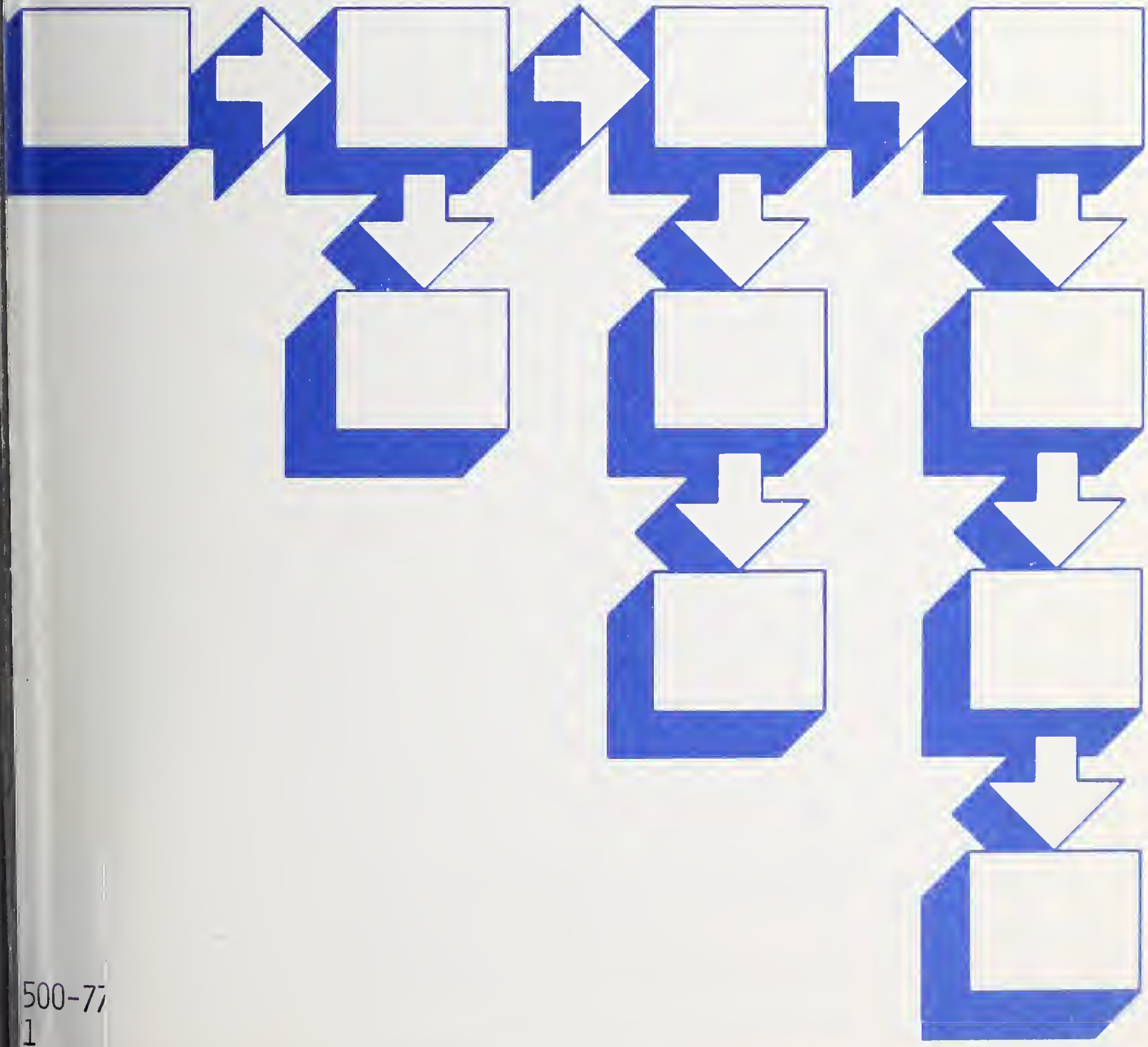
S. Department
Commerce

tional Bureau
Standards

Computer Science and Technology

NBS Special Publication 500-77

Specifications and Test Methods for Numeric Accuracy in Programming Language Standards



NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Thermodynamics and Molecular Science — Analytical Chemistry — Materials Science.

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Mechanical Engineering and Process Technology[2] — Building Technology — Fire Research — Consumer Product Technology — Field Methods.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1] Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2] Some divisions within the center are located at Boulder, CO 80303.

Computer Science and Technology

NBS Special Publication 500-77

Specifications and Test Methods for Numeric Accuracy in Programming Language Standards

John V. Cugini

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued June 1981

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-77
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-77, 42 pages (June 1981)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 81-600056

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1981

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402
Price $2.75
(Add 25 percent for other than U.S. mailing)

Specifications and Test Methods for Numeric Accuracy in Programming Language Standards

by John V. Cugini
Institute for Computer Sciences and Technology
National Bureau of Standards

Abstract. This publication formulates language-independent and machine-independent criteria for assessing the quality of floating-point arithmetic operations and functions. The criteria require that results be within the limits generated by perturbing the arguments or operands by a specified amount, and thus allow for the mathematical instability of some functions at certain arguments and also for the granularity of numeric representation inherent in digital machines. Automatic test methods derive naturally from the accuracy requirements. Model algorithms for testing are included.

Key words: Computer arithmetic; conformance testing; numeric accuracy; programming language standards

Table of Contents

1. INTRODUCTION

1.1 The Problem Of Specifying Numeric Accuracy

1.1.1 Standardization Problems

In order to support program portability, programming language standards [FORT78, BASI78, PL/I76, Jens75, Ada8Ø] endeavor to specify the behavior of standard-conforming programs. In the ideal case, the language standard would assign a unique semantic meaning to each standard program. Thus, a standard-conforming program would produce exactly the same results even when executed on different kinds of computing hardware, given standard-conforming language processors for those systems.

In certain functional areas, such as integer arithmetic, character string manipulation, and control structures, the actual situation closely approaches the ideal. In so-called "real" or "floating point" arithmetic, however, this is not so, largely for two reasons. First, there is no one, obviously "best", way for digital hardware to model the capabilities in question, and so one abstract operation, such as division, has a variety of mutually inconsistent implementations. There will be more detail on the problems of simulating real arithmetic on discrete machines throughout the paper. Second, implementations of real arithmetic in a language are almost always closely bound to underlying hardware, among which there is such diversity. A language primitive, again taking division as an example, is typically handled directly by a corresponding hardware primitive (not, for instance, by a software routine operating on strings whose characters represent the operand values).

How then can a language standard treat real arithmetic? Let us distinguish three approaches (from strictest to most lenient):

1. Specify exactly what the results of any arithmetic expression must be, perhaps in terms of an abstract machine. This approach preserves the strict notion of standardization mentioned above: for each program there is a unique interpretation.

2. Formulate some criterion of accuracy which does not uniquely specify a result for any given operation or function, but rather sets a constraint which processors must obey in order to conform. Thus, for a standard program, there exists a set of "legal" outcomes, any one of which a processor is free to assign.

3. Explicitly or implicitly disavow any criterion of accuracy. As long as a processor recognizes the syntax associated with the language's functions and operations, it may assign any value as the result and still conform, technically, to the standard.

The first approach, while theoretically satisfying, is not practical, in that few if any implementors could be expected to abide by such a strict rule. Moreover, even if there were conforming processors, specification of a unique answer would necessarily place an upper as well as lower bound on the accuracy of the result. That is, under such a rule, processors would be forbidden to achieve a <u>more</u> accurate result than specified, since this would conflict <u>with</u> the standard just as would a less accurate result.

The third approach is, in fact, the one currently being used by many of the popular languages with real arithmetic, e.g., BASIC, FORTRAN, Pascal, and PL/I (Ada does specify an accuracy rule for computation with real numbers which is similar, although not identical, to the approach which will be described herein). The disadvantages of failure to specify accuracy should be apparent; it is not a strategy for standardization, but rather an abstention from standardization. The theoretical problem is that there is no result a processor can generate, no matter how outlandish, for which one can cite the standard to show that the processor has failed to conform. More practically, such a "standard" gives implementors no guidance as to what level of accuracy their processors should achieve, and so lower quality arithmetic is encouraged by default. Furthermore, purchasers have no way to compare language processors with respect to the quality of their mathematical functions. In short, it simply does not suffice for a language standard to appeal to our intuitions about what constitutes reasonable behavior for a processor, nor can a standard naively state that "+" means addition in the formal mathematical sense and let it go at that. Computers can do useful simulations of real arithmetic but simulation is not identity and it is the job of the standard to specify the capabilities and limitations of the processor's implementation of real arithmetic.

It is the contention of this paper that the second approach is the preferable alternative; that it would address the problems of standardization and quality just mentioned, while still being a realistic goal implementors could reasonably be expected to achieve.

1.1.2 Implementation Problems

If setting accuracy limits is a good idea, why hasn't it been done? One practical problem is that word lengths (to hold floating point values) differ among machines and it would be unwise for a language standard to make conformance difficult simply because of hardware constraints. More importantly, there is the general problem of characterizing the behavior (and any useful criterion of accuracy will depend on this characterization) of discrete simulation of the mathematics of real numbers. If there were some simple exact model for such simulation (such as exists for character manipulation, e.g., one character per byte) we would expect that language processors and the underlying hardware would long since have adopted this model, and no problem would ever have arisen. Note that this is the situation that prevails for logical operations which are themselves discrete in nature and thus allow for exact simulation by discrete machines. Few difficulties attend the implementation of string concatenation or substring operations just because exact simulation is possible and therefore all implementors converge on the exact logical specification. But real mathematics is inherently non-discrete. No machine can return the exact value for SIN (Ø.2), since this is an irrational number. The impossibility of exact simulation of real mathematics leads to diverse implementation, hence the difficulty of formulating an acceptable accuracy criterion.

What are the practical implications of these theoretical considerations? What, concretely, are the sources of inaccuracy?

1. The granularity of numeric representations in discrete machines. The general problems discussed above find their hardware expression in the finite (and usually fixed) number of bits for the mantissa and exponent of a floating point number.

2. The inherent instability of some functions and operations for certain arguments. It is easy to produce cases where a small error (relative or absolute) in the argument will generate a much larger error in the result. We will see that this instability interacts with machine granularity, to ill effect.

3. Poor algorithms, both at the hardware level for elementary operations and in subroutine libraries for functions.

Clearly, when testing how well a computer system has implemented its mathematical capabilities, we need somehow to build into our test criterion a recognition of the unavoidable limitations imposed by the first two items. Any inaccuracy beyond that attributable to these causes must be presumed to result from sub-optimal implementation and it is just this that we would like to discourage.

1.2 Design Goals

The objective, then, is to formulate a criterion of accuracy for floating point operations and functions which meets the following specifications:

1. Since the problem of standardization cuts across language and machine boundaries, the criterion should apply generally to floating point arithmetic. It should be based on only minimal assumptions (and these should be explicit; see next

item) about the implementation mechanism. It should not be tied to properties peculiar to individual programming languages or computing hardware. In particular, the criterion will neither imply nor rely on any standardization of mathematical operations at the bit or hardware level. Hardware standardization is a worthy but distinct goal, currently the object of an IEEE effort [Coon8Ø].

2. We will assume only that real values are represented in familiar fixed precision, floating point form. Throughout, the convention shall be that "d" represents the number of digits of precision, and "b" the base. There is no provision for fixed point numbers, nor for variable precision or multi-precision capabilities.

3. The criterion, to be meaningful, must reject poor implementations. On the other hand, it should be possible and practical for a conscientious implementor to meet the criterion, using currently available hardware. This means, for instance, that the problems of machine granularity and mathematical instability must be taken into account.

4. The criterion should provide an explicit and objective metric for the quality of implementation. This is important not only so that we can determine whether or not a processor conforms to the language standard, but also so that purchasers will have a reasonable means of comparison when judging processors' mathematical facilities.

5. The application (even if not the justification) of the criterion ought to be exoteric, especially in that one of its principal purposes is incorporation into language standards.

2. MATHEMATICAL INSTABILITY

In this section, we will focus on functions of a single variable. The results apply equally well, however, to the arithmetic operations, which can be thought of as functions of several variables. This generalization will be explicit in later sections.

It is important to realize that a function may be extremely sensitive to an error in its argument(s), and that this holds for relative as well as absolute error. That is, just as a given absolute error in an argument does not imply a similarly small or large absolute error in the value of the function, so also a given relative error in an argument does <u>not</u> bound the resulting relative error in the function value. In the case of absolute error, we may expect the error in the function to be magnified roughly by the slope of the function in the vicinity of the argument, i.e., approximately:

$$\text{absolute error in } f(x) = f'(x) * \text{absolute error in } x$$

Similarly, there is a relative error magnification factor, such that, approximately:

$$\text{relative error in } f(x) = \frac{f'(x) * x}{f(x)} * \text{relative error in } x$$

For example, sin(3.1416) is quite sensitive to relative errors, and arccos(.ØØØØØØ1) quite insensitive. For further details see Sterbenz [Ster74].

The point is simply that knowing the argument accurate to one part in a million does not guarantee similar accuracy in the resulting function value. This is a proposition of mathematics and has nothing to do with the characteristics of computers. Any reasonable criterion of accuracy for computer evaluation of functions must, however, allow for the relative sensitivity, or insensitivity, of the function and argument in question.

3. MACHINE GRANULARITY

Machine granularity refers to the inability of a discrete machine to represent real numbers exactly. Specifically, arguments to functions will generally be represented with some unknown but bounded error. As discussed in the last section, even a small error in the argument may produce a large error in the function value. Clearly we need to understand the relationships among machine granularity, accuracy, relative error, and numerical computation.

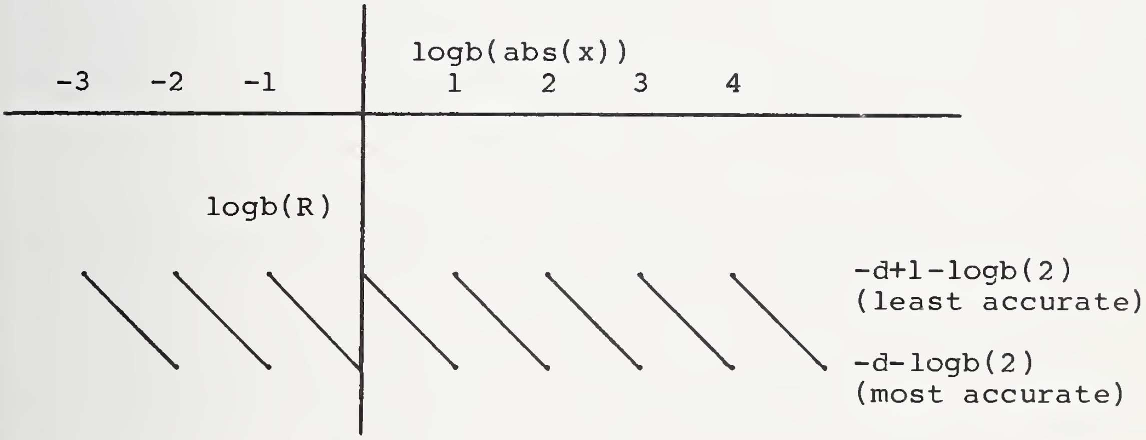
All modern digital computers represent floating point numbers as a series of d digits (with an implicit radix point) times the base, b, raised to some integral power. An implicit design goal is to keep the relative error for representation of any numeric value within some constant bound. Thus, when discussing real arithmetic and accuracy, relative, rather than absolute, error is the appropriate measure. (Why, one may ask, do machines not adopt a direct logarithmic representation, which would keep relative error exactly constant? The d,b form is preferred because there exist reasonable algorithms for both addition and multiplication of numbers in such form, whereas addition of logarithmically represented numbers is difficult.) In particular a "d,b" machine (one whose floating point numbers use d digits of base b) can always represent a value with relative error <= b**(1-d) / 2 [Ster74]. The actual relative error may be quite a bit lower, firstly, just because the value to be represented happens to be very near one of the available hardware encodings (e.g., as with relatively small integer values), and secondly because the value is near the high end of the set of values for a given exponent. Specifically, on this second point, where N is an integer, values just above b**N will be susceptible to a relative error of b**(1-d) / 2, as just mentioned, but values just below b**N can always be encoded with an error <= b**(-d) / 2. Let "x" stand for some value which is to be approximated by the best available hardware representation in a d,b machine. Then, the general formula for the bound, R, on the relative error of the approximation is (where logb is the log base b, P is the logb (abs (x)), and INT(P) is the largest integer <= P):

$$R = \frac{b ** (INT (P) - P - d + 1)}{2}$$

or, logarithmically:

$$logb (R) = INT(logb(abs(x))) - logb(abs(x)) - d + 1 - logb(2)$$

Graphically:



So as x goes through a cycle of values (those within one exponent value), the maximum relative error gets gradually lower, until we begin a new cycle, at which time, the error jumps up to the worst case again.

One other result will be relevant in the next section. Let us ask, what is necessary for one d,b machine to match the performance of another? We will characterize the target machine by d1,b1 and the simulating machine by d2,b2. We must force the least accurate representations within the simulator to be at least as good as the most accurate within the target. Using the previous expressions for relative error bound, we derive:

$$b1**(-d1) / 2 \quad >= \quad b2**(1-d2) / 2$$

$$\frac{b2 ** (d2-1)}{b1 ** d1} \quad >= \quad 1$$

(d2 - 1) - (d1 * logb2 (b1)) >= 0

d2 >= 1 + (d1 * logb2 (b1))

How many bits, for instance, do we need to maintain six decimal digits of precision, for all values? Using the formula:

d2 >= 1 + 6 * log2 (10)

d2 >= 1 + 6 * 3.32193

d2 >= 20.9316

i.e., at least 21 bits.

In the special case where b1 is an integral power of b2, or vice-versa, we can relax the original condition, because their cycles "line up" and thus the most accurate regions for the target will never coincide with the least accurate of the simulator. Such a simulator can always represent any number as accurately as the target as long as d2 >= d1 * logb2 (b1). Thus we see that, in the general case, it costs the simulator exactly one more digit than in this more favorable special case.

4. CRITERIA FOR NUMERIC ACCURACY

4.1 Criteria Based On Argument Perturbation

The set of criteria proposed herein is based on the following ideas:

1. Any conventional machine can encode, in floating point form, any numeric value to some reasonably specifiable accuracy, expressed as a maximum relative error, or in some closely related way, e.g., significant digits.

2. Every such encoding represents some exact value, which can be treated as if it were error-free (although, with respect to the source code, this is not typically so). For any of the familiar functions provided by high-level programming languages, there exists an algorithm (since these functions are, after all, computable) [Bohl75] which will return the "best" approximation of the function value (for the argument as encoded internally). That is, the encoded function value returned will be no farther from the real value than any other available encoding.

3. The effect of the error involved in encoding the argument is highly variable, as suggested in the section on mathematical instability. Accuracy criteria must reflect the sensitivity of the function and argument.

The criterion, then, broadly speaking, is that, within some determined allowance for encoding of numeric values, implementations under test must do no worse than the best possible implementation operating on the worst value(s) allowed for the argument(s). That is, we imagine the behavior of a hypothetical d,b machine (the target) with optimal algorithms for its arithmetic. The implementation under test (the simulator) must do, in some sense, at least as good as the target. We must be careful not to lean too heavily on the above image; though it can be a helpful conceptual aid, we shall see that it is susceptible of varying interpretations.

Let us confine ourselves for the moment to the problem of what we should expect of the evaluation of a function of simple numeric constants. A precise formulation is: we specify two functions, e1(x) and e2(x), which define the endpoints of a real interval containing x. We then require that the result of any computed operation or function be some value actually taken on by the operation or function within the domain [e1(x), e2(x)]. This requirement is represented by set A below. In some cases no internal representation may exist which strictly satisfies this requirement, e.g. arccos (1E-22); we then enlarge the range just enough to allow for this possibility, and this is expressed by set B.

In set notation, then, where cf(x) is the computed value of f(x), the requirement for continuous functions of a single variable is:

$$cf(x) \in A \cup B, \text{ where}$$

$$A = \{ y \mid y=f(x) \wedge e1(x) <= x <= e2(x) \}$$

$$B = \{ y \mid M = \frac{\max(A) + \min(A)}{2} \wedge e1(M) <= y <= e2(M) \}$$

The purpose of the error bound functions, e1 and e2, is to allow for the granularity inherent in digital machines. We will face some difficulties in devising a suitable bound; there are several plausible choices, each with its own advantages and drawbacks (see section 4.2, below). The criterion above is easily generalized to functions of several variables simply by requiring the computed result to be within the range generated by the multi-dimensional domain specified by the application of e1 and e2 to each of the arguments. Also, in cases where A is discontinuous (e.g. as can occur with TAN and MOD), there must be one set B for each continuous interval in A. For the common language-based functions, there will be no more than two such intervals.

More generally, for any numeric expression, we can state the criterion recursively: the computed function must evaluate to some value actually taken on by the true function within the domain formed by the allowed evaluation of the argument(s). If the argument is itself the result of a function (or operation), the criterion must be re-applied to that function. If the argument is a variable, its allowed evaluation is that of the expression last assigned to it. If the argument is a constant, the allowed evaluation is given by e1 and e2.

The motivation for this criterion follows directly from the considerations presented earlier. By allowing the implementation to return any value actually taken on by the function in a given domain, we allow for possible mathematical instability within a relevant vicinity, not just at the nominal argument. Moreover, this scheme not only allows somewhat inaccurate results where precise computation is difficult, but it also requires high accuracy where theoretically achievable.

The case for setting up an optimal implementation as a target rests on two observations. First, *the use of a criterion based on a hypothetical d,b machine does not imply that the implementation under test uses exactly d,b hardware.* Just because the target is conceived as an optimal implementation does not mean that a real implementation must be optimal to *its* last bit to pass. Rather, as we will see in detail later on, we can establish a series of target machines and then implementations under test can be easily classified according to the most stringent criterion they are capable of passing. It is *not* the primary purpose of the criterion to determine whether or not a given system is producing results which are optimal for *its* particular d,b characteristics (although there is a relationship between optimality of the simulator and of the hypothetical target, as we shall see below). Rather we look at the accuracy we could expect from an optimal implementation using, say, six digits and then ask merely whether or not the implementation under test has done at least as well *regardless of the number of digits it actually uses.* If it does as well, then we may characterize it as (at least) a six digit system.

The second point is that, *for a target,* optimal implementation is an appropriate criterion. Indeed, what else could we specify as a logical goal? There is no other obvious choice; even if there were, its adoption would imply acceptance of and encourage sub-optimal implementations - hardly the purpose of a software standard. Such a sub-optimal criterion would also introduce the anomalous possibility that an implementation using d-1,b hardware would pass the d,b test.

It might be worth stressing that this criterion is intended to apply to language-based primitives, *not* to user-defined functions, i.e., functions implemented by writing source code subroutines which can then be invoked elsewhere in the program. The latter are subject to error analysis, based on knowledge of their algorithms. The criteria and test methods of this paper are concerned only with gauging the quality of the computational tools directly embedded in the language itself; error analysis of *algorithms* may then be undertaken, based on the measured accuracy of the facilities with which such algorithms are realized.

Thus, this criterion directly applies only to high-level languages as such, not to algorithms on the one hand or hardware on the other. For instance, no distinction is made between the operations of exponentiation and addition, even though their means of implementation are likely to be quite different, because *linguistically* they are both primitive operations, directly available to users for computing a numeric result based on two operands.

4.2 Bounding The Argument Error

How should we specify the endpoints, e1(x) and e2(x), of the interval for the argument or operands? The choice of e1 and e2 will reflect the conditions we wish to impose on the set of internal numeric representations available to the implementation. There are several reasonable strategies, three of which will be presented here. Further, we will evaluate each approach with respect to the general design criteria outlined in section 1.2. We will see that the choice of formulation for the endpoint functions depends on the purposes for which one wants to employ an accuracy criterion.

4.2.1 Relative Error

If we view the whole process of floating point encoding of numeric values as an attempt to maintain a stable maximum relative error, R, then it seems natural simply to designate some tolerable value for the maximum and then require that all functions and operations be computed within this allowance. Thus, e1 and e2 are simply the min and max of x*(1-R) and x*(1+R). If, for instance, we set the relative error bound, R = 1E-6, then we would require that sin (1.23) evaluate to some value actually taken on by the sin function within [1.22999877, 1.23000123], that is .94248839 < sin(1.23) < .94248922. As we've seen in section 3, this criterion requires a machine such that b**(1-d) / 2 <= R.

This approach has much to recommend it. First of all, it abstracts from any particular hardware design (it is irrelevant whether b=2 or b=16, e.g.), as is appropriate for a specification within a language standard. Second, it is a useful measure of accuracy for many important applications which use floating point computation, e.g. engineering. Third, it is simple to use and understand; relative error is a natural and familiar concept in mathematics. No specialized knowledge of computer arithmetic is necessary. Fourth, such a criterion is compatible with contemporary hardware, in that it is practical for implementations to conform to such a standard (see previous section on granularity and relative error).

There are two problems with the approach, neither very serious, but worth mentioning. First, using relative error gives us a somewhat looser characterization of floating point computation than some of the techniques to follow. Recall that for a d,b machine, the relative error bound in some parts of the number line will be b times smaller than in others. Even for binary arithmetic (b=2), the maximum relative error fluctuates considerably. For higher b, a constant relative error bound provides an even poorer description. Of course, one may counter that this is an argument against a high value of b in the hardware, (that it does not provide a stable relative error) and not a shortcoming of the criterion. The second problem is that of selection: what do we choose as a maximum allowable relative error? There is no obvious choice or series of choices. Of course, we could always find a number motivated by prevailing hardware. If, for instance, we knew that all machines of interest carried at least 2Ø bits, an obvious (though stringent) value for R would be 2 ** (-2Ø). This strategy, however, gets us too bound up with the hardware and too far from a user-oriented approach (i.e., who would choose 2 ** (-2Ø), if he had no familiarity with internal numeric representation?). Choosing integral powers of ten seems reasonable, given the inherent arbitrariness of the problem. It is user-oriented, easy to apply, and gives us a series of criteria to which implementations may aspire. Those interested in comparing various implementations could classify candidates according to the level of accuracy they passed (1E-5, 1E-6, 1E-7, ...). A language standard could adopt one such value as representing the minimally acceptable performance for a conforming processor (e.g., 1E-5 for single precision, 1E-1Ø for double precision, if provided in the language).

4.2.2 Significant Digits

This approach is based more closely on the concept of requiring a candidate implementation to be "at least as good" as some target d,b machine. Specifically, we require that arguments be represented accurate to d significant digits of base b, i.e., an error no greater than one half unit in the d+1 significant digit. Mathematically, e1 and e2 are the min and max of x + or - this half unit, which equals (b ** (int(logb(abs(x)))-d+1)) / 2. Letting b=1Ø and d=6, sin(1.23), for instance, would have to evaluate to some value of the function in [1.229995, 1.23ØØØ5], i.e., .94248713 < sin(1.23) < .94249Ø48. As with simple relative error, this approach is (or at least can be) suitably removed from hardware considerations and reflects a simple model of accuracy with which many people are familiar.

How does the use of significant digits compare with that of relative error? All the advantages of simple relative error apply equally to a significant digit error bound. While relative error gives a somewhat inexact characterization of actual hardware functionality, the significant digits can be used to describe a criterion very closely tailored to a given machine. This can be done, however, only by specifying that d and b of the target be equal to that of the underlying hardware of the system under test. But this would usually entail setting b=2 or 16. If we do this, we lose many of the advantages just alluded to, and also incur severe machine dependence. In short, such a criterion might be appropriate for testing whether a particular system is optimal, but is highly inappropriate in a language standard, or for any machine-independent use, e.g., comparison of several systems for procurement. All high level languages express numbers in decimal. This is the natural model for numeric representations and accuracy requirements should be expressed in a way compatible with users' normal intuitions.

Assuming, then, that b=10, we do have (in contrast to relative error) a very natural discrete series of criteria to apply, simply by varying d over reasonable integer values. Thus implementations could be classified as having passed the 6 or 7 or 8 (decimal) digit test: very handy for comparative evaluation. What is the minimally acceptable number of digits? The ANSI standard for Minimal BASIC, while mandating no particular accuracy for operations and functions, does require that numeric constants be accurate to six decimal digits. Presumably, most machines are capable of meeting this requirement (21 significant bits are necessary and sufficient). Recall from section 3 that for a d,b machine to represent values accurate to N decimal digits, (where b is not an integral power of 10) we must have d >= 1 + (N * logb(10)). The specification for Ada allows declaration of the number of significant digits for floating point variables.

The major drawback of the use of significant digits is simply the variation in relative error. It seems somewhat unnatural to require an argument value like 99E-8 to fall within [98.99995E-8, 99.00005E-8] (relative error about 1/2,000,000) while allowing 101E-8 to be anywhere within [100.9995E-8, 101.0005E-8], (relative error about 1/200,000) especially given that the encoding may well be done in binary hardware. Such an approach is artificial in that it makes a gross distinction based on an accident of decimal representation.

4.2.3 Floating Point Decimal

The third possibility is to adhere exactly to the notion that an implementation under test must match the performance of a target d,b machine. For reasons discussed above, we will assume that b=10. This criterion then becomes simulation of a floating point decimal machine. If we let d=6, for instance, then we require that sin(1.23) evaluate to some value taken on in [1.23, 1.23], i.e., the exact value. As usual, we would have to apply an enlargement rule to this generated range.

Of course, floating point decimal encoding cannot exactly represent many values. How then do we specify accuracy for inexact arguments, e.g., what should we require of the evaluation of sin(1.234567)? There are two choices here. First, we could require that conforming implementations exactly duplicate the operation of the target machine, i.e. since a six digit machine would round the argument to 1.23457, the value returned must be the best representation of sin(1.23457). From a purely formal standpoint, this is somewhat attractive since we have now specified a unique meaning for floating point calculations (of course we would need rules on how to break ties, etc.). As mentioned earlier, however, such definition would represent an upper as well as lower limit on accuracy and is therefore undesirable. A second way is to use the concept of argument perturbation again. We let e1 and e2 be the min and max respectively of the argument itself and the nearest d-digit decimal number. Since the target machine changes the argument from 1.234567 to 1.23457, we will require that the returned value of the function be correct for some argument within [1.234567, 1.234570]. This reduces to .94400543 < sin(1.234567) < .94400643.

Is such a criterion a good idea? It has some strong advantages [Hull78] and equally strong disadvantages. From the point of view of <u>formal language definition,</u> (as opposed to application utility) floating point decimal provides an especially clean model of computation. Users conceive and express numbers in decimal form. Floating point decimal is just a way of making the implementation conform to users' intuitions. Furthermore, a language definition based on floating point decimal has far less need for a multiplicity of numeric data types (with all the ensuing semantic complexity), e.g., real, fixed-point decimal, integer, since their desirable logical properties are encompassed in the model. (The model is even stronger than the combination of the three types mentioned, e.g. it would require 1.1E-33 * 1.1E33 to equal exactly 1.21.) In short, the argument for floating point decimal is that operations which appear to give exact results by a plain reading of the source code, are in fact required to do so. For computations which are not exact within the length of d, a reasonable error bound can still be formulated.

The clear disadvantage is that it is quite difficult to implement with binary hardware. Except for "small" (< b**d) integers, decimal numbers are not generally exactly representable in binary or hexadecimal format. The earlier comments on instability show that even a very small error in an argument or operand can cause a large error in the result. Most binary hardware will <u>not</u> correctly calculate .123456 - .123455 = 1.00000E-6 unless it represents numbers accurate to about <u>eleven</u>

decimal digits (but notice that most hand calculators will; their users would complain otherwise).

To sharpen the issue: is it reasonable or not to require that .1 + .1 + .1 equal .3? Is it reasonable to require that the source code "X = 1.2345E-22" will really assign that value to X? From the user's point of view, it depends largely on whether he regards the numeric constant as an approximation in the first place (in which case it is irrelevant whether he gets a good decimal or good binary encoding), or whether ".1" in the source code really means .1. Any accuracy criterion will implicitly adopt one or the other assumption, neither of which is always true.

4.3 Conclusions

Weighing all the considerations above, it appears that there are two good ways to characterize accuracy requirements, each emphasizing a different purpose. When floating point arithmetic is to be "applications-oriented", i.e. thought of as a tool for solving engineering, scientific, and statistical problems, then a simple relative error bound on the arguments and operands is most appropriate. For such applications, numbers are merely points on a number line, inherently approximate (especially when the numbers represent some physical measurement, itself subject to error) and therefore the incommensurability of binary hardware with decimal numbers can be accommodated for the sake of execution efficiency.

When a programming language is oriented towards the casual, unsophisticated user, then the "hand-calculator" model is a good one, and simulation of a floating point decimal target machine is the appropriate criterion. It may also be useful for the computer scientist or mathematician who needs a strict formal model for the computational semantics of a language. For example, proofs of correctness may be easier if it is guaranteed that when executing a loop and incrementing a counter by .1, the values taken on are exactly -.2, -.1, Ø, .1, .2, ...

A criterion based on significant digits could be used in place of relative error. While open to some objections (see above), it is still a reasonable choice. Also, it is compatible with what little accuracy standardization has been done so far in Minimal BASIC and Ada.

5. TEST METHODS

This section discusses the justification for and practical aspects of automatic testing techniques recommended for use with the above criteria. Appendix B contains model test algorithms and may be used to help clarify the material of this section.

5.1 Design Goals

As design goals, we will posit that:

1. The tests should be machine independent, written in a portable version of the language whose facilities are under examination. This excludes examination of dumps, or the use of non-decimal numeric constants, such as some implementations provide for exact encoding of values.

2. The tests must be "black box" tests, i.e., there is no attempt to analyze the internal algorithm (hardware or software) by which results are computed. It is only the result itself which is to be used. This approach is typical of testing language processors: the standard prescribes only the external behavior of a processor's various facilities, and the test design follows from this.

3. As is true of most testing, we want it to be necessary, and "almost" sufficient for a processor to pass the test in order to conform, i.e., a truly conforming processor will be certain to pass, and a non-conforming processor will very probably fail.

4. The determination of the outcome of the test (pass or fail) must be as automatic as possible. It would be highly undesirable, for instance, to depend on human inspection of the results to discover whether or not the test was passed. This last constraint implies that the tests will need to compare internally generated values and then produce summary reports, indicating perhaps only which, if any, arguments caused failure.

However, we shouldn't think of internal comparison merely as a convenient tool for testing. It is a central part of the semantics of a programming language that certain comparisons will be true or false; indeed the only other way to tell whether a computation is accurate is simply to display the result on some external device - and this display is very often less important than that the behavior of the program during execution be consistent with our normal intuitions about the way comparisons work in the language. "IF 1.4 < SQRT(2)" is not just a way of <u>testing</u> accuracy, but is part of its very meaning.

5.2 Accuracy Of Numeric Constants

The comparison of internal values brings us to a central problem: if we compare the result of a function or operation to some numeric constant, how do we know that the constant itself is accurate? If the criterion specifies that .94248839 must be less than sin(1.23), is it sufficient merely to code "IF .94248839 < SIN(1.23)"? We can decide whether or not this is legitimate by considering the accuracy, not only of functions and operations, but of simple numeric constants.

We must first recognize that constants act very much like arguments of a special function. A reference to a constant must be evaluated by the processor just as surely as a square root. In both cases, only an approximation of the true result is delivered, as the outcome cf some internal process. The "function" to which constants are arguments is *special* because, within a language definition, it is the most primitive mechanism available to the user for expressing a numeric value. This is reflected in constants' role as the termination condition for the recursive statement of the accuracy criterion given earlier. That is, we can state that the accuracy of a function depends on the allowed accuracy of its arguments, which, if they are functions, depends on the accuracy of *their* arguments, and so on. But when an argument is a constant, we are forced to state explicitly what we require of its evaluation, and, as mentioned earlier, this requirement will express our model of the underlying set of values available in the implementation. Calling constants "primitive" does not imply that the actual processing of constants is simple. For example, even though "2 + 3" is, in the abstract, a syntactically and semantically more complex object than "1.2345E-22", the evaluation of the former will likely be a simpler process than that of the latter.

Contrast this with the role of variables. We can normally assume that variables are exact because they are bound to the hardware representation of a numeric value. Thus, variables are, unlike constants, primitive for *evaluation*, though not *definitionally.* We assume here that the "evaluation" of a variable normally consists merely of fetching the hardware word(s) corresponding to it. Throughout the tests, our paradigm of evaluation will be movement from a constant or expression to a variable, in the broad sense of either an assignment statement or some form of input. (We will not distinguish between the evaluation of a constant in the source code and from an external medium.) Thus, we will assume that the following code fragments are equivalent for testing purposes:

```
1.   IF .94248839 < SIN(1.23)...

2.   A = .94248839
     B = SIN(1.23)
     IF A < B ...

3.   READ A,B  (where the input device supplies
                ".94248839" and "1.23")
     IF A < SIN(B)
```

The accuracy criteria presented in the previous section applied to language-embedded functions, such as sin, log, sqr, and operations, such as +, **. We now find that we are compelled to specify some criterion for the accuracy of numeric constants in order to justify fully a specific testing procedure based on the intentionally abstract, machine independent criteria for functions and operations.

If constants are arguments to an implicit function, as argued, it seems natural to apply to it the same criteria we apply to any other function of a single variable. In this case, the mathematical function serving as the ideal for our approximation is of course f(x) = x. If we are using, say, relative error <= 1E-6, then we will require that the constant "1.23" be equal to some value actually taken on by f(x) in [1.22999877, 1.23000123], which of course reduces to simply 1.22999877 <= evaluation of "1.23" <= 1.23000123. If we apply a floating point decimal criterion (with d >= 3), then of course the evaluation must be exact (indeed, it is the exact evaluation of "most" numeric constants that constitutes much of the appeal of floating point decimal). This approach, since it corresponds exactly to that for explicit functions and operations, of course preserves all its advantages such as machine-independence and susceptibility to leveling.

Although this approach is feasible, it unfortunately sometimes precludes coding a test which is both necessary and almost sufficient for conformance. Let us say we want to require relative error <= 1E-6 for all functions, operations, and constants. We've shown earlier that this requires .94248839 < sin(1.23). We cannot, however, simply write "IF .94248839 < SIN(1.23)". The possible error in "1.23" is no problem; indeed it is just the potential variation in the argument that the criterion is based upon. But with relative error <= 1E-6, ".94248839" could legally take on any value within [.94248745, .94248933]. Now suppose there is an actual hardware representation within [.94248839, .94248933]. If both ".94248839" and "SIN(1.23)" evaluated to it (e.g., .942489) then the comparison would indicate failure, even though both evaluations were within their respective boundaries. Since we never want a false indication of failure from a test, we have to weaken the condition. We must choose a constant whose greatest legal value is .94248839, which in this case means backing off by the allowed relative error. This gives us a test which will never fail a conforming processor, but it is a weaker test, more likely to allow non-conforming processors to pass.

5.3 Sufficiency Of Test Method

Note that problems arise only when the granularity of the system under test is considerably finer than necessary. The anomaly above could not arise unless there was at least one hardware numeric representation in an interval corresponding to the maximum relative error, whereas a system could theoretically pass the test with only one representation in an interval twice as large. By contrast if a binary system with 20 bits of precision attempts to pass the criterion for R = 1E-6, the most stringent of which it is capable, the requirement for accuracy of constants will allow an error of at most one bit for some values (in the regions just below integral powers of 2) and optimal encoding for others (in the regions above) - altogether a near optimal situation - and the tests will accordingly be very strong.

Even when the system under test has a set of numeric representations finer than necessary, the tests for some functions and arguments will still be quite strong, although others are weak. The strength of the test for any function and argument has to do with the size of the error allowance for the returned value, as compared to the size of the error allowance for the constant. The error allowance for constants is always the same, R, and so the question reduces to whether the error allowance for the function is large relative to R, in which case, the extra allowance for the constant is not important (and this occurs just where the function is unstable, as described earlier), or whether it is small relative to R, in which case the error allowance for the constant predominates (this, of course, where the function is stable). Thus we get a weak test only if the level of the criterion being applied is weak relative to the system under test, and the function is quite stable at the argument in question.

A system with numeric representations grosser than that implied by the argument endpoint functions, e1 and e2, would almost surely fail. Consider a test such as "IF A < SIN(X) < B THEN PASS ELSE FAIL". For many arguments, the allowed variation in the function value will be approximately equal to that in the constants themselves. Indeed, this is the normal case, where the function is neither especially stable nor unstable. Moreover, the error allowance for constants is, by design, just large enough to contain exactly one hardware numeric representation of a minimally "fine" set of such representations. A minimally fine machine would then often have to produce the optimal result of three adjacent <u>distinct</u> representations for, respectively, A, SIN(X), and B. This is, of course, what we should expect since the criterion was designed around the concept of optimal performance on a machine of given granularity, expressed by e1 and e2. But a machine with fewer than three available distinct representations (and this is sure to arise in many of the individual tests in a sub-minimal machine) could not possibly pass, barring lucky mistakes, even if, especially if, it performs optimally relative to its own capabilities, since this would involve mapping at least two of the three values involved to the same internal representation.

In fact, the general point should be made that even though individual tests may be weak, a processor must pass all the tests at a given level for all functions and operations with all arguments and operands in order to qualify at that level. It is quite unlikely that a processor which generated non-conforming results for any significant proportion of the individual tests would still manage to pass that entire test set.

As for the floating point decimal tests, we will see below that they often require exact results, and are therefore very difficult for a non-conforming processor to pass; it is doubtful that any implementation would even attempt to qualify as a d-digit decimal implementation, unless it truly had an underlying decimal data type (whether this was based on hardware, firmware, or software) of sufficient length.

5.4 Coding Necessary Tests

Given the above constraints, how do we code tests that are as strong as possible and yet are necessary for conformance? The general idea, of course, is to calculate, based on argument perturbation, the limits within which the evaluation of some function should fall. We must code a constant whose legal range of evaluation does not overlap with that of the function in question. To illustrate, let us work through an example of the code necessary to test the accuracy of evaluation of sin(1.23), using both the relative error criterion with R=1E-6, and the floating point decimal criterion, with d=6.

5.4.1 Relative Error Test

The first step is to calculate the mathematical limits, based on allowed argument perturbation. Since the argument may vary within [1.22999877, 1.23000123], we find that:

```
sin(1.22999877) = .9424883908186... = P1
sin(1.23)       = .9424888019317... = P2
sin(1.23000123) = .9424892130434... = P3
```

We now must find one decimal representation (since this is to be written in source code) which is less than P1/(1+R) and another which is greater than P3/(1-R). Given that the test is for level R=1E-6, we should obtain a decimal representation of at least nine digits, which will make the test suitable for systems whose internal encoding carries up to nine significant digits. For systems of higher precision, the test will be weaker than necessary, but presumably such a high precision system would not normally be attempting to pass at such a comparatively low level as R=1E-6. For nine digits, then, we shall use .942487448 and .942490156, giving us the test: "IF .942487448 < SIN(1.23) < .942490156 THEN PASS ELSE FAIL".

By our accuracy criterion for constants, ".942487448" must evaluate to some hardware representation < P1 and ".942490156" > P3. We will strictly require, as the criterion implies, that there always exists at least one actual hardware representation within the legal range of a constant. Indeed, the error bound expresses just this availability of a set of representations in the target machine sufficient to represent any value to that accuracy. Mathematically, we have now arrived at three expressions, a constant, a function, and a constant, whose legal ranges of evaluation according to e1 and e2 = min and max of x*(1 + or - R), are strictly disjoint. Furthermore, the range of the function falls in between those for the constants, and the range for each of the constants must contain at least one hardware representation.

We cannot, however, strictly require "IF .942487448 < SIN(1.23) < .94249Ø156" to be true because we haven't shown that, in contrast to the ranges for constants, there must exist a hardware representation within the legal range of "SIN(1.23)". Indeed, if we know nothing more than that any value can be encoded with relative error <= 1E-6, we must quite often assume that there might not be such a representation. Let M be the midpoint of the allowed range, i.e., M = (P1+P3)/2. If P1 <= (1-R)*M or (equivalently) if P3 >= (1+R)*M, then, because M (as any number) must be representable with an error <= R, we *can* assume a distinct representation exists, and the test is necessary. Another formulation, strictly in terms of P1 and P3 is that we must have (P3-P1)/(P3+P1) >= R for the interval to be large enough. If not, then we must, somewhat artificially, enlarge the allowed interval to guarantee the existence of at least one representation in it. This enlargement corresponds to the actual loss of information that occurs when a stable function is evaluated. For example, most systems will not evaluate cos(arccos(1E-22)) as 1E-22, because of the loss of information when computing arccos. If the allowed interval is too small, then we substitute for it the encompassing interval [M*(1-R), M*(1+R)]. In fact, P1 and P3 are slightly too close to assume the existence of a hardware representation between them and so we must weaken the test as indicated. Thus, the "new" P1 and P3 are .942487859 and .942489745, and as before we must encode the limiting constants such that they do not overlap with our enlarged range.

It is noteworthy that the extrapolation of this principle to the accuracy of constants accounts for their allowed error. That is, if we think of a constant, e.g., "1:23" as equivalent to the invocation of the identity function, e.g., "IDENT(1.23)", and then apply the undersize range principle, we would reason that the value of the function must be within the range generated by the domain [1.23, 1.23]. But the resulting range, also [1.23, 1.23] of course, is too small and must be enlarged to [1.23*(1-R), 1.23*(1+R)]; but this is just the allowable variation for constants. Thus we picture the inaccuracy as stemming, not from the constant itself which is taken to be exact, but from the implicit function evaluation resulting in an undersize range. Thus, constants are treated exactly analogously to functions, and with the very same justifications.

5.4.2 Floating Point Decimal Test

We need to distinguish three cases in the floating point decimal tests: 1) inexact arguments, 2) exact arguments with inexact results, and 3) exact arguments with exact results. Although the underlying rationale of matching the performance of a target is the same for all cases, its application in the tests is slightly different.

In the general case the argument length (i.e., number of significant decimal digits in the argument) is greater than d. As mentioned earlier, we will then establish our argument domain as that between the nominal argument and the nearest d-digit decimal encoding. So, for instance, with d=6, the argument interval for sin(1.234567) is [1.234567, 1.23457]. In case the argument is exactly halfway between two available encodings, we could adopt a tie-breaking rule, or simply allow the interval to contain both adjacent encodings. As with relative error, we then obtain upper and lower limits on the function value, which in this case gives us .94400543 < sin(1.234567) < .94400643. Now, however, we have a simpler rule for writing constants. Since the error allowance for constants will simply be the difference between the constant and the nearest six digit encoding, we simply set the lower limit to the next lowest six digit constant, and the upper limit to the next highest. Thus, we write: "IF .944005 < SIN(1.234567) < .944007 THEN PASS ELSE FAIL".

As with the relative error test, we must assure that at least one distinct hardware representation exists between the encoded limits for a minimal machine. For floating point decimal, this is trivially solved by noting whether or not the encoded limits differ by at least two in the last digit. In our example, they do, and so the test is necessary and valid as it stands. Conceptually, a minimal six digit machine must evaluate sin(1.234567) to some value between .94400543 and .94400643. Since there does exist a six digit number within these limits, namely .944006, the test may be strictly applied.

If, however, the limits were, say, .94400513 and .94400589, then we could not use the resulting "IF .944005 < F(X) < .944006". When such an undersize range (i.e., one not containing a d-digit number) results, we must enlarge the range to include at least one of the two adjacent d-digit representations. One reasonable rule would be that if the original undersize range contains the midpoint of the two adjacent d-digit numbers, it is enlarged to include both; if not, then it is enlarged to include only the unambiguously nearer. Thus, for six digits, [.94400513, .94400589] is enlarged to [.944005, .944006] and is tested with "IF .944004 < F(X) < .944007", but [.94400577, .94400589] is enlarged to [.94400577, .944006] and is tested with "IF .944005 < F(X) < .944007". This rule is a consistent generalization of our rule for evaluation of constants. Note, however, that this is not the same treatment of undersize range as specified in the general criterion in section 4.1. The resolution of the problem of an undersize range is different than with the relative error criterion because we know not only the maximum allowed separation between representations, but also their actual values.

When the argument length does not exceed the number of available decimal digits, the argument is assumed to be exact, and thus the function value may be computed accurate to an arbitrary number of digits. Assuming for the moment that the correct value is not itself expressible in six digits, it will always fall between two adjacent six digit numbers. We then apply the rule of the last paragraph, treating the allowed range as a zero-width interval. Thus, the undersize range [.94400567, .94400567] is enlarged to [.94400567, .944006], and [.9440055, .9440055] to [.944005, .944006]. Again, note that this agrees with our treatment of constants.

When both the argument and result are exact within six digits, e.g., sqrt(1.44) = 1.2, 1.23456 - 1.234 = 5.6E-4, then we may simply code the exact test. This is, of course, the special strength of floating point decimal: we get exact results whenever, in decimal computation, it is possible.

If we wished to impose the criterion of exact simulation of an optimal floating point decimal machine, then all tests would be simple equality tests. For example, when testing sin(1.234567), we note that an optimal six digit machine would round the argument to 1.23457, and then find the nearest approximation to the true value for 1.23457, namely .94400643... Thus our test would be: "IF SIN(1.234567) = .944006 THEN PASS ELSE FAIL".

It should be clear that the tests never require a more accurate result than that which an actual d-digit floating point decimal machine would optimally deliver. Therefore, the tests are necessary for any acceptable simulator.

5.5 Testing Compound Expressions

Although the test method as outlined has been applied only to function evaluation where the argument is a simple numeric constant (or, equivalently, a simple variable to which such a constant has been assigned), it can be generalized to compound expressions as well. As mentioned earlier, the criterion applies recursively to any numeric expression. Of course, as the expression grows, the criterion becomes weaker, since it is based on a worst-case analysis and for this reason testing of large expressions is less important than that of single functions or operations. Nonetheless, an unambiguous bound for large expressions can be computed by successive application of the criterion. There are two rules to keep in mind when doing so: first, if at any step, a generated range is undersize, it must be enlarged at that point. This reflects the model of computation that there may be hardware-bound intermediate results, and thus information may be lost. We do not require symbolic analysis of expressions to avoid this loss. The earlier example of cos(arccos(1E-22)) illustrates this point. Second, only at the end of the worst case analysis do we compute the non-overlapping constants to be compared with the result; this is not done at each step. Note the distinction between the endpoints of the allowed interval, which are computed at each step, and the usually different source constants used in the actual comparison with the result.

Also, note that as long as we abide by the rule of non-overlapping allowed intervals, we can compare functions with other functions as well as with constants, e.g., "IF SIN(X) < SIN(Y)". The choice of X and Y must be done carefully, however. It is not sufficient, as might be thought, merely that X and Y be far enough apart so that their allowed evaluations are disjoint. The ranges for the SIN function might still overlap if, for instance, the slope in the vicinity of X and Y is low enough so as to generate undersize ranges.

5.6 Multiple Operands

Most of the examples given so far have involved functions of a single variable, for ease of discussion. The generalization to binary operations or functions of several arguments is quite natural. We simply apply the allowed argument perturbation to all the operands and allow the result to be anywhere within the range generated by this multi-dimensional domain. For instance, using the relative error criterion with R=1E-6, the expression ".3 ** .2" must evaluate to some value of the exponentiation operator, x**y, within the two-dimensional domain given by .2999997 <= x <= .3000003 and .1999998 <= y <= .2000002. The minimum is .786002739 = .2999997 ** .2000002, and the maximum is .786003432 = .3000003 ** .1999998. We then test the evaluation exactly as we would for an interval generated by a single-argument function, with the usual considerations for undersize range and calculation of source constants.

5.7 Special Cases

5.7.1 Arguments And Operands With Small Integer Values

Although the relative error criterion can be applied consistently to any numeric value, it might be reasonable to strengthen the tests by assuming that all machines can represent "small" integers (e.g., between -1000 and +1000) exactly. This does, of course, go beyond the initial premise that the only thing we know about internal representation is that there must exist at least one hardware encoding within a maximum relative error, R, of any arbitrary value.

The assumption of small integer representations is reasonable because any d,b machine can represent such integers. The assumption is useful because it allows us to require certain desirable results in tests, which we otherwise could not do. For instance, most users would want their machines to satisfy the following equalities:

1. Ø + 5 = 5

2. 5 + 7 = 12

3. 8 / 4 = 2

4. 3 ** 3 = 27

5. cos(Ø) = 1

6. log(1) = Ø

7. exp(Ø) = 1

8. sqrt(9) = 3

Without the assumption of exact integer representation, none of the preceding tests could be required directly in the code. (Of course, the above tests are perfectly valid for the floating point decimal criterion.) Given that small integers are exactly representable, then of course the original philosophy of requiring the implementation under test to match the performance of an optimal machine implies that whenever all the arguments and the result take on such integer values, we can encode a simple equality test. Another special consideration is that raising negative values to other than integral powers is generally not allowed, and so when testing, e.g., -5.12 ** 3.Ø, we would have to assume that the evaluation of "3.Ø" was exact.

5.7.2 Non-monotonicity Within Allowed Domain

The criterion does not depend on the function being monotonic, or even continuous, within the argument domain. When these conditions arise, however, the transition from criterion to test program is somewhat indirect and less susceptible to automatic test data generation. Some examples will illustrate the point.

If we test the evaluation of sin(x) near pi/2, it is important to realize that the maximum value allowed is not simply the value of the function at one of the two endpoints of the domain (assuming one is less than and the other greater than pi/2). It is, of course, one; the minimum allowed value would be that at one of the endpoints. This case is also special in that we would want to take advantage of the assumption that small integers are exactly representable in order to require: "SIN(X) <= 1" rather than the weaker check based directly on the relative error criterion.

The criterion does not always reduce to the requirement that the evaluation of the function be between two limiting values. Testing tan(x) near pi/2, in fact, requires source code in the form: "IF TAN(1.57Ø8) < A OR TAN(1.57Ø8) > B THEN PASS ELSE FAIL" since within the domain, the range of the tan function consists of all values outside of a given interval. Likewise, MOD(Ø.9, Ø.3) could legally give some value either slightly above zero or slightly below Ø.3.

Note, however, that for the five usual binary operations (+, -, *, /, **), extreme values of the operation will always be found at one of the four points representing the extreme values of the operands. This is because all the operations are monotonic along either operand axis, i.e., when holding one operand constant, varying the other gives a montonic function as long as the sign of the varied operand doesn't change. The endpoint functions for variation of the operands never allow a change of sign, so it is always sufficient to examine the value of the operation taken on at the four extreme points of the two dimensional domain.

6. LIMITATIONS

The approach outlined in this paper is intended to be of wide, but not unlimited, applicability. Let us make explicit those problems which are not addressed.

6.1 Overflow And Underflow

Any test programs based on the proposed approach must take into account the finite range of magnitudes which contemporary machines can handle. Most implementations, for instance, will not produce an accurate evaluation of exp(1ØØØ) or exp(-1ØØØ). Most language standards do not specify a minimum range of values to be supported, but the Minimal BASIC standard does mandate a range of 1E-38 through 1E+38 (these figures were chosen based on an eight bit exponent for binary hardware). Since this has not aroused any opposition among implementors, it may represent a reasonable conservative assumption for test programs.

6.2 Not Appropriate For All Functions

The approach of this paper is intended to solve the problems associated with measuring the accuracy of the traditicnal, smooth mathematical functions which typically do not take on "d,b" values (i.e., values exactly expressible in a d,b system), even for d,b arguments. For functions which do take on such values, it is appropriate to adopt a stronger criterion of exact evaluation. Thus, we should require:

1. abs(-2.34) = 2.34

2. int(3.7) = 3

3. max(1.2, 2.3) = 2.3

4. sgn(-.3) = -1

rather than the weaker range tests which would result from the direct application of the relative error criterion.

6.3 No Guidance For Which Arguments

An important design question is the choice of arguments or operands for which one will test a given function or operation (see [Lozi78]). In general, test programs should include arguments of large, medium, small, and zero magnitude, both positive and negative. The function should be tested at any point where its value is especially stable or unstable, such as its zeros, poles, maxima, and minima. The relative error magnification factor mentioned in section 2 can be helpful in identifying stable and unstable domains. For the binary operations, one must test a wide variety of <u>pairs</u> of these conditions.

7. SUMMARY

The entire proposal for numeric accuracy criteria and test methods can be shown to follow as a logical consequence of a few simple principles. An implementation under test is categorized according to its ability to match the performance of a given abstract machine. That is, we characterize an abstract machine, and, based on that characterization, construct source code statements whose outcome is guaranteed. If the candidate implementation always successfully produces those outcomes when presented with the source code, then it passes the test at the level of the abstract machine upon which the test was based.

The foregoing summarizes the justification for the test method. The interesting question is how we choose to characterize the abstract target machine. The contention is that such a characterization is provided by stipulating, first, the minimal properties of the underlying data representation, and second, that within the worst-case constraints imposed by these properties, the implementation is optimal. The various formulations for the criteria are then dependent on the given properties of the data representation.

To illustrate, let us take a series of properties, each added to all the preceding, and, with each, display comparisons which must test as true because of that incremental property:

Property	Comparison
relative error <= .00001	SQRT(4) > 1.999969 2.000041 > 2
data is represented in d,b format and d,b are integers and d>=6, b>=2 (therefore, integers are exact)	SQRT(4) = 2 2.000021 > 2
b is even	SQRT(2.25) = 1.5
b=10	SQRT(2.9929) = 1.73 2.0000051 > 2
d>=8	SQRT(.11108889) = .3333 2.000000051 > 2

Notice that as one increases the strength of the properties, the allowed evaluation of the same expression (e.g. "2") becomes more and more accurate. Test programs can then distinguish among implementations, based on which comparisons they preserve as true. The algorithms in Appendix B exploit this idea.

The properties themselves will be chosen based on the desired model of computation, e.g. relative error vs. floating-point decimal, and degree of precision required e.g. d=6 or 8 or 12.

Appendix A - Sample Wording for Software Standard

The central purpose of this paper has been to justify a practical criterion which can be incorporated into programming language standards so as to make processor conformance contingent upon a certain level of accuracy in the evaluation of numeric expressions. Alternatively, such a criterion could be adopted as a single independent standard which could then be applied across the board to all languages supporting floating point arithmetic.

Following are two drafts suitable for inclusion in the semantic specification of numeric expression evaluation. For the relative error criterion:

> The evaluation of numeric expressions, variables, and constants involving real values shall be as follows. Evaluation means development of an internal numeric representation such as can be stored in a variable. Thus variables are as accurate as the evaluation of the expressions assigned to them and are exact in the sense that no accuracy is lost simply by assignment between variables. All conforming implementations must support a set of internal representations such that any numeric value within the maximum and minimum magnitude can be encoded with a relative error no greater than some implementation defined maximum, R, which shall be at most 1E-5.
>
> Numeric constants, whether in the source code or on an external medium, shall be evaluated with a relative error no greater than R. Numeric functions and operations in programs (except for INT, SGN, ..) shall be evaluated to some value actually taken on by the true mathematical function or operation within the domain of allowed evaluation for their arguments and operands (which may themselves be compound numeric expressions).
>
> If the range of allowed values generated by this specification for any function or operation forms an interval, [L,H] such that abs((H-L)/(H+L)) < R, then, letting M be the midpoint, (L+H)/2, the range shall be enlarged to [M*(1-R), M*(1+R)] to ensure that at least one internal representation exists within the legal range of values.
>
> In addition, all conforming processors shall be capable of exact representation of all integer values between -1000 and +1000. When the arguments or operands of a function or operation all have such integer values, and when the true mathematical value of the function or operation is also such an integer, then the evaluation shall be exact.

For an accuracy criterion based on the floating point decimal model of computation:

> The evaluation of numeric expressions, variables, and constants involving real values shall be as follows. Evaluation means development of an internal numeric representation such as can be stored in a variable. Thus variables are as accurate as the evaluation of the expressions assigned to them and are exact in the sense that no accuracy is lost simply by assignment between variables. All conforming implementations must support a set of internal representations such that any numeric value within the maximum and minimum magnitude can be encoded exactly to an implementation-defined number, d, of significant decimal digits, which shall be at least six, i.e., all implementations must support floating point decimal numeric representation, with at least six significant decimal digits.
>
> Numeric constants, whether in the source code or on an external medium, shall be evaluated exactly, if they contain no more than d significant decimal digits. Otherwise, they shall be evaluated to some value between that of their true value and the nearest d decimal digit representation. If the constant is equidistant from two such representations, then it may be evaluated to any value between the two. Numeric functions and operations in programs shall be evaluated to some value actually taken on by the true mathematical function or operation within the domain of allowed evaluation for their arguments and operands (which may themselves be compound numeric expressions).
>
> If the range of allowed values generated by this specification for any function or operation forms an interval, [L,H], such that no d decimal digit representation exists within the interval, then the interval shall be enlarged as follows. If the interval contains the midpoint of the two adjacent d decimal digit representations, then it shall be enlarged in both directions to contain both those representations. If not, the interval shall be enlarged in only one direction so as to contain only the nearer of the two adjacent representations. It follows from this specification that whenever all the arguments or operands are exactly expressible as d decimal digit values, and the true value of the function or operation is likewise so expressible, then the evaluation is exact. It also follows that if all the arguments or operands are exactly expressible, but the true result is not, the computed result must be within the interval bounded by the true result and the nearest d decimal digit representation. In the special case where the true result is equidistant from two such representations, the allowed interval is bounded by those two values.

Appendix B - Sample Algorithms for Automatic Test Procedures

Any software standard is considerably strengthened by the availability of a reasonable automated procedure for conformance testing. Clearly, it has been the aim of this paper to present a criterion susceptible to such automatic testing. This appendix contains schematic algorithms for test data generation and the complementary test programs themselves. The translation of these algorithms into the popular computational languages (FORTRAN, BASIC, Pascal, Ada, PL/I) is a reasonably straightforward exercise. Note, of course, that the test data generator need not be written in the same language as that whose implementation is under test. This is especially important, in light of the fact that the generator might well need to use some enhanced facility (e.g., extended precision libraries [Bren78, Wyat76]) which provides greater accuracy than that directly available in the language primitives. For instance, a generator of test data which will be used to test for R=1E-9 must itself compute and write results accurate to at least 12 places.

The generator must produce the test data files in external character format. There are two reasons: first, it allows the generator to be in a different language as mentioned above, or even a different system; indeed in the ideal case, we would execute the generator only once and then use the resulting standard test data file as input to any system we wished to test. Second, and more importantly, the whole criterion is centered around the notion of black-box testing of source code and a standard data environment, not an analysis of internal processing. This means that we must measure an implementation's performance, not on data in some internal format, but on data in machine-independent display format. As mentioned earlier, we do not distinguish between an implementation's handling of an argument, such as "1.23", in the source code and from an external medium.

The following algorithms handle only the cases where the function is continuous and monotonic within the allowed domain. Test cases for minima, maxima, and poles must be developed manually and added to the automatically generated test data file. This is also true in cases where we wish to rely on the assumption of exact representation of integers. Additionally, it would be useful to supplement the accuracy test itself with routines to handle the special cases of discontinuous ranges and integer results. These supplementary routines merely involve different forms of the IF test of the computed value and so have not been included here. Although the generation of test data cannot be automated entirely, the accuracy tests themselves (which presumably will be run far more often than the generator) *are* completely automatic, in that, once they are formulated, they produce reliable objective results, with no need for human intervention or discretionary interpretation.

Test Data Generator for Relative Error Criterion

```
/*  set switch for one or two arguments  */
number-of-arguments = 1 or 2
/*  generate test data for relative error
    between 1E-9 and 1E-5  */
log1Ø-min-relative-error = -9
log1Ø-max-relative-error = -5
/* write header on testdata-file to indicate levels of accuracy
   being tested  */
write "testdata-file" from log1Ø-min-relative-error,
   log1Ø-max-relative-error
/*  loop to accept arguments and generate
    corresponding limits  */
read-loop:
if number-of-arguments = 2
then
   read "arguments-file" into argument1, argument2 at eof stop
else
   read "arguments-file" into argument1 at eof stop
/*  loop to generate evaluation limits for each level
    of relative error  */
for log1Ø-relative-error = log1Ø-min-relative-error
       to log1Ø-max-relative-error step 1
   /*  perturb argument(s)  */
   relative-error = 1Ø ** log1Ø-relative-error
   low-argument1  = argument1 * (1 - relative-error)
   high-argument1 = argument1 * (1 + relative-error)
   if number-of-arguments = 2
   then
      low-argument2  = argument2 * (1 - relative-error)
      high-argument2 = argument2 * (1 + relative-error)
   endif
```

```
/*  calculate endpoints of allowed interval  */
if number-of-arguments = 1
then
   /*  f is single-argument function under test  */
   nominal-value = f(argument1)
   low-value = f(low-argument1)
   high-value = f(high-argument1)
   if low-value > high-value
   then
      tmp = low-value
      low-value = high-value
      high-value = tmp
   endif
else
   /*  g is two-argument function under test
       (syntax is incidental - would normally be infix
        for the usual binary operations)  */
   nominal-value = g(argument1, argument2)
   gvalue(1) = g(low-argument1,  low-argument2)
   gvalue(2) = g(low-argument1,  high-argument2)
   gvalue(3) = g(high-argument1, low-argument2)
   gvalue(4) = g(high-argument1, high-argument2)
   low-value  = gvalue(1)
   high-value = gvalue(1)
   for i = 2 to 4 step 1
      if gvalue(i) < low-value
      then
         low-value = gvalue(i)
      else
         if gvalue(i) > high-value
         then
            high-value = gvalue(i)
         endif
      endif
   next i
endif
/*  General test for monotonicity; does not detect all cases.
    This condition should be modified.to reflect function-
    dependent criteria, e.g. argument near odd multiples of
    pi/2 for the SIN function.  Also, function-dependent
    tests for range discontinuity should be done here.  */
if low-value > nominal-value or nominal-value > high-value
then
   if number-of-arguments = 1
   then
      display
         "non-montonic range for argument = ", argument1,
         " and log of relative error = ", log1Ø-relative-error
   else
      display "non-montonic range for arguments = ",
         argument1, argument2,
         " and log of relative error = ", log1Ø-relative-error
   endif
endif
```

```
    /*  if range is undersize, enlarge to ensure existence of at
        least one internal representation in range  */
    if high-value + low-value not = Ø
    then
       if abs((high-value - low-value)/(high-value + low-value))
          < relative-error
       then
          /* condition is true only when low-value and high-value
             have the same sign */
          midpoint = (high-value + low-value) / 2
          if midpoint > Ø
          then
             low-value  = midpoint * (1 - relative-error)
             high-value = midpoint * (1 + relative-error)
          else
             low-value  = midpoint * (1 + relative-error)
             high-value = midpoint * (1 - relative-error)
          endif
       endif
    endif
    /* Low-value and high-value are set to endpoints of the
       nominal allowed range for evaluation of the function.
       Now we must develop constants whose own evaluation does not
       overlap with this range. The slight increase in relative
       error is to prevent the imaccuracy inherent in the
       decimal encoding of constants from causing such overlap. */
    increased-relative-error =
       relative-error + (1Ø ** (log1Ø-min-relative-error - 3))
    if low-value < Ø
    then
       low-limit = low-value / (1 - increased-relative-error)
    else
       low-limit = low-value / (1 + increased-relative-error)
    endif
    if high-value < Ø
    then
       high-limit = high-value / (1 + increased-relative-error)
    else
       high-limit = high-value / (1 - increased-relative-error)
    endif
    /*  write statement must generate external (decimal) data with
        at least three more significant digits than the level of
        accuracy being tested. E.g. for log1Ø-relative-error = -9,
        the data should have at least 12 significant digits.   */
    if number-of-arguments = 1
    then
       write "testdata-file" from argument1, low-limit, high-limit
    else
       write "testdata-file" from argument1, argument2,
          low-limit, high-limit
    endif
next log1Ø-relative-error
go to read-loop
```

Accuracy Test Program

The sample program below uses test data from the file generated by the previous algorithm. It counts how many test points were processed successfully at each level of accuracy and then produces a summary report of results.

```
/*  set switch for one or two arguments  */
number-cf-arguments = 1 or 2
/*  read in range of levels of accuracy to be covered  */
read "testdata-file" into log1Ø-min-relative-error,
   log1Ø-max-relative-error
/*  test-points-passed array will keep track of the number of
    test points passed at each level  */
test-points-passed = Ø
/*  loop to read in arguments and expected values  */
argument-loop:
pass-switch = "no"
for log1Ø-relative-error = log1Ø-min-relative-error to
      log1Ø-max-relative-error step 1
   if number-of-arguments = 1
   then
      read "testdata-file" into argument1, low-limit, high-limit
         at eof go to report-results
      computed-value = f(argument1)
   else
      read "testdata-file" into argument1, argument2,
         low-limit, high-limit
         at eof go to report-results
      computed-value = g(argument1, argument2)
   endif
   if pass-switch = "no"
   then
      /*  The following condition has to be modified for testing
          of discontinuous ranges, e.g. TAN function near odd
          multiples of pi/2.  */
      if low-limit < computed-value < high-limit
      then
         add 1 to test-points-passed ( - log1Ø-relative-error)
         pass-switch = "yes"
      endif
   endif
next log1Ø-relative-error
if pass-switch = "no"
then
   add 1 to test-points-passed ( - log1Ø-max-relative-error - 1)
endif
go to argument-loop
```

```
report-results:
for log1Ø-relative-error = log1Ø-min-relative-error to
      log1Ø-max-relative-error step 1
   relative-error = 1Ø ** log1Ø-relative-error
   display "number of test points passed = ",
      test-points-passed ( - log1Ø-relative-error),
      " based on relative error = ", relative-error
   if test-points-passed ( - log1Ø-relative-error) > Ø
   then
      overall-relative-error = relative-error
   endif
next log1Ø-relative-error
if test-points-passed ( - log1Ø-max-relative-error - 1) > Ø
then
   display
      "number of test points failing for all relative errors = ",
      test-points-passed ( - log1Ø-max-relative-error - 1),
      "; processor failed to qualify at any tested level."
else
   display "processor passed all tests for relative error <= ",
      overall-relative-error
endif
stop
```

References

[Ada80] "Reference Manual for the Ada Programming Language - Proposed Standard Document", United States Department of Defense, Washington DC, July 1980

[BASI78] American National Standard for Minimal BASIC, X3.60-1978, American National Standards Institute, New York, New York, January, 1978

[Bohl75] Bohlender, G., "Floating-Point Computation of Functions with Maximum Accuracy", 3rd Symposium on Computer Arithmetic, IEEE, November 1975, 14-18

[Bren78] Brent, R.P., "A Fortran Multi-Precision Arithmetic Package", ACM Transactions on Mathematical Software, Vol. 4, No. 1, March 1978, 57-70

[Coon80] Coonen, J.T., "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", Computer, Vol. 13, No. 1, Jan. 1980, 68-79

[FORT78] American National Standard Programming Language FORTRAN, X3.9-1978, American National Standards Institute, New York, New York, April, 1978

[Hull78] Hull, T.E., "Desirable Floating-Point Arithmetic and Elementary Functions for Numerical Computation", Proceedings 4th Symposium on Computer Arithmetic, IEEE, October 1978, 63-69

[Jens75] Jensen, K., Wirth, N., Pascal User Manual and Report, Springer-Verlag, New York, New York, 1975

[Lozi78] Lozier, D.W., "A Universal Set of Test Data for Computer Implementations of Elementary Mathematical Functions", NBSIR 78-1478, National Bureau of Standards, Washington, D.C., May 1978

[PL/I 76] American National Standard Programming Language PL/I, X3.53-1976, American National Standards Institute, New York, New York, August, 1976

[Ster74] Sterbenz P.H., Floating-Point Computation, Prentice-Hall, Englewood Cliffs, N.J., 1974, 98-103

[Wyat76] Wyatt, W.T., Lozier, D.W., Orser, D.J., "A Portable Extended Precision Arithmetic Package and Library with Fortran Precompiler", ACM Transactions on Mathematical Software, Vol. 2, No. 3, Sep. 1976, 209-231

[Yohe79] Yohe, J.M., "Software for Interval Arithmetic: A Reasonably Portable Package", ACM Transactions on Mathematical Software, Vol. 5, No. 1, March 1979, 50-63

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM.

BIBLIOGRAPHIC DATA SHEET *(See instructions)*

1. PUBLICATION OR REPORT NO.
NBS SP 500-77

2. Performing Organ. Report No.

3. Publication Date
June 1981

4. TITLE AND SUBTITLE

Specifications and Test Methods for Numeric Accuracy in Programming Language Standards

5. AUTHOR(S)

John V. Cugini

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covere

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as item 6.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 81-600056

□ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This publication formulates language-independent and machine-independent criteria for assessing the quality of floating-point arithmetic operations and functions. The criteria require that results be within the limits generated by perturbing the arguments or operands by a specified amount, and thus allow for the mathematical instability of some functions at certain arguments and also for the granularity of numeric representation inherent in digital machines. Automatic test methods derive naturally from the accuracy requirements. Model algorithms for testing are included.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Computer arithmetic; conformance testing; numeric accuracy; programming language standards.

13. AVAILABILITY

☒ Unlimited

□ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

□ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGE

42

15. Price

$2.75

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, D. C. 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ______________________________

Company ______________________________

Address ______________________________

City ____________ State ____________ Zip Code ____________

(Notification key N-503)

☆ U.S. GOVERNMENT PRINTING OFFICE : 1981 O—340-997 (1650)