Foreword

to the GAMM – IMACS Proposal for Accurate Floating-Point Vector Arithmetic[1]

The *Proposal for Accurate Floating-Point Vector Arithmetic* is an official document of GAMM and IMACS. It was unanimously approved by the GAMM Steering Committee at the annual GAMM meeting in April 1993 and subsequently adopted by the IMACS Board of Directors in May 1993. It is a follow-up on the *IMACS-GAMM Resolution on Computer Arithmetic* which was officially approved by GAMM in 1987 and by IMACS in 1988.

The proposal was drafted and developed by the GAMM technical committee for "Computer Arithmetic and Scientific Computing" and the IMACS working group for "Enhanced Computer Arithmetic". Many have contributed to its development either at working group meetings and open discussion sessions held at various international conferences or by commenting on draft versions of the proposal. In the end, it was decided that a rather rigorous and concise formulation of the essential mathematical principles of computer arithmetic would be the most effective way of conveying the basic ideas.

The former *Resolution on Computer Arithmetic* appealed to computer manufacturers to implement compound (vector) operations with extreme care. The new proposal intends to give more guidance to manufacturers who are trying to assess the current needs of the numerical community and who are confronted with many options for future floating-point hardware design. By extending the principles of elementary floating-point arithmetic to the product spaces of computation (vectors, matrices, complex numbers, etc.), the proposal tries to focus future hardware development efforts on the enhancement of existing arithmetic hardware conforming to such standards as the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754). While it remains applicable to a wide range of hardware platforms and floating-point formats, the new proposal is much more explicit than the former resolution by proposing a consistent and simple mathematical model that allows a concise and uniform definition of computer arithmetic. Various software, firmware, and hardware implementations of this model are available, some of which have been in use for more than ten years. Detailed technical propositions for an efficient realization in hardware exist for many diverse platforms.

The ultimate goal of this proposal is to make floating-point computation more *accurate* and *reliable* and, as a consequence, to make numerical algorithms and their results more *portable* across different platforms — in spite of different

[1]Published in Rundbrief der GAMM, 1993 – Brief 2, pp. 9–16, GAMM, 1993 and in Mathematics and Computers in Simulation, vol. 35, no. 4, pp. 375–382, IMACS, 1993

code generation and optimization strategies. This is achieved by giving a rigorous definition of the mathematical behavior of compound vector operations, independent of any particular hardware and software configuration.

The text of the official GAMM and IMACS document follows:[2]

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

Gesellschaft für Angewandte Mathematik und Mechanik (GAMM) &

Int. Assoc. for Mathematics and Computers in Simulation (IMACS)

Proposal for

Accurate Floating-Point Vector Arithmetic

Abstract

Advances in computer technology are now so profound that the arithmetic capability and repertoire of computers can and should be expanded. The roundings and elementary arithmetic operations (including interval roundings and operations) of the quality provided by the IEEE Standards for Floating-Point Arithmetic [2, 3] should be extended to the product spaces of computation (vectors, matrices, complex numbers, etc.). This proposal gives the essential details for doing this. The new, expanded computational capability is gained at modest cost. Remarkably, the new methodology does not implicate a performance penalty. Moreover, techniques are now available so that with this expanded capability, the computer itself can be used to appraise the quality and the reliability of the computed results over a wide range of applications.

Scope

This proposal is an official document of GAMM and IMACS. It defines the mathematical properties that arithmetic floating-point operations, especially compound vector operations, should satisfy (see also [17]). It can be viewed as an extension of the approach of the IEEE Standards for Floating-Point Arithmetic (754 and 854) [2, 3] from purely scalar to vector/matrix arithmetic [6, 7, 8, 12, 22, 25, 26, 27]. However, rather than being prescriptive to the bit-level, this proposal applies to many hardware platforms and floating-point formats. It may also be used to specify the *mathematical behavior* of arithmetic operations in *programming languages.* A system equipped with the proposed arithmetic is an ideal platform upon which *automatic result verification* techniques can be realized [1, 16, 21, 23, 24, 36, 37].

The principal goals of this proposal are the *reliability* and *security* of numerical results. Part of the proposed features may be provided in software, although good hardware support is strongly recommended. Existing hardware and software implementations demonstrate that accurate floating-point vector arithmetic is feasible using current technology — at modest cost and generally without incurring a runtime penalty.

Definition of Computer Arithmetic

A **floating-point format** is defined by its base b (radix), its precision p (mantissa length), and its minimal and maximal exponent, $emin$ and $emax$. These four parameters define a floating-point format $F(b, p, emin, emax)$ and thus the finite set of real numbers representable in that format, the **floating-point numbers** F. For convenience, it is assumed that a particular floating-point format is given.

A **rounding** $rnd(x)$ is a *monotonic nondecreasing function* from the real numbers $\mathbf{R}$ onto the set of floating-point numbers $F \subset \mathbf{R}$ which leaves all elements of F unchanged. Monotonicity eliminates erratic behavior of a rounding. Formally, the following two properties must hold:

$$rnd(x) = x \qquad \text{if } x \in F \qquad \textit{(projection)} \quad (1)$$

$$x \leq y \;\Rightarrow\; rnd(x) \leq rnd(y) \qquad \text{for } x, y \in \mathbf{R} \qquad \textit{(monotonicity)} \quad (2)$$

Commonly used roundings are also *antisymmetric*:

$$rnd(-x) = -rnd(x) \qquad \text{for all } x \in \mathbf{R} \qquad \textit{(antisymmetry)} \quad (3)$$

Note that (3) imposes the property of symmetry on F, so if $x \in F$, then $-x \in F$. Examples of **antisymmetric roundings** are the rounding *toward zero* (*truncation*), the rounding *away from zero* (*augmentation*), and certain roundings *to nearest*. There are several ways to define a rounding to the nearest floating-point number.

Other important roundings are the **directed roundings** *toward* $-\infty$ (*downward*, denoted by ∇) and *toward* $+\infty$ (*upward*, denoted by Δ), which are both uniquely defined by (1), (2), and the additional property

$$\nabla x \leq x \leq \Delta x \qquad \text{for all } x \in \mathbf{R} \qquad \textit{(directed rounding)} \quad (4)$$

An exponent overflow could be mapped to $+\infty$ or $-\infty$, as the case may be. The directed roundings are not antisymmetric, but satisfy the rule $\nabla(-x) = -\Delta(x)$.

Roundings are defined in exactly the same way for **complex numbers** and for the basic **product spaces** (the real and complex vectors and matrices). In all of these spaces, the order relation $\leq$ is defined *componentwise*, inducing a partial ordering on these sets. If T denotes one of these sets and S its computer-representable subset, a rounding from T onto S fulfills the same properties (1), (2), (3) with $\mathbf{R}$ replaced by T and F by S (e. g., if $T = \mathbf{R}^n$, then $S = F^n$):

$$\begin{aligned} rnd(x) &= x && \text{if } x \in S && \textit{(projection)} && (1') \\ x \leq y \;\Rightarrow\; rnd(x) &\leq rnd(y) && \text{for } x, y \in T && \textit{(monotonicity)} && (2') \\ rnd(-x) &= -rnd(x) && \text{for all } x \in T && \textit{(antisymmetry)} && (3') \end{aligned}$$

Note that (3') imposes the property of symmetry on S. The theory of computer arithmetic shows that these roundings are equivalent to applying the analogous roundings to the individual vector or matrix components or to the real and the imaginary part of a complex number separately [26, 27].

For the corresponding **interval spaces** — the intervals over the real and the complex numbers as well as the intervals over the real and complex vectors and matrices — the order relation is the *subset* relation $\subseteq$ (in (2')). A rounding from any interval set T onto its computer-representable subset S is defined by properties (1'), (2') (with $\leq$ replaced by $\subseteq$), plus the additional property

$$x \subseteq rnd(x) \qquad \text{for all } x \in T \qquad \textit{(inclusion)} \qquad (4')$$

These interval roundings are also antisymmetric, that is, they satisfy (3').

When an arithmetic operation in one of the basic spaces (the real and complex numbers, vectors and matrices and the corresponding interval spaces) is modelled on the computer, it shall be defined as follows.

Let T be one of the spaces under consideration in which arithmetic operations are defined, and let S be its computer-representable subset. If $\circ$ is an arithmetic operation in T, the corresponding *computer operation* $\boxed{\circ}$ in S is given by

$$r \boxed{\circ} s := rnd(r \circ s) \qquad \text{for all } r, s \in S \qquad \textit{(rounded operation)} \qquad (5)$$

where rnd is a rounding. That is, the computer operation must be performed as if the exact result were first computed and then rounded with the selected rounding. The rounding rnd uniquely determines the (rounded) computer operation $\boxed{\circ}$. With the exception of ∇ and Δ, commonly used roundings are *antisymmetric*. Recall that in the case of intervals, the rounding must additionally satisfy the inclusion requirement (4'). A mapping from T onto S satisfying (1'), (2'), (3'), and (5) (and (4') in the case of intervals) is called a **semimorphism**. Many properties of both the order structure and the algebraic structure are invariant under a semimorphism. For more details, refer to [26, 27].

It is easily seen that property (5) guarantees that all arithmetic operations are accurate to at least 1 ulp (unit in the last place). 1/2 ulp is achieved in the case of rounding *to nearest*. This accuracy is achieved by the elementary floating-point operations defined by several implementations including those conforming to one of the IEEE standards (754 or 854) [2, 3].

A careful analysis shows that all the arithmetic operations under consideration (with one exception) can be expressed in terms of the four elementary arithmetic operations $+, -, *, /$ for floating-point numbers plus *one* additional operation (which is a *compound* operation), the (accurate) **dot product** (scalar product, inner product) of two vectors with floating-point components (denoted by $\cdot$). This fifth operation is defined in the box below. To implement the corresponding interval operations, the theory of computer arithmetic shows that the same five operations $(+, -, *, /, \cdot)$ with the directed roundings *toward* $-\infty$ (*downward*, ∇) and *toward* $+\infty$ (*upward*, Δ) are sufficient. The exception, complex (interval) division, can be performed iteratively [26, 27, 29].

Summarizing, semimorphic operations for real and complex numbers, vectors and matrices as well as for real and complex intervals, interval vectors and interval matrices can be realized in terms of the 15 fundamental arithmetic operations of floating-point arithmetic: $+, -, *, /, \cdot$, each with the roundings $\square$, ∇, and Δ, where $\square$ stands for an *antisymmetric* rounding (typically *to nearest*). The symbols $\boxed{\circ}$, $\boxed{\nabla}$, and $\boxed{\Delta}$ (where $\circ \in \{+, -, *, /, \cdot\}$) represent the

operations defined by (5) using an *antisymmetric* rounding $\square$ (*semimorphism*), the rounding *downward* ∇, and the rounding *upward* Δ, respectively.

These 15 fundamental operations are found in a variety of programming systems [14, 15, 16, 23, 24]. Twelve of the 15 are provided by the IEEE standards [2, 3]. By adding just three more, the arithmetic operations in all the basic product spaces of mathematics can be performed with 1 (or 1/2) ulp accuracy (in each component).

In numerical analysis, the dot product is ubiquitous. It appears in complex arithmetic, vector and matrix arithmetic, multiple precision arithmetic, and iterative refinement techniques, among many other places.

To complete the specification of computer arithmetic, we now give a formal definition of an **accurate dot product operation**.

Given two vectors x and y with n floating-point components each and a prescribed rounding rnd, the floating-point result s of the **dot product operation** (applied to x and y) is defined by

$$s := rnd(\overline{s}) := rnd(x \cdot y) = rnd\left(\sum_{i=1}^{n} x_i * y_i\right) \ ,$$

where all arithmetic operations are exact. In other words, s shall be computed as if an intermediate result $\overline{s} = x \cdot y$ correct to infinite precision and with unbounded exponent range were first produced and then rounded to the desired floating-point destination format according to the selected rounding rnd.

This definition of the dot product operation guarantees highest possible accuracy (for the given rounding and the given floating-point destination format). It is analogous to the definition of scalar floating-point arithmetic given in the IEEE standards [2, 3].

It is important to note that the order in which the elementary operations are performed when determining the dot product is not specified, and that this accomodates *pipelined* (and potentially *parallel*) processing. Reordering the summands has no influence on the computed result since all intermediate results are exact.

By way of contrast, note that a traditional computation of the dot product of two vectors with n components each (in ordinary floating-point arithmetic with rounded multiplications and additions) involves $2n-1$ roundings. If catastrophic cancellation occurs, a large number of significant digits may be lost [13, 35]. This may happen even if an extended precision data format is used for the accumulation. Loss of accuracy aside, a considerable amount of processing time may be required to perform gratuitous intermediate steps such as composition, decomposition, normalization, and rounding of intermediate floating-point values. Furthermore, unnecessary load and store operations may have to be performed in the traditional mode.

Implementation Options

Implementations of the elementary floating-point operations with different roundings are commonplace. The computer realization of the new dot product operations can be achieved in several ways [14, 8]. A natural way of adding n numbers or products is via *fixed-point* accumulation. The scaling which accompanies fixed-point computation is eliminated by a hardware implementation which has been in use for more than ten years. This device employs a **long fixed-point register** (**accumulator**) covering twice the exponent range of the floating-point format in which the vector components are given (see for instance [14]). If a relatively small number of extra digits is provided at the front (high-significance) end of such a register to collect extra carries, the intermediate result in such a register is always exact and cannot overflow or underflow for any feasible number of summands. Only one rounding occurs, and this at the end of the accumulation process. This method has the advantage of being rather simple and straightforward, and it always provides the desired accurate answer.

Existing implementations and detailed studies show that a full hardware implementation of the *long register* is routine and inexpensive in units of additional silicon and can be made to perform as fast as conventional floating-point accumulation. The fundamental operations that are needed are: *initialization* of the long register, *addition of a product* of two floating-point numbers to the register, and *rounding* the contents of the register to a floating-point number (using the prescribed rounding). Other desirable operations include: addition of a floating-point number to the register, addition, and comparison of two long registers. Existing programming language extensions provide and exploit these operations [14, 15, 16, 23, 24].

A number of studies in search of a solution that will require less silicon than a long register are under way. One alternative is a **window** (**short register**) where only a segment of the long register is actually provided in hardware. This may be justified since computations involving very large or very small exponents rarely occur. In its simplest form, the window is a segment of fixed width covering a fixed range of exponents. A more sophisticated variant allows the window to be shifted to the left (to accomodate larger intermediate values), but never to the right, thus maintaining a window on the most significant part of the dot product. However, any summand that does not fit fully requires costly extra treatment, most likely through special software.

There are well-known **iterative methods** for computing sums and dot products accurately which use floating-point arithmetic only [5, 33]. One of these proceeds by the successive replacement of the entire set of summands, using a correction technique. In a finite number of steps, the set of summands is driven to a state where no summand overlaps another in a fixed-point sense, and so the summation is correctly completed. It is important to note and exploit the fact that the correctly rounded sum may be available much earlier in the process, often after one iteration. See [5] for an example of a stopping criterion. These methods require *exact* operations $+, -, *$ which can be furnished by representing the result with two floating-point numbers, a high-order and a low-order

part [9]. However, these iterative methods have some major disadvantages: the number of iterations and the amount of memory required to obtain an accurate answer are not known a priori, and intermediate overflow and underflow cannot be avoided without a significant degradation of performance.

¿From the users' point of view, the well-defined, predictable behavior of the *long register* is highly desirable. Any *partial* hardware support requires complementary software. Manufacturers who decline to provide a full implementation of the accurate dot product in hardware must be aware of the fact that there may be a substantial performance penalty in essential applications.

References

[1] Adams, E.; Kulisch, U. (eds.): *Scientific Computing with Automatic Result Verification.* Academic Press, Orlando, 1992.

[2] ANSI/IEEE: *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754–1985, New York, 1985.

[3] ANSI/IEEE: *IEEE Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std 854–1987, New York, 1987.

[4] Bleher, J. H.; Rump, S. M.; Kulisch, U.; Metzger, M.; Ullrich, Ch.; Walter, W.: *FORTRAN–SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH.* Computing **39**, 93–110, Springer-Verlag, 1987.

[5] Bohlender, G.: *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen.* Ph. D. thesis, Univ. Karlsruhe, 1978.

[6] Bohlender, G.: *What Do We Need Beyond IEEE Arithmetic?* In [36], 1–32, 1990.

[7] Bohlender, G.: *A Vector Extension of the IEEE Standard for Floating-Point Arithmetic.* In [21], 3–12, 1991.

[8] Bohlender, G.; Knöfel, A.: *A Survey of Pipelined Hardware Support for Accurate Scalar Products.* In [21], 29–43, 1991.

[9] Bohlender, G.; Kornerup, P.; Matula, D. W.; Walter, W. V.: *Semantics for Exact Floating Point Operations.* Proc. of 10th IEEE Symp. on Computer Arithmetic (ARITH 10) in Grenoble, 22–26, IEEE Comp. Soc., 1991.

[10] Cordes, D.: *Runtime System for a PASCAL–XSC Compiler.* In [21], 151–160, 1991.

[11] Dekker, T. J.: *A Floating-Point Technique for Extending the Available Precision.* Numerical Mathematics **18**, 224–242, 1971.

[12] Hahn, W.; Mohr, K.: *APL/PCXA, Erweiterung der IEEE Arithmetik für technisch wissenschaftliches Rechnen.* Hanser Verlag, München, 1989.

[13] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In [37], 467–482, 1990.

[14] IBM: *System/370 RPQ, High-Accuracy Arithmetic.* SA22-7093-0, IBM Corp., 1984.

[15] IBM: *High-Accuracy Arithmetic Subroutine Library (ACRITH),* General Information Manual. 3rd ed., GC33-6163-02, IBM Corp., 1986.

[16] IBM: *High Accuracy Arithmetic – Extended Scientific Computation (ACRITH–XSC),* General Information. GC33-6461-01, IBM Corp., 1990.

[17] 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, p. T5, 1990; in [36], 301–302, 1990; in [37], 523–524, 1990; in [21], 477–478, 1991.

[18] ISO: *Language Independent Arithmetic Standard (LIA-1).* Second Committee Draft Standard (Version 4.0), ISO/IEC CD 10967-1, 1992.

[19] Kahan, W.: *Further Remarks on Reducing Truncation Errors.* Comm. ACM **8**, no. 1, 40, 1965.

[20] Kahan, W.: *Doubled Precision IEEE Standard 754 Floating-Point Arithmetic.* Mini-Course on "The Regrettable Failure of Automated Error Analysis", Conf. on Computers and Mathematics, MIT, June 13, 1989.

[21] Kaucher, E.; Markov, S. M.; Mayer, G. (eds): *Computer Arithmetic, Scientific Computation and Mathematical Modelling.* IMACS Annals on Computing and Applied Mathematics **12**, J.C. Baltzer, Basel, 1991.

[22] Kirchner, R.; Kulisch, U.: *Arithmetic for Vector Processors.* Proc. of 8th IEEE Symp. on Computer Arithmetic (ARITH 8) in Como, 256–269, IEEE Comp. Soc., 1987.

[23] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: *C–XSC: A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[24] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL–XSC Sprachbeschreibung mit Beispielen.* Springer-Verlag, Berlin, Heidelberg, New York, 1991.

. . . : *PASCAL–XSC Language Reference with Examples.* Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[25] Knöfel, A.: *Fast Hardware Units for the Computation of Accurate Dot Products.* Proc. of 10th IEEE Symp. on Computer Arithmetic (ARITH 10), 70–74, IEEE Comp. Soc., 1991.

[26] Kulisch, U.: *Grundlagen des numerischen Rechnens: Mathematische Begründung der Rechnerarithmetik.* Reihe Informatik **19**, Bibl. Inst., Mannheim, 1976.

[27] Kulisch, U.; Miranker, W. L.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[28] Linnainmaa, S.: *Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums.* BIT **14**, 167–202, 1974.

[29] Lohner, R.; Wolff v. Gudenberg, J.: *Complex Interval Division with Maximum Accuracy.* Proc. of 7th IEEE Symp. on Computer Arithmetic (ARITH 7) in Urbana, Illinois, 332–336, IEEE Comp. Soc., 1985.

[30] Møller, O.: *Quasi Double Precision in Floating-Point Addition.* BIT **5**, 37–50, 1965.

[31] Müller, M.; Rüb, Ch.; Rülling, W.: *Exact Addition of Floating Point Numbers.* Sonderforschungsb. 124, FB 14, Informatik, Univ. des Saarlandes, Saarbrücken, 1990.

[32] NAG: *Basic Linear Algebra Subprograms (BLAS).* The Numerical Algorithms Group Ltd, Oxford, 1990.

[33] Pichat, M.: *Correction d'une somme en arithmétique à virgule flottante.* Numer. Math. **19**, 400–406, 1972.

[34] Priest, D. M.: *Algorithms for Arbitrary Precision Floating Point Arithmetic.* Proc. of 10th IEEE Symp. on Computer Arithmetic (ARITH 10), 132–143, IEEE Comp. Soc., 1991.

[35] Ratz, D.: *The Effects of the Arithmetic of Vector Computers on Basic Numerical Methods.* In [37], 499–514, 1990.

[36] Ullrich, Ch. (ed.): *Computer Arithmetic and Self-Validating Numerical Methods.* Academic Press, San Diego, 1990.

[37] Ullrich, Ch. (ed.): *Contributions to Computer Arithmetic and Self-Validating Numerical Methods.* IMACS Annals on Computing and Applied Mathematics **7**, J.C. Baltzer, Basel, 1990.