On Error Analysis in Arithmetic with Varying Relative Precision

James W. Demmel

Courant Institute
251 Mercer Str.
New York, NY 10012

Abstract. Recently Clenshaw/Olver and Iri/Matsui proposed new floating point arithmetics which seek to eliminate overflows and underflows from most computations. Their common approach is to redistribute the available numbers to spread out the largest and smallest numbers much more thinly than in standard floating point, thus achieving a larger range at the cost of lower precision at the ends of the range. The goal of these arithmetics is to eliminate much of the effort needed to write code which is reliable despite over/underflow. In this paper we argue that for many codes this eliminated effort will reappear in the error analyses needed to ascertain or guarantee the accuracy of the computed solution. Thus reliability with respect to over/underflow has been traded for reliability with respect to roundoff. We also propose a hardware flag, analogous to the "sticky flags" of the IEEE binary floating point standard, to do some of this extra error analysis automatically.

1. Introduction

Two arithmetics have been independently proposed by Clenshaw and Olver [1] and Iri and Matsui [7] whose common goal is the elimination of overflow and underflow from numerical computation. More precisely, their goal is to eliminate much of the effort required to write software which will work reliably despite overflow and underflow. They both accomplish this by significantly extending the range of numbers that can be represented within a given wordsize. Since for a fixed wordsize only a fixed number of different numbers can be represented, they must necessarily spread their numbers much more thinly at the extremes of the range than in conventional floating point. Thus, as long as one computes with numbers of moderate size, one has approximately the same relative precision as conventional floating point (or perhaps somewhat more), in the sense that the results of each basic operation ($+$, $-$, $\times$, $/$) are about as accurate as with conventional floating point. As the numbers become very large (or very small), the relative precision of each operation gradually decreases until it reaches a lower bound that depends on the arithmetic. (We shall give numerical values for the ranges and accuracies below.)

In this paper we argue that as a result of this decreased relative accuracy at the extremes of the range, either more care must be taken in doing the error analysis of many algorithms than is necessary in conventional floating point, or else the algorithms must scale or otherwise be modified to avoid inaccurate results. This should come as no surprise, since the sizes and accuracies of intermediate results (e.g. pivot growth factors) appear frequently in error analyses. As a result, the effort required to write reliable codes which are relatively impervious to overflow, underflow, *and* roundoff remains the same overall; the new arithmetics just shift that effort away from over/underflow towards roundoff. In brief, the new arithmetics are no shortcuts to writing reliable code. We will present a few examples of codes written using these new arithmetics, standard floating point, and IEEE standard binary floating point [6] in order to support our argument.

Some of this extra effort in roundoff error analysis with the new arithmetics can be sidestepped, however, given the appropriate hardware support. In particular, we propose a status word maintained by the hardware and accessible to the user to keep track of the growth of intermediate results. It would approximately track the largest magnitude of any exponent encountered since last being reset by the user (its actual contents will be different for the Clenshaw/Olver and Iri/Matsui arithmetics). Thus, it acts very much like the "sticky flags" of the IEEE standard [6] which keep track of whether an exception has occurred since last being reset, only it contains a small integer rather than a single bit. We will show how this flag can be used to simplify the analyses in our examples.

The rest of this paper is organized as follows. Section 2 discusses the two new arithmetics in more detail. Section 3 contains examples showing that the effort required to write reliable code is shifted around but conserved by the new arithmetics. Section 4 discusses the proposed status word and shows how it can make error analysis with the new arithmetics easier. Section 5 contains conclusions.

2. Details of the New Arithmetics

First we will discuss the Iri/Matsui arithmetic [7]. The "level 0" version of their arithmetic has three fields, the fraction f (including sign), exponent e and pointer p. The pointer p, which occupies a field with a fixed number of bits, is the number of bits which represent e, the remaining bits representing the fraction f. The value represented by these fields is

$$x = f \cdot 2^e \ ,$$

where f has a leading hidden bit (in the one's position). Thus, for very large or very small numbers, p grows until

enough exponent bits are allocated to represent the exponent e exactly, the remaining bits being used for the fraction. At least one bit (the sign bit) is reserved for f, so for the largest exponents f can only be ± 1. Thus the largest and smallest nonzero numbers in level 0 are all powers of 2.

The "level 1" version of their arithmetic would be used for numbers too large or too small to represent using level 0. In this case the exponent is itself represented as a level 0 number. This may be used recursively to define levels 2, 3 and so on. In their paper Iri and Matsui dealt mainly with level 0 and did not specify the exact format for higher levels, so in this paper we shall do the same. They proposed a 64-bit format with 6 bits allocated for p, which is sufficient to designate from 0 to 57 of the remaining 58 bits as exponent bits. Values of p from 58 to 63 are reserved for NANs (not-a-number symbols), higher levels, and so on. This results in a range from approximately $10^{-10^{16}}$ to $10^{10^{16}}$. Note that the relative error at the extremes of the range, i.e.

$$e_{rel}(x) \equiv \frac{|x - \text{nextafter}(x)|}{|x|}$$

where nextafter(x) is the nearest representable number exceeding x, is 1. This means that the relative error in the basic operations is 100% (when chopping) and 50% (when rounding). The relative error is at least as large for higher level representations. At the center of the range, where the relative precision is largest, $e_{rel}(1.)=2^{-57}$. $e_{rel}(x)$ rises to 2^{-52} (IEEE double precision, see below) at $x=2^{15}=32768$, to 2^{-28} (about half the maximum precision) at $x=2^{2^{28}-1}\approx 7.2\cdot 10^{80807123}$, and to 1 (no relative precision) at $x=2^{2^{56}-1}\approx 10^{10^{16}}$.

Clenshaw and Olver [1] represent a number x as follows:

$$x = \pm(\exp(\exp(\cdots(\exp(f))\cdots)))^{\pm 1}$$

where f is a fixed point fraction between 0 and 1. Also stored is the sign bit , a bit for the exponent $e=\pm 1$ at the right above, and the number of "exp"s minus 1, an integer n. Numbers at least 1 in magnitude are represented by $e=1$ and nonzero numbers at most 1 in magnitude are represented by $e=-1$. f and n occupy fixed fields in the word. As with the Iri/Matsui arithmetic, adjacent representable numbers get relatively farther and farther apart until, if n is large enough, adjacent numbers are so far apart that even their exponents (the i in scientific notation $f\cdot 10^i$) differ by over 100%. In a 64-bit format the authors would allocate 3 bits to n, 1 to the sign, 1 to the exponent e, and the remaining 59 to f. This results in a range from at least

$$10^{-10^{10^{10^{10}}}} \quad \text{to} \quad 10^{10^{10^{10^{10}}}} \tag{1}$$

which is almost inconceivably large. The representation error of such numbers is also enormous, best expressed recursively by saying that in scientific notation, the relative error in the exponent of the exponent of the exponent ... is moderate. The relative error in the middle of the range where it is smallest is $e_{rel}(1.)=2^{-59}$. $e_{rel}(x)$ rises to 2^{-52} (IEEE double precision, see below) at $x\approx 1.73\cdot 10^{13}$, to 2^{-30} (about half the maximum precision) at $x\approx 4.9\cdot 10^{5012916}$, and to 1 (no relative precision) at $x\approx 10^{10^{15}}$

By way of contrast the range of the 64-bit double precision IEEE binary format is about $2^{\pm 1023}\approx 10^{\pm 308}$ with an almost constant relative precision for normalized numbers of $e_{rel}(x)=2^{-52}$. Thus both new arithmetics have greater relative precision than IEEE double precision at the center of their ranges and much less at the extremes.

As for eliminating over/underflow in these two arithmetics, care must be taken in stating the assumptions. The Iri/Matsui arithmetic can clearly overflow or underflow but it is very unlikely. The Clenshaw/Olver arithmetic, on the other hand, is impervious to over/underflow by the basic operations $+$, $-$, $\times$, and $/$ [2], although it can still overflow by repeated exponentiation. This is because for the largest and smallest numbers x in the format ($n=6$ or 7, most numbers for $n=5$), the nearest representable number to $2*x$ or x^2 is x. Of course, this means that $2*x-x$ and x^2/x differ greatly from x for very many different x in this arithmetic, which might make anyone wanting to write reliable code in this arithmetic dubious about applying even simple algebraic identities in attempting to prove anything.

3. Examples of "Conservation of Effort"

In this section we present four examples to show how the effort required to write reliable code is shifted by these new arithmetics away from avoiding over/underflow to bounding or limiting the roundoff error. If this shifted effort is not undertaken, the resulting code will be unreliable because of roundoff errors rather than over/underflow as with conventional floating point.

To prove my point it is not necessary that *every* code exhibit this shift of effort, merely that many common codes do. For even if certain problems turn out to be solvable more easily using the new arithmetics, this must be weighed against the effort it takes to rewrite many old, reliable codes which stop working with the new arithmetics.

The examples are extended products, rational function evaluation, computing binomial distributions, and Gaussian elimination.

Consider the extended product $p=\prod_{i=1}^{n} p_i$. We consider computing it by the simplest possible program

```
p := 1
for i=1 to n do p := p*p_i
```

Consider first how this algorithm behaves in the new arithmetics. If an intermediate p is very large or very small, it will be computed to low relative accuracy: $p_{true}=p_{computed}(1+\epsilon)$ where ϵ is large. In Iri/Matsui's arithmetic ϵ can be as large as .5 in rounded arithmetic, and in the Clenshaw/Olver arithmetic it can be as large as a tower of several 10's as in (1). Later $p_{computed}$ inherit this relative error. Thus, the final p may be a reasonable looking but utterly wrong number without a warning having been given. Clenshaw/Olver's arithmetic cannot possibly yield over/underflow, whereas Iri/Matsui's can, although it is much less likely than with conventional floating point.

In conventional floating point, it is well known [9] that in the absence of over/underflow the final computed p can differ from the true product by a factor of $1+\epsilon$ where

ϵ is at most about $n \cdot macheps$, where *macheps* is the machine precision and n is assumed to be not too large. If over/underflow occurs, the programmer will become aware of this typically by getting an overflow interrupt or else computing $p=0$ when no $p_j=0$. In IEEE arithmetic, one might get an overflow or underflow interrupt, or if these are disabled, an underflow flag or a final result of $\pm\infty$ or *NAN*. In fact, when the underflow trap is disabled (the default case), the underflow flag will be turned on only if an intermediate p *underflows inaccurately*, i.e. is less than the underflow threshold (is denormalized) *and* differs from the true result by more than the usual .5 units in the last place. Thus the underflow flag will be on only if underflow could actually have made the final result inaccurate [4,6]. (The standard also permits raising the underflow flag if the underflowed result is merely inexact; this raises the flag somewhat more often). In any event, it is easy to tell when the possibility of a large relative error in the final result exists.

Therefore, the simple code above is unreliable because of roundoff with the Clenshaw/Olver arithmetic, over/underflow with conventional and IEEE floating point, and both roundoff and over/underflow with Iri/Matsui. In order to make it reliable, the following changes are necessary. By reliability, we mean prevention of any interrupts due to over/underflow, and either a reliable warning that the result is inaccurate or a guarantee of accuracy (assuming a reasonable bound on n).

For Iri/Matsui and Clenshaw/Olver the final error will clearly be bounded by n times the error in computing the largest intermediate p. Thus, if a reliable final error bound is desired, an "if" statement must be inserted into the program loop to keep track of the largest intermediate p (an alternative to this is discussed in section 4 below).

To prevent over/underflow using either Iri/Matsui or floating point, a similar "if" statement must be inserted to make sure the next product is neither too large nor too small. To continue computing in such a case there are several possibilities:

(1) scale p and retain the scale factor (this is used in computing determinants in [5])

(2) use a different format with a bigger exponent (level 1 for Iri/Matsui)

(3) sum the logarithms of the factors and exponentiate the sum if it is not too large or too small.

To get an accurate answer with Clenshaw/Olver arithmetic when intermediate results get too large or too small, the same techniques can be used. Thus we see that no effort has been saved in writing a reliable program to compute extended products.

It is interesting to see that for Iri/Matsui and Clenshaw/Olver arithmetic the error analysis for products is analogous to the error analysis of sums in conventional floating point. Products are computed most accurately when all factors are greater than 1 (or all less than 1), just as sums are accurate when all terms are positive (or all are negative). Inaccuracy may arise from cancellation, either from factors greater than 1 and less than 1, or positive and negative terms. This is because Iri/Matsui and Clenshaw/Olver are in some sense summing logarithms of the factors, some of which may be positive and others negative.

The second example is rational function evaluation. Let $r(x)=p(x)/q(x)$ be a rational function, with $p(x)=\sum_{i=0}^{n} p_i x^i$ and $q(x)=\sum_{i=0}^{n} q_i x^i$. Let our algorithm for evaluating r be simply

compute $p(x)$ by Horner's rule
compute $q(x)$ by Horner's rule
compute the quotient of p and q

In conventional floating point it is well known [9] that in the absence of over/underflow, this algorithm is backwards stable in the sense that it will compute r accurately for slightly different $\hat{p}$ and $\hat{q}$, i.e. $\hat{p}(x) = \sum_{i=0}^{n} \hat{p}_i x^i$ where $\hat{p}_i=p_i(1+\epsilon_i)$, $|\epsilon_i| \le 2n \cdot macheps$ for moderate n. Analogous equations hold for $\hat{q}(x)$. The absolute error (again in the absence of over/underflow) in the computed $p(x)$ is thus bounded by $macheps \cdot 2n \cdot \sum_{i=0}^{n} |p_i x^i|$. If, for example, $x>0$ and $p_i>0$, this means $p(x)$ will be computed with a relative error at most $2n \cdot macheps$. If $q_i>0$ as well, $r(x)$ will be computed with a relative error of at most $(4n+1)macheps$, again provided n is moderate and over/underflow does not occur.

What must be done to make this code reliable in case over/underflow occurs? As in the last example, overflow will be indicated by an interrupt, an overflow flag, an ∞ symbol, or a NAN appearing in the answer. If this occurs the intermediate value and remaining coefficients can be scaled down and the scale factor retained. Since the scale factor for p may well cancel with the scale factor for q, the scale factors can be put back in at the end. Alternately, if x is very large, one could rewrite $r(x)$ as

$$r(x) = \frac{\sum_{i=0}^{n} p_i x^i}{\sum_{i=0}^{n} q_i x^i} = \frac{\sum_{i=0}^{n} p_{n-i} z^i}{\sum_{i=0}^{n} q_{n-i} z^i}$$

where $z=1/x$. Underflow is harder to detect, since it usually causes no problem and hence raises no flag. Nonetheless, if all the p_i and q_i are near the underflow threshold, $p(x)$ and $q(x)$ may both be computed inaccurately, and their quotient r may be of moderate size and completely wrong. In [4] it is shown that it is sufficient to scale the coefficients to keep them a short distance above the underflow threshold in conventional arithmetic in order to retain the results of the previous error analysis. In IEEE arithmetic, gradual underflow means that we need only keep the coefficients above the underflow threshold itself.

To guard against absolutely all possibility of over/underflow using these schemes, however, is quite tedious and requires much care. The use of a wider conventional floating point format with more range and precision makes problems more unlikely but does not eliminate them completely. A practical "reliable code" might well settle for eliminating most problems of over/underflow rather than all.

Is it easier to write this program using Iri/Matsui or Clenshaw/Olver arithmetic? As with the example of extended products, relative precision will decay if the com-

puted values of p or q are very large or very small. If p and q are inaccurate but of comparable magnitudes, $r=p/q$ may be inaccurate and of moderate size. Therefore one must track the growth of intermediate terms in p and q to be able to provide a guaranteed error bound on the output. Alternately, to guarantee high accuracy in the output, one must scale the intermediate results to keep them near the center of the range where relative accuracy is high. This is about as much programming effort as it would take to make the conventional floating point algorithm reliable. The Iri/Matsui arithmetic must be guarded against over/underflow as well.

The next example is taken from [2]. The problem is to compute the probability distribution function

$$I(n,k,p) = \sum_{j=0}^{k} \binom{n}{j} p^j (1-p)^{n-j}$$

with $0 \le k \le n$ and $0 < p < 1$. Since the function is a probability distribution, all the terms in the sum are nonnegative and have a sum bounded by 1. Therefore the major worry arises in computing the individual terms, which we call y_j as in [2].

The y_js are given recursively by

$$y_j = \frac{p(n-j+1)}{(1-p)j}\, y_{j-1} \ , \ y_0=(1-p)^n \ , \ I(n,k,p)=\sum_{j=0}^{k} y_j$$

If p is close to 1 and n is large, y_0 will be very tiny, and the y_j will increase rapidly. In conventional floating point, as long as y_0 does not underflow, all later y_j will be computed with high relative accuracy. If y_0 underflows to zero, the y_j must be computed another way. (As before, the test (y_0 *.eq.* 0) or an underflow flag in IEEE arithmetic will indicate whether underflow is a problem.) With the new arithmetics, if y_0 is so tiny that it is inaccurate, this inaccuracy will be propagated forward into the larger y_j which dominate the sum just as in the first example above. Therefore, the program must check to make sure y_0 is not so small as to contaminate the larger y_j, or else the program may compute moderate sized and incorrect $I(n,k,p)$ without any warning.

So what should be done if y_0 is too small? If $y_n = p^n >> y_0$ and k is large one could compute the y_js using a recurrence from the other end. Or one could scale so y_0 lay in a moderate range. Or, in conventional floating point, one could compute $\log y_j$ recursively instead and exponentiate to get y_j.

In any event, it seems that it takes about as much work to write a reliable program to compute $I(n,k,p)$ in the new arithmetics as in conventional floating point.

The final example is Gaussian elimination with partial pivoting. The usual error analysis [9] shows that when trying to solve $Ax=b$ one computes an approximate solution $\hat{x}$ which satisfies $(A+\delta A)\hat{x}=b+\delta b$, where $||\delta A|| \,/\, ||A||$ and $||\delta b|| \,/\, ||b||$ are small, on the order of *macheps* times the cube of the dimension n^3 times a pivot growth factor g. g measures how much larger the intermediate values in the computation were than $||A||$, the original data. In conventional arithmetic, g can be as large at 2^{n-1}, although it almost never attains this bound. These bounds on $||\delta A|| \,/\, ||A||$ and $||\delta b|| \,/\, ||b||$ are usually multiplied by the condition number $||A|| \cdot ||A^{-1}||$ to get a bound on the relative error $||x-\hat{x}|| \,/\, ||x||$.

What can go wrong with Gaussian elimination in conventional floating point? If an intermediate result overflows, one gets either an interrupt or ∞ symbols or NANs. An error analysis taking underflow into account is given in [3,4]. It is shown there that as long the initial data is large enough, underflow will not make the usual error bounds [9] significantly larger. In conventional floating point, $||A||$, $||b||$ and $||b|| \,/\, ||A||$ (if $\hat{x}$ itself underflows) must exceed the underflow threshold divided by *macheps*, whereas with IEEE arithmetic and gradual underflow, they need only exceed the underflow threshold itself (be normalized). It is not hard to achieve these constraints by scaling. It may be necessary to rescale in the middle of the algorithm.

In the new arithmetics the same problem as before arises, namely that if intermediate results are very large, their inherent large relative error can propagate to the output. Preventing this requires monitoring element growth during the algorithm and scaling if necessary. This is as much work as with conventional floating point.

4. Hardware Support for Error Analysis with the New Arithmetics

How likely are the problems discussed in the previous section to occur? For lack of broad experience with the new arithmetics, one can only say that the problems with roundoff which were raised seem quite rare, since one still has half precision (28 to 30 bits) out at $10^{\pm 80807123}$ for Iri/Matsui and at $10^{\pm 5012916}$ for Clenshaw/Olver. Such gargantuan (or minuscule) numbers are unlikely to be input directly, but could result from some (probably erroneous) prior computation. Over/underflow in conventional arithmetic is also unlikely, although frequent enough for certain kinds of problems when the range is only $10^{\pm 38}$ (as it is in many single precision formats including IEEE) for Clenshaw/Olver and Iri/Matsui to have attempted to avoid it. It should be much rarer with a wider exponent range as in IEEE double, although we know of no statistics to corroborate this. It is sufficiently rare that most codes will do no preventive tests or scaling, settling for an interrupt when it does occur. This is because for most codes the development and lifetime costs of unnecessary preventive tests and scaling far outweigh the costs of occasional runs terminating unexpectedly. Therefore any precautions the arithmetic (or language or operating system) offer to prevent these rare problems should be very cheap, or they will be avoided by all but the most demanding or paranoid programmers.

In this spirit we propose a status word S maintained by the arithmetic unit which keeps track of the highest "level" the arithmetic has so far reached. "Level" means the pointer p which measures the width of the exponent field in Iri/Matsui arithmetic and the index n which says how high the tower of exponentials is in Clenshaw/Olver arithmetic. After each operation the hardware would replace S by the maximum of S and the level of the result of the current operation. The user would be able to read S and also reset it to 0 in order to track the maximum level of any sequence of operations. Thus, S has semantics similar to the "sticky flags" in IEEE arithmetic which are

set to 1 whenever an exception occurs and are only resettable by the user (there are separate sticky flags for underflow, overflow, division by zero, invalid operation, and inexact in IEEE arithmetic).

S would provide a measure of the worst relative error in any operation since it was last reset. In Iri/Matsui arithmetic this worst relative error is given by 2^{S-57}, corresponding to a result between $2^{2^{S-1}-1}$ and $2^{2^{S-1}}$. In Clenshaw/Olver arithmetic it is given in the following table (for the 64-bit format described above):

S	$\max(\lvert x\rvert, \lvert x^{-1}\rvert)$	minimum relative error	maximum relative error
0	2.7	$1.7\cdot10^{-18}$	$1.7\cdot10^{-18}$
1	15.	$1.7\cdot10^{-18}$	$4.7\cdot10^{-18}$
2	$3.8\cdot10^{6}$	$4.7\cdot10^{-18}$	$7.1\cdot10^{-17}$
3	$2.3\cdot10^{1656520}$	$7.1\cdot10^{-17}$	$2.7\cdot10^{-10}$
4	$10^{10^{1656520}}$	$2.7\cdot10^{-10}$	$10^{10^{1656520}}$
5-7	–	$10^{10^{1656520}}$	–

Clearly, S is a much coarser measure of relative error for Clenshaw/Olver arithmetic than Iri/Matsui arithmetic. Only if $S\leq3$ is any relative precision guaranteed, but since this includes numbers as large as $2.3\cdot10^{1656520}$ (or as small as the reciprocal of this), larger values of S are very unlikely to arise. For $S\leq2$, almost maximum precision is available, and this includes numbers up to $3.8\cdot10^{6}$ (and their reciprocals). By choosing a base other than e for Clenshaw/Olver arithmetic, one might be able to extract more information from S, perhaps even eliminating some of the very large and apparently useless numbers at the top of their range.

It is easy to see how S would be used in the examples above. For extended products in Clenshaw/Olver arithmetic, for example, one could write

```
S := 0
p := 1
for i = 1 to n do p := p*p_i
if (S≤3) then
    /* enough accuracy, continue computing */
else
    /* scaling needed, retry */
end if
```

Since underflow can often be shown to be less harmful than overflow, one could have two status words S_1 and S_2, S_1 measuring the maximum level of numbers less than 1 in magnitude and S_2 measuring the maximum level of numbers greater than 1 in magnitude [8]. By only testing S_2 one might avoid false alarms caused by harmless underflows.

5. Conclusions

We have demonstrated that if one wants to write truly iron-clad code which delivers guaranteed high accuracy or even just guaranteed error bounds despite roundoff, overflow and underflow, then for many codes its takes about as much work using the new arithmetics proposed by Iri/Matsui and Clenshaw/Olver as with conventional floating point. It is very difficult to give likelihoods to the different problems that can arise (overflow, underflow, reduced relative precision due to extreme operands), other than to say that they are sufficiently unlikely as to almost certainly be ignored by all but the most demanding or paranoid programmers. Some of the work of monitoring growth of operands in the new arithmetics can be done automatically by a hardware maintained status word which keeps track of the largest (or smallest) result computed since last being reset. It plays a very similar role in the proposed new arithmetics as the sticky flags for exceptions do in IEEE arithmetic.

If a wider conventional floating point format with more range were available (as in IEEE arithmetic), many of the problems with over/underflow in this paper could most easily be avoided by rerunning the program with all or some variables redeclared to be of the wider format. This cannot guarantee that over/underflow cannot occur, and may be infeasible for storage reasons. In that case tedious programming with a profusion of scaling tests may be hard to avoid no matter which arithmetic is used.

6. References

[1] C. W. Clenshaw and F. W. J. Olver, *Beyond Floating Point*, JACM, 31 (1984) pp. 319-328

[2] C. W. Clenshaw and F. W. J. Olver, *A Closed Computer Arithmetic*, these proceedings

[3] J. Demmel, *Effects of Underflow on Solving Linear Systems*, Fifth Symposium on Computer Arithmetic, Ann Arbor, MI, May 18-19, 1981

[4] J. Demmel, *Underflow and the Reliability of Numerical Software*, Siam J. Sci. Stat. Comp., Vol. 5, No. 4, Dec 1984

[5] J. Dongarra et al., *LINPACK Users' Guide*, SIAM, Philadelphia, 1979

[6] IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronic Engineers, New York, 1985

[7] M. Iri and S. Matsui, *An Overflow/Underflow-Free Floating Point Representation of Numbers*, J. of Information Processing, Vol. 4, No. 3, pp. 123-133, Nov. 1981

[8] W. Kahan, private communication, 1986

[9] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs N.J., 1963