13 Letters to the IEEE Computer Arithmetic Standards Revision Group

Ulrich Kulisch
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe

Abstract

The IEEE standards 754 and 854 for floating-point arithmetic have been under revision for some time. The work is pushed forward by the Floating-Point Working Group (ca. 100 scientists) of the Standards Committee of the IEEE Computer Society. The author has tried to influence the development of the new standard by several letters he has sent to the working group. The text of these letters follows.

Key words: computer arithmetic, floating-point arithmetic, interval arithmetic, elementary functions, arithmetic standards.

May 2005

Dear colleagues,

recently I have been asked by several colleagues to comment on the ongoing IEEE 754 revision work. So I am trying today to comment a little on interval arithmetic. Perhaps later I will send you comments on other topics.

The IEEE standard, adopted in 1985, seems to support interval arithmetic. It requires the basic four arithmetic operations with rounding to nearest, towards zero, and with rounding downwards and upwards. The latter two are needed for interval arithmetic. But almost all processors that provide IEEE arithmetic separate the rounding from the operation, which proves to be a severe drawback. In a conventional floating-point computation this does not cause any difficulties. The rounding mode is set only once. Then a large number of operations is performed with this rounding mode each one in a single cycle. However, when interval arithmetic is performed the rounding mode has to be switched very frequently. The lower bound of the result of every interval operation has to be rounded downwards and the upper bound rounded upwards. Thus, the rounding mode has to be reset for every arithmetic operation. If setting the rounding mode and the arithmetic operation are equally fast this slows down the computation of each bound unnecessarily by a factor of two in comparison to conventional floating-point arithmetic. On almost all existing commercial processors, however, setting the rounding mode takes a multiple (three, five, ten) of the time that is needed for the arithmetic operation. Thus an interval operation is unnecessarily at least eight (or twenty and even more) times slower than the corresponding floating-point operation not counting the necessary case distinctions for interval multiplication and interval division. This actually kills interval arithmetic. The rounding should be an integral part of the arithmetic operation. Every one of the rounded arithmetic operations with rounding to nearest, downwards or upwards should be equally fast and executed in a single cycle. (Of course I know to code around all that. But I do not need a standard for that).

Interval arithmetic is an essential extension of floating-point arithmetic. It will only be widely accepted, used, and understood if it is equally fast. I append a paper which shows how interval arithmetic should be implemented on computers. The IEEE 754 standard should more precisely specify how interval arithmetic should be supported on computers.

I also append the title page of a new book which in the chapter "Interval Arithmetic Revisited" gives more (easily readable) information on the subject.

I wish you all much success with the IEEE 754 revision work. Progress in scientific computing depends critical on that standard.

With best regards

Ulrich Kulisch

Kulisch, U.: *Advanced Arithmetic for the Digital Computer - Design of Arithmetic Units.* Springer-Verlag 2002.

Kirchner, R. and Kulisch, U.: *Hardware Support for Interval Arithmetic.* Reliable Computing, January 2006.

June 2005

Dear colleague,

In early May I sent you a letter concerning interval arithmetic and the IEEE 754 revision work. I thank everybody who sent me comments. Apart from some misunderstandings about my meaning, all comments have been positive.

Today I would like to comment on another topic. It is related to interval arithmetic and is equally directed at increasing the speed of computation and the reliability of computed results.

Undoubtedly for elementary floating-point computation the IEEE standard arithmetic is thorough, consistent, and well defined. It has been widely accepted and can be found in virtually every processor developed since 1985. This has greatly improved the portability of floating-point programs.

However, computer technology has been dramatically improved since 1985. Arithmetic speed has gone from megaflops to gigaflops to teraflops, and it is already approaching the petaflops range. This is not just a gain in speed. A qualitative difference goes with it. At the time of the megaflops computer a conventional error analysis was recommended in every numerical analysis textbook. Today the PC is a gigaflops computer. For the teraflops computer conventional error analysis is no longer possible. An avalanche of numbers is produced when a teraflops computer runs for a few hours. If these numbers were to be printed they would need a pile of paper that reaches from the earth to the sun. Computing indeed has already reached astronomical dimensions!

This brings to the fore the question of whether the computed result really solves the given problem. The only way to answer this question is by the computer itself. Mathematical methods that provide an answer to this question are available for very many problems. Computers, however, are not at present built in a way that allows these methods to be used effectively.

During the last several decades of numerical analysis, methods for a large variety of problems have been developed which allow the computer itself to validate or verify its computed results. These methods compute unconditional bounds for a solution and can iteratively improve the accuracy of the computed result. Hardly any of these methods needs higher precision floating-point arithmetic. Double precision floating-point arithmetic is the basic arithmetical tool.

Two additional arithmetical features are fundamental and necessary:
1. **fast interval arithmetic** and
2. a **fast and accurate** *multiply and accumulate* **instruction**.[1]

How fast interval arithmetic can be implemented was described in my earlier letter. To obtain **close** bounds for the solution interval arithmetic has to be combined with defect correction or iterative refinement techniques. To be effective these techniques require an accurate *multiply and accumulate* instruction or, what is equivalent to this, an accurate scalar product. It is realized by accumulating products of the full double length into a wide fixed-point register. This fixed-point accumulation is completely free of truncation error. Fast hardware circuitry for an accurate *multiply and accumulate* instruction for all kinds of computers is discussed in the first chapter of my new book: *Advanced Arithmetic for the Digital Computer - Design of Arithmetic Units.* See the Appendix.

A very natural pipelining of the *multiply and accumulate* instruction leads to very fast and simple circuits. The hardware expenditure for it is comparable to that for a fast multiplier with an adder tree, accepted years ago. A speed increase by a factor of four, compared to a possibly wrong accumulation of the products in conventional floating-point arithmetic, is easily achieved.

With the fast and accurate *multiply and accumulate* instruction, fast quadruple or multiple precision arithmetic also easily can be provided. A multiple precision number is represented as an array of floating-point numbers. The value of this number is the sum of its components. It can be represented in the wide fixed-point register. Addition and subtraction of multiple precision variables or numbers can easily be performed in this register. Multiplication of two such numbers is simply a sum of products of floating-point numbers. It can be computed by means of the accurate *multiply and accumulate* instruction which is very fast. For instance in case of a fourfold precision the product of two such numbers $a = (a_1 + a_2 + a_3 + a_4)$ and $b = (b_1 + b_2 + b_3 + b_4)$ is obtained by

$$\begin{aligned} a \times b &= (a_1 + a_2 + a_3 + a_4) \times (b_1 + b_2 + b_3 + b_4) \\ &= a_1b_1 + a_1b_2 + a_1b_3 + a_1b_4 + a_2b_1 + \cdots + a_4b_3 + a_4b_4 \\ &= \sum_{i=1}^{4} \sum_{j=1}^{4} a_i b_j. \end{aligned}$$

By using the accurate *multiply and accumulate* instruction the result is inde-

[1]To achieve high speed all conventional vector processors provide a 'multiply and accumulate' instruction. It is, however, not accurate. By pipelining, the accumulation (continued summation) is executed very swiftly. The accumulation is done in floating-point arithmetic. The pipeline usually has four or five stages. What comes out of the pipeline is fed back to become the second input into the pipeline. Thus four or five sums are built up and are finally added together. This so-called partial sum technique alters the sequence of the summands and causes errors in addition to the usual floating-point errors. A vectorizing compiler uses this 'multiply and accumulate' operation within a user's program as often as possible, since it greatly speeds up the execution. Thus the user loses complete control of his computation.

pendent of the sequence in which the summands are added.

In Summary: Interval arithmetic can bring guarantees into computation while an accurate *multiply and accumulate* instruction can bring high accuracy via defect correction methods and at high speed. It also is the key operation for fast multiple precision arithmetic.

Fast and accurate hardware support for 1. and 2. must be added to conventional floating-point arithmetic. Both are necessary extensions. Instead of the computer being merely a fast calculating tool they would turn it into a scientific instrument of mathematics. Computing that is continually and greatly speeded up makes this step necessary and it is that very speed that calls conventional computing into question.

Of course, often the computer would have to do more work to obtain verified results. But the mathematical safety should be worth it. The step from assembler to higher programming languages or the use of convenient operating systems also consumes a lot of computing power and nobody complains about it. Fast computers in particular are often used for safety critical applications. Severe, expensive, and tragic accidents can occur if the eigenfrequencies of a heavy electricity generator, for instance, are erroneously computed, or if a nuclear explosion is incorrectly simulated.

The IEEE standards 754 and 854 have been under revision for some time. I hope that the revision does not get bogged down in details but that it will also consider the more basic features of high speed scientific computing. The computer should not be just a glorified calculator for another twenty years.

Again, I wish you all much success with the IEEE 754 revision work. A major breakthrough is necessary for scientific computing.

With best regards

Ulrich Kulisch

August 8, 2005

Dear Colleague,

It really is not easy to follow a discussion from far away. So I am not sure whether I correctly understand everything what I read about 'Static Mode Declarations'. Nevertheless I do have a comment about the rounding mode.

I think the old way (of IEEE 754), first to set the rounding mode and then to call the arithmetic operation, is well established and I have no complaint about it nor did my earlier letter request its elimination.

However, for interval arithmetic we need to be able to call each of the operations $+>$, $->$, $*>$, $/>$, and $+<$, $-<$, $*<$, $/<$ (here $>$ means rounding upwards and $<$ means rounding downwards) as one single instruction. (I simply call these operations 'my operations'). This requires new operation codes! These operations are distinct from the other approach where first the rounding mode has to be set and then the arithmetic operation is called.

I see a need for the older mechanism as well, in particular for those applications where the rounding mode is to be selected randomly. I think in this case the status register should provide a new mode for choosing the rounding mode randomly (by a hardware random number generator). Then all standard roundings would be performed with roundings that change at random. This would not have any effect on 'my operations' since they are called by other operation codes.

I would appreciate it very much if, with a new arithmetic standard, denotations for arithmetic operations with different roundings would be introduced. They could be: $+, -, *, /$ for operations with rounding to the nearest floating-point number, $+>$, $->$, $*>$, $/>$ for operations with rounding upwards, $+<$, $-<$, $*<$, $/<$ for operations with rounding downwards, and $+|, -|, *|, /|$ for operations with rounding towards zero (chopping).

We would have to convince the language designers that this is much simpler and leads to much more easily readable expressions than for instance the use of operators like .addup. or .adddown. in Fortran. In languages like C++ which just provide operator overloading and do not allow user defined operators these operations would be called as functions or via assembler. Our C-XSC, for instance, does not provide operations with directed roundings. They are hidden within the interval operations.

With best regards

Ulrich Kulisch

October 2005

HIGH SPEED COMPUTING WITH HIGH ACCURACY

Dear colleague: In my second letter to the IEEE 754 revision group (in June 2005) I proposed an *accurate multiply and accumulate operation* for inclusion in the revised IEEE arithmetic standard. Some of my colleagues have since urged me to expand on the significance of this operation in a further mail. My comments are the following:

A pipelined *multiply and accumulate operation* is the key to obtaining high speed on all existing vector processors. However, the way this operation has been implemented causes errors beyond those of conventional floating-point. I append the *GAMM-IMACS Resolution* and the *GAMM-IMACS Proposal* which address these errors.

The proposed *accurate multiply and accumulate operation*, or equivalently an *accurate scalar product*, comes with the same gain in speed (multiplication and accumulation are performed in one pipeline). In addition it delivers a fully accurate result and it is the key operation for high speed multiple precision arithmetic and multiple precision interval arithmetic. These latter two in turn are the key operations for controlling the accuracy in a computation.

As far as I understand the drafts of the IEEE revision work, quadruple precision arithmetic will be part of the next IEEE arithmetic standard. Of course the entire computing community would be grateful for this. But this should not be the only way to higher accuracy. If the attainable accuracy in a particular class of problems is insufficient with double precision, then it will very often be insufficient with quadruple precision as well. I think it is necessary, therefore, also to provide a basis for improving the accuracy rather than simply providing higher precision. Hardware implementation of a full quadruple precision arithmetic is much more costly than implementation of the accurate scalar product. The latter only requires fixed-point accumulation of double precision products. Fast multiple precision arithmetic and multiple precision interval arithmetic can easily be provided as a by-product.

The *accurate multiply and accumulate operation* is realized by accumulating the products of the full double length into a wide fixed-point register. Fixed-point accumulation is simple, error free, and fast. No under- or overflow can occur during a scalar product computation, if the width of this register is appropriately chosen. There is a specific technique for absorbing a possible carry before it appears (see the first chapter of my book on 'Advanced Computer Arithmetic'). Of course it is most convenient to supply enough local memory on the arithmetic unit for all scalar products of double precision data to be computed to full accuracy. We did this on our chip. However if this much memory or register space is not available, compromises are quite possible. See the section 'Hardware Accumulation Window' in my book. For example in the case of the data format 'long' of the /370 architecture all scalar products that do not cause an over- or underflow can be computed to full accuracy in a register space of 624 bits. If this register space is available on the arithmetic unit the vast majority of scalar products could be computed to full accuracy on very fast hardware. In case of an over- or underflow, an underlying software routine (exception handling) would take over. Thus in every case a fully accurate result is computed.

There are many good reasons to urge very strongly that the *accurate multiply and accumulate operation* be included in a future IEEE arithmetic standard. For obtaining high accuracy it is the ultimate solution. A hardware implementation of the accurate scalar product, by the way, brings a considerable speed up compared to a conventional computation of the scalar product in floating-point, while any software simulation will be considerably slower than the latter. This is also the case if quadruple precision and/or computing the second part of an operation and/or other aids like multiply and add fused are available. Computer arithmetic without the *accurate multiply and accumulate operation* is incomplete and unnecessarily slow.

With best regards

Ulrich Kulisch

IMACS-GAMM Resolution on Computer Arithmetic.
In: Mathematics and Computers in Simulation 31, 297-298, 1989.
In: Zeitschrift für Angewandte Mathematik und Mechanik 70, no. 4, T5, 1990.

GAMM-IMACS Proposal for Accurate Floating-Point Vector Arithmetic.
In: GAMM, Rundbrief 2, 9-16, 1993.
In: Mathematics and Computers in Simulation, Vol. 35, IMACS, North Holland, 1993.
In: News of IMACS, Vol. 35, No. 4, 375-382, Oct. 1993.

January 28, 2006

To: Nelson H. F. Beebe and to stds-754@IEEE.ORG

Dear colleague, in your mail of Nov. 22, 2005 to stds-754@IEEE.ORG and to me you refer to the *GAMM-IMACS Proposal for Accurate Floating-Point Vector Arithmetic*. Methods that realize the proposal are developed in my book [KUL]. In the same mail you mention the paper [ORO], and in another mail of Jan. 4, 2006 the paper [ROO]. I also appreciate these excellent papers. All these papers [ORO], [ROO], and [KUL] claim to provide fast methods that compute accurate sums and dot products. The question is: fast in comparison with what?

[ORO] and [ROO] just use conventional floating-point arithmetic. The methods are fast compared with other software methods that compute highly accurate sums and dot products. The computing time comes surprisingly close to the time T needed for (a possibly wrong) computation in conventional floating-point arithmetic. The speed mildly decreases with increasing condition number.

[KUL] suggests hardware solutions. The methods are fast compared with other hardware solutions. The computing time is independent of the condition number. It is less than T. This high speed is reached because the actions: *loading the data, computing, shifting, and accumulation of the products* are performed in one pipeline. Furthermore fixed-point accumulation of the products is simpler than accumulation in floating-point arithmetic. Many intermediate steps that are executed in a floating-point accumulation such as normalization and rounding of the products and of the intermediate sum, composition into a floating-point number and decomposition into mantissa and exponent for the next operation do not occur in the fixed-point accumulation. It simply is performed by a shift and addition of the products of full double length into a wide fixed-point register. Fixed-point accumulation is error free! If supported by a (conventional) vectorizing compiler the method would boost both the speed of a computation and the accuracy of the result! The [KUL] method does not just solve the problem with faithful rounding. Sums and dot products are computed to full accuracy. This allows an easy and very fast realization of multiple precision floating-point and interval arithmetics.

The second paragraph in your mail of Nov. 22 reads: *One implementation of this proposal would use internal accumulators wide enough to hold the entire exponent range, requiring width of hundreds to tens of thousands of bits (for the 32-bit, 64-bit, 80-bit, and 128-bit formats).*

Other methods only consider the 32-bit and the 64-bit formats. Who knows how they would perform for a 128-bit format? The wording accumulator in your mail may be misleading the understanding of the situation. All that is actually needed in the case of the 64-bit format is a tiny local memory on the arithmetic unit of about 1K bytes. We really can and we should afford this at a time where computer memory is measured in gigabytes. The arithmetic itself is not much different from what is available on a conventional CPU. For larger data formats see the section 'Hardware Accumulation Window' in my book. I very much appreciate your suggestion to further investigate the subject with regard to an emerging revised IEEE 754 standard. A new standard should also consider the more basic features of high speed scientific computing.

With best regards

Ulrich Kulisch

[ORO] Takeshi Ogita, Siegfried M. Rump, Shin'ichi Oishi: *Accurate Sum and Dot Product,* SIAM Journal on Scientific Computing, Vol. 26, No. 6, pp. 1955 - 1988, 2005.

[ROO] Siegfried M. Rump, Takeshi Ogita, Shin'ichi Oishi: *Accurate Floating-Point Summation,* to be published in Reliable Computing, 2006, http://www.ti3.tu-harburg.de/paper/rump/Ru05d.pdf.

[KUL] Ulrich W. Kulisch: *Advanced Arithmetic for the Digital Computer - Design of Arithmetic Units,* Springer-Verlag, 2002.

February 10, 2006

Dear colleagues

My thanks to everybody who sent me comments on my letter of January 28, 2006 to stds-754@IEEE.ORG and to Nelson Beebe. In the responses, the wordings accurate sum and dot product, exact sum and dot product, and faithfully rounded sum and dot product are used more or less synonymously. For me the meaning of the last differs from that of the others. An accurate or exact dot product means that the result is computed to the fullest possible accuracy. Not a single bit is lost.

Here is why I see the issue in this way:

A. The most natural way to accumulate numbers is fixed-point accumulation. It is simple, error free and fast. This is also true for the accumulation of floating-point numbers and of their products. If the result register is wide enough it can be done without exception. The result is exact. Not a single bit is lost. The arithmetic to achieve this is much the same as that of a conventional CPU. Fixed-point accumulation of floating-point sums and dot products can be realized in hardware at low cost. And it is very very fast. If supported by a vectorizing compiler it would boost both the speed of a computation and the accuracy of the result.

Fully accurate sums and dot products improve many applications. As a by-product, multiple precision real and interval arithmetic can be done at very high speed. With operator overloading they are very easy to use. With a long precision interval arithmetic, for instance, highly accurate enclosures of orbits of dynamical systems have been obtained for considerable long durations. Iterated defect correction is another important class of applications. The method can be applied to compute enclosures of arithmetic expressions or of polynomials with very high accuracy. The result is an enclosure as a long precision interval. Verified solution of badly conditioned systems of linear equations by use of the Rump-operator is another large class of important applications. Finally, of course, a faithfully rounded result can be obtained. All these and other applications of an accurate dot product come with very high speed. They can be considered as top-down approaches of a fully accurate dot product.

B. In contrast to this the Rump-Ogita-Oishi method is a bottom-up approach. I mentioned in my mail of January 26, 2006 that I very much admire this method. It achieves faithfully rounded sums and dot products just by using conventional floating-point arithmetic. The methods are fast in comparison with other software methods. This certainly is a great achievement of our field. Applications of these methods are inherently a subset of the applications of A.

I do not see any reason why we need the methods B. as justification for accurate sums and dot products in the next arithmetic standard. Fixed-point accumulation of sums and dot products is the additive equivalent of fast multiplication techniques, for instance by an adder tree. The advantages, the cost and gain in speed of both techniques are similar. I do not see any reason why mathematicians should hesitate to require this mode of operation from a new arithmetic standard. If we do not require it we will never get it. We would not have got floating-point operations with directed roundings if IEEE 754 hadn't required it.

With best regards

Ulrich Kulisch

June 2006

Dear colleagues:

Following the discussion on elementary functions, my colleagues here urged me to comment to you on this subject. These are my comments:

Concepts like correct rounding, well rounded, faithful rounding, monotone rounding, and others have been discussed at length. I think a clear distinction should be made between arithmetic operations and elementary functions.

The IEEE 754 standard requires that arithmetic operations are provided with four different roundings: Rounding downwards, rounding upwards, rounding towards zero, and rounding to the nearest floating-point number (round to nearest even). All these roundings are monotone. Implementation of these operations is well understood and established. It is supported by hardware (guard digits, etc.). The error is less than 1 or 0.5 ulp respectively. But what is most important is that the potential error is known to the user so that he can deal with it.

The situation is very different for elementary functions. Extremely accurate evaluation of elementary functions for all relevant argument values needs several guard digits in each individual case. In general, however, the hardware does not support enough guard digits. Their realization in software results in slow function evaluations. A practical compromise is to evaluate elementary functions just using machine precision. Function evaluation is then fast. Experience shows that for the double precision format this can be done for the conventional 24 elementary functions with an error that is less than 1.5 ulp. Even for a long data type elementary functions can be implemented with an error less than a few ulp.

Of utmost importance, however, is that all elementary functions must be provided with proven and reliable error bounds. This error bound must be made known to the user so that he can deal with it. He should know just what he gets if an elementary function is used in a program. It is the vendor's responsibility to deliver these error bounds. Error estimates or error statements based on speculation are simply not acceptable. Of course, deriving proven reliable *a priori* error bounds can take a lot of work. But this work has to be done only once. A great deal of it can be done by use of the computer and of interval arithmetic. For an individual elementary function different error bounds may be obtained for different evaluation domains. Their maximum then is a global error bound. Proven error bounds must consider all error sources, like the approximation error in a reduced range, errors resulting from range reductions, etc.

Correctly rounded, well rounded, faithfully rounded, or monotone rounded elementary functions would be desirable since they might preserve nice mathematical properties. For a 32 bit data format this may be a realistic requirement.

But they will be very very difficult to achieve for the longer data formats which are expected to be available in a new IEEE arithmetic standard. Of course, elementary functions should be provided for all data formats of a new arithmetic standard and possibly even for a dynamic data format. For a 128 bit data format, for instance, an error bound of 3 bits certainly would be acceptable. Directed rounding, of course, must deliver lower and upper bounds appropriately.

Comments on interval elementary functions have been solicited. Of course, high accuracy is desirable. However, to require results that differ from the correct result by only the monotone directed roundings is quite unrealistic. Speed also is an issue for interval arithmetic! For the double precision format the elementary functions have been implemented for interval arguments (just using double precision arithmetic) with proven reliable error bounds that differ from the best possible interval enclosure of the result by an error that is, say, less than 1.5 ulp for each of the bounds. This is fully acceptable!

A programming language or an arithmetic standard that supports interval arithmetic should provide highly accurate elementary functions for interval data for all data formats of the standard and for dynamic precision. These function evaluations suffer little from overestimation caused by interval arithmetic. For point intervals the computed bounds show immediately how accurately the function has been evaluated.

Lots of experience is available concerning the implementation of elementary and special functions for real and interval arguments with proven reliable error bounds for diverse data formats. I refer to published and unpublished work of Walter Kraemer, Werner Hofschuster, Frithjof Blomquist, and others.

For easy and fast evaluation of special functions and of complex elementary functions it would be extremely useful if a new standard would provide, in addition to the usual elementary functions, a number of auxiliary functions with proven *a priory* error bounds. Examples are: $f(x) = sqrt(1 + x) - 1$, $g(x) = 0.5 * log(x * x + y * y)$, $h(x) = exp(x) - 1$, $r(x) = sqrt(x * x - 1)$, and perhaps some others.

Beyond of these issues, it must be remembered that a most important elementary function is a fully accurate scalar product! As a side effect it would boost both the speed of a computation and the accuracy of its result. A fused multiply and add operation at the matrix level is inherent to it, and is fundamental to the whole of numerical analysis. Also, fast multiple precision arithmetic can easily be provided with it to a certain extent.

The vendor is responsible for the quality of the arithmetic and of the elementary functions, and he has to provide valid (guaranteed) error bounds. The user is responsible for the mapping of his problem onto the computer and for the interpretation of the computed result. These are distinct responsibilities. They should not be confused or conflated.

It is my personal opinion that a new arithmetic standard should primarily standardize certain data formats for ease of data transfer between different platforms. In my opinion it is not desirable that all platforms in all instances should be required to react in an identical way. Pinning details down at too early a stage may greatly hinder further progress. Competition is healthy and continued development depends on it.

This and my earlier letters to the IEEE754 revision group have the support of the GAMM-Fachausschuss on Computer Arithmetic and Scientific Computing.

With best regards

Ulrich Kulisch

October 2006

Dear colleagues,

For many applications a verified solution of a numerical problem with narrow bounds is needed. A fusion reactor is planned to be built in France a few kilometers to our west. To keep the reaction going and stable large systems of equations have to be solved. Computations which we have been asked to verify have sometimes had surprising results. So I want to verify all the computations and simulations essential for this reactor before they are applied. This requires fast hardware support of the necessary arithmetic.

Surprisingly there are problems for which a verified solution can be obtained faster than an approximate solution. For systems of equations, however, a verified solution requires a particular verification step to follow an approximate solution. The verification step needs a different kind of arithmetic (defect correction techniques and interval arithmetic). On existing computers this arithmetic has to be simulated by the floating-point or integer arithmetic provided. Thus the verification step is usually slower by orders of magnitude than the computation of an approximate solution. A consequence of this is that for large problems result verification is simply not possible or it is not applied because of an unrealistic waste of computing power. Quadruple precision arithmetic suffers from the same problem if it is simulated by software.

This need not be so. Verification techniques are not inherently slow. It is the software simulation of the necessary arithmetic which makes them slow. Hardware support for it would drastically change the situation. The two kinds of arithmetic must be brought into balance.

Interval operations can be made as fast as simple floating-point operations. On Intel or AMD processors about 30 additional gates would almost suffice to do this. It cannot be done by putting extra floating-point units on a chip. A hardware implementation of the exact dot product is at least as fast as a (possibly wrong) conventional computation of the dot product in floating-point arithmetic. No clever floating-point simulation of it can come anywhere near this speed.

With respect to the exact dot product Nick Maclaren writes in his mail of Oct. 4, 2006:

Again, this is exactly like mandating 'exact' results for the special functions. 32-bit cos is easy; 64-bit is feasible; 128-bit is not in cases where time and memory are critical; and what about extended? And, if cos must be exact, why not erf?

This seems to me to point out the right way to go. The dot product of two floating-point vectors is the most important and the most frequently used elementary function. Hardware support for the exact dot product for (single and) double precision would be enough for many computational methods to be greatly enhanced. It also would suffice to correctly compute the vast majority of dot products for higher precision data formats. This is the least that should be done. If a very large exponent range is needed software aids (perhaps sorting or other methods) could be used.

Experience has shown that for numerical computing, manufacturers only implement what the arithmetic standard requires. It would be tragic if a certain lack of familiarity with verification techniques were to prevent a future arithmetic standard from providing the arithmetic needed to allow results to be verified.

A more formal specification of the *Basic Requirements ...* is being prepared by the GAMM-Fachausschuss on Computer Arithmetic and Scientific Computing.

Ulrich Kulisch

December 2006

Dear colleagues,

Michel Hack wrote on Nov.29, 2006:
Subject: Suggestion from Prof. Wolff von Gudenberg

Some comments on the Kulisch Accumulator proposal.
I don't think functions should be expected to pass the accumulator by value, and to return it by value. Perhaps some machines will implement a hardware register that is 64K bits wide, but probably not in the near future. Any practical implementation (such as IBM's of 20 years ago) would use accumulators in storage, so they would be passed by reference. (Actually, 64K is too short by 276 bytes, so a hardware register would have an awkward size, to say the least.)

Yes, practical implementations on IBM, SIEMENS, and Hitachi computers 20 years ago placed the accumulator in the user memory. However, this was not done by choice, but by necessity. These computers did not provide enough register space. A disadvantage of this solution is that for each accumulation step, four memory words must be read and written in addition to the two operand loads. (For details see my book). So the scalar product computation speed is limited by the memory to processor transfer bandwidth and speed.

Use of the word *accumulator* here may be misleading. *Long Result Register* would much better describe the situation. The *Result Register* of old mechanical calculators was usually much wider than the *Input Register* or the *Adder*. The *Long Result Register* here is nothing else but the *Result Register* of old calculators adapted to modern data formats.

The theoretical size of this *Long Result Register* grows with the exponent range of the data format. If this range is extremely large, as for instance for an extended precision floating-point format, then only an inner part of this register should be supported by hardware. See the section *Hardware Accumulation Window* in my book. The outer parts of this window must then be handled in software. Probably they would seldom be used. A scalar product computation, in general, does not require an extremely large exponent range (as might be necessary for polynomials with high powers). Restriction of the register size to the single exponent range would reduce the size to about one half. Then the vast majority of scalar products (all those which do not cause an under- or overflow) will exactly execute on fast hardware.

So 64K bytes for a hardware register would be extremely large. 8K bytes would be a very convenient size. It would be a great progress and I would already be happy if 1K bytes would be available. See the *Basic Requirements for a Future Floating-Point Arithmetic Standard*. Then the exact result of two double precision scalar products or the exact result of one scalar product of vectors with double precision interval components could conveniently be computed without any exceptions.

Clearing the long register is done by setting the *(all bits zero)*-flags to zero. These flag bits that are used for the fast carry resolution can be used for the rounding of the long register contents also. (For details see my book).
With best regards
Ulrich Kulisch

January 2007

Dear colleages,

Today I had a close look into the document entitled:
Draft Standard for Floating-Point Arithmetic P 754.
I came to the conclusion that I need to write you again. I was aghast to read that the arithmetic operations with the directed roundings are not required in the current draft. This is fatal for interval arithmetic. It will prevent interval mathematics from being more widely used in the scientific computing community for another 25 years. This is not tolerable.

Let me cite a paragraph from an earlier letter:
"Future **processors** should provide the basic arithmetic operations with the monotone downwardly and upwardly directed roundings by distinct operation codes. Each of these 8 operations must be callable as a single instruction, that is, the rounding mode must be inherent in each.

It is desirable that with a new arithmetic standard, denotations for arithmetic operations with different roundings be introduced. They could be:

`+`, `-`, `*`, `/`	for operations with rounding to the nearest floating-point number,
`+>`, `->`, `*>`, `/>`	for operations with rounding upwards,
`+<`, `-<`, `*<`, `/<`	for operations with rounding downwards, and
`+|`, `-|`, `*|`, `/|`	for operations with rounding towards zero (chopping)."

Frequently used programming languages regrettably do not allow 4 plus, minus, multiply, and divide operators for floating-point numbers. This, however, does not justify separating the rounding from the operation!

Instaed, a future floating-point arithmetic standard should require that **every future processor** shall provide the 16 operations listed above. A future standard even can and should dictate names for the corresponding assembler instructions as for instance:
addp, *subp*, *mulp*, *divp*, *addn*, *subn*, *muln*, *divn*, *addz*, *subz*, *mulz*, and *divz*.
Here p stands for rounding toward positive, n for rounding toward negative, and z for rounding toward zero. With these operators interval operations can easily be programmed. They would be very fast and fully transferable from one processor to another.

This is exactly the way interval arithmetic is provided in our C++ class library C-XSC which has been successfully used for more than 15 years. Since C++ only allows operator overloading the operations with the directed roundings cannot and are not provided in C-XSC. These operations are rarely needed in applications. If they are needed, interval operations are performed instead and the upper or lower bound of the result then gives the desired answer.

Please give this letter serious consideration. Interval arithmetic must be appropriately provided for on future processors at least for the data format double precision.

With best regards
Ulrich Kulisch

June 2007

David Hough wrote on 21 June 2007:

The time has come for decimal to resume its historical role as the default floating-point radix for most computations of most users, and 754R recognizes this by encompassing decimal.

I fully support a move toward use of the decimal number system on computers. It certainly would eliminate many difficulties and unnecessary error sources in computations.

In the same mail follows:

Reproducibility

So 754R has to come to terms with what will eventually become the default requirement as surely as decimal radix - namely floating-point results reproducible on all implementations of the same language. And just as 754R also continues to cater to the small but vital minority willing to pay the price to reap the rewards of binary radix, it must also cater to that minority by providing a way for specifying where "different but equally good" results are allowable for better performance on specific systems, for persons who can accurately ascertain whether "different is equally good".

An example of different but equally good:

$$s = x + y + z$$

where

$$x = 2**N$$
$$y = 1$$
$$z = -(2**N)$$

In the absence of roundoff there is no mathematical difference between evaluating s as (x+y)+z or x+(y+z) or (x+z)+y ; nor is there any difference in a typical error bound on the computed answer, which will depend on the number of rounded additions and the magnitude of the largest addend. But when N is much bigger than the precision P of the variables, then (x+y) rounds to x, and (y+z) rounds to z, and the rounded sum of (x+y)+z or x+(y+z) will be 0, while *the rounded sum of (x+z)+y will be 1 exactly. If either x or z is not regarded as exact but as possibly contaminated by roundoff and thus at least uncertain by at least a half ulp 2**(N-P-1), then that uncertainty swamps the difference between the computed sums 0 and 1. But to recognize that requires some notion of error analysis, while anybody can tell the difference between 0 and 1. Fixing the order of evaluation of x+y+z doesn't eliminate the need for error analysis but widens the applicability of error analyses that are made, because they don't need to be repeated (or the programs debugged) if the results are the same. The goal of 754R's reproducibility requirement is to provide a way to remove the gratuitous variation added by programming environments when it costs more than it benefits. The primary beneficiaries of reproducibility features are end user programmers and ISV programmers who have been under-represented in the 754R meetings.*

My COMMENT on this:
If the numbers in this example are of dollars I think most of us would accept 0 instead of 1, if this occurs only once. But if the 1 is replaced by 8192, I wonder how many of us would accept 0 as 'different but equally good' to 8192.

Not the best education in error analysis can help to foresee whether a situation similar to the example appears in a program that performs 10^{15} floating-point operations a second. The only true interpretation of the different results is: 1 and 8192 are correct while 0 is wrong. This is much more user-friendly.

I think that two things are conflated in the discussion of this simple example:
a) errors in the data that occur during a computation and
b) errors in the arithmetic.
a) is an unavoidable source of errors. Of course, the data are frequently contaminated by roundoff. But an error in the data does not justify to add another error in the arithmetic. The vendor is responsible for the quality of the arithmetic. To be as accurate as possible is essential. Sloppy arithmetic prevents users from solving problems correctly when the data are exact.

Compound arithmetic operations like 'accumulate' (continued addition) and 'multiply and accumulate' are common instructions on all vector processors. By pipelining they greatly speed up the computation. Compilers help to detect them in a program. Within a certain exponent range (double precision) these instructions can easily be executed exactly. This eliminates many difficulties and unnecessary error sources in computations and so must be implemented on future computers. 754R should require this from all future processors. For implementation details I refer to my upcoming book. It is entitled:
Computer Arithmetic and Validity - Theory, Implementation and Applications.

Ulrich Kulisch 27 June 2007

July 2007

David Hough wrote on July 6, 2007:

Concerning my simplest possible example of irreproducible results,

$z = a + b + c$

Prof. Kulisch wrote a few days ago:

"I think that two different things are conflated in the discussion of this simple example:
a) errors in the data that occur during a computation and
b) errors in the arithmetic.
a) is an unavoidable source of errors. Of course, the data are frequently contaminated by roundoff. But an error in the data does not justify error in the arithmetic. The vendor is responsible for the quality of the arithmetic. To be as accurate as possible is essential. Sloppy arithmetic prevents users from solving problems correctly when the data are exact."

In scientific computation the data are seldom exact, so the problem is to get the final uncertainty due to roundoff to be much less than the final uncertainty due to the input data and due to the algorithm. So getting exact or correctly-rounded results for large sums or dot products is usually not economic. Realistic computations usually involve many such operations whose rounded results are combined. The economic case for even 128-bit floating-point hardware is arguable.

Variable precision is indeed an answer for a certain small class of users operating on data they consider exact or very precise, but 754R has repeatedly considered and rejected standardizing variable precision in this revision.

The goal of a typical roundoff error analysis of an approximate process on inexact inputs is not to prove that the result is correct, but that roundoff has not made the result worse than it would have been in exact arithmetic. Those analyses are difficult enough that one would prefer not to repeat them for every optimization level of every release of every compiler for every platform.

So the goal of the reproducibility proposals is to make it possible to do one analysis that could be applied to each platform the program is run upon. "Possible" does not mean "always desirable" because there are very real performance costs associated with constraining results to be reproducible.

My COMMENT on this:
I regret the need to write again, but I disagree with nearly everything that is said above. A new standard will again reign for 20 years or so and we will all be bound by it. So I am deeply concerned that it should be based on a thorough understanding of the issues.

In scientific computation the data are seldom exact.

I give only two counterexamples for this assertion. I could easily give more.

1. For a system of linear equations A x = b, many solvers repeatedly compute the defect d = b - A x' , where x' is an approximation of the solution. In the expression for d all data, and x' in particular, are exact. With an exact scalar product d can be computed exactly.

2. Any algorithm designed to find a root of an equation f(x) = 0 must necessarily compute f for arguments x' which cause cancellation of leading digits. If f is a polynomial, all the data and x' in particular, are exact. With double precision interval arithmetic and an exact scalar product f can be computed to guaranteed full accuracy. This is essential for Newton's method to succeed.

So getting exact or correctly-rounded results for large sums or dot products is usually not economic.

Implementing an exact scalar product (for double precision) in silicon is much cheaper than conventionally implementing a full 128-bit floating-point arithmetic as defined in IEEE754R. Yet implementing a 128-bit arithmetic is very simple on the basis of the exact scalar product and achieving higher precision is not much more difficult for real and for interval data. Once the exact scalar product is included in the hardware, this kind of high precision arithmetic will be fully supported by fast hardware.

The economic case for even 128-bit floating-point hardware is arguable.

I agree with this. It is only really needed for problems which require an extremely large exponent range.

CONCLUSION:

In simplicity lies truth. So my suggestion for a new arithmetic standard is this:

1. A well defined double precision floating-point arithmetic.
2. Fast and direct hardware support for double precision interval arithmetic.
3. An exact scalar product for the double precision format.

With best regards
Ulrich Kulisch   14 July 2002

Craig Nelson wrote on July 21:

In your 1st example, it strikes me that the input variables (b, A) may (unlikely if they arise from a physical problem where data are measured by instruments of finite precision) be precisely known in decimal, but will be imprecise when converted to binary floating point. To be correct, I think you would have to demand that decimal arithmetic would be supported and utilized if the inputs are decimal scientific notation.

YOUR EXAMPLE:

I give only two counterexamples for this assertion. I could easily give more.

1. For a system of linear equations A x = b, many solvers repeatedly compute the defect d = b - A x' , where x' is an approximation of the solution. In the expression for d all data, and x' in particular, are exact. With an exact scalar product d can be computed exactly.

My COMMENT on this:

A computer can only solve problems for data that are in the computer. These data define the problem. In this sense they are exact.

It is the user's responsibility to bring the data in the computer with sufficient accuracy. For a particular problem you may well think that the data do not describe your problem accurately enough if they are read through standard input conversion or if they are otherwise contaminated. Then it would be most natural to read the data into the computer as intervals or to a higher precision and solve the problem for these data with interval or higher precision arithmetic.

Of course, nobody would do this. On existing processors it would slow down the computation by orders of magnitude. The proposed IEEE P754 arithmetic as it is presently specified will not change this situation much.

The simple arithmetic suggested under CONCLUSION in my mail, however, would change the situation dramatically. It provides interval arithmetic at the speed of simple floating-point arithmetic and high speed arithmetic for dynamic precision for real and interval data.

Best regards
Ulrich Kulisch 25 July 2007