A Simple but Realistic Model of Floating-Point Computation

W. S. BROWN
Bell Laboratories

A model of floating-point computation, intended as a basis for efficient portable mathematical software, is presented. The model involves only simple familiar concepts, expressed in a small set of environment parameters. Using these, both a program and its documentation can tailor themselves to the host computer.

First, the *model numbers*, a conventional four-parameter system of floating-point numbers, are introduced. The parameters are the base, the precision, and the minimum and maximum exponents. They must be chosen so that the result of a basic arithmetic operation on model numbers is no less accurate than the result that would be obtained by chopped arithmetic in the model system. Also, the result of a basic operation on operands between model numbers must be bounded by the results of the same operation on the neighboring model numbers.

These ideas are summarized in a few fundamental axioms, which enable the machine-independent properties of numerical programs to be stated and proved. The main conclusion is that the model supports conventional, worst case, forward or backward error analyses with only minor qualifications.

The model is simple in the sense that its axioms are more easily stated and understood than the detailed operational rules of most floating-point processors. It is realistic in the sense that real computers exhibit most of the forms of behavior (or misbehavior) that it allows, but nearly always conform to its rules.

Key Words and Phrases: computer arithmetic, environment parameters, error analysis, Euclidean norm, floating-point arithmetic, software portability
CR Categories: 4.0, 5.11, 5.19

1. INTRODUCTION

This paper presents a model of floating-point computation, intended as a basis for the development and analysis of efficient and portable programs for a wide variety of mathematical algorithms. For ease of learning and of use, the model involves only simple familiar concepts that are fundamental to numerical computation. These concepts are expressed in a small set of parameters that characterize the numerically most important attributes of a computing environment, and in a small set of axioms that describe floating-point computation in terms of these parameters.

In writing portable software, one focuses on the model and avoids thinking about the idiosyncrasies of any particular machine. Since the parameters of the

Author's address: Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

model may be used freely, a program tailors itself to each environment in which it is installed, and a high degree of efficiency is possible. In writing *about* portable software, one also focuses on the model and its parameters, and thus the documentation is portable too.

While the parameters describe a simple and familiar static number system, they are defined to reflect the dynamic behavior of the host computer as well. Taken together, the parameters and the axioms describe a conceptually simple model machine whose numerical behavior is not completely specified. This abstract machine expresses design rules that have evolved in the marketplace, capturing the agreements among designers while avoiding their differences. Among these differences are roundoff procedures, which are often mathematically inelegant, and the treatment of overflow, which is sometimes computationally disastrous. Otherwise the model machine is quite pleasant, and furthermore it supports conventional, worst case, forward or backward error analyses with only minor qualifications, justifies standard scaling strategies designed to avoid overflow or underflow, and rationalizes a programming discipline already familiar to experts in mathematical software.

In general, the effects of anomalous behavior on the part of a particular computer or compiler are accounted for by adjusting the parameters of the model in such a way as to reduce the purported precision or range. Thus the parameters reflect the precision and range that are actually provided to running programs, rather than the precision and range that could be provided with the given number representation. When the model parameters, which measure the dynamic performance of the host computer, differ from the more customary ones, which depend solely on its static number representation, we shall refer to the differences as *penalties*. Once the model parameters are available, programmers can forget about floating-point anomalies, and think about the model machine instead of the actual machines on which their programs will run. Furthermore, they can prove machine-independent properties of a numerical program, with confidence that the program will have those properties on any computer that conforms to the model. While any penalties on the parameters of a given computer obviously weaken the error bounds or restrict the domains of data for portable programs installed on that computer, neither the programmers nor the users need be aware of the reasons for the penalties.

As presented in this paper, the model is limited to a single system of floating-point numbers and floating-point arithmetic. If a computing environment offers two or more floating-point systems (for example, single precision and double precision), the model can be applied separately to each, and augmented with new rules to cover the interactions. To develop a portable library, one must also be concerned with integer arithmetic, character strings, input, and output. Of course, someone must determine the actual values of all the parameters for each site, and make them conveniently available both to programs and to their human users, but fortunately there is a program (see Section 5) to help in this task.

Section 2 of the paper introduces the parameters, and discusses the representation of floating-point numbers. Section 3 presents inequalities needed to ensure that the floating-point system defined by the parameters is meaningful, and provides a usable range. Section 4 proposes axioms for arithmetic, and discusses

some of their implications. Section 5 shows that the model is not restricted to computers of mathematically elegant design, but can encompass a wide variety of anomalies. Section 6 discusses arithmetic comparisons. Section 7 examines overflow and underflow. Section 8 shows that the model supports conventional, worst case, forward or backward error analyses with only minor qualifications. To illustrate the usefulness of the theory, Section 9 presents and analyzes an algorithm to compute the Euclidean norm of a vector. With this example in mind, Section 10 discusses the need for auxiliary number systems with extended range and precision. The final section describes the evolution of the theory, and acknowledges the author's indebtedness to numerous colleagues.

2. ENVIRONMENT PARAMETERS

The model includes a few basic parameters and a few derived parameters for each floating-point system that is supported by the host computer. If a computer supports two or more such systems (e.g., single and double precision), then each has its own parameters. The basic parameters, all integers, are

(1) the *base*, b;
(2) the *precision*, p;
(3) the *minimum exponent*, $e_{\min}$;
(4) the *maximum exponent*, $e_{\max}$.

These define a system of *model numbers* consisting of zero and all numbers of the form

$$x = fb^e \tag{1}$$

where

$$f = \pm (f_1 b^{-1} + \cdots + f_p b^{-p}), \qquad f_1 = 1, \ldots, b-1$$
$$f_2, \ldots, f_p = 0, \ldots, b-1 \tag{2}$$

and

$$e_{\min} \leq e \leq e_{\max}. \tag{3}$$

The parameters must be chosen so that these model numbers are exactly representable in the machine, and so that operations on them behave according to the following simple and desirable rules, which are restated as axioms in Sections 4 and 6.

(1) The result of a *basic arithmetic operation* (addition, subtraction, multiplication, negation, or division by a power of the base) on model numbers must be no less accurate than the result that would be obtained by chopped arithmetic in the model system. (Chopping away from zero, as in the case of negative results in a two's complement system, is permitted.)
(2) The result of a division when both operands are model numbers but the divisor is not a power of the base may be less accurate than the model-chopped result by up to one unit in the last (pth) place.
(3) The result of a comparison of model numbers must be exact.
(4) The result of an arithmetic operation (or comparison) on operands between

model numbers must be within the interval (or set) of permitted results of the same operation on the neighboring model numbers.

Note that if x is a model number, then so is $-x$, but $1/x$ need not be either in range (see (3)) or exactly representable (see (2)). On many computers $e_{\max} + e_{\min} \approx 0$, but on machines with the implicit radix point at the right, it is likely that $e_{\max} + e_{\min} \approx 2p$.

We do not assume that the computer actually uses a normalized sign-magnitude representation for floating-point numbers. In fact, the details of the hardware representation are of no concern to us. What we require is simply that the model numbers be possible values for program variables, and that arithmetic operations be at least accurate enough to satisfy the axioms.

Usually the base is chosen to be whatever the manufacturer says it is, and the remaining parameters are chosen to make the range and precision as large as possible. In some cases the resulting model numbers coincide exactly with the machine numbers. However, anomalies in the behavior of a computer may require that the parameters be penalized to reduce the purported range or precision. For example, on some computers multiplication by 1.0 causes the last b-digit of the multiplicand to be replaced by 0. On such a computer, a precision penalty of 1 is sufficient to ensure that model numbers are not affected by the anomaly. Each penalty reduces the size of the set of model numbers, thus creating machine numbers that are not model numbers and need not behave so well in computation. Any anomalies that cannot be accommodated by modest penalties are usually recognized by the manufacturer as design errors, and repaired in due course.

Since the model numbers with a given exponent e are equally spaced on an absolute scale, the relative spacing decreases as the magnitude of the fraction-part f increases. For error analysis, the maximum relative spacing

$$\epsilon = b^{1-p} \tag{4}$$

is of critical importance. Also of interest throughout this paper are the smallest positive model number

$$\sigma = b^{e_{\min}-1}, \tag{5}$$

and the largest model number

$$\lambda = b^{e_{\max}}(1 - b^{-p}). \tag{6}$$

3. PARAMETER INEQUALITIES

Our definition of the system of model numbers clearly presupposes that $b \geq 2$ and $e_{\min} \leq e_{\max}$. To ensure that 1 is a model number, we must require that $e_{\min} \leq 1 \leq e_{\max}$. To ensure that $\epsilon < 1$, we must require that $p \geq 2$. These inequalities are sufficient to guarantee a meaningful system of floating-point numbers. However, to write portable numerical programs with provable numerical properties, one needs a reasonably large and reasonably balanced range.

In the remainder of this section we present simple range-related inequalities that are weak enough to be realistic, but strong enough to be useful. If a computer fails to obey these inequalities, then the person who evaluates the parameters should issue a warning to users and send a copy to the salesman. Even though

such computers do not fully conform to the model and do not meet the requirements of many important algorithms, one can use the model to analyze the numerical behavior of programs that are intended to run on them. On the other hand, if a program needs more range or a more balanced range than these inequalities guarantee, then its author should analyze and document its special requirements and warn recipients to check before installing it.

To provide a usable range for any given precision, we require that

$$\sigma < \epsilon^2 \tag{7}$$

and

$$\lambda > \epsilon^{-2}. \tag{8}$$

In Brown's algorithm [2] for the mean of a vector, one must have $\sigma < \epsilon^4\lambda$ to avoid the possibility of overflow when summing scaled small components. Clearly (7) and (8) are just sufficient for this purpose. In Lawson's algorithm for the Euclidean norm of a vector (see Section 9, especially (80)), one must have $\lambda > \epsilon^{-3/2}$ to avoid the possibility of overflow when summing the squares of scaled small components, and $\sigma < \epsilon^2$ to avoid the possibility of lost accuracy due to underflow when squaring a scaled small or medium component after the first large component has been encountered. (Although the inequalities in (80) are not strict, a more careful derivation starting from (81) and (82) makes them so.) Thus (7) is essential for Lawson's algorithm, while (8) provides a modest safety factor. Once (7) is accepted, symmetry suggests (8). Furthermore, in practice (7) implies that (8) is true or nearly true, since $\sigma\lambda \approx 1$, if the implicit radix point is at the left, and $\sigma\lambda > 1$, otherwise.

Assuming that these examples demonstrate the usefulness of (7) and (8), we must still consider their realism. To make these inequalities fail, one would need a small range and a relatively high precision. An extreme example of small range is provided by the BESM series computers, where $b = 2$ and only 7 bits are allocated for the signed exponent from a 48-bit word. On these machines, ϵ^2 and ϵ^{-2} are out of range in single precision, and even ϵ and ϵ^{-1} are out of range in the obvious software simulation of double precision. Since the BESM computers are quite old and their small range is unlikely to recur, we are not distressed by their failure to obey (7) and (8).

To proceed further, it is convenient to restate (7) and (8) as

$$e_{\min} \leq 2 - 2p \tag{9}$$

and

$$e_{\max} \geq 2p - 1. \tag{10}$$

In the important case of an 8-bit signed exponent, an extreme example of high precision is provided by the DEC PDP-10 and Honeywell 6000 series computers, where in double precision 64 bits are allocated for the signed fraction part from a 72-bit word. These computers satisfy (9) and (10) by a very small margin. (On the obsolete PDP-10 KA processor, where double precision operations are simulated in software, (9) fails because of a penalty on $e_{\min}$, but the damage can be repaired by a compensating penalty on p.) Assuming the same word size (72 bits)

and the same split (8 bits for the signed exponent and 64 bits for the signed fraction-part), one could use an implicit normalization bit to increase the precision to 64. Likely exponent ranges (after setting one value aside for the number zero) would then be [−127, +127] or [−126, +128]. The proposed inequalities are the tightest that would permit both of these plausible systems.

Besides being large enough, the exponent range must also be reasonably balanced. While we share Reinsch's preference [21] for $\sigma\lambda \approx 1$, such a strict rule is neither essential nor realistic. Since $\sigma\epsilon^{-2} < \sigma\lambda < \epsilon^2\lambda$ by (7) and (8), it is tempting to require that $\sigma\epsilon^{-2} < (\sigma\lambda)^{-1} < \epsilon^2\lambda$. However, we shall settle for the weaker requirement that

$$\sigma\epsilon^{-1} < (\sigma\lambda)^{-1} < \epsilon\lambda, \tag{11}$$

which may be rewritten as

$$\sigma^2\lambda < \epsilon \tag{12}$$

and

$$\sigma\lambda^2 > \epsilon^{-1}. \tag{13}$$

In Lawson's algorithm for the Euclidean norm of a vector, one must have $\sigma^2\lambda < \epsilon^{1/2}$ and $\sigma\lambda^2 > 1$ to ensure that the scale factors are in range. Thus (12) provides a modest safety factor, and (13) provides a larger one. Once (12) is accepted, symmetry suggests (13). Furthermore, in practice (12) implies that (13) is true or nearly true, since $\sigma\lambda \approx 1$, if the implicit radix point is at the left, and $\sigma\lambda > 1$, otherwise.

The inequalities (12) and (13) may be restated as

$$2e_{\min} + e_{\max} \leq 3 - p \tag{14}$$

and

$$e_{\min} + 2e_{\max} \geq p + 1. \tag{15}$$

If $e_{\min} + e_{\max} = 0$, these follow from (9) and (10). However, if $e_{\min} + e_{\max} = 2p$, as might be the case on a machine with the implicit radix point at the right, then by (14), $e_{\min} \leq 3 - 3p$. It follows that $e_{\max} = 2p - e_{\min} \geq 5p - 3$, and $e_{\max} - e_{\min} \geq 8p - 6$, which is exactly twice the minimum exponent range implied by (9) and (10). Fortunately, machine designers who choose to put the implicit radix point at the right also seem to prefer a relatively large exponent range, and we know of no floating-point system with $\sigma\epsilon^{-1} < \epsilon\lambda$ that fails to satisfy (11).

4. PROPERTIES OF ARITHMETIC

In a conventional model of floating-point computation (for example, see [16]), one begins by postulating a computer with machine numbers that are exactly the model numbers given above, and with floating-point operations that are exact up to rounding or chopping. Suppose x and y are numbers in such a computer, and $*$ is a binary arithmetic operator (+, −, ×, or /). Let $\mathrm{fl}(x * y)$ denote the result of computing $x * y$ in floating-point arithmetic. Assuming that $x * y$ is in range, it follows that

$$\mathrm{fl}(x * y) = (x * y)(1 + \delta), \qquad |\delta| < \epsilon, \tag{16}$$

where ϵ is the maximum relative spacing, given by (4). This theorem is usually taken as the starting point for error analyses, and one never goes back to the original (and much stronger) postulate from which it was derived.

In this paper we take a rather different tack. Instead of assuming that the machine numbers are exactly the model numbers, we require only that they include the model numbers, and instead of assuming that all floating-point operations are as accurate as possible, we adopt axioms that are somewhat weaker and much more realistic. By defining ϵ in terms of model numbers rather than machine numbers, we are able to retain (16) for its normal uses in error analyses. However, we also use the axioms directly to justify standard scaling strategies and for other purposes.

For operations on model numbers, the axioms imply that the relative error is always less than ϵ, the maximum relative spacing between model numbers. Furthermore, if the exact result is a model number, then there is no error in the computed result. In the terminology of Dekker [9], one might say that arithmetic on the model machine is "faithful", but not necessarily "rounded".

If a machine number x of magnitude less than λ is not a model number, we view it as an arbitrary representative of the closed interval x' bounded by the neighboring model numbers. If the value of a variable at one instant is x, its value at another instant may be any other number in x'. This assumption is realistic because a number in an extended register may suffer roundoff when it is moved to another register or to a storage location, and any perturbed value that is obtained in this way may remain available for subsequent use. When an operation is performed on machine numbers of magnitude less than λ, the axioms imply that there exist *effective values* of the operands in these model intervals such that the relative difference between the computed result and the exact result determined by these effective values is less than ϵ, assuming no overflow or underflow. In an error analysis, the error assigned to an operand is sufficient to bound both the error in its computed value when it is produced and the error in its effective value when it is consumed. Of course, one must allow for some input error in the case of an initial datum that is not a model number, just as one would conventionally allow for some input error in the case of an initial datum that is not a machine number.

Unfortunately, there is almost nothing that can safely be said about an operation with an operand or exact result of magnitude greater than λ. Hence for portability, and in fact for correct performance on some computers, such operations must be rigorously avoided.

While small relative error is sufficient for worst case error analyses, more is needed to guarantee that a straightforward scaling strategy will always be successful in avoiding overflow or underflow. Appealing directly to the axioms, we will show that scaling by a power of the base may be considered to be exact, and that mathematical inequalities of the type that are customarily used in choosing scale factors and range boundaries remain valid in floating-point arithmetic.

To make the above ideas precise, we need some definitions and notation. Since error analysis is closely akin to interval analysis [19], it is convenient to formulate our axioms in terms of intervals. In particular, if the end points of a closed

interval are both model numbers, we call it a *model interval*; if they are adjacent model numbers, we call it an *atomic model interval.*

For any real number x, we say that x is λ-*bounded* if $|x| \leq \lambda$, while x is *in range* if $x = 0$ or $\sigma \leq |x| \leq \lambda$. More generally, we say that a real interval X is λ-*bounded* if x is λ-bounded for all $x \in X$, and *in range* if x is in range for all $x \in X$.

If x is a λ-bounded real number, we let x' denote the smallest model interval containing x. Thus, if x is a model number, then $x' = x$; otherwise, x' is the atomic model interval that contains x. (As a special case, if $0 < x < \sigma$, then $x' = [0, \sigma]$.) More generally, if X is a λ-bounded real interval, we let X' denote the smallest model interval containing X. It is important to note that $x \in X'$ implies $x' \subseteq X'$.

For given X, we also define an interval X^+ that is generally a little larger than X'. If neither end point of X' is zero or $\pm\lambda$, then X^+ is obtained from X' by adjoining an atomic model interval at each end. If an end point of X' is $\pm\lambda$, then it is impossible to adjoin an atomic model interval at that end, and X^+ is undefined. (We shall say that X^+ *overflows* in this case.) On the other hand, if an end point of X' is zero, we let X^+ share that end point, instead of making the extension to $\pm\sigma$.

Recall from Section 2 that the *basic arithmetic operations* are addition, subtraction, multiplication, negation, and division by $B = \pm b^k$, where k is any integer such that B is in range. The parameters of a computing environment must be chosen so that these operations conform to Axioms 1 and 2 below. If other operations (for example, division, exponentiation, elementary functions, or special functions) are implemented to this same standard, we say that they too are *strongly supported* by the host environment.

Since division is sometimes implemented as a composite of two or more suboperations, each susceptible to roundoff, it cannot realistically be considered basic. For this and perhaps other operations that are not strongly supported, we provide Axioms 1a and 2a as alternatives to Axioms 1 and 2. Any operation that conforms to Axiom 1a or Axiom 2a will be called *supported.* Any supported operation that is not strongly supported will be called *weakly supported.*

We now present the formal axioms that govern arithmetic in our model, and some theorems to demonstrate that these axioms do indeed meet the requirements discussed above. Note that all operands and results are required to be λ-bounded, since otherwise there would be almost nothing that could safely be asserted. In cases where the model numbers coincide exactly with the machine numbers, this requirement need not be mentioned, and the axioms and theorems can be simplified in other ways as well.

Axiom 1 *(For Addition, Subtraction, Multiplication, and Perhaps Other Binary Operators).* Let x and y be λ-bounded machine numbers, and let $*$ be a strongly supported binary operator. Then

$$\mathrm{fl}(x * y) \in (x' * y')', \tag{17}$$

provided that the interval $x' * y'$ is λ-bounded.

Axiom 2 *(For Negation, Division by $\pm b^k$, and Perhaps Other Unary Operators).* Let x be a λ-bounded machine number, and let $*$ be a strongly supported

unary operator. Then

$$\mathrm{fl}(*x) \in (*(x'))', \tag{18}$$

provided that the interval $*(x')$ is λ-bounded.

Axiom 1a *(For Composite Division and Perhaps Other Binary Operators).* Let x and y be λ-bounded machine numbers, and let $*$ be a supported binary operator. Then

$$\mathrm{fl}(x * y) \in (x' * y')^+, \tag{19}$$

provided that the interval $(x' * y')^+$ does not overflow.

Axiom 2a *(For Composite Reciprocation and Perhaps Other Unary Operators).* Let x be a λ-bounded machine number, and let $*$ be a supported unary operator. Then

$$\mathrm{fl}(*x) \in (*(x'))^+, \tag{20}$$

provided that the interval $(*(x'))^+$ does not overflow.

4.1 Exactness Theorems

The following theorems, which follow directly from Axioms 1 and 2, show that strongly supported operations are exact whenever the operands and results are all model numbers.

THEOREM 1. *Let x and y be model numbers, and let $*$ be a strongly supported binary operator. If $x * y$ is also a model number, then*

$$fl(x * y) = x * y. \tag{21}$$

THEOREM 2. *Let x be a model number, and let $*$ be a strongly supported unary operator. If $*x$ is also a model number, then*

$$fl(*x) = *x. \tag{22}$$

COROLLARY. *If x is a model number, then $fl(-x) = -x$. Also, if x and x^{-1} are both model numbers, and if reciprocation is strongly supported, then $fl(x^{-1}) = x^{-1}$.*

Remark. For both x and x^{-1} to be model numbers, x must be a product of (positive or negative) powers of prime factors of the base b. To see this, let $x = \xi b^i$ and $x^{-1} = \eta b^j$, where ξ and η are integers. Then $\xi\eta = b^k$ with $k = -(i + j) \geq 0$, so ξ divides b^k, and each prime factor of ξ divides b.

4.2 Small-Relative-Error Theorems

We shall now show that all strongly supported operations may be considered accurate to within ϵ, and all supported operations to within 2ϵ, whenever the operands are λ-bounded and the exact results are in range and not too close to the limits of the range. If the operands are model numbers, the theorems support this claim without any qualifications. However, if an operand x is not a model number, then the operation may effectively replace it by a different value $\hat{x} \in x'$, and the relative error of the computed result will be small if the exact result is

obtained from $\hat{x}$ rather than x. Since we view x as an arbitrary number in the interval x', we are consistent in viewing $\hat{x}$ as equivalent to x. Another way to understand the situation is to note that the axioms describe the arithmetic behavior of a computer in such a way that the exact position of a relevant quantity x within x' is insignificant; only model numbers and model intervals are fundamental.

We begin with a pair of lemmas for strongly supported operators, in which the generalizations to weakly supported operators are informally suggested. A pair of theorems then summarize the main results in terms of the *accuracy parameter*

$$\alpha = \begin{cases} 0_+, & \text{if } * \text{ is special unary (see Theorem 7 below), else} \\ 1, & \text{if } * \text{ is strongly supported, else} \\ 2, & \text{if } * \text{ is supported} \end{cases} \tag{23}$$

where 0_+ is an arbitrarily small positive number. (Note that $|\delta| < 0_+$ implies $\delta = 0$.)

LEMMA 1. *Let x and y be λ-bounded machine numbers, and let $*$ be a strongly supported binary operator. Then there exist (real)* effective values $\hat{x} \in x'$ *and* $\hat{y} \in y'$ *such that*

$$fl(x * y) \in (\hat{x} * \hat{y})', \tag{24}$$

provided that the interval $x' * y'$ is λ-bounded. (If $*$ is only weakly supported, a similar lemma holds with the right side replaced by $(\hat{x} * \hat{y})^+$.)

PROOF. By Axiom 1, $\mathrm{fl}(x * y) \in (x' * y')'$. If $\mathrm{fl}(x * y) \in x' * y'$, then by the definition of an interval operation there exist $\hat{x} \in x'$ and $\hat{y} \in y'$ such that $\mathrm{fl}(x * y) = \hat{x} * \hat{y}$. Otherwise, $\mathrm{fl}(x * y) \in u'$, where u is an endpoint of $x' * y'$, and we obtain (24) by choosing $\hat{x}$ and $\hat{y}$ so that $u = \hat{x} * \hat{y}$. □

LEMMA 2. *Let x be a λ-bounded machine number, and let $*$ be a strongly supported unary operator. Then there exists an* effective value $\hat{x} \in x'$ *such that*

$$fl(*x) \in (*\hat{x})', \tag{25}$$

provided that the interval $*(x')$ is λ-bounded. (If $*$ is only weakly supported, a similar lemma holds with the right side replaced by $(*\hat{x})^+$.)

PROOF. By Axiom 2, $\mathrm{fl}(*x) \in (*(x'))'$. If $\mathrm{fl}(*x) \in *(x')$, then by the definition of an interval operation there exists $\hat{x} \in x'$ such that $\mathrm{fl}(*x) = *\hat{x}$. Otherwise, $\mathrm{fl}(*x) \in u'$, where u is an endpoint of $*(x')$, and we obtain (25) by choosing $\hat{x}$ so that $u = *\hat{x}$. □

THEOREM 3. *Let x and y be λ-bounded machine numbers, and let $*$ be a supported binary operator with accuracy parameter α. In computing $x * y$, let $\hat{x}$ and $\hat{y}$ be the effective values of the operands, and suppose all possible computed approximations to the result (that is, all numbers in $(x' * y')'$ or $(x' * y')^+$, whichever is relevant) are in range. Then for every $\hat{z} \in fl(x * y)'$ there is a δ such that*

$$\hat{z} = (\hat{x} * \hat{y})(1 + \delta), \qquad |\delta| < \alpha\epsilon. \tag{26}$$

THEOREM 4. *Let x be a λ-bounded machine number, and let $*$ be a supported unary operator with accuracy parameter α. In computing $*x$, let $\hat{x}$ be the effective value of the operand, and suppose all possible computed approximations to the result (that is, all numbers in $(*(x'))'$ or $(*(x'))^+$, whichever is relevant) are in range. Then for every $\hat{z} \in fl(*x)'$ there is a δ such that*

$$\hat{z} = (*\hat{x})(1 + \delta), \qquad |\delta| < \alpha\epsilon. \tag{27}$$

4.3 Special Unary Operators

Suppose a unary operator is continuous and strictly monotonic from each of the intervals $(-\infty, 0)$ and $(0, \infty)$ into one of these intervals, and is either

(a) continuous at the origin, or
(b) undefined at the origin and unbounded in its neighborhood.

If such an operator maps the p-precision base-b numbers from each half of its domain onto the p-precision base-b numbers in the corresponding part of its range, and if it is strongly supported in the local computing environment, then we shall call it a *special unary operator.* Examples are negation, absolute value, scaling by a power of the base, and a curious operator R that roughly approximates the reciprocal.

Referring to (1), the operator R is defined by

$$R(fb^e) = (1 + b^{-1} - f)b^{1-e}, \qquad b^{-1} \le f < 1. \tag{28}$$

It is easy to see that $R(x) = x^{-1}$ when x is a power of b, and that $R(x)$ is linear between powers of b. Since the p-precision base-b numbers are equally spaced between powers of the base, it follows that R maps this set onto itself. Also, since $R(x)$ roughly approximates x^{-1}, it is clear that the continuity and monotonicity requirements are satisfied. Finally, if $R(x)$ is evaluated using the *fraction, exponent,* and *synthesize* functions as defined by Brown and Feldman [5], then $\mathrm{fl}(R(x))$ satisfies Axiom 2, and therefore R is a special unary operator.

For scaling, we introduce $B = \pm b^k$, where k is any integer such that B is in range. Then we can state the following theorem.

THEOREM 5. *Both Bx and x/B are special unary operators.*

PROOF. Both Bx and x/B clearly satisfy the continuity and monotonicity requirements, and both map the set of p-precision base-b numbers onto itself. It remains only to prove that both operators conform to Axiom 2. For division by B, this is a fundamental requirement of the model, since division by B is a basic arithmetic operator. For multiplication by B, Axiom 1 asserts that $\mathrm{fl}(Bx) \in (B'x')' = (Bx')'$, as required by Axiom 2 for the unary operator Bx. □

LEMMA 3. *Let x be a machine number and $*$ a special unary operator such that x and $*x$ are in range. Then*

$$*(x') = (*x)', \tag{29}$$

so we can use the notation $*x'$ without ambiguity.

PROOF. If x is a model number, then so is $*x$, and both sides of (29) reduce to $*x$. Otherwise, $x' = [u, v]$, where u and v are adjacent model numbers with $u <$

$x < v$. Assuming without loss of generality that $*$ is increasing, we have $*u < *x < *v$ by strict monotonicity. Since u and v are adjacent p-precision base-b numbers, and $*$ is onto, it follows that $*u$ and $*v$ are also adjacent. Hence both sides of (29) reduce to the interval $[*u, *v]$. □

THEOREM 6. *Let x be a λ-bounded machine number and $*$ a special unary operator, such that $*x$ is in range; then*

$$fl(*x) \in *(x'). \tag{30}$$

PROOF. By Axiom 2, $\mathrm{fl}(*x) \in (*(x'))'$. Since $*$ is special, the endpoints of $*(x')$ are model numbers, and (30) follows immediately. □

COROLLARY. *If both x and $*x$ are in range, then we also have*

$$fl(*x) \in (*x)'. \tag{31}$$

THEOREM 7. *Let x be a λ-bounded machine number and $*$ a special unary operator, such that $*x$ is in range. Then for every $\hat{z} \in fl(*x)'$ there is an* effective value $\hat{x} \in x'$ *such that*

$$\hat{z} = *\hat{x}. \tag{32}$$

PROOF. By Theorem 6, $\mathrm{fl}(*x) \in *(x')$. Since the endpoints of $*(x')$ are model numbers, we have $\mathrm{fl}(*x)' \subseteq *(x')$; hence $\hat{z} \in *(x')$. Finally, by the definition of an interval operation, there exists $\hat{x} \in x'$ such that $\hat{z} = *\hat{x}$, as was to be shown. □

Remark. This proves Theorem 4 for special unary operators. Because $\alpha = 0_+$ for these operators, we show in Section 8 that they are *effectively exact* in the sense that they do not contribute to the error bounds in Theorems 12 and 13.

5. MACHINE ANOMALIES

In this section we show that our model is not restricted to computers of mathematically reasonable design, but can also encompass a variety of anomalies. In general, the effects of anomalies in a computer are accounted for by adjusting its parameters in such a way as to reduce its purported precision or range. Since the model, for simplicity, omits many details of the behavior of real machines, these penalties may be somewhat unfair to certain computers. However, any penalties are usually small, and the total unfairness in $e_{\min}$, $e_{\max}$, or p rarely exceeds unity. This seems a small price to pay to avoid modeling the details of number representations and built-in algorithms.

Unfortunately, there is no way to adjust the parameters to reward a computer for a locally unbiased roundoff procedure. While local bias has no effect on worst case error bounds, it can cause a significant loss of achieved accuracy. However, the advantage of *rounding* (where the roundoff direction changes at the midpoint of any interval between adjacent floating-point numbers) over *chopping* (where the roundoff direction is constant throughout any such interval) is not quite worth one bit of precision. To see this, let us compare rounding in a base-2 machine of precision p with the locally unbiased chop-to-even rule

```
if(guard-bits ≠ 0) then
  guard-bits := 0
  if (low-bit = 1) then {add a unit in the last place}
```

in a base-2 machine of precision $p + 1$. Both give identical results, except when the number to be rounded or chopped is exactly midway between adjacent p-precision numbers. In this case, the first machine must round the number one way or the other, while the second machine (which has an extra bit of precision) can leave it alone. In practice this case occurs frequently, because floating-point operations on p-precision numbers frequently produce such results. Therefore, the second machine is noticeably more accurate than the first. Lest the chop-to-even rule seem completely contrived, we remind the reader that locally unbiased chopping is not a new idea. Recognizing the importance of avoiding local bias, Burks et al. [6] in 1947 proposed a variant of the chop-to-odd rule

if (*guard-bits* $\neq 0$) then {*guard-bits* := 0; *low-bit* := 1}

Except for the unnecessary loss of almost a bit of precision, this rule is attractive to both numerical analysts and hardware designers.

While a thorough study of real-world floating-point anomalies may lead one to despair, the situation can be summarized rather neatly, and with little exaggeration, by stating that any behavior permitted by the axioms of the model is actually exhibited by at least one commercially important computer. To begin a discussion of the subject, let us consider the possibility of machine numbers that are not model numbers. In general, these come in two classes; those that are *extra-precise*, and those that are *out of range*.

As an example of extra-precise numbers, there is a commercial computer that provides $9\frac{1}{2}$ hexadecimal digits of precision. If this is modeled by setting $b = 16$ and $p = 9$, then the least significant two bits of a model number must both be zero, so three-fourths of the machine numbers that are in range are extra-precise. An alternative way to model this computer is to set $b = 2$ and $p = 35$. In this case the model numbers and the machine numbers are identical wherever the leading hexadecimal digit is 1: elsewhere adjacent model numbers are separated by 1, 2, or 4 extra-precise machine numbers. The fact that a hexadecimal computer can be modeled with $b = 2$ should not be surprising. It simply reflects the familiar description of hexadecimal arithmetic as "binary arithmetic with fluctuating precision."

As a second example, consider a computer with extended working registers that are wider (more precise) than its memory registers. Since model numbers must be possible values for program variables (see Section 2), the precision p is determined by the memory registers. However, any number that is representable in a working register is a machine number, since it can arise as the value of a subexpression and then be used immediately as an operand. Most of these machine numbers are extra-precise in the sense that their fraction parts (ignoring trailing zeros) are more than p bits wide. Although it is intuitively clear that extended working registers generally increase the accuracy of computed results, we shall prove only that they (like other extra-precise numbers) rarely increase the worst case error. When common subexpressions are involved, extended working registers may increase the worst case error slightly as noted in Section 8 (see Example 2 and the discussion following it).

As a third example, consider a three-digit decimal computer with no guard digit

in its accumulator. Such a machine would probably yield

$$\text{fl}(1.00 \times .999) = \text{fl}(.100 \times .999) \times 10 = \text{fl}(.099) \times 10 = .990 \tag{33}$$

and

$$\text{fl}(1.00 - .999) = \text{fl}(.100 = .099) \times 10 = .01. \tag{34}$$

If we set $p = 3$, then both of these results contradict Axiom 1. However, if we set $p = 2$, thus designating the last digit of every machine number as a guard digit, then .999 is an extra-precise machine number, not a model number, and Axiom 1, which requires only that

$$\text{fl}(1.00 \times .999) \in .999' = [.99, 1.00] \tag{35}$$

and

$$\text{fl}(1.00 - .999) \in (1.00' - .999')' = [0, .01], \tag{36}$$

is satisfied.

On some computers such phenomena can occur despite the presence of guard digits, because it is inconvenient to use them and the compiler writer prefers to sacrifice a base-b digit of precision rather than suffer a loss of speed in object programs. Thus the parameters of the model are not necessarily determined solely by the computer, but may also depend on the compiler that is used.

As an example of out-of-range numbers, consider a computer that maintains and uses machine numbers with magnitudes too small to normalize. To guarantee full precision throughout the range of the model, the minimum exponent must be large enough to exclude these "tiny" unnormalized numbers, which are therefore out of range.

As a second example, consider a computer with no double-precision hardware, and let $b, p, e_{\min}$, and $e_{\max}$ be the single-precision environment parameters. In the usual software simulation of double precision, the low half of a number $x = fb^e$ with $e_{\min} \leq e < e_{\min} + p$ underflows, and precision is lost. Hence the minimum exponent in double precision is $e_{\min} + p$. Smaller numbers are less precise, and therefore outside of the double-precision range.

As a third example, consider a binary computer that uses normalized two's-complement numbers with p bits of precision. Now it is easy to show that a nonzero fraction part f in the machine representation satisfies $2^{-1} \leq f < 1$ or $-1 \leq f < -2^{-1}$. If we choose $e_{\min}$ to be the smallest machine exponent, then the negative model number with model fraction -2^{-1} and model exponent $e_{\min}$ cannot be represented in the machine, since its machine fraction would be -1 and its machine exponent would have to be $e_{\min} - 1$, which is out of range. To avert this disaster, we must increase $e_{\min}$ by unity, and thus decree that all but one of the machine numbers with the smallest possible machine exponent are out of range.

This collection of anomalies is only a beginning; others are mentioned elsewhere in this paper, and there are still more that could be discussed. While the use of the model eliminates the need for most programmers to study these exotic phenomena, someone must determine the values of the parameters for each host environment, make them conveniently available in relevant programming languages, and publish them in human-readable form. To aid in this task, N. L. Schryer has developed a test program [22] that performs arithmetic operations and comparisons on carefully chosen pairs of operands, and tests whether the results conform to the axioms of the model. Obviously, such a test cannot be

exhaustive, because there are far too many pairs of numbers in any useful floating-point system to test them all. Nevertheless, Schryer's program exercises the hardware quite thoroughly.

The possibility of discovering the parameter values automatically by simple numerical experiments has been discussed in the literature [18], but we feel that this approach has inherent limitations [14]. By contrast, Schryer's program goes to far greater lengths to achieve the more limited objective of testing whether a given computer and a given set of parameter values conform to the model. While we feel that a detailed analytical justification of the final values is essential, the test program can help us to acquire the necessary understanding, and can strengthen our confidence that we have not overlooked any relevant phenomena.

6. ARITHMETIC COMPARISONS

Most scientific programming languages permit conditional statements such as

if $(x < y)$ then {*action*}

where *action* is to be executed or skipped depending on whether $x < y$ is *true* or *false*. We shall refer to the condition $x < y$ as an *arithmetic comparison*, since it compares two arithmetic quantities. The possible *comparison operators* are

$$<, \quad \leq, \quad =, \quad \neq, \quad \geq, \quad >.$$

In performing an arithmetic comparison, great care is required, since any error in either operand may reverse the result. Nevertheless, the result does convey information, which can be made precise by analyzing the possible error in each operand and then using the axioms and theorems of this section.

When presented with an arithmetic comparison, the computer evaluates the two operands, x and y, and decides whether $x < y$, $x = y$, or $x > y$. The truth value of the comparison is then determined in the obvious way. In discussing the outcome, we shall say that *the computer reports* $x < y$, $x = y$, or $x > y$, and leave it to the reader to consider the resulting truth value.

On many computers a comparison is evaluated by an instruction that compares the bit patterns, and the outcome corresponds exactly to the mathematical relationship between the two machine numbers. Unfortunately, some computers lack such an instruction, and must therefore evaluate the relation by comparing $x - y$ to zero. In this case, the compiler writer has a heavy responsibility to ensure that the subtraction does not cause overflow or underflow. Overflow can be avoided by omitting the subtraction when x and y have opposite signs. Otherwise, $e_{\max}$ must be reduced by one to ensure conformance to the model. Underflow can be avoided by using unnormalized subtraction in this context. Otherwise, $e_{\min}$ must be increased by $\hat{p}$, the precision of the relevant system of machine numbers, to ensure conformance to the model. As an example, imagine a 3-digit decimal computer with $e_{\min} = -99$, and consider a comparison between the model numbers $x = .199 \times 10^{-99}$ and $y = .100 \times 10^{-99}$. Since the normalized difference, $x - y = .99 \times 10^{-100}$, would underflow, the computer might report that $x = y$, even though the two numbers differ by almost a factor of 2. However, by setting $e_{\min} = -96$, we can guarantee that the difference between two machine numbers is never too small to be normalized unless both of them are less than $\sigma = 10^{-97}$.

In applying the axiom and theorems of this section, there is another subtle pitfall to be avoided. Since an expression may yield different values for different uses as discussed in Section 4, the results of different comparisons composed from the same expressions may be inconsistent. For example, consider the program fragment

if $(x + y = u + v)$ write "yes"
if $(x + y < u + v)$ write "no"
if $(u + v < x + y)$ write "no"

and suppose that $x = u$ and $y = v$. If the compiler assigns the first operand in each comparison to a memory location and the second to an extended working register, then the output may be "no no" instead of the expected "yes", or there may be no output at all. Even a variable may yield different values for different uses without any intervening assignment, since an optimizing compiler may deliver one value from an extended register and another from a memory location.

Lest the reader think that all such anomalies can be blamed on poorly designed hardware or software, we remark that a very similar phenomenon could occur on a computer conforming to the excellent standard recently proposed by Coonen et al. [7]. If the compiler stores constants in extended precision (relative to the precision of u) as Kahan has advocated [15], then the program fragment

u := 1.0/3.0
if $(u = 1.0/3.0)$ write "yes"
else write "no"

would write "no" instead of the intuitively expected "yes".

With this background we are now prepared to present the axioms for comparisons.

Axiom 3. In comparing λ-bounded machine numbers x and y, the computer may report any result obtainable by an exact comparison of any $\hat{x} \in x'$ and any $\hat{y} \in y'$, but it may not report any other result.

Axiom 4. Consider a comparison of two variables whose values are a λ-bounded machine number x and a model number y, respectively. If the computer reports $x = y$, then after the comparison both variables have the value y.

6.1 Discussion

In the spirit of Axiom 1, we want to compare the intervals x' and y', and in the spirit of interval analysis we do this by considering all pairs $(\hat{x}, \hat{y})$ with $\hat{x} \in x'$ and $\hat{y} \in y'$. Axiom 3 is realistic because x and y may be perturbed within x' and y' between the time they are produced and the time they are compared, as noted in the preceding examples.

In Axiom 4, if the first variable has an extra-precise value before the comparison, then the extra digits must either affect the result or be discarded. Fortunately, it is usually easier for a compiler writer to obey this rule than to disobey it. In any case the rule has obvious appeal, even to compiler writers, and failure to obey it would make comparisons intolerably fuzzy.

The following theorems are immediate consequences of these axioms. Theorems 8 and 9 describe what the computer may report, while Theorems 10 and 11 describe what the user can safely conclude from its report. Theorem 11 is

especially helpful when a tight bound on the computed value of an expression is needed. For examples of its use, see the underflow analysis in Section 9.

THEOREM 8. *Consider a comparison of λ-bounded machine numbers, x and y. Let Z be the closed interval [x, y] if x ≤ y, or [y, x] otherwise. If there are at least two model numbers in Z, then the computer reports the correct result. If there is exactly one model number in Z, then the computer may report either the correct result or x = y. Finally, if there are no model numbers in Z, then the computer may report x < y, x = y, or x > y.*

THEOREM 9. *Consider a comparison of two variables whose values are a λ-bounded machine number x and a model number y, respectively. If x = y or if x′ does not include y, then the computer reports the correct result. On the other hand, if x ≠ y and y ∈ x′, then the computer may report the correct result or it may report x = y.*

THEOREM 10. *Consider a comparison of λ-bounded machine numbers, x and y. If the computer reports x < y, then either x < y or x′ = y′. Similarly, if the computer reports x > y, then either x > y or x′ = y′. Finally, if the computer reports x = y, then x′ and y′ have a nonempty intersection.*

THEOREM 11. *Consider a comparison of two variables whose values are a λ-bounded machine number x and a model number y, respectively. If the computer reports x < y, then in fact x < y, and the value of the first variable after the comparison cannot be greater than y. Similarly, if the computer reports x > y, then in fact x > y, and the value of the first variable after the comparison cannot be less than y. Finally, if the computer reports x = y, then in fact y ∈ x′, and the value of the first variable after the comparison is y.*

7. OVERFLOW AND UNDERFLOW (THE *BÊTES NOIRES* OF PORTABILITY)

To write a portable numerical program, one must design the algorithm and scale the data so as to permit a *proof* that no overflow will occur. Furthermore, if underflow is not similarly prevented, one must allow for it in the error analysis.

These rules are easy to remember, but not always easy to follow. This section shows why they are necessary, and discusses how they can be followed. Even if the alternative strategies were conveniently and portably available, we claim that other considerations, such as simplicity and efficiency, would favor the proposed strategy for many purposes.

We begin by reviewing some of the basic facts of overflow and underflow on actual computers. If a computer is asked to produce a number that is outside the range of the model but not outside the range of the machine, then the computation will continue with the result replaced by an approximate floating-point value. Such an event is called a *model underflow* or *model overflow* to distinguish it from the more easily recognized *machine underflow* or *machine overflow.* Unfortunately, the accuracy of the approximation may be less than one would otherwise expect, and in certain cases the subsequent use of the value may cause further difficulties.

On machine underflow, continued computation is always permitted with the result replaced by zero, a small floating-point number with the correct sign, an *infinitesimal* with the correct sign, or an unsigned *infinitesimal.* On machine

overflow, continued computation may be permitted with the result replaced by zero, a large floating-point number with the correct sign, an *infinity* with the correct sign, or an unsigned *infinity*.

When continued computation is permitted after machine underflow or machine overflow, there may, however, be a costly delay while a trap handler prepares for all sorts of irrelevant disasters, and discovers how little it is really expected to do. In any event, the numerical result may be wrong by many orders of magnitude, so any use of this option (even on a mathematically reasonable machine) must be highly disciplined to avoid utterly meaningless results. If continued computation is not desired, it may be possible instead to abort the operation, and on overflow the operation may be aborted regardless of the user's desires.

To clarify the impact of these diverse phenomena, we now present a few examples of the resulting hazards.

(a) If the machine range is not symmetric, as in a typical two's-complement system, then the statement

$y := -x$

may cause overflow or underflow. Since x is formally out of range in all such situations, this misfortune is not inconsistent with Axiom 2.

(b) If division is implemented via reciprocation and multiplication, then the statement

$y := x/x$

may cause overflow or underflow. Since x is formally out of range in all such situations, this misfortune is not inconsistent with Axiom 1a.

(c) If floating-point multiplication involves the production of a temporary unnormalized product, then this intermediate result may overflow even though the final normalized product is in range. Reducing $e_{\max}$ by unity will ensure conformance to the model.

(d) If x is an *infinitesimal*, then the statement

$y := 1.0 \times x$

may cause machine underflow because of an attempted normalization. Since x is not a model number, this misfortune is not inconsistent with Theorem 1.

(e) If x is an *infinitesimal*, then the statement

if $(x \neq 0)$ then $y := 1.0/x$

may cause division by zero, because the special bit pattern may appear nonzero to the comparison operator but be recognized as a zero by the division operator. Although the exception should be labeled as division by an infinitesimal rather than division by zero, the computer is essentially correct. What the programmer should have written is

if $(|\, x \,| > s)$ then $y := 1.0/x$

where s is a model number greater than λ^{-1}.

(f) If x and y are both equal to *infinity*, then the statement

if $(x = y)$ write "yes"

need not produce output, since the difference, $x - y$, may be *undefined* rather than zero. Although this phenomenon may be unfortunate, it is unfair to blame the computer. Programmers should be aware that *infinities* are inherently dangerous.

Some numerical analysts feel that a well-designed algorithm will rarely produce out-of-range numbers unless unreasonable demands are made on it. Hence they advocate that overflow and underflow should be detected by the computer, and treated as fatal errors. From the viewpoint of a designer of mathematical software, this strategy is attractive because it succeeds in avoiding most of the responsibility for overflow and underflow without risking incorrect results.

Others feel that this strategy restricts the domain of applicability of an algorithm too severely, and should be modified to permit recovery after an overflow or an underflow. Thus "normal" cases would be handled with maximum efficiency, while testing, scaling, and other costly special actions would be taken only when the most straightforward approach fails.

Most hold intermediate views, favoring a relatively strict treatment of overflow and a relatively permissive treatment of underflow. For the sake of portability, this paper advocates avoiding overflow altogether, and using underflow only with great care. Although this strategy is not unattractive, a software designer might well prefer one of the alternatives, and it is regrettable that they cannot be implemented portably (and on some computers, not at all) for the following reasons:

(a) Machine numbers outside the model range may be produced without detection, and later use of these numbers may cause a loss of precision or perhaps some more spectacular disaster.
(b) Some computers are incapable of interrupting the flow of control when an overflow or an underflow occurs.
(c) Following a trap, there may be no way for the user's program to ask for the return of control, or no way for the operating system to grant it.

8. ERROR ANALYSIS

The purpose of this section is to show that the model supports conventional worst case error analyses with the following minor qualifications:

(A) On a computer with rounded arithmetic, the model's use of ϵ as the unit of error is too conservative by a factor of 2.
(B) For computers with division as a strongly supported operation, the model's assumption that it might be only weakly supported is too conservative. If necessary, one can perform separate error analyses for the two classes of machines.
(C) Conventional error analyses often assume an infinite range, but the model requires attention to the limits of the range.
(D) If an initial datum x is not a model number, then enough error must be assigned to it to include the interval x', even though x may be exactly representable in the machine.
(E) If a given expression has a common subexpression, then the subexpression

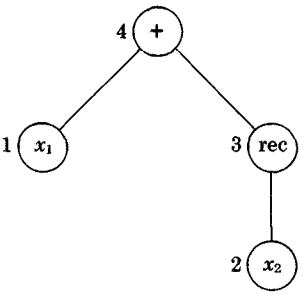

Fig. 1. Graph of $x_1 + x_2^{-1}$.

may yield different values for different uses, but the relative difference between any two values is less than ϵ.

We shall confine our attention to forward error analyses starting from (16). Since backward analyses in the style of Wilkinson [23] assume less, they are *a fortiori* supported by the model.

Let us begin by considering an expression f with no common subexpressions. Such an expression may be represented graphically by an m-node tree whose leaves are the initial data $x_1, \ldots, x_n$, and whose remaining $m - n$ nodes are supported (unary or binary) operators. For example, the expression

$$f(x_1, x_2) = x_1 + x_2^{-1} \tag{37}$$

may be represented by the 4-node tree shown in Figure 1, where rec denotes the reciprocation operator ($\mathrm{rec}(x) = x^{-1} = 1/x$).

Now let f_i be the exact mathematical value of the subtree rooted at node i. Also let $\tilde{f}_i = \mathrm{fl}(f_i)$ be the computed value of f_i, and let $\hat{f}_i$ be the effective value (in the sense of Lemma 1, Lemma 2, or Theorem 7) of f_i when it is used. By convention, we use the highest index for the root of the tree, so $f = f_m$, $\tilde{f} = \tilde{f}_m$, and $\hat{f} = \hat{f}_m$. If f is never used, then we define $\hat{f} = \tilde{f}$.

Let each initial datum x_i be approximated by a machine number $\tilde{x}_i$, and let $\hat{x}_i \in \tilde{x}_i'$ be its effective value when it is used. Suppose that the exact value of x_i is known, and that a well-designed input procedure is available to compute the approximation $\tilde{x}_i$. Then

$$\hat{x}_i = x_i(1 + \delta_i), \qquad |\delta_i| < \alpha_i\epsilon, \tag{38}$$

where

$$\alpha_i = \begin{cases} 0_+, & \text{if } x_i \text{ is a model number, else} \\ 1, & \text{if } x_i \text{ need not be converted} \\ & \quad \text{into the base-}b\text{ representation, else} \\ 2, & \end{cases} \tag{39}$$

provided that all possible values of $\tilde{x}_i$ (that is, all numbers in x_i' or x_i^+, whichever is relevant) are in range.

If node i represents an initial datum, we have seen that δ_i is the relative error introduced by approximating x_i in the computer. Otherwise, node i represents an operator, and we let δ_i be its relative error in the sense of Theorem 3 or Theorem 4, and α_i its accuracy parameter as defined in (23). Finally, we let $\phi_i = \hat{f}_i - f_i$ be the total error in $\hat{f}_i$.

We are now prepared for two formulations of the fundamental theorem of error analysis. In the main part of the proof, it is essential that all possible computed approximations to a given subexpression be in range. To state this briefly and precisely, we shall say that a subexpression h is *portably in range* (for given environment parameters and initial data) if every approximation $\tilde{h}$ consistent with the axioms of the model is in range. Similarly, we shall say that h is *portably λ-bounded* if every possible approximation $\tilde{h}$ is λ-bounded. If h is portably λ-bounded but not portably in range, due to the possibility or certainty of underflow, then h may only be used in contexts where an accurate value is not needed. To define such a context, let $\tilde{h}$ be the result of an underflow, and note that $|\tilde{h}| \leq \sigma$. Also, let $\hat{h}$ be any real number such that $0 < |\hat{h}| < \sigma$ and $\tilde{h} \in \hat{h}'$. Now we shall say that h is *negligible* in the context $g \pm h$ if $|\tilde{g}|$ is large enough that for every $\hat{z} \in \mathrm{fl}(\tilde{g} \pm \tilde{h})'$ there is a δ (with $|\delta| < \epsilon$) such that $\hat{z} = (\hat{g} \pm \hat{h})(1 + \delta)$ for some $\hat{g} \in \tilde{g}'$. The word *negligible* is appropriate because typically $\tilde{h} = 0$ (except on computers that provide denormalized numbers [7], and therefore h is literally neglected in the evaluation of $g \pm h$.

THEOREM 12. *Let $f(x_1, \ldots, x_n)$ be an expression without common subexpressions, represented by a tree as discussed above. Assume all of the subexpressions $f_1, \ldots, f_m$ are either portably in range or negligible. Then for every $\hat{f} \in \tilde{f}(\mathbf{x})'$ there is a $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)$ such that*

$$\hat{f} = f(\mathbf{x}) + \phi(\mathbf{x}, \boldsymbol{\delta}), \qquad |\delta_l| < \alpha_l \epsilon, \quad l = 1, \ldots, m, \tag{40}$$

where $\phi = \phi_m$ is recursively defined by the following rules.

(1) *If f_i is portably in range and $f_i \neq 0$, then there are three possible subcases.*

(a) *If node i is an initial datum so that $f_i = x_i$, then*

$$\phi_i = \delta_i f_i. \tag{41}$$

(b) *If node i is a binary operator so that $f_i = f_j * f_k$, then*

$$\phi_i = ((f_j + \phi_j) * (f_k + \phi_k))(1 + \delta_i) - f_i. \tag{42}$$

(c) *If node i is a unary operator so that $f_i = *f_j$, then*

$$\phi_i = (*(f_j + \phi_j))(1 + \delta_i) - f_i. \tag{43}$$

(2) *If f_i is portably in range and $f_i = 0$, then $\phi_i = 0$, since the neighborhood of zero is out of range.*

(3) *If f_i is negligible, then ϕ_i can be obtained from* (42) *or* (43) *with $\delta_i = 0$.*

PROOF. First, we shall assume that all of the subexpressions $f_1, \ldots, f_m$ are portably in range, and we shall prove the theorem for f_i, using induction on the height of the corresponding subtree. There are three cases:

(a) If node i is an initial datum so that $f_i = x_i$, then the height is zero, and (41) follows from (38).

(b) If node i is a binary operator so that $f_i = f_j * f_k$, then the computation to be performed is $\tilde{f}_i = \mathrm{fl}(\tilde{f}_j * \tilde{f}_k)$. Hence, by Theorem 3 there exist effective values $\hat{f}_j \in \tilde{f}_j'$ and $\hat{f}_k \in \tilde{f}_k'$ such that

$$\hat{f}_i = (\hat{f}_j * \hat{f}_k)(1 + \delta_i), \qquad |\delta_i| < \alpha_i \epsilon. \tag{44}$$

Now, by induction, $\hat{f}_j = f_j + \phi_j$ and $\hat{f}_k = f_k + \phi_k$, so (42) follows immediately.

(c) If node i is a unary operator so that $f_i = *f_j$, then Theorem 4 implies (43) by similar reasoning.

On the other hand, suppose some subexpression f_k underflows, but is negligible in the context $f_i = f_j \pm f_k$ where (40) holds for f_j and the operands of f_k. Let $\tilde{f}_k$ be the result of the underflow, and let $\hat{f}_k$ be the result of reevaluating the underflowed operation in exact real arithmetic with its operands replaced by their effective values. There are two subcases:

(i) If node k is a binary operator so that $f_k = f_p * f_q$, then $\hat{f}_k = \hat{f}_p * \hat{f}_q$ by definition, and $\tilde{f}_k \in \hat{f}_k'$ by Lemma 1.

(ii) If node k is a unary operator so that $f_k = *f_p$, then $\hat{f}_k = *\hat{f}_p$ by definition, and $\tilde{f}_k \in \hat{f}_k'$ by Lemma 2.

In each subcase it is easy to show that (40) holds for f_k with $\delta_k = 0$. Also, by the definition of negligibility, for every $\hat{f}_i \in \mathrm{fl}(\tilde{f}_j \pm \tilde{f}_k)'$ there is a δ_i such that

$$\hat{f}_i = (\hat{f}_j \pm \hat{f}_k)(1 + \delta_i), \qquad |\delta_i| < \epsilon, \tag{45}$$

for some $\hat{f}_j \in \hat{f}_j'$. Since $\alpha_i = 1$ for addition and subtraction, this formula can be used at node i in place of (44). □

8.1. Generalizations

While Theorem 12 views $\tilde{f}$ as an approximation to $f(\mathbf{x})$, it is sometimes advantageous to view it instead as an approximation to $f(\hat{\mathbf{x}})$. If $x_1, \ldots, x_n$ are model numbers, the two formulations are, of course, identical. In the general case, however, the effective-value formulation permits us to relax the constraints on $f_1, \ldots, f_m$, and to set $\delta_1, \ldots, \delta_n$ equal to zero. To permit a concise statement of the relaxed constraints, we shall say that a subexpression h is *effectively exact* if for every $\hat{h} \in \tilde{h}(\mathbf{x})'$ there is an $\hat{\mathbf{x}} \in \mathbf{x}'$ such that $\hat{h} = h(\hat{\mathbf{x}})$.

Another convenient generalization is to express the theorem in terms of the relative errors $\eta_i = \phi_i / f_i$.

To save space, we shall omit making these two generalizations separately, and proceed directly to the fundamental theorem that is obtained by making them together. If f is used as an n-ary operator in a larger expression, this theorem provides the appropriate generalization of Theorem 3 for the error analysis.

THEOREM 13. *Let $f(x_1, \ldots, x_n)$ be an expression without common subexpressions, represented by a tree as discussed above. Assume that all of the subexpressions $f_1, \ldots, f_m$ are portably λ-bounded, and that each of the subexpressions $f_{n+1}, \ldots, f_m$ is portably in range, effectively exact, or negligible. Then for every $\hat{f} \in \tilde{f}(\mathbf{x})'$ there exist $\hat{\mathbf{x}} \in \mathbf{x}'$ and $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)$ such that*

$$\hat{f} = f(\hat{\mathbf{x}})(1 + \eta(\hat{\mathbf{x}}\ \boldsymbol{\delta})), \qquad \delta_l = 0, \quad l = 1, \ldots, n \tag{46}$$
$$|\delta_l| < \alpha_l \epsilon, \quad l = n+1, \ldots, m,$$

where $\eta = \eta_m$ is recursively defined by the following rules.

(1) *If f_i is portably in range and $f_i \neq 0$, then there are three possible subcases.*

(a) *If node i is an initial datum so that $f_i = x_i$, then*

$$\eta_i = \delta_i. \tag{47}$$

(*b*) *If node i is a binary operator so that* $f_i = f_j * f_k$, *then*

$$\eta_i = f_i^{-1}((f_j(1 + \eta_j)) * (f_k(1 + \eta_k)))(1 + \delta_i) - 1. \tag{48}$$

(*c*) *If node i is a unary operator so that* $f_i = *f_j$, *then*

$$\eta_i = f_i^{-1}(*(f_j(1 + \eta_j)))(1 + \delta_i) - 1. \tag{49}$$

(2) *If* f_i *is portably in range and* $f_i = 0$, *then* $\phi_i = 0$, *and so we can set* $\eta_i = 0$.
(3) *If* f_i *is effectively exact, then* $\phi_i = 0$, *and so we can set* $\eta_i = 0$.
(4) *If* f_i *is negligible, then the relative error* η_i *may be large or even infinite, but only the absolute error* $\phi_i = \eta_i f_i$ *is needed, and it can be obtained from* (48) *or* (49) *after formally multiplying both sides by* f_i *and setting* $\delta_i = 0$.

PROOF. If all of the subexpressions $f_{n+1}, \ldots, f_m$ are portably in range or negligible, then the proof is the same as the proof of Theorem 12 with the substitutions

$$\begin{gathered} \mathbf{x} = \hat{\mathbf{x}} \\ \delta_1, \ldots, \delta_n = 0 \\ \phi_i = \eta_i f_i. \end{gathered} \tag{50}$$

Otherwise, for each f_i that is effectively exact, the relevant components of $\mathbf{x}$ can be chosen so that (46) holds for f_i with $\eta_i = 0$. □

8.2 First-Order Approximations

Since explicit error formulas derived from any of the preceding recurrence relations are usually quite complicated, expansions to first order in ϵ are of considerable interest. If node i is a binary operator, we rewrite (42) as

$$\phi_i = \delta_i f_i + \hat{f}_j * \hat{f}_k - f_i. \tag{51}$$

Noting that the second term on the right is $f_i(\hat{f}_j, \hat{f}_k)$, we can expand it in a Taylor series, and thus obtain

$$\phi_i = \delta_i f_i + \frac{\partial f_i}{\partial f_j}\phi_j + \frac{\partial f_i}{\partial f_k}\phi_k. \tag{52}$$

Similarly, if node i is a unary operator, (43) yields

$$\phi_i = \delta_i f_i + \frac{\partial f_i}{\partial f_j}\phi_j. \tag{53}$$

Finally, be repeated application of (41), (52), and (53), we obtain the first-order approximation

$$\phi = \sum_{i=1}^{n} \frac{\partial f}{\partial f_i}(\delta_i f_i) \tag{54}$$

to the absolute error in $\tilde{f}$. Dividing through by f, we obtain the equivalent approximation

$$\eta = \sum_{i=1}^{n} \left(\frac{f_i}{f}\frac{\partial f}{\partial f_i}\right)\delta_i \tag{55}$$

Table I. Recurrence Formulas for ϕ_i

Operation	ϕ_i (exact)	ϕ_i (first order)
$f_i = f_j \pm f_k$	$\delta_i f_i + (1 + \delta_i)(\phi_j \pm \phi_k)$	$\delta_i f_i + \phi_j \pm \phi_k$
$f_i = f_j \times f_k$	$(1 + \delta_i)(f_j + \phi_j)(f_k + \phi_k) - f_i$	$\delta_i f_i + \phi_j f_k + \phi_k f_j$
$f_i = f_j / f_k$	$(1 + \delta_i)(f_j + \phi_j)/(f_k + \phi_k) - f_i$	$\delta_i f_i + (f_k \phi_j - f_j \phi_k)/f_k^2$
$f_i = \pm f_j b^k$	$\pm \phi_j b^k$	$\pm \phi_j b^k$

Table II. Recurrence formulas for η_i

Operation	η_i (exact)	η_i (first order)
$f_i = f_j \pm f_k$	$(1 + \delta_i)[1 + f_i^{-1}(f_j \eta_j \pm f_k \eta_k)] - 1$	$\delta_i + f_i^{-1}(f_j \eta_j \pm f_k \eta_k)$
$f_i = f_j \times f_k$	$(1 + \delta_i)(1 + \eta_j)(1 + \eta_k) - 1$	$\delta_i + \eta_j + \eta_k$
$f_i = f_j / f_k$	$(1 + \delta_i)(1 + \eta_j)/(1 + \eta_k) - 1$	$\delta_i + \eta_j - \eta_k$
$f_i = \pm f_j b^k$	η_j	η_j

to the relative error. Note that the coefficient of δ_i in this formula is simply the relative derivative of f with respect to f_i. If all of the operators are strongly supported, then $|\delta_i| < \epsilon$ for $i = 1, \ldots, n$, and it follows that

$$|\eta| < \epsilon \sum_{i=1}^{n} \left| \frac{f_i}{f} \frac{\partial f}{\partial f_i} \right|. \tag{56}$$

8.3 Universally Supported Operators

The special cases of the recurrence formulas for ϕ_i and η_i that correspond to the elementary arithmetic operations and to scaling by a power of the base are given in Tables I and II, where the first-order approximations express the familiar rules of everyday error analysis.

Example 1. To illustrate the application of these formulas, let us return to (37) and Figure 1. For convenience, we shall assume that reciprocation is strongly supported. Clearly

$$\begin{aligned} f_1 &= x_1 \\ f_2 &= x_2 \\ f_3 &= x_2^{-1} \\ f \equiv f_4 &= x_1 + x_2^{-1}, \end{aligned} \tag{57}$$

and by (55) the first-order relative error in f is

$$\eta = x_1 f^{-1} \delta_1 - x_2^{-1} f^{-1} \delta_2 + x_2^{-1} f^{-1} \delta_3 + \delta_4. \tag{58}$$

Clearly $|\eta|$ is large if and only if $|f|$ is small compared to $|x_1|$ or $|x_2^{-1}|$, which is true if and only if the two terms of f are nearly equal in magnitude and opposite in sign. In this case we also see that the "catastrophic cancellation" at node 4 amplifies the errors in its operands, but does not add any important new term. Furthermore, since $\max |\delta_i| < \epsilon$, we have

$$|\eta| < (|x_1 f^{-1}| + 2|x_2^{-1} f^{-1}| + 1)\epsilon, \tag{59}$$

which is our final result.

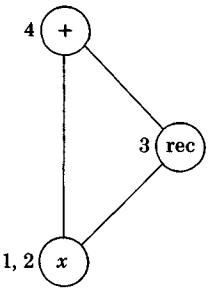

Fig. 2. Graph of $x + x^{-1}$.

Example 2. If a rational expression has common subexpressions, it is not represented by a tree, but by a more general directed acyclic graph (DAG). However, the preceding discussion is essentially unchanged except for Qualification (E). Instead of reformulating Theorems 12 and 13 and their proofs in full generality, we shall illustrate the required changes by analyzing the expression

$$f(x) = x + x^{-1}, \tag{60}$$

which may be obtained from (37) by setting $x_1 = x_2 = x$, and may be represented by the DAG shown in Figure 2.

In numbering the nodes of this DAG, we have assigned two node numbers to the leaf x, because it is a common subexpression with two uses, and Qualification (E) requires us to assign a different δ for each use. We now observe that

$$\begin{aligned} f_1 &= f_2 = x \\ f_3 &= x^{-1} \\ f &\equiv f_4 = x + x^{-1}. \end{aligned} \tag{61}$$

For machines that deliver the same value of x for each use, we can set $\delta_1 = \delta_2 = \delta$, and apply (55) directly to obtain

$$\eta = (x - x^{-1})f^{-1}\delta + x^{-1}f^{-1}\delta_3 + \delta_4, \tag{62}$$

from which it follows that

$$|\eta| < \epsilon h(x) \tag{63}$$

with

$$\begin{aligned} h(x) &= h_1(x) \\ &= |xf^{-1} - x^{-1}f^{-1}| + x^{-1}f^{-1} + 1, \\ &= \begin{cases} xf^{-1} + 1, & \text{if } |x| \geq 1, \\ (2x^{-1} - x)f^{-1} + 1, & \text{if } |x| \leq 1, \end{cases} \\ &= \begin{cases} 2 - x^{-1}f^{-1}, & \text{if } |x| \geq 1, \\ 3 - 3xf^{-1}, & \text{if } |x| \leq 1. \end{cases} \end{aligned} \tag{64}$$

In the more general case when Qualification (E) must be taken into account, we replace Figure 2 by Figure 1 with x_1 and x_2 set equal to x. In this case, (55) yields

$$\eta = xf^{-1}\delta_1 - x^{-1}f^{-1}\delta_2 + x^{-1}f^{-1}\delta_3 + \delta_4. \tag{65}$$

However, instead of setting $\delta_1 = \delta_2$, we now impose the weaker condition that $|\delta_1 - \delta_2| < \epsilon$, and thus obtain (63) with

$$\begin{aligned} h(x) &= h_2(x) \\ &= \max(xf^{-1}, x^{-1}f^{-1}) + x^{-1}f^{-1} + 1, \\ &= \begin{cases} (x + x^{-1})f^{-1} + 1, & \text{if } |x| \geq 1, \\ 2x^{-1}f^{-1} + 1, & \text{if } |x| \leq 1, \end{cases} \\ &= \begin{cases} 2, & \text{if } |x| \geq 1, \\ 3 - 2xf^{-1}, & \text{if } |x| \leq 1. \end{cases} \end{aligned} \tag{66}$$

An interesting fact about this example is that if $\tilde{x}$ is too large, then $\tilde{x}^{-1}$ is too small and vice versa, so the errors tend to compensate. However, when Qualification (E) is taken into account, the delivered values of $\tilde{x}$ may be different for its two uses, and this compensation may be partially defeated. Nevertheless, the constraint $|\delta_1 - \delta_2| < \epsilon$ has permitted us to retain half of the advantage. Without this constraint, (65) would have yielded

$$h(x) = h_3(x) = 2 + x^{-1}f^{-1} = 3 - xf^{-1}, \tag{67}$$

and it is easy to see that h_2 is midway between this and h_1.

We remark that some expressions, such as $f(x) = x - x^{-1}$, do not exhibit the phenomenon of compensation, and in such cases Qualification (E) has no effect on the analysis.

9. AN ALGORITHM FOR THE EUCLIDEAN NORM OF A VECTOR

To explore the issues involved in using the model, we present an accurate, efficient, and portable algorithm to compute the Euclidean norm of the components of a vector $\mathbf{x} = (x_1, \ldots, x_n)$. By appropriate scaling, the algorithm avoids all overflow, except possibly at the last step where the procedure itself may detect and signal it. However, the algorithm tolerates underflow if the resulting loss of accuracy is negligible. Although the algorithm is essentially due to Lawson [17], his analysis is considerably less detailed than the one presented here. In pleasant contrast to this analysis, the algorithm itself is quite simple. A similar algorithm due to Blue [1] avoids underflow as well as overflow, but is less simple and less efficient.

The algorithm is accurate in the sense that the simple (and achievable) error bound is not affected by scaling or by the possibility of underflow.

The algorithm is efficient in the sense that it makes only one pass over the vector, and uses only a single accumulator. The overhead due to scaling consists of at most $n + 1$ comparisons and $n + 1$ multiplications.

Finally, the algorithm is portable in the sense that it depends on the host computer only through the environment parameters and arithmetic axioms of the model. Thus a program implementing the algorithm can be used without change on any computer that conforms to the model, provided only that the parameters receive correct values by some mechanism such as a compiler, a preprocessor, or a set of library functions.

THEOREM 14. *Let* $\mathbf{x} = (x_1, \ldots, x_n)$ *be a vector of* λ*-bounded machine numbers, let* $f(\mathbf{x})$ *be the Euclidean norm of* $\mathbf{x}$, *and let* $\tilde{f}$ *be the approximation*

computed by the following algorithm. Also, let $\hat{\mathbf{x}}$ *be the effective value of* $\mathbf{x}$ *as defined in Section 6, and assume that the square-root function is a supported unary operator. Then for every* $\hat{f} \in \tilde{f}'$ *there is a real number* η *such that*

$$\hat{f} = f(\hat{\mathbf{x}})(1 + \eta), \tag{68}$$

with

$$|\eta| < (n + 2)\epsilon/2 + O(\epsilon^2). \tag{69}$$

9.1 The Algorithm

```
procedure norm(x, n)

check:
  assert (0 ≤ n < N)

small:
  sum := 0
  i := 1
  while (i ≤ n)
    if (|x_i| < t) then sum := sum + (Sx_i)^2
    else go to medium
    i := i + 1
  return (√sum/S)

medium:
  sum := (sum/S)/S
  while (i ≤ n)
    if (|x_i| < T) then sum := sum + x_i^2
    else go to large
    i := i + 1
  return (√sum)

large:
  sum := (sum × s) × s
  while (i ≤ n)
    sum := sum + (sx_i)^2
    i := i + 1

return:
  if (√sum < λs) then return (√sum/s)
  else {signal overflow; return (λ)}

end
```

PROOF OF THEOREM 14. This is an application of Theorem 13. Because of branching, the evaluated expression is data dependent. However, all variations are mathematically equivalent (any scaling is compensated by appropriate unscaling), and we will show that Theorem 13 applies in all cases.

First we present an approximate analysis to motivate the defintions of s, S, t, T, and N, and to guide the formal proof. Then we show that all subexpressions are portably λ-bounded, and that all underflowing subexpressions are either effectively exact or negligible. Finally, we derive (69), which is a first-order bound on the relative error. □

9.2 Approximate Analysis

The approximate analysis given here is intended solely for motivation and guidance, and is not part of the formal proof.

First, to guarantee that at least half of the digits in the computed norm are significant, we require that $N\epsilon \leq \epsilon^{1/2}$ (see the error analysis below), or

$$N \leq \epsilon^{-1/2}. \tag{70}$$

Next, to avoid unnecessary scaling, we want the medium range to be as large as possible. To avoid overflow in forming the sum of the squares of the medium components, we need $NT^2 \leq \lambda$, so we set

$$T = (\lambda/N)^{1/2}. \tag{71}$$

On the other hand, once a medium component has been encountered, subsequent small components are not scaled away from zero, and their squares may underflow. To ensure that any such terms are negligible, we need $t^2 \geq \sigma/\epsilon$. Hence we set

$$t = (\sigma/\epsilon)^{1/2}. \tag{72}$$

Turning to the small range, S must be large enough to avoid underflow when a scaled nonnegligible component is squared. This implies that $(S\sigma)^2 \geq \sigma$, or $S \geq \sigma^{-1/2}$. Hence we set

$$S = \sigma^{-1/2}. \tag{73}$$

On the other hand, the sum of the squares of scaled small components must not overflow. This implies that $N(St)^2 \leq \lambda$, or

$$N \leq \epsilon\lambda. \tag{74}$$

In the large range, s must be small enough to avoid overflow in forming the sum of the squares of the large components. This implies that $N(s\lambda)^2 \leq \lambda$, or $s \leq (\lambda N)^{-1/2}$. Hence we set

$$s = (\lambda N)^{-1/2}. \tag{75}$$

Once a large component has been encountered, subsequent medium components are scaled toward zero, and their squares may underflow. To ensure that any such terms are negligible, we need $(sT)^2 \geq \sigma/\epsilon$, or

$$N \leq (\sigma/\epsilon)^{-1/2}. \tag{76}$$

Finally, to ensure that S is in range, we need $\sigma^{-1/2} \leq \lambda$ or

$$\sigma\lambda^2 \geq 1, \tag{77}$$

and to ensure that s is in range, we need $\sigma \leq (\lambda N)^{-1/2}$ or

$$N \leq (\sigma^2\lambda)^{-1}. \tag{78}$$

In choosing N, we must satisfy (70), (74), (76), and (78). To satisfy (70), we set

$$N = \epsilon^{-1/2}, \tag{79}$$

and the remaining inequalities reduce to

$$\begin{aligned} \lambda &\geq \epsilon^{-3/2} \\ \sigma &\leq \epsilon^2 \\ \sigma^2\lambda &\leq \epsilon^{1/2}. \end{aligned} \tag{80}$$

As noted in Section 3, (77) and (80) are implied by (7), (8), (12), and (13).

9.3 Definitions

To complete the specification of the algorithm, we now define s, S, t, T, and N by rounding the approximate values to powers of the base. To compensate for the possibility of adverse effects from the rounding of s, S, t, and T, we define N to be strictly less than the approximate value suggested by (79). Thus we obtain

$$\begin{aligned} s &= b^{e_s} \\ S &= b^{e_S} \\ t &= b^{e_t} \\ T &= b^{e_T} \\ N &= b^{e_N}, \end{aligned} \tag{81}$$

with

$$\begin{aligned} e_s &= \lfloor -\tfrac{1}{2}(e_{\max} + e_N) \rfloor \\ e_S &= \lceil \tfrac{1}{2}(1 - e_{\min}) \rceil \\ e_t &= \lceil \tfrac{1}{2}(e_{\min} + p - 2) \rceil \\ e_T &= \lfloor \tfrac{1}{2}(e_{\max} - e_N) \rfloor \\ e_N &= \lfloor \tfrac{1}{2}(p - 2) \rfloor. \end{aligned} \tag{82}$$

Note that the algorithm is useful ($N > 1$)only when $p \geq 4$.

9.4 Preliminary Results

To prove that all evaluated subexpressions are acceptable, we need the following inequalities and lemmas. First by (82),

$$2e_s \leq -(e_{\max} + e_N) \tag{83}$$

$$2e_S \geq 1 - e_{\min} \tag{84}$$

$$2e_t \geq e_{\min} + p - 2 \tag{85}$$

$$2e_T \leq e_{\max} - e_N \tag{86}$$

$$2e_N \leq p - 2. \tag{87}$$

In each case, the two sides are equal if the right side is even, and they differ by unity otherwise. Hence

$$2(e_S + e_t) \leq p + 1 \tag{88}$$

and similarly

$$2(e_s + e_T) \geq -2(e_N + 1). \tag{89}$$

Since $-2(e_N + 1) \geq -p \geq e_{\min} + p - 2$ by (87) and (9), it follows that

$$2(e_s + e_T) \geq e_{\min} + p - 2. \tag{90}$$

Also, since $p \geq 2$ in all meaningful floating-point systems, we have $p + 1 \leq 3p/2 = (2p - 1) - (p - 2)/2$, and it follows from (88), (10), and (87) that

$$2(e_S + e_t) \leq e_{\max} - e_N. \tag{91}$$

We now turn to the needed lemmas.

LEMMA 4. *Let u, v, and $u * v$ be positive model numbers, where $*$ is addition or multiplication, and let x and y be positive machine numbers with $x \le u$ and $y \le v$. Then*

$$fl(x * y) \le u * v, \tag{92}$$

and similarly when all the inequalities are reversed.

PROOF. By Axiom 1. □

LEMMA 5. *Let n be an integer with $0 < n < b^p$, let B be a power of the base with $\sigma \le B \le \lambda/n$, and let $y_1, \ldots, y_n$ be machine numbers with $0 \le y_i \le B$. Then*

$$z \equiv fl(y_1 + \cdots + y_n) \le nB. \tag{93}$$

PROOF. The proof is by induction on n. If $n = 1$, the theorem is trivial. Otherwise, the last step is to compute $z = \mathrm{fl}(z_1 + z_2)$ where z_1 and z_2 are computed partial sums. By induction we may assume that $z_1 \le mB$ and $z_2 \le (n - m)B$ for some $m < n$. Now since $n \le b^p$, n is a model number, and Lemma 4 implies that $z \le nB$ as was to be shown. □

LEMMA 6. *If n is an integer in the interval $0 \le n < N$, then*

$$\frac{n}{\lambda} < b^{e_N - e_{max}}. \tag{94}$$

PROOF. By hypothesis, $n \le N - 1 = b^{e_N}(1 - b^{-e_N})$. Since $e_N < p$, it follows that $n < b^{e_N}(1 - b^{-p})$. Finally, dividing this by (6), we obtain (94). □

LEMMA 7. *Let x be a λ-bounded machine number, let u be a model number, and let B be a power of the base. Suppose B, B^{-1}, and uB are all in range, and hence also model numbers. If the computed value of the relation $x < uB$ is* true, *then in fact $x < uB$, and $fl(x/B) \le u$.*

PROOF. Since uB is a model number, we have $x < uB$ by Theorem 11. Hence $x/B < u$, and $\hat{x}/B \le u$ for all $\hat{x} \in x'$. Also, $\mathrm{fl}(x/B) \in x'/B$ by Theorem 5, and therefore $\mathrm{fl}(x/B) \le u$, as was to be shown. □

LEMMA 8. *Let g and h be expressions with computed values $\tilde{g}$ and $\tilde{h}$. Suppose $\tilde{g} \ge \sigma/\epsilon$, while the interval $\tilde{g}' + \sigma$ is λ-bounded. Also suppose $\tilde{h}$ is the result of an underflow, with $0 < \tilde{h} \le \sigma$. Finally, let $\hat{h}$ be any real number in the interval $0 < \hat{h} < \sigma$. Then for every $\hat{z} \in fl(\tilde{g} + \tilde{h})'$ there exist $\hat{g} \in \tilde{g}'$ and δ such that*

$$\hat{z} = (\hat{g} + \hat{h})(1 + \delta), \qquad |\delta| < \epsilon, \tag{95}$$

and therefore h is negligible in the context $g + h$.

PROOF. We shall prove that there exists $\hat{g} \in \tilde{g}'$ such that

$$\mathrm{fl}(\tilde{g} + \tilde{h}) \in (\hat{g} + \hat{h})', \tag{96}$$

and the conclusion follows immediately.

To prove it, let $\tilde{g}' = [u, v]$, and let w be the next larger model number. Since $u \ge \sigma/\epsilon$, *we have* $w - v \ge v - u \ge \sigma$. Hence $\mathrm{fl}(\tilde{g} + \tilde{h}) \in [u, w]$ by Axiom 1. Now if $\mathrm{fl}(\tilde{g} + \tilde{h}) \in [v, w]$, then (96) holds with $\hat{g} = v$. Otherwise, $\mathrm{fl}(\tilde{g} + \tilde{h}) \in [u, v]$, and (96) holds with $\hat{g} = u$. □

9.5 Overflow Analysis

To prove that the medium-range summation does not overflow, we note that $|x_i| < T$ by Theorem 11, and therefore

$$\tilde{y}_i \equiv \mathrm{fl}(x_i^2) \leq T^2 \tag{97}$$

by Lemma 4. It follows that

$$\tilde{z} \equiv \mathrm{fl}(\sum y_i) \leq nT^2 \tag{98}$$

by Lemma 5. Also,

$$e_N - e_{\max} + 2e_T \leq 0 \tag{99}$$

by (86), and therefore

$$\frac{n}{\lambda} T^2 < 1 \tag{100}$$

by Lemma 6. We conclude that $\tilde{z} < \lambda$ by (98) and (100).

In considering the small-range summation, we note that $|x_i| < t$ by Theorem 11, and therefore

$$|\mathrm{fl}(Sx_i)| \leq St \tag{101}$$

and

$$\tilde{y}_i \equiv \mathrm{fl}((Sx_i)^2) \leq (St)^2 \tag{102}$$

by successive applications of Lemma 4. It follows that

$$\tilde{z} \equiv \mathrm{fl}(\sum y_i) \leq n(St)^2 \tag{103}$$

by Lemma 5. Also

$$e_N - e_{\max} + 2(e_S + e_t) \leq 0 \tag{104}$$

by (91), and therefore

$$\frac{n}{\lambda} (St)^2 < 1 \tag{105}$$

by Lemma 6. We conclude that $\tilde{z} < \lambda$ by (103) and (105).

In considering the large-range summation, we begin with $|x_i| < b^{e_{\max}}$ rather than $|x_i| \leq \lambda$ because λ is not a power of the base. Following the same pattern as before, we have

$$\tilde{y}_i = \mathrm{fl}((sx_i)^2) \leq (sb^{e_{\max}})^2. \tag{106}$$

and

$$\tilde{z} \equiv \mathrm{fl}(\sum y_i) \leq n(sb^{e_{\max}})^2. \tag{107}$$

Also,

$$e_N - e_{\max} + 2(e_s + e_{\max}) \leq 0 \tag{108}$$

by (83), and therefore

$$\frac{n}{\lambda}(sb^{e_{\max}})^2 < 1 \tag{109}$$

by Lemma 6. We conclude that $\tilde{z} < \lambda$ by (107) and (109).

Finally, to prove that the precautions taken in the *return* section of the algorithm are sufficient to avoid overflow when $\sqrt{sum}$ is unscaled, we simply apply Lemma 7.

9.6 Underflow Analysis

In the *small* section of the algotithm, let $y_i = (Sx_i)^2$ by the ith term of the sum. If $|x_i| \geq \sigma$, then $\tilde{y}_i \geq (S\sigma)^2 \geq \sigma$ by Lemma 4 and (84), so underflow cannot occur. On the other hand, if $|x_i| < \sigma$, then $0 \leq \tilde{y}_i \leq (S\sigma)^2$ by Lemma 4. Hence, for every $\hat{y}_i \in \tilde{y}_i'$ there is an $\hat{x}_i \in x_i'$ such that $\hat{y}_i = (S\hat{x}_i)^2$, and therefore y_i is effectively exact. Finally, if several leading components of $\mathbf{x}$ have magnitude less than σ, their sum is effectively exact by similar reasoning.

In the *medium* section of the algorithm, the scaled sum from the *small* section may underflow when it is unscaled, and any term x_k^2 after the first may underflow when it is produced. In either case, the next step is to add the underflowed subexpression h to a sum g of one or more terms x_j^2. Since g includes the square of at least one component x_j whose magnitude is reportedly greater than or equal to t, we have $\tilde{g} \geq t^2 \geq \sigma/\epsilon$ by Theorem 11, Lemma 4, and (85), and therefore h is negligible by Lemma 8.

In the *large* section of the algorithm, the sum from the *medium* section may underflow when it is scaled, and any term $(sx_k)^2$ after the first may underflow when it is produced. In either case, the next step is to add the underflowed subexpression h to a sum g of one or more terms $(sx_j)^2$. Since g includes the scaled square of at least one component x_j with a magnitude that is reportedly greater than or equal to T, we have $\tilde{g} \geq (sT)^2 \geq \sigma/\epsilon$ by Theorem 11, Lemma 4, and (90), and therefore h is negligible by Lemma 8.

9.7 First-Order Error Analysis

For simplicity, we first consider the case where no scaling is needed. Using Table I, we assign an error bounded by ϵy_i to the computed value of the term $y_i = x_i^2$, and these contributions add up to an error bounded by ϵz in the sum $z = \sum y_i$. Each of the $n - 1$ nontrivial additions also contributes an error bounded by ϵz, so the absolute error in z is bounded by $n\epsilon z$, and the relative error by $n\epsilon$. This contributes a relative error of $n\epsilon/2$ to the norm, and the square root adds another ϵ, since it is assumed to be a supported unary operator.

If scaling and unscaling are required, they are special unary operators (see Theorem 5), and therefore no additional error terms are required. We conclude that the first-order relative error is bounded by $(n + 2)\epsilon/2$, as was to be shown.

10. EXTENDED RANGE AND PRECISION

The preceding algorithm demonstrates a very high standard of provable performance, achieved by techniques that are familiar and appropriate in nonportable programs on a variety of computers. Broadly speaking, the price of portability is

to adopt the overflow/underflow strategy recommended in Section 7 and to give up the conscious exploitation of the extra information in extra-precise numbers. Since the algorithm, the error bound, and the approximate analysis are all quite simple, this price may seem affordable. However, one should not lose sight of the fact that the whole purpose of the algorithm is to evaluate the expression $(x_1^2 + \cdots + x_n^2)^{1/2}$.

So far, we have implicitly accepted the constraint of having only a single floating-point number system. If an auxiliary *extended* system is available, providing both extended precision and extended range, then it is very easy to implement most elementary numerical operations so that overflow and destructive underflow are avoided and the results are accurate to within a single rounding error. For example, the following algorithm computes the Euclidean norm of a short vector $\mathbf{x} = (x_1, \ldots, x_n)$ to within ϵ, avoiding both overflow and underflow, except possibly when the extended result is shortened.

```
procedure norm (x, n)
extended sum := 0
assert (0 ≤ n < N)
i := 1
while (i ≤ n)
  sum := sum + x_i^2
  i := i + 1
return (short (√sum))
end
```

By the analysis in Section 9, the extended result is accurate to within a relative error of $(n + 2)\epsilon_{ext}/2$, where ϵ_{ext} is the maximum relative spacing between model numbers in the extended floating-point system. Hence the desired accuracy of the short result is guaranteed provided that we choose $n + 2 < 2\epsilon/\epsilon_{ext}$. Thus we can allow for N up to about 2×10^6 if the extended system provides 20 extra bits or 6 extra decimal digits in the fraction field.

To avoid overflow in this algorithm, we require only that $\lambda_{ext} \geq n\lambda^2$. Since $n < \lambda$ in all plausible contexts, it suffices for this algorithm to have $\lambda_{ext} \geq \lambda^3$. For most practical computations, it will probably suffice to have 2 extra bits in the exponent field ($\lambda_{ext} \approx \lambda^4$), but for still greater safety we recommend 3 extra bits ($\lambda_{ext} \approx \lambda^8$) or one extra decimal digit ($\lambda_{ext} \approx \lambda^{10}$).

To avoid underflow in this algorithm, we require only that $\sigma_{ext} \leq \sigma^2$. Again, for most practical computations, it will probably suffice to have 2 extra bits in the exponent field ($\sigma_{ext} \approx \sigma^4$), but for still greater safety we recommend 3 extra bits ($\sigma_{ext} \approx \sigma^8$) or one extra decimal digit ($\sigma_{ext} \approx \sigma^{10}$).

Once we have an extended number system at our command, we will need library procedures for computing with extended numbers. For occasions when maximal accuracy and robustness are essential we will need another auxiliary number system with still greater precision and range. Thus we may need an arbitrarily long sequence of number systems, each offering more precision (at least 20 extra bits or 6 extra decimal digits in the fraction field) and more range (at least 3 extra bits or one extra decimal digit in the exponent field) than its predecessor. Probably only the first two or three of these systems would be provided in hardware, but all of them should be conveniently available at the

language level. To avoid an infinite sequence of libraries, we will need a language facility for determining the length of a formal parameter and declaring auxiliary variables of greater length.

It is interesting to consider the number formats that are suggested by these requirements. Typically we would allocate one word for short numbers, two words for long numbers, three words for long long numbers, and so forth, with the bits or digits of each word after the first being divided in some fixed ratio between the exponent and fraction fields. As an example, for binary computers with 32-bit words, we are enthusiastic about current proposals [7, 20, 13] to partition the first word into an 8-bit signed exponent and a 24-bit signed fraction with an implicit normalization bit, and for long numbers to allocate 3 bits from the second word to the exponent field and the remaining 29 bits to the fraction field.

On a decimal computer with nine digits and perhaps 1 or 2 sign bits per word (all of which could easily be encoded in 32 bits), we would suggest partitioning the first word into a 2-digit exponent and a 7-digit fraction, and allocating one digit from each subsequent word to the exponent field and the remaining 8 digits to the fraction field.

On a computer with exceptionally long words (e.g., 60 bits or 18 decimal digits), it may be desirable to call the single-word numbers "long." In this case a reasonable facsimile of a "short" system can be provided by performing a range check and an optional rounding on every assignment of a long value to a short variable.

ACKNOWLEDGMENTS

While working with Andrew D. Hall on the Altran symbolic algebra system [3] during the period 1968–1973, I first became aware of the importance of modeling computers and of the surprising value of a simple parametrization. The same approach was used by Phyllis A. Fox, Andrew D. Hall, and Norman L. Schryer in their work on the Port mathematical subroutine library [12], starting in 1973.

The floating-point parameters that are used in Port and in this paper have been familiar since the early days of automatic computing. Their sufficiency for the analysis of machine-independent algorithms is implicit in Wilkinson's theory [24], and is made explicit in the restatement of that theory by Forsythe and Moler [11]. A similar set has been proposed by the IFIP Working Group on Numerical Software [10], though this group does not suggest including the parameters in programming languages. Forsythe and Moler assume that all arithmetic operations yield correctly rounded or correctly chopped results in the same floating-point number system. By contrast, the Port and IFIP groups have broadened their definitions to include many computers that are less simple or less accurate than this assumption implies. However, the IFIP definitions depend solely on the static number representation of the host computer, while the current Port definitions, given in Section 2 of this paper, reflect its dynamic behavior as well.

By 1976, interest in software portability had become so intense that two workshops were held to discuss it—one at Los Alamos, New Mexico, in May, sponsored by the Los Alamos Scientific Laboratory, and the other at Oak Brook, Illinois, in June, sponsored by the Argonne National Laboratory [8]. At these

workshops it became clear to me that numerical analysts want much more than the ability to move a program easily into a new computing environment; they also want to have a detailed understanding of its numerical behavior when it gets there. That is, numerical analysts want portable programs with portable documentation and portable error bounds, and they want to reach this goal at negligible cost in complexity or inefficiency. This challenge spurred me to test and sharpen my ideas at the workshops, and to pursue them with determination afterwards.

In March 1977, I presented a preliminary version of the model [4] at the Mathematics Research Center Symposium on Mathematical Software in Madison, Wisconsin. I also became a member of the IFIP Working Group on Numerical Software, and attended meetings in May 1977, May 1978, December 1978, and September 1979. While this paper has benefited greatly from discussions with numerous colleagues at these meetings and at Bell Laboratories, the underlying concepts remain essentially unchanged.

I am grateful to all those who have helped me during the lengthy evolution of this theory. I especially thank Christian H. Reinsch for incisive mathematical insights, Stuart I. Feldman for helping to transfer the benefits of the model into the world of programming languages (see [5]), Norman L. Schryer for developing a program to help in determining the parameter values for real computers (see Section 5), and M. Douglas McIlroy for many helpful comments on the paper as a whole.

REFERENCES

1. BLUE, J.L. A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Softw. 4,* 1 (March 1978), 15–23.
2. BROWN, W.S. Some fundamentals of performance evaluation for numerical software. In *Performance Evaluation of Numerical Software*, L.D. Fosdick (Ed.), North-Holland, Amsterdam, 1979, pp. 17–30.
3. BROWN, W.S. *Altran User's Manual*, 4th ed. Bell Laboratories, Murray Hill, N.J., 1977.
4. BROWN, W.S. A realistic model of floating-point computation. In *Mathematical Software III*, John R. Rice (Ed.), Academic Press, New York, 1977, pp. 343–360.
5. BROWN, W.S. AND FELDMAN, S.I. Environment parameters and basic functions for floating-point computation. *ACM Trans. Math. Softw. 6,* 4 (Dec. 1980), 510–523.
6. BURKS, A.W., GOLDSTINE, H.H., AND VON NEUMANN, J. Preliminary discussion of the logical design of an electronic computing instrument. In *Collected Works of John Von Neumann*, A.H. Taub (Ed.), vol. 5, Pergamon Press, New York, 1963, pp. 34–79. (See especially p. 57.)
7. COONEN, J., KAHAN, W., PALMER, J., PITTMAN, T., AND STEVENSON, D. A proposed standard for binary floating point arithmetic. *ACM SIGNUM Newsl.* (Special Issue) (Oct. 1979), 4–12.
8. COWELL, W. (Ed.) *Portability of Numerical Software.* Springer-Verlag, New York, 1977.
9. DEKKER, T.J. Correctness proof and machine arithmetic. In *Performance Evaluation of Numerical Software*, L.D. Fosdick (Ed.), North-Holland, Amsterdam, 1979, pp. 31–44.
10. FORD, B. Parameterization of the environment for transportable numerical software. *ACM Trans. Math. Softw. 4,* 2 (June 1978), 100–103.
11. FORSYTHE, G.E., AND MOLER, C.B. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1967. (See especially p. 87.)
12. FOX, P.A., HALL, A.D., AND SCHRYER, N.L. The PORT mathematical subroutine library. *ACM Trans. Math. Softw. 4,* 2 (June 1978), 104–126.
13. FRALEY, R., AND WALTHER, S. Proposal to eliminate denormalized numbers. *ACM SIGNUM Newsl.* (Special Issue) (Oct. 1979), 22–23.
14. GENTLEMAN, W.M., AND MAROVICH, S.B. More on algorithms that reveal properties of floating point arithmetic units. *Commun. ACM 17,* 5 (May 1974), 276–277.

15. KAHAN, W. Private communication.
16. KNUTH, D.E. Accuracy of floating-point arithmetic. In *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1969, pp. 195–210.
17. LAWSON, C.L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5*, 3 (Sept. 1979), 324–325.
18. MALCOLM, M.A. Algorithms to reveal properties of floating-point arithmetic. *Commun. ACM 15*, 11 (Nov. 1972), 949–951.
19. MOORE, R.E. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1966.
20. PAYNE, M., AND STRECKER, W. Draft proposal for a binary normalized floating point standard. *ACM SIGNUM Newsl.* (Special Issue) (Oct. 1979), 24–28.
21. REINSCH, C.H. Principles and preferences for computer arithmetic. *ACM SIGNUM Newsl. 14*, 1 (March 1979), 12–27.
22. SCHRYER, N.L. A test of a computer's floating-point arithmetic unit. *Computing Science Tech. Rep. No. 89*, Bell Laboratories, Murray Hill, N. J., Feb. 1981.
23. WILKINSON, J.H. *The Algebraic Eigenvalue Problem*. Oxford University Press, New York, 1965.
24. WILKINSON, J.H. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1963.

Received July 1980; revised April 1981; accepted June 1981