means that

*the choice of base is 10.* (1)

With this choice, a number of annoying problems disappear immediately: for example, the "constant" 0.1 will really be exactly one tenth, and any value that is input will be exactly the same on output. Moreover, with base 10, not even a perverse compiler writer would dare allow the compiled value for a number to differ from its input value.

Requiring the base to be 10 does not mean that each decimal digit must be represented internally as a separate entity. The hardware could, for example, use 10 binary digits to represent 3 decimal digits, if that were an economical thing to do. The point is that the results must appear to the user to be the same as if the base of the number system were 10.

Input-output will be simplified with many programs, especially in date processing, if base 10 is used internally. Programmer efficiency will improve if the programmer does not have to keep in mind the peculiarities of other number bases. It may even happen that a base-10 system, including a base-10 internal representation, would turn out to be, overall, the most efficient, besides being the simplest and most convenient for the user.

Questions about which numbers are to be included in the number system, and questions about the results of arithmetic operations on these numbers will be considered in later sections.

## Precision

The main questions related to precision can be considered independently of the number base. It is possible to describe specifications that would be desirable, whatever the base. However, for purposes of illustration, we will do so only in terms of base 10, but the corresponding statements for other bases will be quite obvious.

Earlier versions of what will be proposed in this section, including the description of a preprocessor for implementing the main ideas, have been discussed elsewhere by Hull and Hofbauer[2,3]. It is hoped that further results will also be developed in a future paper. We will therefore consider only the essential ideas in this section.

It is important that the user have control over the precision. In an ideal system, we believe that

*the user should be able to specify separately the number of digits to be used for the exponent of his floating-point values, and the number of digits to be used for the fraction part.* (2)

Ideally he should be able to make a declaration such as

```
FLOAT(2,12) X
```

and as a result have the value of X composed of a 2-digit exponent part along with a 12-digit fraction part. (The exact details will be discussed in the next section.)

It should also be possible that

*variables or expressions, as well as constants, be allowed in the declarations.* (3)

For example,

```
FLOAT(2,I+1) X
```

would have the obvious meaning.

The most important part of our proposal with respect to precision is that

*the user should be able to specify the precision of the operations to be carried out on the operands, quite apart from, and independently of the precision of the operands themselves.* (4)

For example, he should be able to write something like

```
BEGIN PRECISION(3,14)
    ____
    ____
    Y = X + .51 * SIN(X)
    ____
END
```

and mean that every operation in the expression is to be carried out in (3,14)-precision arithmetic, the result of the calculation finally being adjusted to fit the precision of Y, whatever the precision of Y has been declared to be, before the result is assigned to Y.

It is of course intended that

*the precision of such "precision blocks" be allowed to change between one execution of a block and the next.* (5)

Examples are given in the references by Hull and Hofbauer referred to earlier; however, the preprocessor mentioned there handles only the special case in which only the fraction parts (of the variables and precision blocks) are declared, and their values denote numbers of word lengths rather than numbers of decimal digits.

The specifications we propose for precision control provide a considerable degree of flexibility. In particular, they allow the user to carry out intermediate calculations in higher precision (as may be required, for example, in computing scalar products, or in computing residuals), and they allow for the repetition of a particular calculation in different precisions (as is required, for example, in some iterative procedures, or in attempting to measure the effect of roundoff errors).

The proposed specifications are also simple. For example, separating the precision of the operations from the precisions of the quantities entering into the calculations avoids having to remember a lot of rules about how quantities of different precisions combine. (No satisfactory rules for such calculations can be devised in any event; for example, no such rules would enable us to compare the results of doing a particular calculation twice, at two different precisions.)

It must be acknowledged that very high precision calculations would be used only rarely. This means that all precisions up to something like (2,12) or perhaps (3,15) should be done very efficiently, but, beyond that, a substantial reduction in efficiency would be quite acceptable.

One point is perhaps worth emphasizing. It is intended that precision 12, say, means exactly precision 12, and not at least precision 12. We cannot measure roundoff error if precision 12 and precision 15 give the same results. Further details about the arithmetic will be considered later.

One further point is perhaps worth mentioning. Our requirements for precision control could lead to thinking of the machine as being designed to handle character strings, a number being just a special case in which most of the characters in a string are decimal digits. However, as indicated earlier, we are concerned here primarily with the functional specifications, and not with any details about how those specifications are to be implemented.

## Representation

Quite independently of how the base is specified, or of what sort of flexibility is allowed with the precision, it is possible to state specific requirements about the representation of floating-point numbers. We will describe what we consider to be desirable requirements in terms which may appear to be hardware specifications but, once again, the proposal is not meant to restrict the details of the hardware representation in any way except in so far as the results appear to the user.

The proposal is that

*a sign and magnitude representation be used for both the exponent part and the fraction part, and that the fraction part be normalized.* (6)

Thus, if a value is declared to be FLOAT(2,3) in our earlier notation, its largest possible magnitude is $.999 \times 10^{99}$. Its smallest possible positive magnitude is $.100 \times 10^{-99}$.

The reason for proposing a sign and magnitude representation is that it is simple, and probably easiest to keep in mind. The reason for allowing only normalized numbers is so that the fundamental rule regarding error bounds that is discussed in the next section can then be relatively simple.

We deliberately do not propose any axioms, such as "if x is in the system then so is -x", to be satisfied by the numbers in the system. Any such statements that are valid are easily derived, and there is no need to state them explicitly. In fact, it might be somewhat misleading to begin with statements of this sort and perhaps give the impression that one might be able to derive the system from a collection of such desirable properties.

Besides the normalized floating-point numbers proposed above

*it will be necessary to allow a few other values as well, such as OVERFLOW, UNDERFLOW, ZERODIVIDE, INDETERMINATE, and UNASSIGNED to be used in special circumstances.* (7)

We will return to this question in a later section when we discuss the requirements for exception handling.

Although what we have proposed as allowed values for floating-point numbers is, for the purpose of simplicity, very restricted, the hardware can carry much more in the way of extended registers, guard digits, sticky bits, and so on, if that should be convenient for meeting the requirements of the following sections. However, if this is done, it will ordinarily be only for temporary purposes, and, in any event, the user would under no circumstances have access to such information. (We are continuing to think of the user as programming in a higher level language such as Fortran.)

## Arithmetic operations

Whatever the base or method of representation, we can still be precise about the kind of arithmetic that is most desirable. For various reasons we propose that,

> *in the absence of overflow, underflow, indeterminate, and zero-divide, the results of all arithmetic operations be properly rounded to the nearest representable number. (Some further detail is needed to make this requirement completely precise. In case of a tie, we might as well have the normalized fraction part rounded to the nearest even value.)* (8)

Note that, for numbers of the form FLOAT(2,3), this will mean that every result $\geq .9995 \times 10^{99}$ will overflow, whereas every result $< .09995 \times 10^{-99}$ will underflow.

There are several reasons for preferring this specification:

(1) It is simple, and easy to remember.

(2) Since unnormalized numbers are not allowed, the basic rule required for error analysis is easy to derive and, in the absence of overflow, underflow, indeterminate, and zero-divide, takes the simple form:

$$f\ell(x \circ y) = (x \circ y)(1+\varepsilon),$$

where $\circ$ is an operation and $|\varepsilon| < u$, u being the relative roundoff error bound for the precision that is currently in effect.

(3) Rounding is better than chopping, not because the value of u is smaller (although that happens to be the case), but primarily because of the resulting lack of bias in the errors.

There is a considerable advantage to stating directly what outcome one is to expect from an arithmetic operation, and then deriving any properties that one needs to use, rather than to start off with a list of desirable properties. For example, from the simple specification we have given, it is a straightforward matter to prove that (sign preservation):

$$(-x)*y = -(x*y),$$

or that (monotonicity):

$$x \leq y \text{ and } z \geq 0 \text{ implies } x*z \leq y*z.$$

It is misleading to write down a list of such desirable properties and to suggest that rules might be derived from them. (After all, if we did write down <u>all</u> of the most desirable properties we would of course want to include associativity!)

It is undesirable to allow any exceptions to the specifications - even such small ones as the exception to true chopping arithmetic that occurs with IBM 360/370 computers. The reason is that it is important for the user to <u>know</u> what happens under all circumstances. A simple rule, that is easy to remember and to which there are no exceptions, is a good way to ensure this knowledge.

A consequence of this insistence on no exceptions is that no tricks, such as arranging to evaluate A*B+C with at most one rounding error, can be allowed. It is essential that the product A*B be rounded before C is added. If the rule could be violated in this case (for a "bit" of extra accuracy!), how many other exceptions would you allow, and would the user have to remember? And would it be desirable to have the proof of what a program does depend on such a trick? (If extra accuracy is needed in the evaluation of a particular expression, that requirement should be made quite explicit in the program itself - and the programming language should provide facilities for doing just that, as proposed in the earlier section on precision.)

To complete the programming language specifications with regard to floating-point arithmetic, we also require that

> *some conventions be adopted, such as the left to right rule for resolving ambiguities in expressions such as A+B+C.* (9)

A further discussion of overflow, underflow, etc., is also required, but that will be postponed to the section on exception handling.

## Other basic operations

Besides the arithmetic operations, a programming language must of course also provide various other basic operations. These should include such standard operations as

> *absolute value*
> *the floor function,*
> *quotient, remainder,* (10)
> *max, min,*

as well as

> *the relational operators.* (11)

With the latter it is essential that they work properly over the entire domain, and that, for example, nothing ridiculous happen such as allowing IF(A > B) to cause overflow.

There would also be a need for functions to perform special rounding operations, such as *round the result of an arithmetic operation to a specified number of places in the fraction part, round up, or round down, similarly, round a result to a specified number of places after the point* (12)

and to carry out other special operations, such as

*get precision of fraction part, get precision of exponent part.* (13)

Finally, a special operation may be needed to denote

*repeated multiplication.* (14)

The purpose of this operation is to distinguish $x^n$, where n is an integer and it is intended that x be multiplied by itself n-1 times, from the case where it is intended that $x^n$ be approximated by first determining log x and then computing $e^{n \log x}$. Being able to make this distinction would be helpful in calculating expressions such as $(-1)^n$, or $(3.1)^3$. But whether this part of the proposal is accepted depends to some extent on how strongly one feels about dropping the fixed-point or integer type, as discussed in a later section.

## Elementary functions

For the elementary functions, such as SQRT(X), EXP(X), SIN(X), etc., we propose requiring only that

$$fl(f(x)) = (1+n_1\varepsilon)f(x(1+n_2\varepsilon))$$

*over appropriate ranges of x, where $n_1$ and $n_2$ are small integers. (Of course, each ε satisfies $|\varepsilon| < u$, and the value of u would depend on the precision.)* (15)

It would be a nice feature if the n's were relatively easy to remember. For example, it might be possible to require $n_1 = 2$ for each function, and $n_2 = 0$ for at least most of the functions of interest. Unfortunately, the "appropriate ranges" will differ, although they will be obvious for some functions (for example, they should contain all possible non-negative values of x for the square root function, and all possible positive values for the logarithmic function).

There is a temptation to require more restrictions on the approximations to the elementary functions, such as

```
SIN(0) = 0,        COS(0) = 1
LOG(1) = 0,   ABS(SIN(X)) ≤ 1
```

or that some relations be satisfied, at least closely, such as

$$\text{SQRT}(X^2) = X,$$
$$(\text{SQRT}(X))^2 = X,$$
$$\text{SIN}(-X) = -\text{SIN}(X),$$
$$\text{SIN}(\text{ARCSIN}(X)) = X,$$
$$\text{SIN}^2(X) + \text{COS}^2(X) = 1,$$

or that some monotonicity properties be preserved, such as

$$0 \le X \le Y \text{ implies SQRT}(X) \le \text{SQRT}(Y).$$

A few such properties follow from the proposed requirement (for example, SIN(0) = 0), but we propose not requiring anything beyond what can be derived from the original specification. This proposal is made in the interests of simplicity. The original specification is easy to remember, and any proofs about what programs do should depend only on a relatively few "axioms" about floating-point arithmetic and the elementary functions. No one is required to remember a potentially long list (and perhaps changing list!) of special properties of the elementary function routines.

In those cases where something special is required, the programmer should take appropriate measures. For example, if it appears that we might want to require that $|\sin(x)| \le 1$, as we might in trying to approximate the integral $\int_0^\pi \sqrt{1-\sin x}\,dx$, we can simply replace 1-sin x with $|1-\sin x|$. Moreover, separate function subroutines can always be developed in order to provide function approximations that satisfy special properties; for example, there could be a special sine subroutine, say SSIN, which produces approximations to sin(x) with special properties such as being guaranteed not to exceed 1 in absolute value.

We should have to remember only a small number of basic "axioms" - and then program defensively.

## Exception handling

We turn now to the question of what should be required in our programming language to enable a user to cope effectively with exceptions. There are quite a few different kinds of exceptions that can arise. Overflow, underflow, indeterminate, and zero-divide have already been mentioned. (It may be that one would like to make further distinctions here, between positive and negative overflow, for example.) It should be pointed out that overflow and underflow can occur when precision is changed, especially if the user can change the exponent range. Other exceptions that can arise include trying to compute with an as yet unassigned value, or using a function argument that is out of range.

The first rule should be that,

*if an exception arises and the programmer makes no special provision for handling it, the computation should be stopped, along with an appropriate message about where and why.* (16)

If the user is aware that an exception might arise, and knows what he wants to do about it, he can often "program around" the difficulty. One example has already been mentioned in connection with an argument getting out of range in $\sqrt{1-\sin x}$. Another arises in trying to calculate $\min(|y/x|,2)$ where $y/x$ might overflow. Here, unless $2x$ can overflow, the calculation can be replaced with "if $|2x| < |y|$ then 2 else $|y/x|$".

The second of these two examples is somewhat harder to read, and there clearly is a limit beyond which a user should not be expected to go. It certainly would be advantageous to have a simple way of stating what is to be done in case a particular exception should arise. In the second example mentioned above, it would be clearer if we could write $\min(|y/x|,2)$ and have some way of indicating that overflow is to be ignored if it happens to occur.

Probably most cases where the user knows that an exception can occur and knows what he wants to do about it are of this sort. They occur in a single statement of the program, and one simple solution is to replace an overflowed quantity with the largest number that can be represented (at that precision), or to replace an underflowed quantity with zero. A simple way of appending one such "fix" or the other to a statement is all that is required.

However, a more general capability will sometimes be needed as well, for example when underflows may occur at many places in a relatively large block of code, or when the user might wish to save data, and/or output a substantial amount of information if an exception should arise.

We believe that it is more important to insist on a general capability as our main requirement. Our second rule with regard to exception handling is therefore that

*the user should be able to specify the scope over which he is prepared to state what is to be done, and he should be able to detect the cause of the interrupt, in a way such as is suggested in the following:*

```
BEGIN
     ON(OVERFLOW)
          ——— } what to do in case
          ——— } of overflow
       (UNDERFLOW)                         (17)
          ———
          ———
       (————)
          ———
     END  ———
     ——— } scope
     ——— }
END
```

*Besides OVERFLOW and UNDERFLOW, the other possible causes of interrupts are ZERODIVIDE, INDETERMINATE, UNASSIGNED, and OUTOFRANGE (i.e., argument of a function out of range).*

Third, it is to be understood that

*control will be returned to the point of interruption, after the specified action has been taken, unless the programmer has provided for an alternative to be followed, such as stopping the calculations altogether, or perhaps making an exit from that block of instructions.* (18)

Fourth, it is also proposed that

*the programmer be able to assign a value to RESULT as part of the action to be taken. For example, he could write* (19)

```
ON(OVERFLOW) RESULT = 10**50
  (UNDERFLOW) RESULT = 0
END
```

Not allowing the user to have access to the operands, other than through his access to the program variables themselves, has been deliberate. In particular, if the operands that caused the interrupt were "temporaries", it is difficult to see how he could make use of them.

A fifth part of our proposal is designed to provide, in a convenient way, for those special cases like $\min(|y/x|,2)$ that are likely to arise quite frequently. We merely propose that

*there be a short form that can be used in such special cases, as illustrated in the following two examples:*

```
Z = MIN(ABS(Y/X),2) EXCEPT                    (20)
       ON(OVERFLOW) RESULT = 100

W = U*SIN(V) EXCEPT ON(OUTOFRANGE) EXIT
```

Finally, we believe it would be helpful to require that

*output should be provided at the end of any calculation in which ON clauses are executed, the output being designed to indicate which such clauses were executed along with some statistics about the number of such executions, and so on.* (21)

## Fixed-point arithmetic

In conclusion, we would like to comment that, at least to us, it appears that we do not need to have any type of arithmetic in our programming language other than the floating-point arithmetic described in the preceding sections (except, of course, for complex arithmetic). In particular, there does not appear to be any compelling need for fixed-point or integer arithmetic.

The floating-point arithmetic already described can be used for subscripts, and for counting (with base 10 we could even have the convenience of DO I = 1 TO 2 IN STEPS OF .1). In the case of subscripts, a non-integer value would be an error, of course - just as in the case of an out-of-range subscript, or an out-of-range value of a function argument.

Floating-point arithmetic is difficult enough. We have to get used to non-associativity, and to such peculiarities as the possibility that (A+B)/2 may be less than both A and B. Let us not be forced to learn another system as well - along with all the messy details about conversion between the two systems.

We pay a price for the simplicity of having only one system, but the price may well be worthwhile. One part of the price to be paid, at least from the point of view of the user (as opposed to the designer of the hardware), is that the "integers" would have to be declared as, for example, FLOAT(1,4). This seems to be a small price to pay, and, in fact, even this small price could be avoided if we were willing to provide a suitable default meaning for FLOAT. Another part of the price has already been mentioned: it may be necessary to have a special way of denoting $x^n$ when it is understood that n is an integer and successive multiplications are intended. A third part of the price to be paid for the simplicity of having only one numerical type would be the need for some way of determining when the result of an arithmetic operation was not exact. For example, instead of causing overflow, the product of two integers might cause a roundoff error, and there would have to some way of determining that this has happened; another ON condition may be what is needed.

If there were to be a second kind of real-valued arithmetic, it would be useful to consider rational arithmetic. Within the scheme described in this paper, rational arithmetic can be viewed as a variable-precision arithmetic in which the precision is determined by the numerical quantities themselves, rather than by the programmer.

## Acknowledgements

Much of this work is based on material that was prepared for discussion by members of the IFIP Working Group 2.5 on Numerical Software, beginning with the Oak Brook meeting in 1976.[1] I would like to acknowledge the helpful criticism of members of that committee, particularly of W.S. Brown and T.J. Dekker.

## References

1. T.E. Hull, "Semantics of Floating Point Arithmetic and Elementary Functions", Portability of Numerical Software (edited by W.R. Cowell), Springer-Verlag, N.Y., 1977, pp.37-48.

2. T.E. Hull and J.J. Hofbauer, "Language Facilities for Multiple Precision Floating Point Computation, with Examples, and the Description of a Preprocessor", Technical Report No.63, Department of Computer Science, University of Toronto (1974).

3. T.E. Hull and J.J. Hofbauer, "Language Facilities for Numerical Computation", Proceedings of the ACM-SIAM Conference on Mathematical Software II, Purdue University (1974), pp.1-18.