ROUNDOFF ERRORS

Rounding errors are the errors arising from the use of floating-point arithmetic on digital computers. Since the computer word has only a fixed and finite number of bits or digits, only a finite number of real numbers can be represented on a computer, and the collection of those real numbers that can be represented on the computer is called the *floating-point system* for that computer. Since only finitely many real numbers can be represented exactly, it is possible, indeed likely, that the exact solution to any particular problem is not part of the floating point system and hence cannot be represented exactly. Ideally, one would hope that one could obtain the *representable number* closest to the true exact answer. With simple computations this is usually possible, but is more problematic after long or complicated computations. Even the four basic operations, addition, subtraction, multiplication, and division, cannot be carried out exactly, so the intermediate results in any computation will suffer from contamination of *rounding errors,* and the final results will suffer from the accumulated effects of all the intermediate rounding errors. The field of *numerical analysis* is the study of the behavior of various algorithms when implemented in the floating-point system subject to rounding errors. In this article, we describe the main features typically found in floating-point systems in computers today, give some examples of unusual effects that are caused by the presence of rounding errors, and discuss techniques developed to perform accurate fault tolerance in the presence of these errors.

REPRESENTATION OF FLOATING-POINT NUMBERS

Mantissa Plus Exponent

All computers today represent floating-point numbers in the form mantissa $\times$ base$^{\text{exponent}}$, where the mantissa is typically a number less than 2 in absolute value, and the exponent is a small integer. The base is fixed for all numbers and hence is not actually stored at all. Except for hand-held calculators, the base is usually 2 except for a few older computers where the base is 8 or 16. The mantissa and exponent are represented in binary with a fixed number of bits for each. Hence a typical representation is

$$[s \quad e_7 \quad e_6 \ldots e_0 \quad m_{23} \quad m_{22} \ldots m_1 \quad m_0] \tag{1}$$

where s is the sign bit for the mantissa, $e_7, \ldots, e_0$ are the bits for the exponent, and $m_{23}, \ldots, m_0$ are the bits for the mantissa. If the base is fixed at 2, then the number represented by the bits in Eq. (1) is

$$(-1)^s \times (m_{23} \cdot 2^0 + m_{22} \cdot 2^{-1} + \cdots + m_0 \cdot 2^{-23}) \times 2^{\text{exponent}} \tag{2}$$

where the exponent is an 8-bit signed integer. In this example, we have fixed the number of bits for the mantissa and the exponent to 24 and 8, respectively, but in general these vary from computer to computer, and even within the computer vary from *single* to *double precision.* Notice that the mantissa represented in Eq. (2) has the "binary point" (analog to the usual decimal point) right after the leftmost digit. Regarding the exponent as a signed integer, it is not typically represented as a ones or twos complement number but more often in *excess 127* notation, which is essentially an unsigned integer representing the number 127 larger than the true exponent. Again, if we have k bits instead of 8 as in this example, then the 127 is replaced by $2^{k-1} - 1$.

We illustrate this with a few examples, where we shorten the mantissa to 7 bits plus a sign and the exponent to 4 bits. Hence the exponent is in *excess 7* notation:

decimal	binary	bits	remarks
+5/2	$+1.01 \times 2^1$	0 1000 *1*010000	
−5/2	-1.01×2^1	1 1000 *1*010000	
+20	$+1.01 \times 2^4$	0 1011 *1*010000	
1/3	$+1.010101 \times 2^{-2}$	0 0101 *1*010101	inexact
1/10	$+1.100110 \times 2^{-3}$	0 0100 *1*100110	inexact

(3)

We remark that this representation, using normalized mantissas and excess notation for the exponents, allows one to compare two positive floating-point numbers using the usual integer compare instructions on the bit patterns.

Normalization

Notice that in Eq. (3) there can be multiple ways to represent any particular decimal number. If the leading digit of the mantissa is zero, or more generally, if the digit(s) of the mantissa to the left of its binary point do not represent 1 (e.g., 0. *xxx*. . ., 10.*xxx*. . .), then the number is said to be *unnormalized,* otherwise it is said to be *normalized.* So we could also use the representation

decimal	binary	bits	remarks
+20	$+0.00101 \times 2^7$	0 1110 0001010	unnormalized
1/3	$+0.101010 \times 2^{-1}$	0 0110 0101010	inexact and unnormalized

(4)

When the number is unnormalized, we lose space for significant digits; hence floating-point numbers are always stored in normalized fashion. We see that in Eq. (3), the normalized representation for the number 1/3 captures more nonzero bits than the unnormalized representation in Eq. (4). When the base is equal to 2, then the leading digit of the mantissa is just a bit whose only possible nonzero value is 1, and hence it is not even stored. So in the representation in Eq. (1), the bit m_{23} is always 1 and is not actually stored in the computer. When not stored in this way, the bit m_{23} is called an *implicit bit.* These bits are written in *italics* in Eq. (3).

Special Numbers, Overflow, Underflow

The representation in Eq. (1) with the implicit bit m_{23} does not admit the number 0, since 0 would have an all-zero mantissa that must be unnormalized. To accommodate this, certain special bit patterns are reserved to zero and certain other special "numbers." A zero is often represented by a word of all zero bits, which would otherwise represent the smallest representable positive floating-point number. If a calculation gives rise to an answer less than the smallest representable number (in absolute value), then an *underflow* condition is said to exist. In the past, the result was simply set to zero, but in the recent IEEE standard, the result is denormalized.

The use of gradually denormalized numbers involves those floating-point numbers which are less (in absolute value) than the smallest representable normalized number. As discussed in Ref. (1), there is a relatively big gap between the smallest representable normalized number and zero. To fill this gap, the IEEE decided to allow for the use of unnormalized numbers. We can illustrate this with the representation in Eq. (3). The smallest normalized number representable in Eq. (3) is $+1.00_{\text{binary}} \times 2^{-7}$. However, we can represent smaller numbers in an unnormalized manner, such as $+0.10_{\text{binary}} \times 2^{-7}$. Since we have adopted the convention of using the implicit bit, such an unnormalized number cannot be encoded in this format. The solution is to provide that the smallest representable normalized number be actually $+1.00_{\text{binary}} \times 2^{-6}$, reserving the smallest possible exponent value for unnormalized numbers. This was adopted in the IEEE standard (see below). Since this smallest exponent value has all its bits equal to zero, the representation of the number zero in this format becomes just a special case of such unnormalized numbers. As pointed out by Goldberg (1), the use of denormalized numbers also guarantees that the computed difference of two unequal numbers will never be zero.

A more serious problem occurs if the result of the calculation is larger than the largest representable number. This is called an *overflow* condition, and in most older computers, this would generate an error. However, in the recent IEEE floating point standard (discussed below), such a result would be replaced with a special bit pattern representing *plus-infinity* or *minus-infinity*. When two such infinities are combined, the result can be totally undefined, so yet another special bit pattern is reserved for such a result. This last result is called *Not A Number,* and is often printed by most computer systems as `NaN`. By not generating an exception upon overflow, programs may fail more gracefully.

Rounding versus Chopping

Another issue affecting rounding errors is the choice of rounding strategy. Given any particular real number, which nearby floating point number should one use? For example, in Eq. (3), when we represented 1/3 as an unnormalized number, we chopped away the last bit, but an alternative choice would be to round up to the next higher number to yield $+ 0.101011_{\text{binary}} \times 2^{-1}$. The error committed in chopping in this case is .0052, but in rounding is only .0026. But rounding requires slightly more computation since the digits being removed must be examined. This issue arises when converting a number from an external decimal representation and when trying to fit the result of an intermediate arithmetic operation into a memory word. This is because the arithmetic logic units on most computers actually operate on more digits than can fit in a word, the extra digits being called *guard digits,* discussed below.

The IEEE standard actually provides that the default rounding strategy should be a "round-to-even" strategy. The round-to-even mode is exactly the rounding strategy described above, except when the number being rounded lies exactly half-way between two representable numbers, as in rounding 12.5 to an integer. The default round-to-even strategy selects the representable number whose last digit is even, so that 12.5 would round to 12 and not 13. If the rounding in this case were always up, then more numbers would end up being increased than decreased during the rounding process, on average. If the combinations of trailing digits occur equally likely, it is generally desirable that the number of times the rounding is up is about equal to the number of times the rounding is down, to try to cancel out this bias as much as possible.

Guard Digits

Guard digits are extra digits kept only within the Arithmetic Logic Unit (ALU) during the course of individual floating-point operations. They are never stored in memory. The ALU carries out the operation using at least one extra guard digit, then the result is rounded to fit in the register of a memory word. We illustrate the effect of guard digits using the simple addition of two decimal floating point-numbers, $1.01 \times 10^{+1}$ and -9.93×10^{0} (this example is from Ref. 1), where we keep 3 decimal digits in the mantissa. To accomplish this, the first step for the ALU is to shift the decimal point in the second operand to make the exponents match, yielding $-.993 \times 10^{+1}$. Then the mantissas may be added together directly. The accuracy of the answer is greatly affected by the number of digits kept for the computation. The simplest approach is to use simple chopping and to keep only the digits corresponding to the larger operand. The result in this case is $1.01 \times 10^{+1} - 0.99 \times 10^{+1} = 2.00 \times 10^{-1}$. If, however, we keep at least one extra guard digit, then we obtain $1.010 \times 10^{+1} - 0.993 \times 10^{+1} = 1.70 \times 10^{-1}$. The latter answer is exact, whereas the former result has no correct digits.

The reader may ask whether keeping just one guard digit suffices to make a significant enhancement to the accuracy of floating-point arithmetic operations. The answer can be found in Ref. (1), in which it is proved that if no guard digit is kept during additions, then the error could be so large as to yield no correct digits in the answer, whereas if just one guard digit is kept during the operation, the result being rounded to fit in the memory word, then the error will be at most the equivalent of 2 units in the last significant digit. In this context, the "correct answer" is regarded as the answer computed using all available digits and keeping "infinite precision" for the intermediate results.

IEEE Standard

The previous discussion has shown that there are many choices to be made in representing floating-point numbers, and in the past different manufacturers have made different, incompatible, choices. The result is that the behavior of floating-point algorithms can vary from computer to computer, even if the precision (number of bits used for exponent and mantissa) stays the same. In an attempt to make the behavior of algorithms more uniform across platforms, as well as to improve the performance of such algorithms, the IEEE has established a *floating-point standard* which specified some of these choices (2,3). This standard specifies the kind of rounding that must be used, the use of guard digits, the behavior when underflow or overflow occurs, etc. The first standard (2) was limited to 32- and 64-bit floating-point words, and provided for optional extended formats for computers with longer words. The second standard (3) extended this to general length words and bases. The principal choices made in (2) include the following:

- Rounding to nearest (also known as round to even)
- Base 2 with a sign bit and an implicit bit
- Single precision with 8-bit exponent and 23-bit mantissa fields (not including the implicit bit)
- Double precision with 11 bit exponent and 52 bit mantissa fields (not including the implicit bit)
- The presence of $\pm\infty$ and `NaN`, as well as ± 0
- Gradually denormalized numbers for those numbers unrepresentable as normalized numbers
- User-settable bits to turn on exception handling for overflow, underflow, etc. and to vary the rounding strategies

We have tried to explain the reasons for some of these choices with the above discussion, but detailed formal analyses of these choices can be found in Ref. (1).

Usual Model for Round-Off Error

In order to analyze the behavior of algorithms in the presence of round-off errors, a mathematical model for round-off errors is defined. The usual model is as follows, where $\odot$ represents any of the four arithmetic operations:

$$fl(a \odot b) = (a \odot b) \cdot (1 + \epsilon) \tag{5}$$

where $|\epsilon| \leq$ macheps, and macheps is called the *unit roundoff* or *machine epsilon* for the given computer. The motivation behind this model is that the best any computer could do is to perform any individual arithmetic operation exactly, and then round or chop to the nearest floating-point number when finished. The rounding or chopping involves changing the last bit in the (base 2) mantissa, and hence the macheps is the value of this last bit—always relative to the size of the number itself. This model can be expensive to implement, so some computer manufacturers have designed arithmetic operations that do not obey it, but one can show that one or two guard digits suffice to be consistent with this model.

In most higher level languages, the details of the floating-point representation (especially the length of a computer word) are generally hidden from the user. Hence the macheps has a definition that can be computed in a higher level language, not specifically by the number of bits in a word. The macheps is defined as the value of ϵ yielding the minimum in

$$\min_{\epsilon > 0} fl(1 + \epsilon) > 1 \tag{6}$$

This formula can be used to calculate macheps by trying a sequence of trial values for ϵ, each entry one-half the previous, until equality in Eq. (6) is achieved. The specific value of macheps depends on the rounding strategy. This can be most easily illustrated with 3 digit decimal floating point arithmetic. The smallest s such that $fl(1 + s) > 1$ is 1.00×10^{-3} in chopping, 5.00×10^{-4} if a traditional rounding strategy is used, and 5.01×10^{-4} if rounding to even is used. In general, macheps in rounding is approximately half that obtained using chopping.

CATASTROPHIC EFFECTS OF ROUND-OFF ERROR

To illustrate how rounding errors can accumulate catastrophically in unexpected ways, we give two examples adapted from Ref. (4). An extensive introductory discussion on the effects of rounding error in scientific computations involving the use of floating point can be found in Ref. (4,5).

Of the four arithmetic operations, subtraction and addition are really the same operation. Most loss of significance and cancellation errors described below arise from these two operations. Multiplication and division give rise to problems only if the results overflow, underflow, or must be denormalized. An unusual effect of the fact that floating-point numbers are discrete in nature is that the operations no longer obey the usual laws of real numbers. For example, the associative law for addition does not hold for floating-point numbers. If s is a positive number less than macheps, but more than macheps $\div$ 2, then $1 + (s + s)$ will be strictly bigger than 1, but $(1 + s) + s$ will equal 1. This is an extreme case, but the order in which numbers are added up can affect the computed sum markedly. This is further illustrated by the first example below.

It has been pointed out (1) that the use of the denormalized numbers means that programs can depend on the fact that $fl(a - b) = 0$ implies $a = b$. However, it can still happen that

$fl(a * b) = a$ when $a \neq 0$ and $b \neq 1$. This can happen, for example, when a is the smallest representable floating-point number, and b is a number between .6 and 1, when rounding is used. Programs whose logic depend on $fl(a * b)$ being always different from a can suffer very mysterious failures. However, generally, multiplication and division do not give rise to catastrophic rounding errors unless numbers near the ends of the exponent range are involved, or when combined with other operations.

Taylor Series for e^{-40}

A simple algorithm to compute the exponential function e^x is to use its well-known Taylor series:

$$e^x = \sum_{i \geq 0} \frac{x^i}{i!}$$

When $x \geq 0$, this can yield accurate results if one is willing to take enough terms, but if used when $x < 0$, this can yield catastrophic results, all due to the finite word length of the machine. To take an extreme case, let $x = -40$. Then all the terms after the 140th term are much less than 10^{-16} and decay rapidly, and the result is also very small: $e^{-40} = 4.2484 \times 10^{-18}$. But simply adding up the terms of the Taylor series will yield 1.8654, which is nowhere near the true answer. The problem is the terms in this series alternate in sign, and the intermediate terms reach $1.4817 \times 10^{+16}$ in magnitude, and we end up subtracting very large numbers that are almost equal and opposite. This results in severe cancellation.

Numerical Derivative of e^x at $x = 1$

Suppose we take the naive approach to approximate the numerical derivative of a function f:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

for some suitable small h. Applying this to $f(x) = e^x$ and taking the derivative at $x = 1$, we find that we get as much accuracy with $h = 2 \times 10^{-6}$ as with $h = 10^{-10}$ on a machine with approximately 16 decimal digits in the mantissa. In both cases, the error is about 3×10^{-6}, and less than half the computed digits are good. Here again we have severe cancellation from subtracting numbers that are almost equal. Hence, simply making the step size h smaller does not lead to more accuracy.

EFFECT ON ALGORITHMS

Round-Off Causes Perturbation to Data and Intermediate Results

The examples above are extreme cases showing that catastrophic loss of accuracy can result if floating-point arithmetic is not used carefully. The effect of round-off error is applied to each intermediate result and is guaranteed to be small relative to those intermediate results. However, in some cases, those intermediate results can be larger than the final desired results, leading to errors much larger than would be expected from just the sizes of the input and final output of a particular algorithm. However, in some algorithms, such as when simulating an ordinary differential equation (such as a control system) $\dot{x} = Ax + f$ where f is a forcing function, the intermediate results may not be any larger than than the final or initial values, yet severe loss of accuracy can result. One source of error is the propagation of intermediate errors, and in nasty cases, the effect of those intermediate errors can grow, becoming more and more significant as the algorithm proceeds.

Algorithm Stability versus Conditioning of Problem

In an attempt to analyze and alleviate the effects of rounding errors, numerical analysts have developed paradigms for the analysis of the behavior of numerical algorithms and have used these paradigms to develop algorithms themselves for which one can prove that the effect of rounding errors is bounded. It is useful to describe these paradigms. The most fundamental is the concept of algorithm stability versus conditioning of the problem. The latter refers to the ill posedness of the problem. If a problem is ill posed, then slight variations to the coefficients in the problem will yield massive changes to the exact solution. In this case, no floating point algorithm will be able to compute a solution with high accuracy. If the problem is well posed, then one would expect a good algorithm to compute a solution with full accuracy. An algorithm that fails that requirement is called *unstable*. An algorithm that is able to compute solutions with reasonable accuracy for well-posed problems, and that does not lose more accuracy on ill-posed problems than the ill-posed problems deserve, is called *stable*.

Relevance to Fault Tolerance

The study of rounding errors is relevant to fault tolerance in two ways. At the most elementary level, the presence of rounding errors means that no computed solution will be exact, and we cannot check for the presence of faults by checking if the computed solution satisfies some condition *exactly*. Any fault detection system would have to allow for the presence of errors in the solution arising naturally from normal rounding errors. This thus leads to the difficult task of distinguishing between errors arising from natural rounding errors and errors arising from faults. If the underlying problem is ill posed to any degree (called *ill-conditioned*) then the accuracy of the computed solution will be very poor, even if that solution were computed correctly.

On the other hand, many numerical algorithms have been shown to be stable in a certain sense. Algorithms arising in matrix computations have been especially well studied. In particular, in the domain of solving systems of linear equations, certain algorithms have been shown to compute the exact solution to a system within a small multiple of *macheps* of the original system of equations, even when the system is moderately ill posed. In some cases, precise bounds on the possible discrepancy have been derived. These can be used to develop conditions that then can be used to check for faults. Note that even if the computed solution exactly satisfies a nearby system of equations, that does not imply that the error in the solution is small, unless the system of equations is very well conditioned. As a consequence, any validation procedure for fault detection can only check for the correctness of the computed solutions indirectly, and not by computing the accuracy of the solution itself.

The result of this analysis has been the development of conditions to check the correctness of numerical computations, mainly in the domain of matrix computations and signal processing. These conditions all involve the determination of a set of precise tolerances that are tight enough to enforce sufficient accuracy in the solutions, yet guaranteed to be loose enough to be satisfiable even when solving problems that are moderately ill posed. The principal approaches in this area involve the use of checksums, backward error assertions, and mantissa checksums. In all cases, it has been found that applying these techniques to series of operations instead of checksumming each individual operation has been the most successful.

Instead of using tolerances, an alternative approach that has been used with some success is interval arithmetic. Space does not permit a full treatment here, since most software, languages, compilers, and architectures do not provide interval arithmetic as part of their built-in features. A synopsis of interval arithmetic, including its uses and applications can be found in Ref. 6. In this article, we limit our discussion to a short description. The easiest way to view interval arithmetic is to consider replacing each real number or floating-point number in the computer with two numbers representing an interval $[a, b]$ in which the "true" result is supposed to lie. Arithmetic operations are performed on the intervals. For example, addition would result in $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$. If all endpoints are positive, then multiplication of intervals would be computed by $[a_1, b_1] \cdot [a_2, b_2] = [a_1 \cdot a_2, b_1 \cdot b_2]$. All the other arithmetic operations and more general situations can be defined similarly. However, if no special precautions are taken, the size of the intervals can grow too large to give useful bounds on the location of the "true" answers. So most successful applications involve more sophisticated analysis of whole series of arithmetic operations such as an inner product rather than analysis of each individual operation, or else use some statistical techniques to narrow the intervals. As pointed out in Ref. 1, in order to maintain the guarantee that computed intervals contain the "true" answer, it is necessary to round down the left endpoint and round up the right endpoint of each computed interval. This requires the user to vary the rounding strategy used within the computer. The IEEE standards require that the hardware provide a way for the user to vary the rounding strategy as well as some other parameters of the arithmetic, but, as pointed out by Kahan (7) most compilers and systems today do not actually provide the user access to that level of hardware control.

SYNOPSIS OF FAULT TOLERANCE TECHNIQUES FOR LINEAR ALGEBRA

We present a short synopsis of various techniques that have been proposed for the verification of floating-point computations, mostly in linear algebra. The use of checksums was made popular by Abraham (8). This method takes advantage of the fact that the result of most computations in linear algebra bears a linear relation to the arguments originally supplied. So a linear combination of those results bears the same linear relation to that same linear combination of the original data. For example, the row operations in Gaussian elimination (used to solve systems of linear equations) can be checksummed by taking linear combinations of the entries in each row. When two rows are added in a row operation, the checksums are also added and compared with the checksum generated from scratch from the newly computed row. In a floating-point environment, the checksums will be corrupted by round-off error, and hence a tolerance must be used to decide if they match. This tolerance depends on the condition number of the matrix of *checksum coefficients* (9).

Another class of methods involves comparing the results with certain error tolerances. For matrix multiplication, the error tolerances are forward error bounds ("how far is the computed answer from the true answer?") (10). For solving systems of linear equations, the error tolerances are backward error bounds ("how well does the computed answer fit the original problem?" or more precisely, "how much must the original problem be changed so that the computed answer fits it exactly?") (11). In these methods, the error bounds used depend critically on the properties of the arithmetic, particularly the macheps, and in some cases on the conditioning of the underlying system being solved. Hence these techniques can sometimes detect violations of the mathematical assumptions of solvability that are due to ill posedness of the problem.

Yet a third class of methods is derived by considering the mantissas alone. It turns out that for certain floating-point operations (like multiplication), one can compute checksums of the mantissas alone, treating them as integers (12,13). Then the checksum computed the same way derived from the mantissa of the result must match the combination of the original mantissa checksums. Since the checksums are computed using integer arithmetic, round-off errors do not apply. The only limitation to this approach is that this technique cannot be applied to all floating-point operations (like addition), but can be used to check the multiplication part of inner products. However, when both the floating-point and integer mantissa checksum tests are applied in a "hybrid test," all operations are covered and much higher error coverages are obtained compared to using only the floating-point test.

The latter two techniques are discussed further below.

ANALYSIS OF ERROR PROPAGATION

The research area of numerical analysis is devoted to the study of the behavior of algorithms that must emulate continuous mathematics on a digital computer using floating-point arithmetic. Such analyses are based on the previously mentioned model for the error in floating-point arithmetic in Eq. (5):

$$fl(a \odot b) = (a \odot b) \cdot (1 + \epsilon)$$

where $|\epsilon| \leq$ macheps, and macheps is called the *unit round-off* or *machine epsilon* for the given computer. We illustrate with a couple of examples how the propagation of errors is typically analyzed. Space does not permit a complete derivation of error bounds, but we refer the reader to Refs. 14 and 15 for complete discussions on error analysis of numerical algorithms.

The dot product or inner product of two vectors provides a simple example of how round-off errors can propagate. The inner product of two vectors $\boldsymbol{x}, \boldsymbol{y}$ can be computed by

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

In floating-point arithmetic, however, one will obtain

$$\begin{aligned} fl(\boldsymbol{x} \cdot \boldsymbol{y}) &= fl\{\ldots fl[fl(x_1 y_1) + fl(x_2 y_2)] + \cdots + fl(x_n y_n)\} \\ &= \{[(x_1 y_1)(1+\epsilon_1) + (x_2 y_2)(1+\epsilon_2)](1+\delta_2) \\ &\quad + \cdots + (x_n y_n)(1+\epsilon_n)\}(1+\delta_n) \\ &= (x_1 y_1)(1+\epsilon_1)(1+\delta_2)\ldots(1+\delta_n) \\ &\quad + (x_2 y_2)(1+\epsilon_2)(1+\delta_2)\ldots(1+\delta_n) \\ &\quad + \cdots + (x_n y_n)(1+\epsilon_n)(1+\delta_n) \end{aligned}$$

where the ϵ_i, δ_i's are quantities bounded by the macheps of the machine. Carrying out the analysis in [Ref. 15, sec. 2.4] one can obtain the relation, for some δ:

$$(1+\epsilon_1)(1+\delta_2)\ldots(1+\delta_n) = (1+\delta) \quad \text{such that} \quad |\delta| \le 1.01n\mathrm{u}$$

where n is the dimension of the vectors and u = macheps is the *unit round-off* for the machine, under the assumption that $n\mathrm{u} < .01$. This leads to the bound on the error in the dot product (15, sec. 2.4.5).

$$|fl(\boldsymbol{x} \cdot \boldsymbol{y}) - \boldsymbol{x} \cdot \boldsymbol{y}| \le 1.01n\mathrm{u}(|\boldsymbol{x}| \cdot |\boldsymbol{y}|)$$

where $|\boldsymbol{x}|$ denotes the vector of absolute values of the entries in $\boldsymbol{x}$. This formula can be interpreted as saying that if two vectors are accumulated together, the accumulated error is bounded by the machine unit round-off amplified by a factor growing only linearly in the dimension n. Applying this result to matrix-matrix multiplication using the usual inner product algorithm, we obtain the bound

$$fl(A \cdot B) = A \cdot B + E \quad \text{with} \quad |E| \le 1.01n\mathrm{u}|A| \cdot |B|$$

where $\le$ here denotes elementwise inequality.

Most algorithms, even in linear algebra, do not consist solely of inner products, and in such cases a different approach to error analysis based on the *backward error analysis* has been very successful. We consider the example of Gaussian elimination, used to solve systems of linear equations expressed in matrix terms as $A\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{x}$ is the vector of unknowns. The Gaussian elimination algorithm with row interchanges (e.g., partial pivoting) (15, sec. 3.2) can be viewed as computing the factorization of the matrix A of the form $PA = LU$, where P is a permutation matrix encoding the row interchanges occurring during the elimination process, L is a lower triangular matrix holding the multipliers, and U is an upper triangular matrix encoding the coefficients of the eliminated equations. This factorization of A into a product of simpler matrices then permits the solution of the original set of equations $A\boldsymbol{x} = \boldsymbol{b}$ by forward and back substitution.

The Gaussian elimination algorithm must compute multiples of certain rows to be added to other rows in order to eliminate variables one at a time, but in floating-point arithmetic, the multiples computed will be subject to round-off error. This means that variables will be eliminated only approximately. It becomes extremely complicated to analyze the effect of such approximations on the values of subsequent multipliers and eliminated rows. In an extreme case, slight perturbations may affect the row interchanges performed during the algorithm, yielding very different results. Hence it is possible that the computed L and U will not be close to the L, U that would be obtained in exact arithmetic. Thus it is not possible to obtain a tight *forward error bound* of the form $\|U_{\text{computed}} - U_{\text{exact}}\| \le$ some_bound. However it has been shown that a tight *backward error bound* can be obtained. One such bound has the form (15, sec. 3.3.1).

$$L_{\text{computed}} U_{\text{computed}} = PA + H \quad \text{with} \quad |H| \le 3n\mathrm{u}|A|\rho + O(\mathrm{u}^2)$$

where ρ is a growth factor depending on the pivoting strategy used, and is typically a small number. This bound does not say anything about how close U_{computed} is to the "true" U, but does say that L_{computed}, U_{computed} are the *exact* factors for a matrix $A + P^{\mathrm{T}}H$ that is very close to the original one. When used to compute the solution to the original system of linear equations, this will guarantee that the computed solution will almost satisfy that system of equations, or exactly satisfy a nearby system of equations, even if there is no guarantee that the solution obtained will be anywhere close to the solution that would be obtained in exact arithmetic.

INTEGER CHECKSUMS FOR FLOATING-POINT COMPUTATIONS

Floating-Point Checksum Test

Many previous approaches for error detection and correction of linear numerical computations have been based on the use of checksum schemes (8,16–18). A function f is *linear* if $f(\boldsymbol{u} + \boldsymbol{v}) = f(\boldsymbol{u}) + f(\boldsymbol{v})$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors. We discuss here a commonly used checksum technique for a frequently encountered computation, matrix multiplication.

The floating-point checksum technique for matrix multiplication due to Ref. 8 is as follows. Consider an $n \times m$ matrix A with elements $a_{i,j}$, $1 \le i \le n$, $1 \le j \le m$. The *column checksum matrix* A_c of the matrix A is an $(n+1) \times m$ matrix whose first n rows are identical to those of A, and whose last row *rowsum*(A) consists of elements $a_{n+1,j} := \sum_{i=1}^{n} a_{i,j}$ for $1 \le j \le m$. Matrix A_c can also be defined as

$$A_c := \left[\frac{A}{\boldsymbol{e}^{\mathrm{T}}A}\right]$$

where $\boldsymbol{e}^{\mathrm{T}}$ is the $1 \times n$ row vector (1, 1, . . ., 1). Similarly, the *row checksum matrix* A_r of the matrix A is an $n \times (m+1)$ matrix whose first m columns are identical to those of A, and whose last column *colsum*(A) consists of elements $a_{i,n+1} := \sum_{j=1}^{n} a_{i,j}$ for $1 \le i \le n$. Matrix A_r can also be defined as $A_r := [A|A\boldsymbol{e}]$, where $A\boldsymbol{e}$ is the column summation vector. Finally, a *full checksum matrix* A_f of A is defined to be the $(n+1) \times (m+1)$ matrix, which is the column checksum matrix of the row checksum matrix A_r. Corresponding to the matrix multiplication $C := A \times B$, the relation $C_f := A_c \times B_r$ was established in Ref. 8. This result leads to their ABFT scheme for error detection in matrix multiplication, which can be described as follows:

Algorithm Mult_Float_Check(A,B)
/* A is an $n \times m$ matrix and B an $m \times l$ matrix. */

1. Compute A_c and B_r.
2. Compute $C_f := A_c \times B_r$.
3. Extract the $n \times l$ submatrix D of C_f consisting of the first n rows and l columns. Compute D_f.
4. Check if $\boldsymbol{c}_{n+1} \stackrel{?}{=} \boldsymbol{d}_{n+1}$, where $\boldsymbol{c}_{n+1}$ and $\boldsymbol{d}_{n+1}$ are the $(n+1)$th rows of C_f and D_f, respectively.
5. Check if $\boldsymbol{c}^{n+1} \stackrel{?}{=} \boldsymbol{d}^{n+1}$, where $\boldsymbol{c}^{n+1}$ and $\boldsymbol{d}^{n+1}$ are the $(n+1)$th columns of C_f and D_f, respectively.
6. If any of the above equality tests fail then return ("error") else return ("no error").

The following result was proved indirectly in Theorem 4.6 of Ref. 8.

Theorem 1 At least three erroneous elements of any full checksum matrix can be detected, and any single erroneous element can be corrected.

Theorem 1 implies that Mult_Float_Check can detect at least three errors and correct a single error in the computation of $C_f = A_c \times B_r$, as long as all operations, especially floating-point additions, have a large enough precision such that no round-off inaccuracies are introduced. Of course, such an "infinite" precision assumption is unrealistic, and thus the above checksum scheme is susceptible to round-off introduced by finite-precision floating-point arithmetic, as described earlier. In particular, there can be *false alarms* in which the checksum test fails because of round-off in spite of the absence of real errors (those occurring due to hardware glitches or failures) in the computation. Alternatively, real errors could be masked/canceled by round-off leading to nondetection of a potential problem in the hardware.

Integer Checksum Test

The susceptibility of the floating point checksum test to roundoff inaccuracies can be largely mitigated by applying integer checksums to various (linear) computations that are "mantissa preserving." This results in high error coverage and zero false alarms stemming from the fact that integer checksums do not have to contend with the round-off error problem of floating-point checksums. The integers involved are derived from the mantissas of the intermediate floating-point results of the floating-point computation. To date, we have successfully applied integer checksums (hereafter also called mantissa checksums) to two important matrix computations, matrix–matrix multiplication and **LU** decomposition (using the Gaussian elimination algorithm) (12,13). Here we briefly discuss the general theory of mantissa checksums and how they are applied to these two computations.

General Theory. In the following discussion, we use $\boldsymbol{u} = (u_1, \ldots, u_n)^T$ to represent column vectors and a, b, c, etc., for scalars. Unless otherwise specified, these variables will denote floating-point quantities. We use the notation $mant(a)$ to denote the mantissa of the floating-point number a treated as an integer. For example, considering 4-bit mantissas and integers, if 1.100 is the mantissa portion of a, with its implicit binary point shown, then the value of the mantissa is 1.5 in decimal. However, $mant(a) = 1100.$, and has value 12 in decimal. Furthermore, for a vector $\boldsymbol{v} := (v_1, \ldots, v_n)^T$, $mant(\boldsymbol{v}) := [mant(v_1), \ldots, mant(v_n)]^T$, and for a matrix $A := [a_{i,j}]$, $A^{mant} := mant(A) := [mant(a_{i,j})]$—we use the $:=$ symbol to denote equality by definition, $=$ to denote the standard (derived) equality, and $\stackrel{?}{=}$ to denote an equality test of two quantities that are theoretically supposed to be equal, but may not be because of errors and/or round-off.

Let f be any linear function on vectors. The linearity of f allows us to apply the following floating-point checksum test on the computation of f on a set S of vectors:

$$f\left(\sum_{\boldsymbol{v}\in S}\boldsymbol{v}\right) \stackrel{?}{=} \sum_{\boldsymbol{v}\in S} f(\boldsymbol{v}) \tag{7}$$

Ignoring the round-off problem, the left- hand side (LHS) and right-hand side (RHS) of the above equation should be equal, if there are no errors in computing the $f(\boldsymbol{v})$'s for all $\boldsymbol{v} \in S$ (which is the original computation), in summing up these $f(\boldsymbol{v})$'s to get the RHS, and in summing up the $\boldsymbol{v}$'s and applying f to the sum to get the LHS. If they are not equal, then an error is detected. Unfortunately, because of round-off, the test of Eq. (7) often fails to hold in the absence of computation errors. Therefore, we want to seek an integer version of this test that is not susceptible to round-off problems. Of course, this integer checksum test should involve integers derived from the floating-point quantities.

Now, since f is a linear function, irrespective of whether the vectors are floating points or integers, the following checksum property also holds:

$$f\left[\sum_{\boldsymbol{v}\in S} mant(\boldsymbol{v})\right] = \sum_{\boldsymbol{v}\in S} f[mant(\boldsymbol{v})] \tag{8}$$

where the $mant(\boldsymbol{v})$'s are integer quantities, as we saw above. Note that Eq. (8) is in general not related to the original floating-point computation $f(\boldsymbol{v})$, and can be used to check it only if f is *mantissa preserving,* that is, $f[mant(\boldsymbol{v})]$ is equal to $mant[f(\boldsymbol{v})]$, $[f(\boldsymbol{v})]$, which is derived from the original computation $f(\boldsymbol{v})$. Then the above equation becomes

$$f\left[\sum_{\boldsymbol{v}\in S} mant(\boldsymbol{v})\right] = \sum_{\boldsymbol{v}\in S} mant[f(\boldsymbol{v})] \tag{9}$$

Thus, if there are errors introduced in the mantissas of the $f(\boldsymbol{v})$'s, then those errors are also present in the $mant[f(\boldsymbol{v})]$'s and these will be detected by the integer checksum test of Eq. (9). Furthermore, this test is not susceptible to round-off. Hence it will not cause any false alarms, and very few computation errors will go undetected vis-a-vis the floating-point test of Eq. (7). In practice, since an integer word can store a finite range of numbers, integer arithmetic is effectively done modulo q where $q - 1$ is the largest integer that can be stored in the computer. Some higher-order bits can be lost in a modulo summation. However, as we will establish shortly, a single error on either side of Eq. (9) will always be detected even in the presence of overflow.

The crucial condition that must be satisfied to apply a mantissa-based integer checksum test on f is that $f[mant(\boldsymbol{v})] = mant[f(\boldsymbol{v})]$. To check if f is mantissa preserving, we have to look at the basic floating-point operations like multiplication, division, addition, subtraction, square-root, etc. that f is composed of, and see if they are mantissa preserving. A binary operator $\odot$ is said to be *mantissa preserving* if $mant(a) \odot$

$mant(b) = mant(a \odot b)$. Let a floating-point number a be represented as $a_1 \times 2^{a_2}$, where a_1 is the mantissa and a_2 the exponent of a. Ignoring the position of the implicit binary point, that is, in terms of just the bit pattern of numbers, floating-point multiplication is mantissa preserving, since

$$mant(a) \cdot mant(b) := a_1 \cdot b_1$$

while

$$mant(a \cdot b) = mant(a_1 \cdot b_1 \times 2^{a_2+b_2}) := a_1 \cdot b_1$$

Note that sometimes the mantissa c_1 of the product $c = a \cdot b$ is "forcibly" normalized by the floating-point hardware when the "natural" mantissa of the resulting product is unnormalized (e.g., $1.100 \times 1.110 = 10.101000$; the product mantissa is unnormalized, and is normalized to 1.010100, assuming 6 bits of precision after the binary point, and the exponent is incremented by 1). In such a case, c_1 is either equal to $(a_1 \cdot b_1) \div 2$ as in the previous example, or is equal to $(a_1 \cdot b_1) \div 2 - 1$ when the unnormalized mantissa has a 1 in its least-significant bit. When normalization is performed, the exponent of c becomes $a_2 + b_2 + 1$. However, this normalization done by the floating-point multiplication unit is easy to detect and reverse in c (a process we call *denormalization*) so that floating-point multiplication is effectively mantissa preserving. Similarly, floating-point division is also mantissa preserving. However, floating-point addition and subtraction are not mantissa preserving.

Thus, if f is composed of only floating-point multiplications and/or divisions, it is mantissa preserving, and we can apply the integer checksum test to it. On the other hand, if f has floating-point additions also, and there is no guarantee that the exponents of all numbers involved are equal, then f is not mantissa preserving. However, all is not lost in such a case, since it might be possible to formulate f as a composition $g \circ h(g \circ h(\boldsymbol{u}) := g[\mathrm{h}(\boldsymbol{u})])$ of two (or more) linear functions g and h, where, without loss of generality h is mantissa preserving, while g is not. In such a case, we can apply an integer checksum test to the h portion of f, that is, after computing $h(\boldsymbol{u})$, and a floating-point checksum test to f, that is, after computing $g[h(\boldsymbol{u})] := f(\boldsymbol{u})$. Since errors in $h(\boldsymbol{u})$ are caught precisely, this will still increase the error coverage and reduce the false alarm rate in checking f vis-a-vis just applying the floating-point checksum test to f. This type of a combined mantissa and floating-point checksum is called a *hybrid* checksum.

Application to Matrix Multiplication. We discuss here the application of integer mantissa checksums to matrix multiplication; the description of this test for LU decomposition can be found in Ref. 12. Matrix multiplication is not mantissa preserving, since it contains floating-point additions. However, we can formulate matrix multiplication as a composition of two functions, one mantissa preserving and the other not, as shown below.

First of all, matrix multiplication can be thought of as a sequence of vector-matrix multiplications, that is,

$$A_{n\times m} \cdot B_{m\times l} := \begin{bmatrix} \boldsymbol{a}_1^{\mathrm{T}} \cdot B \\ \boldsymbol{a}_2^{\mathrm{T}} \cdot B \\ \vdots \\ \boldsymbol{a}_n^{\mathrm{T}} \cdot B \end{bmatrix}$$

where $\boldsymbol{a}_i^{\mathrm{T}}$ is the ith row of A, and $\boldsymbol{a}_i^{\mathrm{T}} \cdot B$ is a vector-matrix multiplication. We have that $f_B(\boldsymbol{a}_i^{\mathrm{T}}) := \boldsymbol{a}_i^{\mathrm{T}} \cdot B$ is a linear function. This property leads to the *floating-point row checksum test* for matrix multiplication. In terms of f_B, the row checksum test is:

$$f_B\left(\sum_{i=1}^{n} \boldsymbol{a}_i^{\mathrm{T}}\right) \stackrel{?}{=} \sum_{i=1}^{n} f_B(\boldsymbol{a}_i^{\mathrm{T}}) \tag{10}$$

Matrix multiplication can also be thought of as a sequence of matrix-vector products $A \cdot B = (A \cdot \boldsymbol{b}_1, A \cdot \boldsymbol{b}_2, \ldots, A \cdot \boldsymbol{b}_l)$. This leads to a similar *column checksum test.*

We define a vector-vector *component-wise* product $\diamond$ for two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ as the vector

$$\boldsymbol{n} \diamond \boldsymbol{v} := (u_1 \cdot v_1, u_2 \cdot v_2, \ldots, u_n \cdot v_n)^T$$

For a matrix $B_{m\times l}$, and an m-vector $\boldsymbol{u}$, we define $\boldsymbol{u}^T \diamond B$ as

$$\boldsymbol{u}^T \diamond B := (\boldsymbol{u}^T \diamond \boldsymbol{b}_1, \boldsymbol{u}^T \diamond \boldsymbol{b}_2, \ldots, \boldsymbol{u}^T \diamond \boldsymbol{b}_l)$$

where $\boldsymbol{b}_i$ denotes the i-th column of B. Thus $\boldsymbol{u}^T \diamond B$ is an $m \times l$ matrix. For example,

$$(5,2)^T \diamond \begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix} := \left((5,2)^T \diamond (2,1)^T, (5,2)^T \diamond (3,4)^T\right) = \begin{pmatrix} 10 & 15 \\ 2 & 8 \end{pmatrix}$$

It is easy to see that h_B defined by $h_B(\boldsymbol{u}) := \boldsymbol{u}^T \diamond B$ is linear and mantissa preserving.

Finally, defining function *rowsum*(C) for a matrix $C = (\mathbf{c}_1, \ldots, \mathbf{c}_m)$ as *rowsum*$(C) := [\vec{+}(\mathbf{c}_1), \ldots, \vec{+}(\mathbf{c}_m)]$ where $\vec{+}(\mathbf{v}) := \sum_{j=1}^{m} v_i$, we obtain the decomposition:

Theorem 2 (12) The vector-matrix product $\boldsymbol{u}^T \cdot B := f_B(\boldsymbol{u}) = rowsum \circ h_B(\boldsymbol{u})$.

Since matrix multiplication $A \cdot B$ is a sequence of $f_B(\boldsymbol{a}_i)$ computations, one for each row of A, we can apply a mantissa-based integer row checksum test to the $h_B(\boldsymbol{a}_i)$ components to precisely check for errors in the floating-point multiplies in $A \cdot B$. This integer row checksum test is:

$$h_{B_{mant}}\left[\sum_{i=1}^{n} mant(\boldsymbol{a}_i)\right] \stackrel{?}{=} \sum_{i=1}^{n} mant[h_B(\boldsymbol{a}_i)] \tag{11}$$

or, in other words,

$$rowsum[mant(A)] \diamond mant(B) \stackrel{?}{=} \sum_{i=1}^{n} mant(\boldsymbol{a}_i^T \diamond B) \tag{12}$$

Note that the RHS of Eq. (12) is obtained almost for free from the floating-point computations $\boldsymbol{a}_i^T \diamond B$ that are computed as part of the entire floating-point vector matrix product $\boldsymbol{a}_i^T \times B$. A similar derivation can be made for an integer column checksum test.

The floating-point additions have to be tested by applying the floating-point checksum tests to $rowsum \circ h_B(\boldsymbol{u}) := f_B(\boldsymbol{u})$,

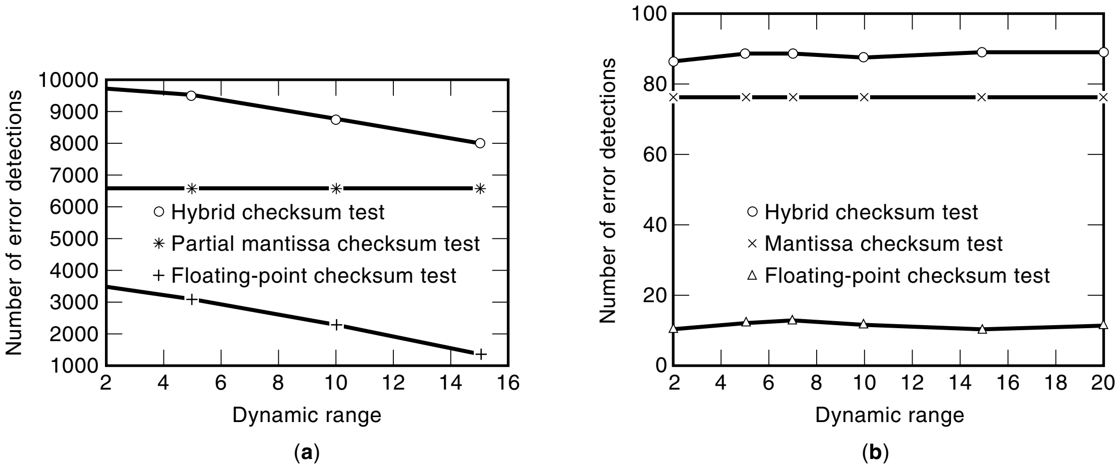

Figure 1. Error coverage vs. dynamic range of data for the mantissa checksum test, a properly thresholded floating-point checksum test, and the hybrid checksum test for (a) matrix multiplication, and (b) **LU** decomposition.

that is, to the final matrix product $A \cdot B$, to give rise to the hybrid test for matrix multiplication.

Error Coverage Results

Analytical Results. Two noteworthy results that have been obtained regarding the error coverage of the mantissa checksum method are given in the two theorems below.

Theorem 3 (12) If either modulo or extended-precision integer arithmetic is used in a mantissa checksum test of the form of Eq. (9) shown again below

$$f\left[\sum_{\boldsymbol{v}\in S} mant(\boldsymbol{v})\right] \stackrel{?}{=} \sum_{\boldsymbol{v}\in S} mant[f(\boldsymbol{v})]$$

then any single-bit error in each scalar component of this test will be detected even in the presence of overflow in modulo (or single-precision) integer arithmetic.

In Eq. (9), we compare scalars a_i and b_i, where $\boldsymbol{a} := (a_1, \ldots, a_n)^T$ and $\boldsymbol{b} := (b_1, \ldots, b_n)^T$ are the LHS and RHS, respectively, of Eq. (9). The above result means that we can detect single-bit errors in either a_i or b_i, for each i, even when single-precision integer arithmetic is used. We also have the following two results regarding the maximum number of arbitrarily distributed errors (i.e., not necessarily restricted to one error per scalar component of the check) that can be detected by the mantissa checksum test.

Theorem 4 (13) The row and column mantissa checksums for matrix multiplication can detect errors in any three elements of the product matrix $C = A \cdot B$ that are due to errors in the floating-point multiplications used to compute these elements.

The mantissa checksum test also implicitly detects errors in the exponents of the floating point products. This is done during the denormalization process by checking if $exp(a) + exp(b) = exp(a \cdot b)$ (this occurs when the floating-point multiplier did not need to normalize the product $a \cdot b$) or if $exp(a) + exp(b) = exp(a \cdot b) + 1$ (this means that a normalization was needed and the mantissa of $a \cdot b$ needs to be denormalized for use in the mantissa checksum test). If neither of these conditions hold, then an error is detected in the exponent of $a \cdot b$.

Empirical Results. A *dynamic range* of x means that the exponents of the input data lie in the interval $[-x, x]$. In Fig. 1 coverage or the number of detection events (for single errors) is plotted against different dynamic ranges of the input data for the following tests.

1. The thresholded floating-point checksum test (with the lower 24 bits masked in the checksum comparison for matrix multiplication, and 12 bits for **LU** decomposition). The threshold of the floating-point checksum test component of the hybrid checksum test was chosen to

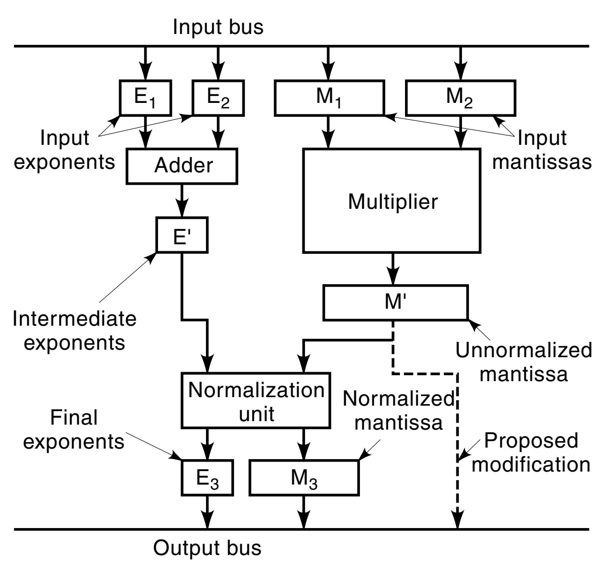

Figure 2. A simple modification of a floating-point multiplier, shown by the dashed line from internal register M' to the output bus, to make the unnormalized mantissa of the product available at no extra time penalty.

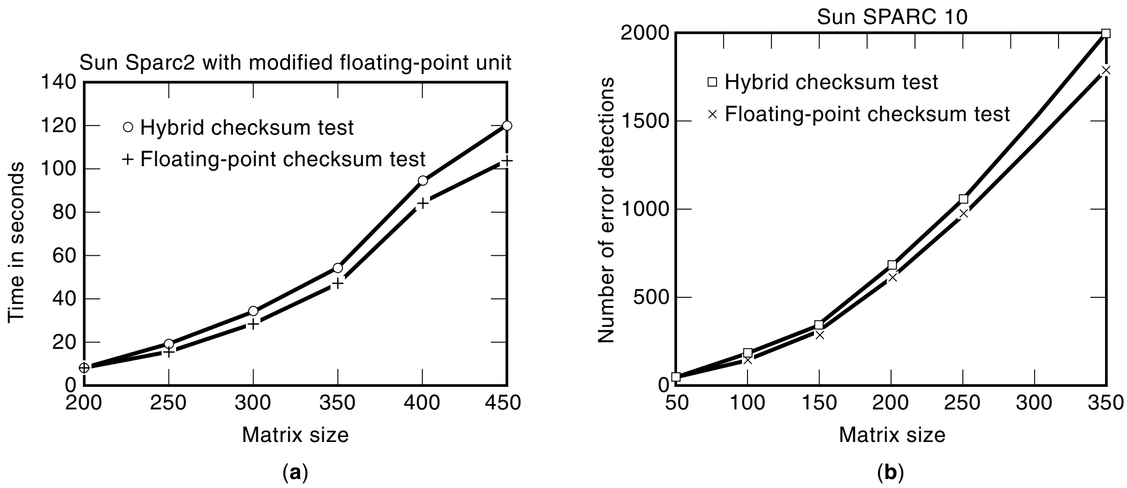

Figure 3. Timing results with a simulated modification of the floating point multiplier for (a) matrix multiplication and (b) **LU** decomposition.

correspond to masking the lower 24 (12) bits, which guarantees almost zero false alarm in matrix multiplication (**LU** decomposition).

2. The mantissa checksum test alone as described above.
3. The hybrid checksum test that uses both the thresholded floating-point test and the mantissa checksum test—an error is detected in the hybrid test if either an error is detected in its mantissa checksum test or in its floating-point checksum test.

The plots clearly show the significant improvements in coverage of the hybrid checksum test with respect to both the mantissa and the floating-point checksum tests. They also show that for the low false alarm case, the mantissa checksum test has a superior coverage compared to the floating-point checksum test. An important point to be noted that is not apparent from the plots of Fig. 1 is that the mantissa checksum test detects 100% of all multiplication errors for both matrix multiplication and **LU** decomposition.

Note that for matrix multiplication, the error coverage of the hybrid test is as high as 97% for a dynamic range of 2, and is 80% for a dynamic range of 15; this is much higher error coverage than the technique of forward error propagation used with the floating point checksum test in Ref. 10. For **LU** decomposition, we obtain error coverage of 90% for a dynamic range of 7 and Roy-Chowdhury and Banerjee (10) report a comparable coverage.

Timing Results. Note that part of the overhead of a mantissa checksum test is extracting the mantissas of the input matrices or vectors and also extracting and denormalizing the mantissas of the intermediate multiplications $a_{i,j} \cdot b_{j,k}$. The latter overhead can be eliminated by a very simple modification to the floating-point multiplication unit that is shown in Fig. 2. With this modification, the unnormalized mantissa is also available (along with the normalized mantissa) as an output of the floating-point multiplier. In many computers, the floating-point product is also available in unnormalized form by using the appropriate multiply instruction—this requires tinkering with the compiler in such a machine in order to use the unnormalized multiply instruction where appropriate. No hardware modification is needed in this case to extract the mantissa for free.

Assuming the above scenarios in which mantissa extraction and denormalization is available without any time penalty, Fig. 3 shows the plots of the times of the fault-tolerant computations that use the hybrid checksum test and that use only the floating-point checksum test. The average overhead of the hybrid checksum for matrix multiplication is 15%, while that for **LU** decomposition is only 9.5%. Thus the significantly higher error coverages yielded by the mantissa checksum test are obtained at only nominal time overheads, which are lower than those of previous techniques (10) developed for addressing the susceptibility of the floating-point checksum test to roundoff.

BIBLIOGRAPHY

1. D. Goldberg, What every computer scientist should know about floating point arithmetic. *ACM Comput. Surveys,* **23** (1): 5–48, 1991.
2. IEEE. *ANSI/IEEE Standard 754-1985 for Binary Floating Point Arithmetic.* IEEE, 1985.
3. IEEE. *ANSI/IEEE Standard 854-1987 for Radix-Independent Floating Point Arithmetic.* IEEE, 1987.
4. D. K. Kahaner, C. Moler, and S. Nash, *Numerical Methods and Software.* Englewood Cliffs, NJ: Prentice-Hall, 1989.
5. M. Heath, *Scientific Computing, An Introductory Survey.* New York: McGraw-Hill, 1997.
6. R. B. Kearfott, Interval computations: Introduction, uses, and resources. *Euromath Bull.,* **2** (1): 95–112, 1996.
7. W. Kahan, The baleful effect of computer languages and benchmarks upon applied mathematics, physics and chemistry. Presented at the SIAM Annual Meeting, 1997.
8. K. H. Huang and J. A. Abraham, Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.,* **C-33** (6): 518–528, 1984.
9. D. L. Boley and F. T. Luk, A well conditioned checksum scheme for algorithmic fault tolerance. *Integration, VLSI J.,* **12**: 21–32, 1991.
10. A. Roy-Chowdhury and P. Banerjee, Tolerance determination for algorithm based checks using simple error analysis techniques.

In *Fault Tolerant Comput. Symp. FTCS-23,* IEEE Press, 1993, pp. 290–298.

11. D. L. Boley et al., Floating point fault tolerance using backward error assertions, *IEEE Trans. Computers,* **44** (2): 302–311, February 1995.
12. S. Dutt and F. Assaad, Mantissa-preserving operations and robust algorithm-based fault tolerance for matrix computations, *IEEE Trans. Comput.,* **45**: 408–424, 1996.
13. F. T. Assaad and S. Dutt, More robust tests in algorithm-based fault-tolerant matrix multiplication, *22nd Fault-Tolerant Comput. Symp.,* July 1992, pp. 430–439,.
14. N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* Philadelphia, PA: SIAM, 1996.
15. G. H. Golub and C. F. Van Loan, *Matrix Computations,* 3rd ed. Baltimore, MD: Johns Hopkins Univ. Press, 1996.
16. P. Banerjee et al., Algorithm-based fault tolerance on a hypercube multiprocessor, *IEEE Trans. Comput.,* **39**: 1132–1145, 1990.
17. J. Y. Jou and J. A. Abraham, Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. *Proc. IEEE,* **74**: 732–741, 1986.
18. F. T. Luk and H. Park, An analysis of algorithm-based fault tolerance. *J. Parallel Distr. Comput.,* **5**: 172–84, 1988.

SHANTANU DUTT
University of Illinois at Chicago
DANIEL BOLEY
University of Minnesota

ROUTH HURWITZ STABILITY CRITERION. See STABILITY THEORY, INCLUDING SATURATION EFFECTS.

ROUTING. See NETWORK ROUTING ALGORITHMS.

RULE-BASED SYSTEMS. See KNOWLEDGE ENGINEERING.