THE IEEE STANDARDS COMMITTEE P1788 FOR INTERVAL ARITHMETIC REQUIRES AN EXACT DOT PRODUCT

Ulrich Kulisch
Institut für Angewandte und Numerische Mathematik, Universität Karlsruhe, D-76128 Karlsruhe, Germany

Keywords: High speed computing, Computing with guarantees, Vector processing, Interval arithmetic, Exact dot product.

Abstract: Computing with guarantees is based on two arithmetical features. One is fixed (double) precision interval arithmetic. The other one is dynamic precision interval arithmetic, here also called long interval arithmetic. The basic tool to achieve high speed dynamic precision arithmetic for real and interval data is an exact multiply and accumulate operation and with it an exact dot product. Actually the simplest and fastest way for computing a dot product is to compute it exactly. Pipelining allows to compute it at the same high speed as vector operations on conventional vector processors. Long interval arithmetic fully benefits from such high speed. Exactitude brings very high accuracy, and thereby stability into computation. This document is intended to provide some background information, to increase the awareness, and to informally specify the implementation of an exact dot product.

1 INTRODUCTION

It is well known that evaluation of a real arithmetic expression (a sequence of arithmetic operations) for an interval $\boldsymbol{x}$ leads to a superset of the range of the expression over the interval $\boldsymbol{x}$. The distance between this superset and the range decreases with the width of the interval $\boldsymbol{x}$ and it tends to zero with the width of $\boldsymbol{x}$. This simple observation is the basis for many successful and simple applications of long interval arithmetic. Practically this means that results of real arithmetic expressions can always be guaranteed to a number of correct digits by using variable precision interval arithmetic. Variable length interval arithmetic can be made very fast by an exact dot product and complete arithmetic (Kulisch, 2008; Kulisch and Snyder, 2009). There is no way to compute a dot product faster than the exact result. By pipelining, it can be computed in the time the processor needs to read the data, i.e., it comes with utmost speed. Variable length interval arithmetic fully benefits from such high speed. No software simulation can go as fast. By operator overloading variable length interval arithmetic is very easy to use. It allows evaluation of real arithmetic expressions with guaranteed bounds (naive interval arithmetic). This has the potential of greatly increasing the acceptance of interval arithmetic in the scientific computing community.

In a letter to the IEEE 754 revision group (IFIP WG - IEEE 754R letter) (Sept. 4, 2007) the IFIP Working Group on Numerical Software requested that a future arithmetic standard should consider and specify the exact dot product as basic ingredient of variable precision real and interval arithmetic.

In another letter to IEEE P1788 (IFIP WG - IEEE P1788 letter) (Sept. 9, 2009) the IFIP Working Group strongly supports inclusion of an exact dot product in the IEEE standard P1788 for interval arihtmetic. *The exact dot product is essential for fast long real and long interval arithmetic, as well as for assessing and managing uncertainty in computer arithmetic. It is a fundamental tool for computing with guarantees and can be implemented with very high speed.* In Numerical Analysis the dot product is ubiquitous.

On Nov. 18, 2009 the IEEE standards committee P1788 on interval arithmetic accepted a proposal (Kulisch and Snyder, 2009) for including the exact dot product into the future interval arithmetic standard.

2 INFORMAL DESCRIPTION FOR REALIZING AN EXACT DOT PRODUCT

Actually the simplest and fastest way for computing a dot product is to compute it exactly. Take a chest of drawers with 67 numbered drawers. Each one holds

Kulisch U. (2010).
THE IEEE STANDARDS COMMITTEE P1788 FOR INTERVAL ARITHMETIC REQUIRES AN EXACT DOT PRODUCT.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Information Systems Analysis and Specification*, pages 475-478
DOI: 10.5220/0002906204750478

64 bits. The exponent of the summand (the product) consists of 12 bits. The leading 6 bits give the address of the three consecutive drawers to which the summand of 106 bits is added. The low end 6 bits of the exponent are used for the correct positioning of the summand within the selected drawers. The addition affects at most 170 bits of these drawers, i.e., an adder of 170 bits could execute the addition in a single add cycle.

A carry is absorbed by the next more significant drawer in which not all bits are 1. For fast detection of this word a flag is attached to each drawer. It is set 1 if all bits of the word are 1. This means that a carry will propagate through the entire word. As soon as the exponent of the summand is available the flags allow selecting and incrementing the carry word. This can be done simultaneously with adding the summand into the selected drawers. If the addition produces a carry the incremented word is written into the carry word. Otherwise it is left as it was. A zero flag may serve the same purpose in case of subtraction.

There is indeed no simpler way of accumulating a dot product. Any method that just computes an approximation also has to consider the relative values of the summands. This results in a more complicated method.

The fascinating property of the exact dot product is the fact that it can be computed with extreme speed, ideally in the time the processor needs to read the data. No special cases have to be dealt with. The technique of adding a medium sized bit string to a very long one may have applications in other areas of computing as well.

Ideally on the computer the chest of drawers would consist of register memory. This leads to the fastest solution. To add a product to the register memory only two memory words (the two factors of the product) must be read. The result of the accumulation of the product appears in the register memory.

The basic idea discussed here was realized on IBM, SIEMENS, and Hitachi computers about 25 years ago (IBM System/370 RPQ, 1984; ACRITH-XSC). These computers did not provide enough register space. So here the chest of drawers was placed in the user memory. The disadvantage of this solution is that for each accumulation step, four memory words (the three words to which the product is added and the word which absorbs the carry) must be read and written in addition to the two operand loads. So the scalar product computation is slower than by use of register memory. However, in practice today the necessary memory space would probably be in cache. So the loss in speed would be marginal. Anyway, compared with any (even fast) software solution the gain in speed would still be tremendous. Generally, realization of an exact dot product may be architecture dependent. For more details see (Kulisch, 2008).

All conventional vector processors provide a *multiply and accumulate* operation (the dot product) to achieve high speed. In a pipeline the arithmetic (the multiplication and the accumulation) is done in the time the processor needs to read the data. Very effective vectorizing compilers have been developed that use this *multiply and accumulate* operation within a user's program as often as possible, since this greatly speeds up program execution. However, the accumulation is done in floating-point arithmetic. The so-called *partial sum technique* alters the sequence of the summands and causes errors in addition to the usual floating-point errors. The exact dot product avoids all numerical errors and at the same speed. The hardware needed for it is comparable to that for a fast multiplier by an adder tree (also called Wallace tree), accepted years ago and now standard technology in every modern processor. The exact dot product brings the same speed up for accumulations at comparable costs. In speed a hardware implementation of the exact dot product exceeds computing an accurately rounded dot product in software by several orders of magnitudes.

3 A FEW SAMPLE APPLICATIONS

A very impressive application is considered in (Blomquist et al., 2009), an iteration with the logistic equation (dynamical system)

$$x_{n+1} := 3.75 \cdot x_n \cdot (1 - x_n), \quad n \geq 0.$$

For the initial value $x_0 = 0.5$ the system shows chaotic behavior.

Double precision floaint-point or interval arithmetic totally fail (no correct digit) after 30 iterations while long interval arithmetic after 2790 iterations still computes correct digits of a guaranteed enclosure.

In numerical analysis the scalar or dot product is ubiquitous. It is not merely a fundamental operation in all vector and matrix spaces. The process of residual or defect correction, or of iterative refinement, is composed of scalar products. There are well known limitations to these processes in floating-point arithmetic. The question of how many digits of a defect can be guaranteed in single, double or extended precision arithmetic has been carefully investigated. With an exact scalar product the defect can always be computed to full accuracy. It is the exact scalar product which makes residual correction effective. This has a

direct and positive influence on all iterative solvers of systems equations.

A simple example may illustrate the advantages of what has been said: Solving a system of linear equations $Ax = b$ is the central task of numerical computing. Large linear systems can only be solved iteratively. Iterative system solvers repeatedly compute the defect d (sometimes called the residual) $d := b - A\tilde{x}$ of an approximation $\tilde{x}$. It is well known that the error e of the approximation $\tilde{x}$ is a solution of the same system with the defect as the right hand side: $Ae = d$. If $\tilde{x}$ is already a good approximation of the solution, the computation of the defect suffers from cancellation in floating-point arithmetic, and if the defect is not computed correctly the computation of the error does not make sense. In the computation of the defect it is essential that, although $\tilde{x}$ is just an approximation of the solution x^*, $\tilde{x}$ is assumed to be an exact input and that the entire expression for the defect $b - A\tilde{x}$ is computed as a single exact scalar product. This procedure delivers the defect to full accuracy and by that also to multiple precision accuracy. Thus the defect can be read to two or three or four fold precision as necessary in the form $d = d_1 + d_2 + d_3 + d_4$ as a long real variable. The computation can then be continued with this quantity. This often has positive influence on the convergence speed of the linear system solver. It is essential to understand that apart from the exact scalar product, all operations are performed in double precision arithmetic and thus are very fast. If the exact scalar product is supported by hardware it is faster than a conventional scalar product in floating-point arithmetic[1].

But also direct solvers of systems of linear equations profit from computing the defect to full accuracy. The step of verifying the correctness of an approximate solution is based on an accurate computation of the defect. If a first verification attempt fails, a good enough approximation can be computed with the exact scalar product.

The so called *Krawczyk-operator* which is used to verify the correctness of an approximate solution is able to solve the problem only in relatively stable situations. Matrices from the so called Matrix Market usually are very ill conditioned. In such cases the Krawczyk-operator almost never finds a verified answer. Similarly, for instance, in the case of a Hilbert matrix of dimension greater than eleven the Krawczyk-operator always fails to find a verified solution. In all these cases, however, the Krawczyk-operator recognizes that it cannot solve the problem and then automatically calls a more powerful operator, the so called *Rump-operator* which then in almost all cases, even for extemely ill conditioned problems solves the problem satisfactorily (Klatte et al., 1993).

The Krawczyk-operator first computes an approximate inverse R of the matrix A and then iterates with the matrix $I - RA$, where I is the identity matrix. The Rump-operator inverts the product RA in floating-point arithmetic again and then multiplies its approximate inverse by RA. This product is computed with the exact scalar product and from that it can be read to two or three or four fold precision as necessary as a long-real matrix, for instance $(RA)^{-1}RA \approx I_1 + I_2 + I_3 + I_4$ in case of a four fold precision. The iteration then is continued with the matrix $I - (I_1 + I_2 + I_3 + I_4)$. It is essential to understand that even in the *Rump-algorithm* all arithmetic operations except for the (very fast) exact scalar product are performed in double precision arithmetic and thus are very fast. This linear system solver is an essential ingredient of many other problem solving routines with automatic result verification.

To be more successful conventional floating-point and interval arithmetic have to be complemented by some easy way to use multiple or variable precision arithmetic. This enables the use of higher precision operations in numerically critical parts of a computation. The fast and exact scalar product is the tool to provide this very easily.

If one runs out of precision in a certain problem class, one often runs out of quadruple precision very soon as well. It is preferable and simpler, therefore, to provide a high speed basis for enlarging the precision rather than to provide any fixed higher precision or to simulate higher precision in software. A hardware implementation of a full quadruple precision arithmetic is more costly than an implementation of the exact scalar product. The latter only requires fixed-point accumulation of the products. On the computer, there is only one standardised floating-point format that is double precision.

For many applications it is necessary to compute the value of the derivative of a function. Newton's method in one or several variables is a typical example of this. Modern numerical analysis solves this problem by automatic or algorithmic differentiation. The so called reverse mode is a very fast method of automatic differentiation. It computes the gradient, for instance, with at most five times the number of operations needed to compute the function value. The memory overhead and the spatial complexity of the reverse mode can be significantly reduced by the exact scalar product if this is considered as a single, always correct, basic arithmetic operation in the vector

[1] An iterative method which converges to the solution in infinite precision arithmetic often converges more slowly or even diverges in finite precision arithmetic.

spaces (Shiriaev, 1993). The very powerful methods of global optimization are impressive applications of these techniques.

Many other applications require that rigorous mathematics can be done with the computer using floating-point arithmetic. As an example, this is essential in simulation runs (eigenfrequencies of a large generator, fusion reactor, simulation of nuclear explosions) or mathematical modelling where the user has to distinguish between computational artifacts and genuine reactions of the model. The model can only be developed systematically if errors resulting from the computation can be excluded.

Nowadays computer applications are of immense variety. Any discussion of where a dot product computed in quadruple or extended precision arithmetic can be used to substitute for the exact scalar product is superfluous. Since the former can fail to produce a correct answer an error analysis is needed for all applications. This can be left to the computer. As the scalar product can always be executed exactly with moderate technical effort it should indeed always be executed exactly. An error analysis thus becomes irrelevant. Furthermore, the same result is always obtained on different computer platforms. An exact scalar product eliminates many rounding errors in numerical computations. It stabilises these computations and speeds them up as well. It is the necessary complement to floating-point arithmetic.[2]

REFERENCES

IEEE Floating-Point Arithmetic Standard 754, 2008.

The *IFIP WG - IEEE 754R letter*, dated September 4, 2007.

The *IFIP WG - IEEE P1788 letter*, dated September 9, 2009.

U. Kulisch, V. Snyder. *The Exact Dot Product as Basic Tool for Long Interval Arithmetic*, passed on Nov. 18, 2009 as official IEEE P1788 document.

U. Kulisch. *Computer Arithmetic and Validity – Theory, Implementation, and Applications*, de Gruyter, Berlin, New York, 2008.

U. Kulisch. *Implementation and Formalization of Floating-Point Arithmetics* IBM T. J. Watson-Research Center, Report Nr. RC 4608, 1 - 50, 1973. Invited talk at the Caratheodory Symposium, Sept. 1973 in Athens, published in: The Greek Mathematical Society, C. Caratheodory Symposium, 328 - 369, 1973, and in Computing 14, 323–348, 1975.

U. Kulisch. *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik.* Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim/Wien/Zürich, 1976.

S. M. Rump. *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Universität Karlsruhe, 1980.

R. Lohner. *Interval Arithmetic in Staggered Correction Format*. In: E. Adams and U. Kulisch (eds.): *Scientific Computing with Automatic Result Verification*, pp. 301–321. Academic Press, (1993).

F. Blomquist, W. Hofschuster, W. Krämer. *A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range.* In: A. Cuyt et al. (eds.): *Numerical Validation in Current Hardware Architectures*, Lecture Notes in Computer Science LNCS, vol. 5492, Springer-Verlag Berlin Heidelberg, 41-67, 2009.

R. Klatte, U. Kulisch, C. Lawo, M. Rauch, A. Wiethoff, *C–XSC, A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, Berlin/Heidelberg/New York, 1993. See also: http://www.math.uni-wuppertal.de/∼xsc/ resp. http://www.xsc.de/.

D. Shiriaev. *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*. Dissertation, Universität Kalrsuhe, 1993.

S. Oishi; K. Tanabe; T. Ogita; S.M. Rump. *Convergence of Rump's method for inverting arbitrarily ill-conditioned matrices*. Journal of Computational and Applied Mathematics 205, 533-544, 2007.

IBM System/370 RPQ. High Accuracy Arithmetic. SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1984.

ACRITH-XSC: IBM High Accuracy Arithmetic, Extended Scientific Computation. Version 1, Release 1. IBM Deutschland GmbH (Schönaicher Strasse 220, D-71032 Böblingen), 1990.
1. General Information, GC33-6461-01.
2. Reference, SC33-6462-00.
3. Sample Programs, SC33-6463-00.
4. How To Use, SC33-6464-00.
5. Syntax Diagrams, SC33-6466-00.

[2] A disappointing feature is the failure of the numerical analysts to influence computer hardware and software in the way they should. It is often said that the use of computers for scientific work represents a small part of the market and numerical analysts have resigned themselves to accepting facilities "designed" for other purposes and making the best of them. J. H. Wilkinson: Turing Lecture 1970, J. ACM 18 (1971), 146.