THE EXACT DOT PRODUCT AS BASIC TOOL FOR LONG INTERVAL ARITHMETIC

ULRICH KULISCH AND VAN SNYDER

Abstract. Computing with guarantees is based on two arithmetical features. One is fixed (double) precision interval arithmetic (see Motion 5). 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. 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 specifies the implementation of an exact multiply and accumulate operation.

1. Introduction

If x_i, $i = 1(1)n$, $n \geq 0$, are floating-point numbers and $X = [x_{\text{low}}, x_{\text{high}}]$ is an interval with floating-point bounds x_{low} and x_{high}, then an element of the form

$$\boldsymbol{x} = \sum_{i=1}^{n} x_i + X$$

is called a *long interval* of *length n*. The x_i are called the *components* of $\boldsymbol{x}$ and X is called the *interval component*.

In this definition n is permitted to be zero. Then the sum is empty and $\boldsymbol{x} = X$ is just an interval with standard floating-point bounds. In the representation of a long interval it is desirable that the components do not overlap, i.e., the exponents of two successive summands differ at least by the number of mantissa digits. The operations for long intervals are written so that they produce results with this property. With an exact dot product the operations $+, -, *, /$, and the square root for long reals and long intervals are easily programmed, see [6–9]. The width of the interval $\boldsymbol{x}$ is a direct measure of its accuracy. With increasing width of a long interval the length n may decrease.

It is well known that evaluation of an arithmetic expression for an interval $\boldsymbol{x}$ leads to a superset of the range of the expression. The distance between this superset and the range decreases with the width of the interval $\boldsymbol{x}$ and it tends to

Part of this work supported by Institut für Angewandte und Numerische Mathematik, Universität Karlsruhe.

Part of this work supported by NASA at Jet Propulsion Laboratory, California Institute of Technology.

zero with the width of $\boldsymbol{x}$. This simple observation is the basis for many successful and simple applications of long interval arithmetic.

In a letter to IEEE 754R [3] (Sept. 4, 2007) the IFIP Working Group on Numerical Software requested that a future arithmetic standard should consider and specify the basic ingredients of variable precision real and interval arithmetic. A paragraph in that letter reads: *The basic tool to achieve high speed dynamic precision arithmetic for real and interval data is an exact multiply and accumulate (i.e., continued addition) operation for the data format double precision. Pipelining gives it high speed and exactitude brings very high accuracy into computation.*

A paragraph in another letter of the IFIP Working Group to IEEE P1788 [4] (Sept. 9, 2009) reads: *The IFIP WG strongly supports inclusion of an exact dot product in the IEEE standard P1788. 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.*[1]

The floating-point arithmetic standard IEEE 754 [1] provides functions sum(p, n) and dot(p, q, n) for accumulation[2] of the components of a vector p and for the dot product[3] of two vectors p and q of length n with no accuracy requirement. Vector processors use such functions to gain high speed. The Vienna proposal [2] also provides functions for these operations and it requires computation of the tightest interval enclosure of the sum and the dot product of vectors with real and interval components.

In contrast to this the IFIP WG - IEEE 754R and the IFIP WG - IEEE P1788 letters [3, 4] require exact accumulation of the sum and the dot product for vectors with real and interval components. An exact dot product for the double precision format is the base for long real and long interval arithmetic. The

[1]In Numerical Analysis the dot product is ubiquitous. It is not merely a fundamental operation in matrix and vector arithmetic. The process of residual or defect correction, or of iterative refinement, is composed of scalar products. With the exact scalar product the defect can always be computed to full accuracy. It is the exact scalar product which makes residual correction effective. It has a direct and positive influence on all iterative solvers of systems of equations. But also direct solvers of systems of equations profit from computing the defect to full accuracy. The step of verifying the correctness of an approximate solution is based on accurate computation of the defect and on the exact dot product in critical situations, see [10, 11] and [9]. Computational geometry is another area where the exact dot product turns out to be extremely useful.

If one runs out of precision using double precision in a certain problem class, one often runs out of quadruple precision very soon as well. It is preferable, therefore, to provide a high speed basis for enlarging the precision rather than to provide any fixed higher precision. With the exact scalar product high speed multiple precision arithmetic can easily be provided for real and for interval data. Long interval arithmetic has been successfully applied to the computation of very ill conditioned problems. Validated numerics requires additional arithmetical features. The exact dot product is the fundamental tool for this purpose. For implementation see [8, 9, 12, 13].

[2]Accumulation means continued addition.

[3]Also called scalar or inner product.

exact dot product can be implemented with the same high speed as dot products on conventional vector processors. Long real and long interval arithmetic fully benefit from such high speed.

Experience has shown that computing the dot product by function calls leads to clumsy and awkward programming. It is reasonable, therefore, that the exact dot product be computed by use of a new data format which is called *complete* associated with each floating-point format together with a few simple operations. A complete format is a signed fixed-point format which has enough digit positions such that multiplication of pairs of non-exceptional floating-point numbers and accumulation of such summands have exact results. They are *complete* in that all possible information from non-exceptional floating-point operands of multiplication and accumulation operations is represented. Not a single bit is lost. The result contains the exact sum in its entirety. [8, 9, 12, 13].

A complete expression is composed of operands and operations. Only three kinds of operands are permitted: floating-point numbers, exact products of two such numbers, and data of the format *complete*. Such operands can be added or subtracted in an arbitrary order. All operations (multiplication, addition, and subtraction) are to be exact. The result is a datum of type *complete*. For non-exceptional floating-point operands it is always exact. No truncation or rounding is performed during execution of a complete expression. The result of *complete arithmetic* reflects the complete information as given by the input data and the operations in the expression.

2. Complete Arithmetic

2.1. **Complete format.** A floating-point format $\mathbb{F}$ is characterized by four parameters, the radix or base b of the number system in use, the number l of digits in the mantissa, and the least and the greatest exponents $emin$ and $emax$; thus $\mathbb{F} = \mathbb{F}(b, l, emax, emin)$.

The product of two floating-point numbers with l digits has $2l$ digits. In order to represent all products of floating-point numbers with negative exponents exactly another $|2emin|$ digits are required after the point for the fractional part of the complete format. In order to represent all products of floating-point numbers with positive exponents exactly $2emax$ digits before the point are required for the integer part of the complete format. Thus all products of floating-point numbers are representable in a fixed-point register of length $2emax + 2l + 2|emin|$ without loss of information. Products of floating-point numbers can be added into a register of this size. Each accumulation can result in an (intermediate) overflow of the integer part by one bit. To accommodate possible overflows another k bits before the point allows b^k accumulations to be computed without loss of information due to overflow. The final result of the accumulation if rounded into a floating-point number is assumed to have an exponent between $emax$ and $emin$; otherwise the problem needs to be scaled.

Complete format is characterized by two numbers $m = k + 2emax$, which is the number of digits before the point, and $d = 2l + 2|emin|$, which is the number of digits after the point. In detail, it is composed of four parts, *viz.* a three-bit status, a one-bit sign, an m-bit integer part, and a d-bit fractional part, where k is chosen such that $L = m + d + 4$ is a multiple of eight. For IEEE 754 binary64 format with $l = 53$, $emin = -1022$, and $emax = 1023$ the recommended value for $L = m + d + 4$ is 4288 bits (536 bytes or 67 words of 64 bits) allowing 2^{88} accumulations before overflow could occur. The status field has one of the values *exact*, *inexact*, $-\infty$, $+\infty$, *overflow*, *sNaN*, or *qNaN*.

3 bits	1 bit	MSB m bits LSB	MSB d bits LSB
Q (status)	S (sign)	I (integer part)	F (fractional part)

Figure 1—Binary interchange, binary complete format

For IEEE 754 decimal64 format with 20 bits of overflow, $m \geq 789$ and $d = 798$. Using the encoding specified in IEEE 754 (ten bits representing three decimal digits), and again assuming the total number of bits to be a multiple of eight, 5312 bits (664 bytes) are recommended.

An implementation shall at least provide complete formats corresponding to binary64 if it provides binary floating-point and decimal64 if it provides decimal floating-point. Complete formats corresponding to other floating-point formats may be provided. Table 1 shows values of m and d for other formats.

Table 1: Complete format parameters for floating-point operands

	Complete formats associated with:				
Parameter	**binary32**	**binary64**	**binary128**	**decimal64**	**decimal128**
k bits	≥ 65	≥ 65	≥ 65	≥ 20	≥ 20
m digits	≥ 319	≥ 2111	≥ 32831	≥ 789	≥ 12308
d digits	300	2150	16608	798	12354
$m + d$	≥ 619	≥ 4261	≥ 49439	≥ 1587	≥ 24663
$k_e^\dagger$ bits	640	4288	49536	5312	82304

† k_e is the recommended width in bits of the exact interchange format encoding. See sub clause 3.6 of IEEE 754.

2.2. Complete operations.

2.2.1. ***Convert***. An implementation shall provide the following operation, producing a result of complete or floating-point format, as specified by the operation. The *source* operand may be of complete format, floating-point format, or integer format. The radix of the operand shall be the same as the radix of the result.

- *formatOf*-**convert**(*source*)
  Conversion of an exceptional floating-point operand (*sNaN*, *qNaN*, $-\infty$, $+\infty$) to complete format shall cause the corresponding status of the result to be set, and the mantissa of the operand shall be copied to the high-order l bits of the fraction part of the result. Conversion of an exceptional complete-format operand (*overflow*, *sNaN*, *qNaN*, $-\infty$, $+\infty$) to another complete format shall preserve the status; if the value of m or d of the result is less than the corresponding value for the operand, the low-order bits of the integer part and the high-order bits of the fraction part shall be preserved. Conversion of a complete operand with overflow status to floating-point format shall produce floating-point infinity with the same sign. Otherwise conversion of an exceptional complete-format operand (*sNaN*, *qNaN*, $-\infty$, $+\infty$) to floating-point format shall produce a corresponding exceptional floating-point result, with the mantissa equal to the high-order l bits of the fraction part. Conversion of a normal floating-point datum to complete format, or of a complete-format datum to a complete format with values of m and d not less than those of the operand, shall be exact. Conversion of a normal (*exact*, *inexact*) complete format datum to a floating-point format shall round as specified in clause 4 of IEEE 754. Floating-point overflow might signal; if floating-point overflow signals, the result shall be a floating-point infinity of the same sign as the operand. The result of conversion of a normal (*exact, inexact*) complete format datum to a complete format with a value of m less than that of the operand might have overflow status. Conversion of a normal (*exact*, *inexact*) complete format datum to a complete format with a value of d less than that of the operand shall round as specified in Clause 4 of IEEE 754, and the result shall have *inexact* status, unless the result has *overflow* status. If the operand has *inexact* status the result shall have *inexact* status, unless the result has *overflow* status. Bits of the integer and fraction parts that are not derived from the operand shall be zero.

2.2.2. ***Addition and Subtraction***. In addition to the indicators of operand and result formats specified in subclause 5.1 of IEEE 754, this document specifies

- *completeFormatOf* indicates that the name of the operation specifies a complete destination format.

An implementation shall provide the following operations, producing a complete-format result. The operands shall be of complete, floating-point, or integer format, and of the same radix as the result. The result shall be computed using complete arithmetic, as if any operand that is not of complete format, or of a complete format different from the result, were first converted to the same complete format as the result using the **convert** operation.

- *completeFormatOf*-**completeAddition**(*source1*, *source2*)
  The operation **completeAddition**(x, y) computes $x + y$.
- *completeFormatOf*-**completeSubtraction**(*source1*, *source2*)
  The operation **completeSubtraction**(x, y) computes $x - y$.

2.2.3. ***Multiply and Add.*** An implementation shall provide the following operation, producing a complete-format result. The *source1* and *source2* operands shall be of floating-point format, and of the same radix as the result. The *source3* operand shall be a complete-format operand of the same radix as the result. The multiplication shall be computed without loss of any digits, the addition shall be computed using complete arithmetic in the complete format corresponding to the floating-point operands, and the result converted if necessary to the specified result format as if by application of the convert operation.

- *completeFormatOf*-**completeMultiplyAdd**(*source1*, *source2*, *source3*)
  The operation **completeMultiplyAdd**(x, y, z) computes $(x \times y) + z$.

If any of the operands in this operation is an exceptional floating-point datum ($sNaN, qNaN, -\infty, +\infty$) the status field is set appropriately.

3. The Interval Dot Product

Let $\boldsymbol{a}=(\boldsymbol{a}_\nu)$ and $\boldsymbol{b}=(\boldsymbol{b}_\nu)$ be vectors with floating-point interval components $\boldsymbol{a}_\nu=[a_{\nu 1}, a_{\nu 2}]$ and $\boldsymbol{b}_\nu=[b_{\nu 1}, b_{\nu 2}]$, $a_{\nu 1}, a_{\nu 2}, b_{\nu 1}, b_{\nu 2} \in \mathbb{F}$. Computing the dot product requires evaluation of the formulas

$$\begin{aligned}\boldsymbol{a} \diamondsuit \boldsymbol{b} &= ([a_{\nu 1}, a_{\nu 2}]) \diamondsuit ([b_{\nu 1}, b_{\nu 2}]) \\ &= \left[\bigtriangledown \sum_{\nu=1}^{n} \min_{i,j=1,2}(a_{\nu i} b_{\nu j}),\ \bigtriangleup \sum_{\nu=1}^{n} \max_{i,j=1,2}(a_{\nu i} b_{\nu j}) \right].\end{aligned}$$

Here the products of the bounds of the vector components $a_{\nu i} b_{\nu j}$ are to be computed without loss of any digits. Then the minima and maxima have to be selected. These selections can be done by distinguishing the nine cases as usual for interval multiplication. The selected products are then accumulated in complete arithmetic each one as an exact scalar product. Finally the sum of products is rounded only once by $\bigtriangledown$ (resp. $\bigtriangleup$) from the complete format onto $\mathbb{F}$ as specified in clause 4 of IEEE 754.

If any of the components of the interval vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is the empty set, the result of the dot product $\boldsymbol{a} \diamondsuit \boldsymbol{b}$ shall be the empty set.

References

[1] IEEE Floating-Point Arithmetic Standard 754, 2008.
[2] A. Neumaier: *Vienna Proposal for Interval Standardization.*
[3] The *IFIP WG - IEEE 754R letter*, dated September 4, 2007.

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

[5] V. Snyder: *P1788, draft standard*, dated July 20, 2008.

[6] 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).

[7] 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.

[8] 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/`.

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

[10] S. M. Rump: *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Universität Karlsruhe, 1980.

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

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

[13] *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.

Institut für Angewandte und Numerische Mathematik, Universität Karlsruhe, Englerstrasse 2, D-76128 Karlsruhe GERMANY
E-mail address: `Ulrich.Kulisch@math.uka.de`

California Institute of Technology, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Mail Stop 183-701, Pasadena, CA 91109-8099 USA
E-mail address: `Van.Snyder@jpl.nasa.gov`