Mathematics and Speed for Interval Arithmetic
A Complement to IEEE P1788

Ulrich Kulisch

Institut für Angewandte und Numerische Mathematik
Karlsruher Institut für Technologie
Kaiserstrasse 12
D-76128 Karlsruhe GERMANY
Ulrich.Kulisch@kit.edu

Abstract. The paper begins with an axiomatic definition of rounded arithmetic. The concepts of rounding and of rounded arithmetic operations are defined in an axiomatic manner fully independent of special data formats and encodings. Basic properties of floating-point and interval arithmetic can directly be derived from this abstract model. Interval operations are defined as set operations for elements of the set $\overline{\mathbb{IR}}$ of closed and connected sets of real numbers. As such they form an algebraically closed subset of the powerset of the real numbers. This property leads to explicit formulas for the arithmetic operations of floating-point intervals of $\overline{\mathbb{IF}}$, which are executable on the computer. Arithmetic for intervals of $\overline{\mathbb{IF}}$ forms an exception free calculus, i.e., arithmetic operations for intervals of $\overline{\mathbb{IF}}$ always lead to intervals of $\overline{\mathbb{IF}}$ again.

Later sections are concerned with programming support and hardware for interval arithmetic. Both are a must and absolutely necessary to move interval arithmetic more into the center of scientific computing. With some minor hardware additions interval operations can be made as fast as simple floating-point operations. Section 6 shows that nearly everything that is needed for high speed interval arithmetic is already available on current x86-processors for multimedia applications (parallel operations like $+, -, \cdot, /, min, max$, and *compare*, as well as data paths for pairs of numbers).

In vector and matrix spaces for real, complex, and interval data the dot product is a fundamental arithmetic operation. Computing the dot product of two vectors with floating-point components **exactly** substantially speeds up floating-point and interval arithmetic as well as the accuracy of the computed result. Hardware needed for the exact dot product is very modest. The exact dot product is essential for long real and long interval arithmetic.

Section 9 illustrates that interval arithmetic as developed in this paper has already a long tradition. Products based on these ideas have been available since 1980. Implementing what the paper advocates would have a profound effect on mathematical software. Modern processor architecture comes quite close to what is requested in this paper [36].

1 Introduction

A standard for floating-point arithmetic IEEE 754 was established in 1985. A revision was published in 2008. A standard for interval arithmetic IEEE P1788 is just under development. While these two standards consider data formats, encodings, implementation, exception handling, and so on, this study focuses on the mathematics of the arithmetic for computers and on hardware support for the basic arithmetic operations. As usual mathematics is suited to focus the understanding on the essentials. High speed by hardware support is a must and vital for interval arithmetic to be more widely accepted by the scientific computing community.

Interval mathematics has been developed to a high standard over the last few decades. It provides methods which deliver results with guarantees. However, the arithmetic on existing processors makes these methods very slow. This paper deals with the question of how interval arithmetic can be provided effectively on computers. With more suitable processors, rigorous methods based on interval arithmetic could be comparable in speed to today's approximate methods. Interval arithmetic is a natural extension to floating-point arithmetic, not a replacement for it. As computers speed up, from gigaflops to teraflops to petaflops, interval arithmetic becomes a principal and necessary tool for controlling the precision of a computation as well as the accuracy of the computed result.

Central for the considerations in this paper are operations for intervals of $\overline{\mathbb{IR}}$. These are closed and connected sets of real numbers. Arithmetic for intervals of $\overline{\mathbb{IF}}$ leads to a calculus that is free of exceptions, i.e., arithmetic operations for intervals of $\overline{\mathbb{IF}}$ always lead to intervals of $\overline{\mathbb{IF}}$ again. In $\overline{\mathbb{IR}}$ and $\overline{\mathbb{IF}}$ division by an interval that includes zero is defined. With it the Newton method reaches its ultimate elegance and power. The *extended interval Newton method* yields enclosures of all zeros of a function in a given domain. It is globally convergent and never fails, not even in rounded arithmetic [7, 23].

In vector and matrix spaces[1] the dot product of two vectors is a fundamental arithmetic operation in addition to the four operations $+, -, \cdot, /$ for the components. It is fascinating that this fundamental arithmetic operation is also a mean to increase the speed of computing as well as the accuracy of the computed result. Actually the simplest and fastest way for computing a dot product is to compute it exactly. Here the products are just shifted and added into a wide fixed-point register. By pipelining, the exact dot product can be computed in the time the processor needs to read the data, i.e., it comes with utmost speed. Any method that just computes a correctly rounded dot product also has to consider the values of the summands. This results in a more complicated method. The hardware needed for the exact dot product is comparable to that for a fast multiplier by an adder tree, accepted years ago and now standard technology in every modern processor. The exact dot product brings the same speedup for accumulations at comparable costs.

The reader may be well advised to begin reading this paper with section 9.

2 Axiomatic Definition of Rounded Arithmetic

Frequently mathematics is seen as the science of structures. Analysis carries three kinds of structures: an algebraic structure, an order structure, and a topological or metric structure. These are coupled by certain compatibility properties, as for instance: $a \le b \Rightarrow a + c \le b + c$.

It is well known that floating-point numbers and floating-point arithmetic do not obey the rules of the real numbers $\mathbb{R}$. However, the rounding is a monotone function. So the changes to the order structure are minimal. This is the reason why the order structure plays a key role for an axiomatic definition of computer arithmetic.

This study is restricted to the two elementary models that are covered by the two IEEE arithmetic standards, computer arithmetic on the reals and on real intervals. Abstract settings of computer arithmetic for higher dimensional spaces like complex numbers, vectors and matrices for real, complex, and interval data can be developed following similar schemes. For more details see [23] and the literature cited there.

We begin by listing a few well-known concepts and properties of ordered sets.

Definition 1. *A relation $\le$ in a set M is called an* order relation*, and $\{M, \le\}$ is called an* ordered set[2] *if for all $a, b, c \in M$ the following properties hold:*

(O1) $a \le a$, (reflexivity)
(O2) $a \le b \ \wedge \ b \le c \ \Rightarrow \ a \le c$, (transitivity)
(O3) $a \le b \ \wedge \ b \le a \ \Rightarrow \ a = b$, (antisymmetry)

An ordered set M is called linearly *or* totally ordered *if in addition*

(O4) $a \le b \vee b \le a$ *for all* $a, b \in M$. (linearly ordered)

An ordered set M is called

(O5) *a* lattice *if for any two elements $a, b \in M$, the* $\inf\{a, b\}$ *and the* $\sup\{a, b\}$ *exist.* (lattice)

(O6) *It is called* conditional completely ordered *if for every bounded subset $S \subseteq M$, the* $\inf S$ *and the* $\sup S$ *exist.*

(O7) *An ordered set M is called* completely ordered *or a* complete lattice *if for every subset $S \subseteq M$, the* $\inf S$ *and the* $\sup S$ *exist.* (complete lattice)

[1] for real, complex, interval, and complex interval data
[2] Occasionally called a partially ordered set.

With these concepts the real numbers $\{\mathbb{R}, \leq\}$ are defined as a conditional complete linearly ordered field.

In the definition of a complete lattice, the case $S = M$ is included. Therefore, $\inf M$ and $\sup M$ exist. Since they are elements of M, every complete lattice has a least and a greatest element.

If a subset $S \subseteq M$ of a complete lattice $\{M, \leq\}$ is also a complete lattice, $\{S, \leq\}$ is called a *complete sublattice* of $\{M, \leq\}$ if the two lattice operations inf and sup in both sets lead to the same result, i.e., if

$$\text{for all} \quad A \subseteq S, \quad \inf_M A = \inf_S A \quad \text{and} \quad \sup_M A = \sup_S A.$$

Definition 2. *A subset S of a complete lattice $\{M, \leq\}$ is called a* screen *of M, if every element $a \in M$ has upper and lower bounds in S and the set of all upper and lower bounds of $a \in M$ has a least and a greatest element in S respectively. If a minus operator exists in M, a screen is called* symmetric*, if for all $a \in S$ also $-a \in S$.*

As a consequence of this definition a complete lattice and a screen have the same least and greatest element. It can be shown that a screen is a complete sublattice of $\{M, \leq\}$ with the same least and greatest element, [23].

Definition 3. *A mapping $\square : M \to S$ of a complete lattice $\{M, \leq\}$ onto a screen S is called a* rounding *if (R1) and (R2) hold:*

(R1) *for all* $a \in S, \quad \square\, a := a.$ (projection)

(R2) $a \leq b \Rightarrow \square\, a \leq \square\, b.$ (monotone)

A rounding is called downwardly directed *resp.* upwardly directed *if for all $a \in M$*

(R3) $\square\, a \leq a \quad$ *resp.* $\quad a \leq \square\, a.$ (directed)

If a minus operator is defined in M, a rounding is called antisymmetric *if*

(R4) $\square\,(-a) = -\square\, a, \quad$ *for all* $a \in M.$ (antisymmetric)

The monotone downwardly resp. upwardly directed roundings of a complete lattice onto a screen are unique. For the proof see [23].

Definition 4. *Let $\{M, \leq\}$ be a complete lattice and $\circ : M \times M \to M$ a binary arithmetic operation in M. If S is a screen of M, then a rounding $\square : M \to S$ can be used to approximate the operation $\circ$ in S by*

(RG) $a \boxdot b := \square\,(a \circ b)$, *for* $a, b \in S.$

If a minus operator is defined in M and S is a symmetric screen of M, then a mapping $\square : M \to S$ with the properties (R1,2,4) and (RG) is called a semimorphism[3]*.*

Semimorphisms with antisymmetric roundings are particularly suited for transferring properties of the structure in M to the subset S. It can be shown [23] that semimorphisms leave a number of reasonable properties of ordered algebraic structures (ordered field, ordered vector space) invariant.

If an element $x \in M$ is bounded by $a \leq x \leq b$ with $a, b \in S$, then by (R1) and (R2) the rounded image $\square\, x$ is bounded by the same elements: $a \leq \square\, x \leq b$, i.e., $\square\, x$ is either the least upper (supremum) or the greatest lower (infimum) bound of x in S. Similarly, if for $x, y \in S$ the result of an operation $x \circ y$ is bounded by $a \leq x \circ y \leq b$ with $a, b \in S$, then by (R1), (R2), and (RG) also $a \leq x \boxdot y \leq b$, i.e., $x \boxdot y$ is either the least upper or the greatest lower bound of $x \circ y$ in S. If the rounding is upwardly or downwardly directed the result is the least upper or the greatest lower bound respectively.

[3] The properties (R1,2,4) and (RG) of a semimorphism can be shown to be necessary conditions for a homomorphism between ordered algebraic structures. For more details see [23]

3 Two Elementary Models

3.1 Floating-Point Arithmetic

The set $\{\overline{\mathbb{R}}, \leq\}$ with $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$ is a complete lattice. Let $\mathbb{F}$ denote the set of finite floating-point numbers and $\overline{\mathbb{F}} := \mathbb{F} \cup \{-\infty, +\infty\}$. Then $\overline{\mathbb{F}}$ is a screen of $\{\overline{\mathbb{R}}, \leq\}$. The least element of the set $\overline{\mathbb{R}}$ and of the subset $\overline{\mathbb{F}}$ is $-\infty$ and the greatest element is $+\infty$.

Definition 5. *With a rounding* $\square : \overline{\mathbb{R}} \to \overline{\mathbb{F}}$ *arithmetic operations* $\boxdot$ *in* $\overline{\mathbb{F}}$ *are defined by*

(RG) $a \boxdot b := \square (a \circ b)$, *for* $a, b \in \overline{\mathbb{F}}$ *and* $\circ \in \{+, -, *, /\}$,

with $b \neq 0$ *in case of division.*[4]

If a and b are adjacent floating-point numbers and $x \in \mathbb{R}$ with $a \leq x \leq b$, then because of (R1) and (R2) also $a \leq \square x \leq b$, i.e., there is never an element of $\mathbb{F}$ between an element $x \in \mathbb{R}$ and its rounded image $\square x$. The same property holds for the operations defined by (RG): If for $x, y \in \mathbb{F}$, $a \leq x \circ y \leq b$ then by (R1), (R2), and (RG) also $a \leq x \boxdot y \leq b$, for all $\circ \in \{+, -, *, /\}$, i.e., $x \boxdot y$ is either the greatest lower or the least upper bound of $x \circ y$ in $\mathbb{F}$.

Frequently used roundings $\square : \overline{\mathbb{R}} \to \overline{\mathbb{F}}$ are antisymmetric. Examples are the rounding to the nearest floating-point number, the rounding toward zero, or the rounding away from zero. A semimorphism transfers a number of useful properties of the real numbers to the floating-point numbers. The mathematical structure of $\mathbb{F}$ can even be defined as properties of $\mathbb{R}$ which are invariant with respect to semimorphism, [23].

For the monotone downwardly resp. upwardly directed roundings of $\overline{\mathbb{R}}$ onto $\overline{\mathbb{F}}$ often the special symbols $\bigtriangledown$ resp. $\bigtriangleup$ are used. These roundings are not antisymmetric. They are related by the property:

$$\bigtriangledown (-a) = - \bigtriangleup a \quad \text{and} \quad \bigtriangleup (-a) = - \bigtriangledown a. \tag{1}$$

Arithmetic operations defined by (RG) and these roundings are denoted by $\bigtriangledown$ and $\bigtriangleup$, respectively, for $\circ \in \{+, -, *, /\}$. These are heavily used in interval arithmetic.

3.2 Interval Arithmetic

The set of nonempty, closed and bounded real intervals is denoted by $\mathbb{IR}$. An interval of $\mathbb{IR}$ is denoted by an ordered pair $[a_1, a_2]$, with $a_1, a_2 \in \mathbb{R}$. The first element is the lower bound and the second is the upper bound.

Interval arithmetic over the real numbers deals with *closed*[5] and connected sets of real numbers. Such intervals can be bounded or unbounded. For the notation of unbounded intervals $-\infty$ and $+\infty$ are used as bounds. Since $-\infty$ and $+\infty$ are not real numbers a bound $-\infty$ or $+\infty$ of an unbounded interval is not an element of the interval. Thus unbounded real intervals are frequently written as $(-\infty, a]$ or $[b, +\infty)$ or $(-\infty, +\infty)$ with $a, b \in \mathbb{R}$.

The set of bounded and unbounded real intervals is denoted by $\overline{\mathbb{IR}}$. With respect to the subset relation as an order relation the set of real intervals $\{\overline{\mathbb{IR}}, \subseteq\}$ is a complete lattice. The subset of $\overline{\mathbb{IR}}$ where all finite bounds are floating-point numbers of $\mathbb{F}$ is denoted by $\overline{\mathbb{IF}}$. $\{\overline{\mathbb{IF}}, \subseteq\}$ is a screen of $\{\overline{\mathbb{IR}}, \subseteq\}$. In both sets $\overline{\mathbb{IR}}$ and $\overline{\mathbb{IF}}$ the infimum of a subset of $\overline{\mathbb{IF}}$ is the intersection and the supremum is the interval hull[6]. The least element of both sets $\overline{\mathbb{IR}}$ and $\overline{\mathbb{IF}}$ is the empty set $\varnothing$ and the greatest element is the set $\mathbb{R} = (-\infty, +\infty)$.

Definition 6. *For intervals* $\boldsymbol{a}, \boldsymbol{b} \in \overline{\mathbb{IR}}$ *arithmetic operations* $\circ \in \{+, -, *, /\}$ *are defined as set operations*

$$\boldsymbol{a} \circ \boldsymbol{b} := \{a \circ b \mid a \in \boldsymbol{a} \wedge b \in \boldsymbol{b}\}. \tag{2}$$

Here for division we assume that $\boldsymbol{b} \neq [0, 0]$. $\boldsymbol{a}/[0, 0]$ *is defined to be the empty set* $\varnothing$ *for any* $\boldsymbol{a} \in \overline{\mathbb{IR}}$.

[4] In real analysis division by zero is not defined. It does not lead to a real number.

[5] A set of real numbers is called *closed*, if its complement in $\mathbb{R}$ is open. A set $M \subset \mathbb{R}$ is called *open* if for all $a \in M$ also an ϵ-neighborhood $(a - \epsilon, a + \epsilon)$ is entirely in M.

[6] Which is the convex hull in the one dimensional case.

For $\boldsymbol{a} = [a_1, a_2] \in \overline{\mathbb{IR}}$ we obtain by (2) immediately

$$-\boldsymbol{a} := {}^{7}(-1)*\boldsymbol{a} = [-a_2, -a_1] \in \overline{\mathbb{IR}}. \tag{3}$$

If in (3) $\boldsymbol{a} \in \overline{\mathbb{IF}}$, then also $-\boldsymbol{a} \in \overline{\mathbb{IF}}$, i.e., $\overline{\mathbb{IF}}$ is a symmetric screen of $\overline{\mathbb{IR}}$.

Between the complete lattice $\{\overline{\mathbb{IR}}, \subseteq\}$ and its screen $\{\overline{\mathbb{IF}}, \subseteq\}$ the monotone upwardly directed rounding $\diamondsuit : \overline{\mathbb{IR}} \to \overline{\mathbb{IF}}$ is uniquely defined. It is characterized by the following properties:

(R1) $\diamondsuit\, \boldsymbol{a} = \boldsymbol{a}$, for all $\boldsymbol{a} \in \overline{\mathbb{IF}}$. (projection)
(R2) $\boldsymbol{a} \subseteq \boldsymbol{b} \Rightarrow \diamondsuit\, \boldsymbol{a} \subseteq \diamondsuit\, \boldsymbol{b}$, for $\boldsymbol{a}, \boldsymbol{b} \in \overline{\mathbb{IR}}$. (monotone)
(R3) $\boldsymbol{a} \subseteq \diamondsuit\, \boldsymbol{a}$, for all $\boldsymbol{a} \in \overline{\mathbb{IR}}$. (upwardly directed)

For $\boldsymbol{a} = [a_1, a_2] \in \overline{\mathbb{IR}}$ the result of the monotone upwardly directed rounding $\diamondsuit$ is

$$\diamondsuit\, \boldsymbol{a} = [\, \triangledown\, a_1,\ \triangle\, a_2]. \tag{4}$$

Using (1) and (2) it is easy to see that the monotone upwardly directed rounding $\diamondsuit : \overline{\mathbb{IR}} \to \overline{\mathbb{IF}}$ is antisymmetric, i.e.,

(R4) $\diamondsuit\, (-\boldsymbol{a}) = -\diamondsuit\, \boldsymbol{a}$, for all $\boldsymbol{a} \in \overline{\mathbb{IR}}$. (antisymmetric).

An interval $\boldsymbol{a} = [a_1, a_2]$ is frequently interpreted as a point in $\mathbb{R}^2$. This very naturally induces the order relation $\leq$ of $\mathbb{R}^2$ to the set of intervals $\overline{\mathbb{IR}}$. For two intervals $\boldsymbol{a} = [a_1, a_2]$ and $\boldsymbol{b} = [b_1, b_2]$ the relation $\leq$ is defined by $\boldsymbol{a} \leq \boldsymbol{b} :\Leftrightarrow a_1 \leq b_1 \wedge a_2 \leq b_2$.

For the $\leq$ relation for intervals compatibility properties hold between the algebraic structure and the order structure in great similarity to the real numbers. For instance:

(OD1) $\boldsymbol{a} \leq \boldsymbol{b} \Rightarrow \boldsymbol{a} + \boldsymbol{c} \leq \boldsymbol{b} + \boldsymbol{c}$, for all $\boldsymbol{c}$.
(OD2) $\boldsymbol{a} \leq \boldsymbol{b} \Rightarrow -\boldsymbol{b} \leq -\boldsymbol{a}$.
(OD3) $[0,0] \leq \boldsymbol{a} \leq \boldsymbol{b} \wedge \boldsymbol{c} \geq [0,0] \Rightarrow \boldsymbol{a} * \boldsymbol{c} \leq \boldsymbol{b} * \boldsymbol{c}$.
(OD4) $[0,0] < \boldsymbol{a} \leq \boldsymbol{b} \wedge \boldsymbol{c} > [0,0] \Rightarrow [0,0] < \boldsymbol{a}/\boldsymbol{c} \leq \boldsymbol{b}/\boldsymbol{c} \wedge \boldsymbol{c}/\boldsymbol{a} \geq \boldsymbol{c}/\boldsymbol{b} > [0,0]$.

With respect to set inclusion as an order relation arithmetic operations in $\{\overline{\mathbb{IR}}, \subseteq\}$ are inclusion isotone by (2), i.e., $\boldsymbol{a} \subseteq \boldsymbol{b} \Rightarrow \boldsymbol{a} \circ \boldsymbol{c} \subseteq \boldsymbol{b} \circ \boldsymbol{c}$ or equivalently

(OD5) $\boldsymbol{a} \subseteq \boldsymbol{b} \wedge \boldsymbol{c} \subseteq \boldsymbol{d} \Rightarrow \boldsymbol{a} \circ \boldsymbol{c} \subseteq \boldsymbol{b} \circ \boldsymbol{d}$, for all $\circ \in \{+, -, *, /\}, 0 \notin \boldsymbol{b}, \boldsymbol{d}$ for $\circ = /$. (inclusion isotone)

Setting $\boldsymbol{c}, \boldsymbol{d} = -1$ in (OD5) delivers immediately $\boldsymbol{a} \subseteq \boldsymbol{b} \Rightarrow -\boldsymbol{a} \subseteq -\boldsymbol{b}$ which differs significantly from (OD2).

Definition 7. *With the upwardly directed rounding $\diamondsuit : \overline{\mathbb{IR}} \to \overline{\mathbb{IF}}$ binary arithmetic operations in $\overline{\mathbb{IF}}$ are defined by semimorphism:*

(RG) $\boldsymbol{a} \diamondsuit \boldsymbol{b} := \diamondsuit\, (\boldsymbol{a} \circ \boldsymbol{b})$, *for all $\boldsymbol{a}, \boldsymbol{b} \in \overline{\mathbb{IF}}$ and all $\circ \in \{+, -, *, /\}$.*

Here for division we assume that $\boldsymbol{a}/\boldsymbol{b}$ is defined.

If an interval $\boldsymbol{a} \in \overline{\mathbb{IF}}$ is an upper bound of an interval $\boldsymbol{x} \in \overline{\mathbb{IR}}$, i.e., $\boldsymbol{x} \subseteq \boldsymbol{a}$, then by (R1), (R2), and (R3) also $\boldsymbol{x} \subseteq \diamondsuit\, \boldsymbol{x} \subseteq \boldsymbol{a}$. This means $\diamondsuit\, \boldsymbol{x}$ is the least upper bound, the supremum of $\boldsymbol{x}$ in $\overline{\mathbb{IF}}$. Similarly if for $\boldsymbol{x}, \boldsymbol{y} \in \overline{\mathbb{IF}}$, $\boldsymbol{x} \circ \boldsymbol{y} \subseteq \boldsymbol{a}$ with $\boldsymbol{a} \in \overline{\mathbb{IF}}$, then by (R1), (R2), (R3), and (RG) also $\boldsymbol{x} \circ \boldsymbol{y} \subseteq \boldsymbol{x} \diamondsuit \boldsymbol{y} \subseteq \boldsymbol{a}$, i.e., $\boldsymbol{x} \diamondsuit \boldsymbol{y}$ is the least upper bound, the supremum of $\boldsymbol{x} \circ \boldsymbol{y}$ in $\overline{\mathbb{IF}}$. Occasionally the supremum $\boldsymbol{x} \diamondsuit \boldsymbol{y}$ of the result $\boldsymbol{x} \circ \boldsymbol{y} \in \overline{\mathbb{IR}}$ is called the tightest enclosure of $\boldsymbol{x} \circ \boldsymbol{y}$.

Arithmetic operations in $\overline{\mathbb{IF}}$ are inclusion isotone, i.e.,

(OD5) $\boldsymbol{a} \subseteq \boldsymbol{b} \wedge \boldsymbol{c} \subseteq \boldsymbol{d} \Rightarrow \boldsymbol{a} \diamondsuit \boldsymbol{c} \subseteq \boldsymbol{b} \diamondsuit \boldsymbol{d}$, for $\circ \in \{+, -, *, /\}, 0 \notin \boldsymbol{b}, \boldsymbol{d}$ for $\circ = /$. (inclusion isotone)

This is a consequence of the inclusion isotony of the arithmetic operations in $\overline{\mathbb{IR}}$, of (R2) and of (RG).

Since the arithmetic operations $\boldsymbol{x} \circ \boldsymbol{y}$ in $\overline{\mathbb{IR}}$ are defined as set operations by (3) the operations $\boldsymbol{x} \diamondsuit \boldsymbol{y}$ for intervals of $\overline{\mathbb{IF}}$ defined by (RG) are not directly executable. The step from the definition of interval arithmetic by set operations to computer executable operations still requires some effort. This holds in particular for product sets like complex intervals and intervals of vectors and matrices of real and complex numbers, [23]. We sketch it here for the most simple case of floating-point intervals.

[7] An integral number a in an interval expression is interpreted as interval $[a, a]$.

3.3 Executable Interval Arithmetic

We first derive executable formulas for the arithmetic operations for nonempty, closed and bounded real intervals $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$. These are then extended to unbounded intervals of $\overline{\mathbb{IR}}$. Arithmetic operations for intervals are defined as set operations by $\boldsymbol{a} \circ \boldsymbol{b} := \{a \circ b \mid a \in \boldsymbol{a} \wedge b \in \boldsymbol{b}\}$. For $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$ with $0 \notin \boldsymbol{b}$ in case of division for all arithmetic operations $\circ \in \{+, -, *, /\}$ the function $a \circ b$ is a continuous function of both variables. The set $\boldsymbol{a} \circ \boldsymbol{b}$ is the range of the function $a \circ b$ over the product set $\boldsymbol{a} \times \boldsymbol{b}$. Since $\boldsymbol{a}$ and $\boldsymbol{b}$ are closed and bounded intervals $\boldsymbol{a} \times \boldsymbol{b}$ is a simply connected, closed and bounded subset of $\mathbb{R}^2$. In such a region the continuous function $a \circ b$ takes a minimum and a maximum as well as all values in between. Therefore

$$\boldsymbol{a} \circ \boldsymbol{b} = [\min_{a \in \boldsymbol{a}, b \in \boldsymbol{b}} \{a \circ b\}, \max_{a \in \boldsymbol{a}, b \in \boldsymbol{b}} \{a \circ b\}] = [\inf(\boldsymbol{a} \circ \boldsymbol{b}), \sup(\boldsymbol{a} \circ \boldsymbol{b})],$$

i.e., for $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$, $0 \notin \boldsymbol{b}$ in case of division, $\boldsymbol{a} \circ \boldsymbol{b}$ is again an interval of $\mathbb{IR}$. In other words: Arithmetic in $\mathbb{IR}$ is an algebraically closed subset of the arithmetic in the powerset of the real numbers.

Actually the minimum and maximum is taken for operations with the bounds. For bounded intervals $\boldsymbol{a} = [a_1, a_2]$ and $\boldsymbol{b} = [b_1, b_2]$ the following formula holds for all operations with $0 \notin \boldsymbol{b}$ in case of division:

$$\boldsymbol{a} \circ \boldsymbol{b} = \left[\min_{i,j=1,2}(a_i \circ b_j), \max_{i,j=1,2}(a_i \circ b_j)\right] \text{ for } \circ \in \{+, -, *, /\}. \tag{5}$$

We demonstrate this in case of addition. By (OD1) we obtain $a_1 \leq \boldsymbol{a}$ and $b_1 \leq \boldsymbol{b} \Rightarrow a_1 + b_1 \leq \inf(\boldsymbol{a} + \boldsymbol{b})$. On the other hand $\inf(\boldsymbol{a} + \boldsymbol{b}) \leq a_1 + b_1$. From both inequalities we obtain by (O3): $\inf(\boldsymbol{a} + \boldsymbol{b}) = a_1 + b_1$. Analogously one obtains $\sup(\boldsymbol{a} + \boldsymbol{b}) = a_2 + b_2$. Thus

$$\boldsymbol{a} + \boldsymbol{b} = [\min_{a \in \boldsymbol{a}, b \in \boldsymbol{b}} \{a + b\}, \max_{a \in \boldsymbol{a}, b \in \boldsymbol{b}} \{a + b\}] = [\inf(\boldsymbol{a} + \boldsymbol{b}), \sup(\boldsymbol{a} + \boldsymbol{b})] = [a_1 + b_1, a_2 + b_2].$$

Similarly by making use of (OD1,2,3,4) for intervals of $\mathbb{IR}$ and the simple sign rules $-(\boldsymbol{a} * \boldsymbol{b}) = (-\boldsymbol{a}) * \boldsymbol{b} = \boldsymbol{a} * (-\boldsymbol{b}), -(\boldsymbol{a}/\boldsymbol{b}) = (-\boldsymbol{a})/\boldsymbol{b} = \boldsymbol{a}/(-\boldsymbol{b})$ explicit formulas for all interval operations can be derived, [23].

Now we get by (RG) for intervals of $\mathbb{IF}$

$$\boldsymbol{a} \diamondsuit \boldsymbol{b} := \diamondsuit(\boldsymbol{a} \circ \boldsymbol{b}) = \left[\bigtriangledown \min_{i,j=1,2}(a_i \circ b_j), \bigtriangleup \max_{i,j=1,2}(a_i \circ b_j)\right]$$

and by the monotonicity of the roundings $\bigtriangledown$ and $\bigtriangleup$:

$$\boldsymbol{a} \diamondsuit \boldsymbol{b} = \left[\min_{i,j=1,2}(a_i \mathbin{\bigtriangledown} b_j), \max_{i,j=1,2}(a_i \mathbin{\bigtriangleup} b_j)\right].$$

For bounded and nonempty intervals $\boldsymbol{a} = [a_1, a_2]$ and $\boldsymbol{b} = [b_1, b_2]$ of $\mathbb{IF}$ the unary operation $-\boldsymbol{a}$ and the binary operations addition, subtraction, multiplication, and division are shown in the following tables. For details see [23]. Therein the operator symbols for intervals are simply denoted by $+, -, *, /$.

Minus operator $\quad -\boldsymbol{a} = [-a_2, -a_1]$.

Addition $\quad [a_1, a_2] + [b_1, b_2] = [a_1 \triangledown b_1, a_2 \triangle b_2]$.

Subtraction $\quad [a_1, a_2] - [b_1, b_2] = [a_1 \triangledown b_2, a_2 \triangle b_1]$.

Multiplication $[a_1, a_2] * [b_1, b_2]$	$[b_1, b_2]$ $b_2 \leq 0$	$[b_1, b_2]$ $b_1 < 0 < b_2$	$[b_1, b_2]$ $b_1 \geq 0$
$[a_1, a_2], a_2 \leq 0$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$
$a_1 < 0 < a_2$	$[a_2 \triangledown b_1, a_1 \triangle b_1]$	$[min(a_1 \triangledown b_2, a_2 \triangledown b_1),$ $max(a_1 \triangle b_1, a_2 \triangle b_2)]$	$[a_1 \triangledown b_2, a_2 \triangle b_2]$
$[a_1, a_2], a_1 \geq 0$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$	$[a_2 \triangledown b_1, a_2 \triangle b_2]$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$

Division, $0 \notin \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$[b_1, b_2]$ $b_2 < 0$	$[b_1, b_2]$ $b_1 > 0$
$[a_1, a_2], a_2 \leq 0$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$[a_2 \triangledown b_2, a_1 \triangle b_2]$	$[a_1 \triangledown b_1, a_2 \triangle b_1]$
$[a_1, a_2], 0 \leq a_1$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$

In real analysis division by zero is not defined. In interval arithmetic, however, the interval in the denominator of a quotient may contain zero. So this case has to be considered also.

The general rule for computing the set $\boldsymbol{a}/\boldsymbol{b}$ with $0 \in \boldsymbol{b}$ is to remove its zero from the interval $\boldsymbol{b}$ and perform the division with the remaining set.[8] Whenever zero in $\boldsymbol{b}$ is an endpoint of $\boldsymbol{b}$, the result of the division can be obtained directly from the above table for division with $0 \notin \boldsymbol{b}$ by the limit process $b_1 \to 0$ or $b_2 \to 0$ respectively. The results are shown in the table for division with $0 \in \boldsymbol{b}$. Here, the round brackets stress that the bounds $-\infty$ and $+\infty$ are not elements of the interval.

Division, $0 \in \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$\boldsymbol{b} =$ $[0, 0]$	$[b_1, b_2]$ $b_1 < b_2 = 0$	$[b_1, b_2]$ $0 = b_1 < b_2$
$[a_1, a_2] = [0, 0]$	$\varnothing$	$[0, 0]$	$[0, 0]$
$[a_1, a_2], a_1 < 0, a_2 \leq 0$	$\varnothing$	$[a_2 \triangledown b_1, +\infty)$	$(-\infty, a_2 \triangle b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$\varnothing$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], 0 \leq a_1, 0 < a_2$	$\varnothing$	$(-\infty, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, +\infty)$

When zero is an interior point of the denominator, the set $[b_1, b_2]$ splits into the distinct sets $[b_1, 0]$ and $[0, b_2]$, and the division by $[b_1, b_2]$ actually means two divisions. The results of the two divisions are already shown in the table for division by $0 \in \boldsymbol{b}$.

However, in the user's program the two divisions appear as a single operation, as division by an interval $[b_1, b_2]$ with $b_1 < 0 < b_2$, an operation that delivers two distinct results.

A solution to the problem would be for the computer to provide a flag for *distinct intervals*. The situation occurs if the divisor is an interval that contains zero as an interior point. In this case the flag would be raised and signaled to the user. The user may then apply a routine of his choice to deal with the situation as is appropriate for his application. This routine could be: return the entire set of real numbers $(-\infty, +\infty)$ as result and continue the computation, or set a flag and continue the computation with one of the sets and

[8] This is in full accordance with function evaluation: When evaluating a function over a set, points outside its domain are simply ignored.

ignore the other one, or put one of the sets on a list and continue the computation with the other one, or modify the operands and recompute, or stop computing, or some other action.

An alternative would be to provide a second division which in case of division by an interval that contains zero as an interior point generally delivers the result $(-\infty, +\infty)$. Then the user can decide when to use which division in his program.

Deriving explicit formulas for the result of interval operations $\boldsymbol{a} \circ \boldsymbol{b}, \circ \in \{+, -, *, /\}$ it was assumed so far that the intervals $\boldsymbol{a}$ and $\boldsymbol{b}$ are nonempty and bounded. Four kinds of *extended intervals* come from division by an interval of $\mathbb{IF}$ that contains zero:

$$\varnothing, \quad (-\infty, a], \quad [b, +\infty), \quad \text{and} \quad (-\infty, +\infty).$$

To extend the operations to these more general intervals the first rule is that any operation with the empty set $\varnothing$ returns the empty set. Then the above tables extend to possibly unbounded intervals of $\overline{\mathbb{IF}}$ by using the standard formulae for arithmetic operations involving $\pm\infty$ together with one rule that goes beyond standard rules for $\pm\infty$:

$$0 * (-\infty) = (-\infty) * 0 = 0 * (+\infty) = (+\infty) * 0 = 0.$$

This rule is not a new mathematical law, it is merely a short cut to compute the bounds of the result of multiplication on unbounded intervals.[9]

In the previous section intervals of $\overline{\mathbb{IR}}$ are defined as closed and connected sets of real numbers. Arithmetic for intervals of $\overline{\mathbb{IF}}$ as derived here leads to an exception free calculus, i.e., arithmetic operations for intervals of $\overline{\mathbb{IF}}$ always lead to intervals of $\overline{\mathbb{IF}}$ again.

This is in sharp contrast to other models of interval arithmetic which consider intervals over the extended real numbers $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$. In such models obscure arithmetic operations like $\infty - \infty, \infty/\infty, 0 * \infty$ occur which require introduction of unnatural superficial objects like NaI (Not an Interval).

4 Interval Arithmetic in Higher Dimensions

The axioms for computer arithmetic shown in section 2 also can be applied to define computer arithmetic in higher dimensional spaces like complex numbers, vectors and matrices for real, complex, and interval data.

If M in section 2 is the set of real matrices and $S \subseteq M$ the set of floating-point matrices, then the definition of arithmetic operations in S by (RG) ideally requires exact evaluation of scalar products of vectors with floating-point components. Indeed very effective algorithms have been developed for computing scalar products of vectors with floating-point components exactly, [23]. For a brief sketch see section 7.

If M in section 2 is the set of intervals of real vectors or matrices, respectively, then the set definition of arithmetic operations in M by (2) does not lead to an interval again. The result $\boldsymbol{a} \circ \boldsymbol{b} := \{a \circ b \mid a \in \boldsymbol{a} \wedge b \in \boldsymbol{b}\}$ is a more general set. It is an element of the power set[10] of vectors or matrices, respectively. To obtain an interval the upwardly directed rounding $\square$ from the power set onto the set of intervals of M has to be applied. With it arithmetic operations for intervals $\boldsymbol{a}, \boldsymbol{b} \in M$ are defined by

$$\text{(RG)} \quad \boldsymbol{a} \boxdot \boldsymbol{b} := \square (\boldsymbol{a} \circ \boldsymbol{b}), \circ \in \{+, -, \cdots\}.$$

The set S of intervals of floating-point vectors or matrices, respectively, is a screen of M. To obtain arithmetic for intervals $\boldsymbol{a}, \boldsymbol{b} \in S$ once more the monotone upwardly directed rounding, now denoted by $\diamondsuit$ is applied:

$$\text{(RG)} \quad \boldsymbol{a} \diamondsuit \boldsymbol{b} := \diamondsuit (\boldsymbol{a} \boxdot \boldsymbol{b}), \circ \in \{+, -, \cdots\}.$$

[9] Intervals of $\overline{\mathbb{IF}}$ are closed and connected sets of real numbers. So multiplication of any such interval by 0 can only have 0 as the result.

[10] The power set of a set M is the set of all subsets of M.

This leads to the best possible operations in the interval spaces M and S. So for intervals in higher dimensional spaces generally an additional rounding step is needed. It can be shown that in all relevant cases these best possible operations can be expressed by executable formulas just using the bounds of the intervals. Showing this for more complicated cases (e.g. complex intervals or intervals of real and complex vectors and matrices) requires a more detailed study. For details see [23] and the literature cited there. The dot product of two vectors the components of which are floating-point numbers is a key operation in all these spaces. Computing it exactly can considerably speed up floating-point and interval arithmetic.

5 Operations with Directed Roundings and Programming Languages

The IEEE standard 754 for floating-point arithmetic seems to support interval arithmetic. It requires the four basic arithmetic operations to be available with rounding to nearest, towards zero, downwards and upwards. The latter two are essential for interval arithmetic. But almost all processors that provide IEEE 754 arithmetic separate the rounding from the basic 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, one in every 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 is fatal for interval arithmetic. The rounding must 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.

For interval arithmetic we need to be able to call each of the operations $\overline{\nabla}$, ∇, $\overline{\nabla}$, ∇ and $\overline{\Delta}$, Δ, $\overline{\Delta}$, Δ as one single operation, that is, the rounding must be inherent to each. Therefore in programming languages notations for arithmetic operations with different roundings shall be provided. 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).

New operation codes are necessary! Each of these operations shall be executed as one single operation.

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

Instead, a floating-point arithmetic standard should require that **every future processor** shall provide the 16 operations listed above. In languages like C++ which just provide operator overloading and do not allow user defined operators these operations could be written as functions:

$$addp, subp, mulp, divp, \quad \text{and} \quad addn, subn, muln, divn.$$

Here p stands for rounding toward positive and n for rounding toward negative infinity. With these operators interval operations can easily be programmed. They would be very fast and fully transferable from one processor to another.

In such languages the operations with the directed roundings are hidden within the interval operations. If operations with directed roundings are needed, interval operations are performed instead and the upper and lower bound of the result then gives the desired answer.

This is exactly the way interval arithmetic is provided in the C++ class library C-XSC which has been successfully used for more than 20 years [16, 17]. Since C++ only allows operator overloading, the operations with the directed roundings cannot and are not provided in C-XSC. These operations

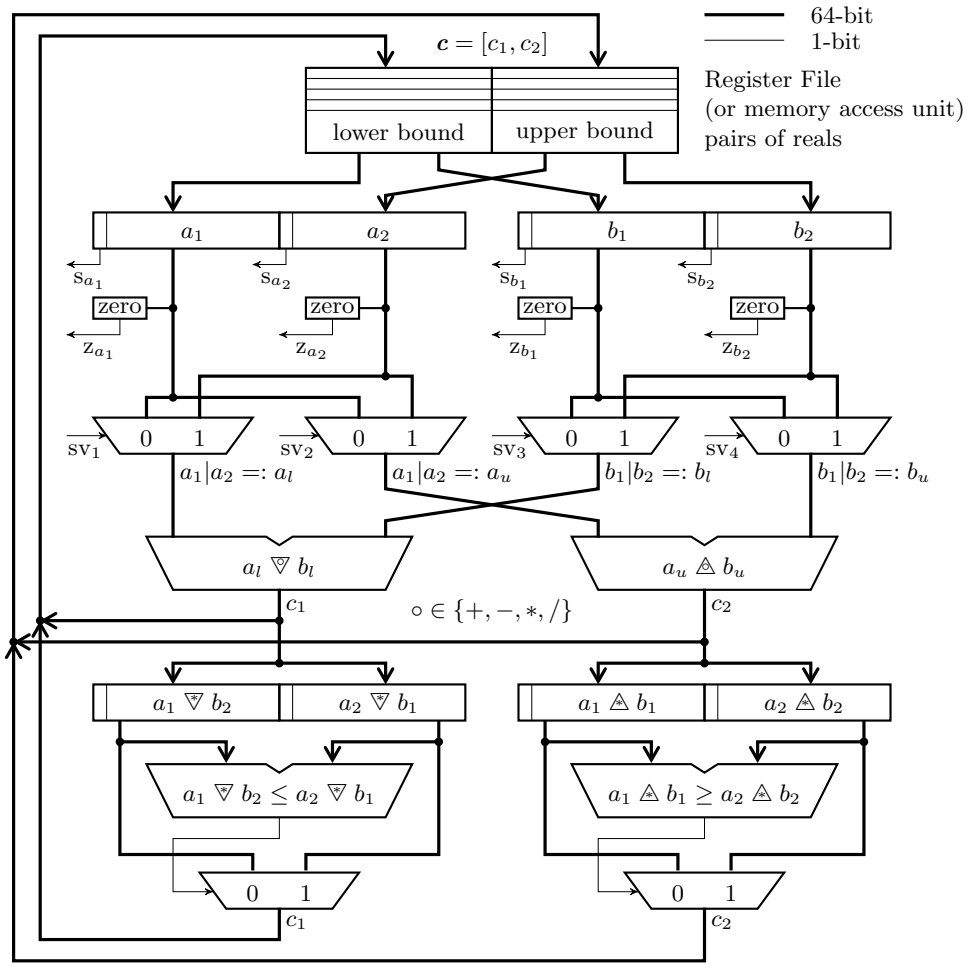
are rarely needed in applications. If they are needed, interval operations are performed instead and the upper or lower bound of the result then give the desired answer.

In Fortran operators like .addp. could be used for instance.

6 Hardware for Interval Arithmetic

Figure 1 gives a brief sketch of what hardware support for interval arithmetic may look like. It would not be hard to realize it in modern technology.

The circuitry broadly speaks for itself. The lower and the upper bound of the result of an operation are computed simultaneously. The interval operands are loaded in parallel from a register file or a memory access unit. Then, after multiplexers have selected the appropriate operands, the lower bound of the result is computed with rounding downwards and the upper bound with rounding upwards with the selected operands. If in case of multiplication both operands contain zero as an interior point a second multiplication is necessary. In the figure it is assumed that the two multiplications are performed sequentially and that the results of both multiplications are delivered to the lower part of the circuitry. They are then forwarded to a comparison unit. Here for the lower bound of the result the lower and for the upper bound the higher of the two products is selected. This lower part of the circuitry could also be used to perform comparison relations.



operands: $\boldsymbol{a} = [a_1, a_2]$, $\boldsymbol{b} = [b_1, b_2]$, result: $\boldsymbol{c} = [c_1, c_2]$.
s: sign, z: zero, o: operand select.

Fig. 1. Circuitry for Interval Operations

Multiplication	$B = 0$	$b_1 < 0, b_2 \leq 0$	$b_1 < 0 < b_2$	$0 \leq b_1, 0 < b_2$
$A = 0$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$
$a_1 < 0, a_2 \leq 0$	$[0,0]$	$[a_2 \bigtriangledown b_2, a_1 \bigtriangleup b_1]$	$[a_1 \bigtriangledown b_2, a_1 \bigtriangleup b_1]$	$[a_1 \bigtriangledown b_2, a_2 \bigtriangleup b_1]$
		$sv = 1100$	$sv = 0100$	$sv = 0110$
$a_1 < 0 < a_2$	$[0,0]$	$[a_2 \bigtriangledown b_1, a_1 \bigtriangleup b_1]$	$[min(a_1 \bigtriangledown b_2, a_2 \bigtriangledown b_1),$ $max(a_1 \bigtriangleup b_1, a_2 \bigtriangleup b_2)]$	$[a_1 \bigtriangledown b_2, a_2 \bigtriangleup b_2]$
		$sv = 1000$	$sv = 0100,\ sv = 1011$	$sv = 0111$
$0 \leq a_1, 0 < a_2$	$[0,0]$	$[a_2 \bigtriangledown b_1, a_1 \bigtriangleup b_2]$	$[a_2 \bigtriangledown b_1, a_2 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, a_2 \bigtriangleup b_2]$
		$sv = 1001$	$sv = 1011$	$sv = 0011$

Table 1. Interval multiplication.

Division	$B = 0$	$b_2 < 0$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$	$0 < b_1$
$A = 0$	$\varnothing$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$
$a_1 < 0, a_2 \leq 0$	$\varnothing$	$[a_2 \bigtriangledown b_1, a_1 \bigtriangleup b_2]$	$[a_2 \bigtriangledown b_1, +\infty)$	$(-\infty, a_2 \bigtriangleup b_2]$ $\cup [a_2 \bigtriangledown b_1, +\infty)$	$(-\infty, a_2 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, a_2 \bigtriangleup b_2]$
		$sv = 1001$	$sv = 10xx$	$sv = 1011$	$sv = xx11$	$sv = 0011$
$a_1 < 0 < a_2$	$\varnothing$	$[a_2 \bigtriangledown b_2, a_1 \bigtriangleup b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1 \bigtriangledown b_1, a_2 \bigtriangleup b_1]$
		$sv = 1101$				$sv = 0010$
$0 \leq a_1, 0 < a_2$	$\varnothing$	$[a_2 \bigtriangledown b_2, a_1 \bigtriangleup b_1]$	$(-\infty, a_1 \bigtriangleup b_1]$	$(-\infty, a_1 \bigtriangleup b_1]$ $\cup [a_1 \bigtriangledown b_2, +\infty)$	$[a_1 \bigtriangledown b_2, +\infty)$	$[a_1 \bigtriangledown b_2, a_2 \bigtriangleup b_1]$
		$sv = 1100$	$sv = xx00$	$sv = 0100$	$sv = 01xx$	$sv = 0110$

Table 2. Interval division.

In the figure it is assumed that the second product is computed with the same circuitry sequentially. This slows down interval multiplication in comparison with the other interval operations. This can be avoided by doubling the circuitry for multiplication and performing the two multiplications in parallel.

The selector signals sv_1, sv_2, sv_3, and sv_4 control the multiplexers. The circuitry shown in the figure performs all arithmetic operations.

In **addition** the lower bounds of $\boldsymbol{a}$ and $\boldsymbol{b}$ are simply added with rounding downwards and the upper bounds with rounding upwards $[a_1 \bigtriangledown b_1, a_2 \bigtriangleup b_2]$. The selector signals are set to $sv_1 = 0$, $sv_2 = 1$, $sv_3 = 0$, and $sv_4 = 1$.

In **subtraction** the bounds of $\boldsymbol{b}$ are exchanged by the operand selection. Then the subtraction is performed, the lower bound being computed with rounding downwards and the upper bound with rounding upwards $[a_1 \bigtriangledown b_2, a_2 \bigtriangleup b_1]$. The selector signals are set to $sv_1 = 0, sv_2 = 1, sv_3 = 1$, and $sv_4 = 0$.

These two operations, addition and subtraction, reveal a close relationship between the arithmetic operations shown in Tables 1 and 2 and the operand selector signals which select the lower or the upper bound of the operands $\boldsymbol{a}$ and $\boldsymbol{b}$ for the operation to be performed. The index of the operand component to be selected minus 1 gives the value of the selector signal $sv_i, i = 1, 2, 3, 4$ for the selection of the operand.

This simple rule also holds for the operand selections in the cases of multiplication and division. In the Tables 1 and 2 the selector signals sv_1, sv_2, sv_3, sv_4 are summarized into a selection vector $sv = (sv_1, sv_3, sv_2, sv_4)$ which is shown directly below the formulas for the arithmetic operations. In Table 2 both cases $0 \notin \boldsymbol{b}$ and $0 \in \boldsymbol{b}$ are merged into a single table.

For intervals $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$ subtraction is defined by $\boldsymbol{a} - \boldsymbol{b} := \boldsymbol{a} + (-\mathbf{1}) * \boldsymbol{b}$. Thus subtraction could be reduced to addition by negation of the operand $\boldsymbol{b}$.

6.1 Interval Arithmetic on X86-Processors

It is interesting to note that most of what is needed for fast hardware support for interval arithmetic is already available on current x86-processors. Figure 2 shows figures from various publications by Intel.

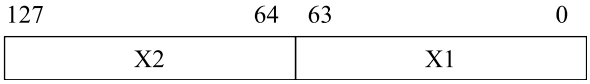


Figure 6. Packed double precision floating-point data type

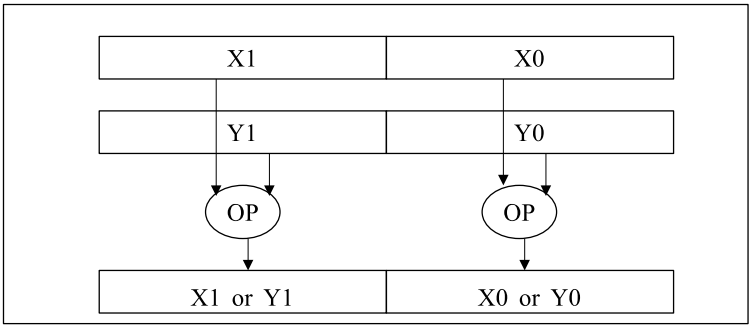


Figure 11-3. Packed Double-Precision Floating-Point Operation

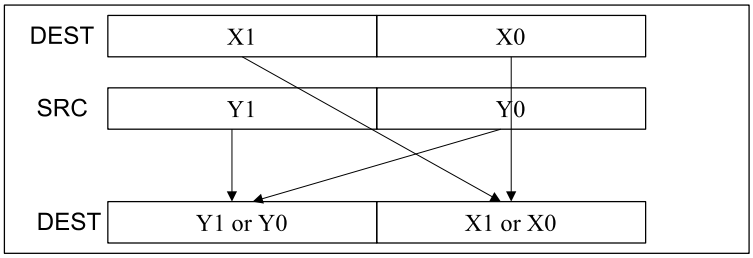


Figure 11-5. SHUFFD Instruction Packed Shuffle Operation

Fig. 2. Figures from various Intel publications.

On an Intel Pentium 4, for instance, eight registers are available for words of 128 bits (xmm0, xmm1, ..., xmm7). The x86-64 processors even provide 16 such registers. These registers can hold pairs of double precision floating-point numbers. They can be viewed as bounds of intervals. Parallel operations like $+$, $-$, $\cdot$, $/$, min, max, and compare can be performed on these pairs of numbers. What is not available and would be needed is for one of the two operations to be rounded downwards and the other one rounded upwards. The last picture in Figure 2 shows that even shuffling of bounds is possible under certain conditions. This is half of operand selection needed for interval arithmetic. So an interval operation would need two such units or to pass this unit twice. Also nearly all of the data paths are available on current x86-processors. Thus full hardware support of interval arithmetic would probably add very little to a current Intel or AMD x86 processor chip.

Full hardware support of fast interval arithmetic on RISC processors may cost a little more as these lack pairwise processing. But most of them have two arithmetic units and use them for super scalar processing. What has to be added is some sophisticated control.

7 High Speed Computing by an Exact Dot Product

The IFIP Working Group on Numerical Software and other scientists repeatedly requested that a future arithmetic standard should consider and specify an exact dot product (EDP) [5, 6], [1,2]. In Nov. 2009 the IEEE standards committee P1788 on interval arithmetic accepted a motion for including the EDP into a future interval arithmetic standard. In Numerical Analysis the dot product is ubiquitous. It is a fundamental arithmetic operation in vector and matrix spaces for real, complex, and interval data. It is the EDP which makes residual correction effective. This has a direct and positive influence on all iterative solvers of systems of equations. The EDP is essential for fast long real and long interval arithmetic [23]. By operator overloading variable precision interval arithmetic is very easy to use. With it the result of every arithmetic expression can be guaranteed to a number of correct digits.

A later motion revised the motion for the EDP. It now requires a correctly rounded dot product instead. Computing this via an EDP is recommended. The literature contains many excellent and intelligent methods for computing "accurate" sums and dot products or a correctly rounded dot product in particular. Several of these methods are claimed to be *fast* or even *ultimately fast*. All these methods use conventional floating-point arithmetic. The speed of the faster of these methods comes considerably close to the time T needed for (a possibly wrong) computation of the dot product in conventional floating-point arithmetic. The speed mildly decreases with increasing condition number.

Computing dot products exactly on the contrary considerably speeds up computing. Actually the simplest and fastest way for computing a dot product is to compute it exactly. By pipelining the stages: loading the data, computing, shifting, and accumulation of the products the EDP is computed in the time the processor needs to read the data, i.e., it comes with utmost speed. Here the computing time is independent of the condition number. It is much less than T. The summands are just shifted and added into a long fixed-point register. Fixed-point accumulation of the products is simpler than accumulation in floating-point. 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 addition is error free! This brings a substantial speed increase compared to a possibly wrong accumulation of the dot product using conventional floating-point arithmetic. So the EDP is a mean to considerably speed up floating-point and interval arithmetic.

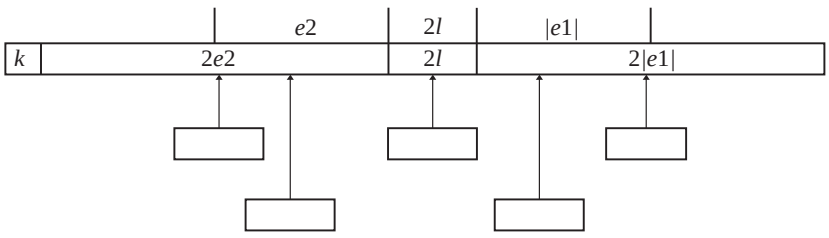
This section briefly illustrates how the EDP can be implemented on computers. There is indeed no simpler way of accumulating a dot product. Any method that just computes an approximation also has to consider the relative values of the summands. This results in a more complicated method.

Let $(a_i), (b_i) \in \mathbb{F}^n$ be two vectors with n components which are floating-point numbers $a_i, b_i \in \mathbb{F}(b, l, e1, e2)$, for $i = 1(1)n$. Here b is the base of the number system in use, l the number of digits of the mantissa, $e1$ is the least and $e2$ the greatest exponent respectively. We compute the sum

$$s := \sum_{i=1}^{n} a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n, \qquad a_i \cdot b_i \in \mathbb{F}(b, 2 \cdot l, 2 \cdot e1, 2 \cdot e2), \text{ for } i = 1(1)n,$$

where all additions and multiplications are the operations for real numbers.

All summands can be taken into a fixed-point register of length $2 \cdot e2 + 2 \cdot l + 2 \cdot |e1|$ without loss of information. We call it a *complete register*, CR for short.



Complete register for exact scalar product accumulation.

If the register is built as an accumulator with an adder, all summands could be added in without loss of information. To accommodate possible overflows, it is convenient to provide a few, say k, more

digits of base b on the left, allowing b^k accumulations to be computed without loss of information due to overflow. k can be chosen such that no overflows of the long register will occur in the lifetime of the computer.

For IEEE-arithmetic double precision we have:
$b = 2$; 64 bits word length; 1 bit sign; 11 bits exponent; $l = 53$ bits; $e1 = -1022$, $e2 = 1023$. With $k = 92$ the entire unit consists of
$L = k + 2 \cdot e2 + 2 \cdot l + 2 \cdot |e1| = k + 4196$ bits $= 4288$ bits. It can be represented by 67 words of 64 bits. L is independent of n.

Figure 3 informally describes the implementation of an EDP. The long register (here represented as a chest of drawers) is organized in words of 64 bits. The exponent of the products 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. A possible carry is absorbed by the next more significant word in which not all bits are 1. For fast detection of this word a flag is attached to each word. It is set 1 if all bits of the word are 1. This means that a carry will propagate through the entire word. In the figure the flag is shown as a red (dark) point. 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 positions. Figure 4 shows a sketch for the parallel accumulation of a product into the complete register.

By pipelining, the accumulation of a product into the complete register can be done in the time the processor needs to read the data. Since every other method computing a dot product also has to read the data this means that no such method can exceed computing the EDP in speed. For the pipelining and other solutions see [23]. Rounding the EDP into a correctly rounded dot product is done only once at the very end of the accumulation.

In [23] three different solutions for pipelining the dot product computation are dealt with. They differ by the speed with which the vector components for a product can be read into the scalar product unit. They can be delivered in 32- or 64-bit portions or even at once as one 128 bit word. With increasing bus width and speed more hardware has to be invested for the multiplication and the accumulation to keep the pipeline in balance.

The hardware cost needed for the EDP is modest. It is comparable to that for a fast multiplier by an adder tree, accepted years ago and now standard technology in every modern processor. The EDP brings the same speedup for accumulations at comparable costs.

Figures 3 and 4 informally specify the implementation of the EDP on computers. For more details see [23] and [25].

Possibilities to compute the EDP by only making use of a hardware *complete register* window are discussed in [23].

To be completely successful interval arithmetic occasionally has to be complemented by some easy way to use multiple precision real and interval arithmetic. The fast and exact dot product is the tool to provide these operations. By operator overloading multiple precision arithmetic can be used very easily. Multiple precision real and interval operations for addition, subtraction, multiplication, division, the scalar product, and the square root are defined in [23]. In [30] the authors show that even extremely ill conditioned matrices with a condition number of about 10^{100} can successfully be inverted using such tools.

In 1984 a hardware unit for the EDP was developed in bit-slice technology [12]. This technology was expensive and did not allow production of the unit in large quantities. Then in 1995 the EDP was implemented on a single chip using a VLSI gate-array technology [9]. It was connected with the PC via the PCI-bus and embedded into powerful PASCAL-XSC and C-XSC programming environments. Although the technology available for the chip was considerably weaker than that available for the PC, the chip computed the EDP faster than a (possibly wrong) evaluation in conventional floating-point arithmetic.

Ultimately the EDP must be integrated into the arithmetic unit of the processor. A little register space on the arithmetic unit will be rewarded by a substantial increase in speed and reliability of computing.

Interval arithmetic brings guarantees and mathematics into computing, while the EDP brings higher precison, accuracy, and speed.

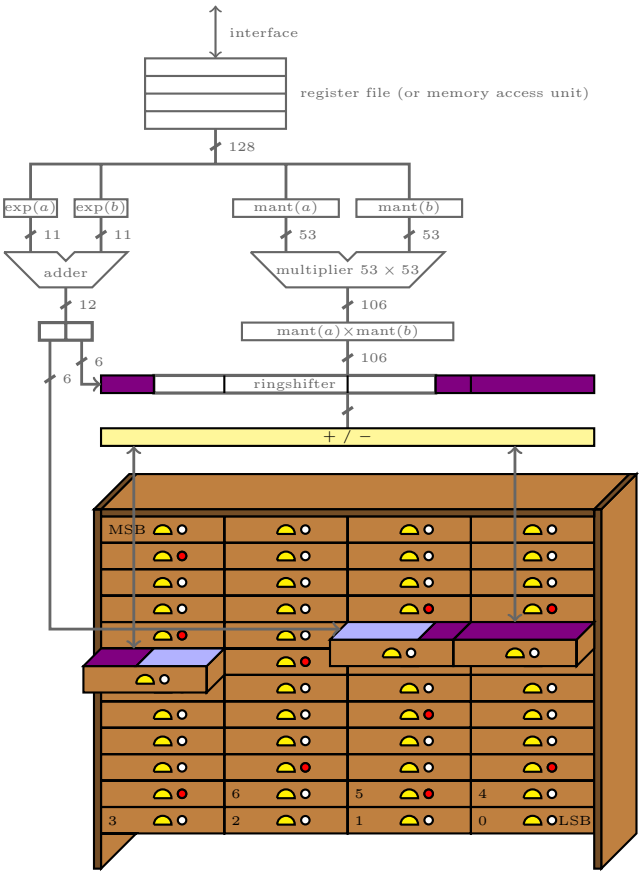


Fig. 3. Illustration for computing the exact dot product

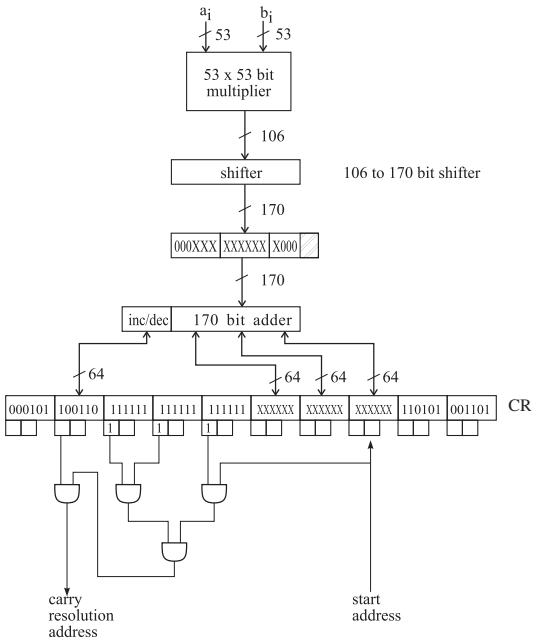


Fig. 4. Parallel accumulation of a product into the CR.

8 Early Super Computers

It is interesting that the technique for computing dot products exactly is not new at all. It can be traced back to the early computer by G. W. Leibniz (1673). Also old mechanic calculators added numbers and products of numbers into a wide fixed-point register. It was the fastest way to use the computer. So it was applied as often as possible. No intermediate results needed to be written down and typed in again for the next operation. No intermediate roundings or normalizations had to be performed. No error analysis was necessary. As long as no underflow or overflow occurred, which would be obvious and visible, the result was always exact. It was independent of the order in which the summands were added. Rounding was only done, if required, at the very end of the accumulation.

This extremely useful and fast fifth arithmetic operation was not built into the early floating-point computers. It was too expensive for the technologies of those days. Later its superior properties had been forgotten. Thus floating-point arithmetic is still comparatively incomplete.

Fig. 5. Mechanical computing devices equipped with the desired capability:
Burkhart Arithmometer, Glashütte, Germany, 1878; **Brunsviga**, Braunschweig, Germany, 1917; **MADAS**, Zürich, Switzerland, 1936; **MONROE**, New Jersey, USA, 1956.

The two lower calculators in Figure 5 are equipped with more than one long result register. In the previous section it was called a complete register CR. It is an early version of the recommended new data format *complete.*

In summary it can be said: The technique of speeding up computing by accumulation of numbers and of products of numbers exactly into a wide fixed-point register is as old as technical computing itself. It proves an excellent feeling of the old mathematicians for efficiency in computing.

Also early super computers (until ca. 2005) got their high speed by pipelining the dot product (vector processing). They provided so-called compound operations like *accumulate* or *multiply and accumulate.* The second computes the sum of products, the dot product of two vectors. Advanced programming languages offered these operations. Pipelining made them really fast. A vectorizing compiler filled them into a users program as often as possible. However, the accumulation was done in floating-point arithmetic by the so-called partial sum technique. This altered the sequence of the summands and caused errors beyond of the conventional floating-point errors. So finally this technique was abolished.

Fixed-point accumulation of the dot product, as discussed in the previous section, is simpler and faster than accumulation in floating-point arithmetic. In a very natural pipeline the accumulation is done in the time that is needed to read the data into the arithmetic unit. This means that no other method of computing a dot product can be faster, in particular not a conventional computation in

double or quadruple precision floating-point arithmetic. Fixed-point accumulation is error free! It is high speed vector processing in its perfection.

9 Early Products and Motivation

Between 1976 and 1980 a powerful programming environment PASCAL-XSC for support of interval arithmetic was developed at the author's institute together with a major number of problem solving routines for all kinds of standard problems of numerical analysis. These routines delivered close bounds for the solution. The project was supported by Nixdorf Computers. The Z-80 based system was exhibited at the Hannover Exhibition in April 1980. Nixdorf Computers donated a number of such systems to the University at Karlsruhe. This made it possible to decentralize the programming education in the summer semester 1980 while batch processing and punch cards and punch tapes were standard elsewhere.

When IBM became aware of this development a ten years lasting cooperation with the author's institute was started in 1980. The aim was to develop corresponding Fortran based products for their /370 architecture which at that time dominated the market. The corresponding IBM program products were called ACRITH (1983) and ACRITH-XSC (1990) [32–34]. Siemens and Hitachi followed soon by developing similar products for their /370 based systems. The Siemens product was called ARITHMOS [35]. The commercial success of these new products, however, was modest. During the development time other architectures appeared. The PC entered the market in 1982 and several new workstations in 1983/84. All these new systems made use of the new computer arithmetic standard IEEE 754 which was officially available since 1985. Scientific computing slowly but steadily moved from the old /370 systems to these more modern looking architectures.

In adapting the new situation the old Z-80 based PASCAL-XSC system was immediately transferred to the PC and to workstations at the author's institute. Books on PASCAL-XSC were published in German, English and Russian. See [13–15]. The book and the PASCAL-XSC system are now freely available. Further, after appearance of C and C++ a corresponding C-XSC was developed as a C++ class library again with a major number of problem solving routines [16, 17]. These books are also freely available.

High speed by support of hardware and programming languages is vital for interval arithmetic to be more widely accepted by the scientific computing community. The IEEE standard 754 seems to support interval arithmetic. It requires floating-point operations with the directed roundings. But in practice these have to be simulated by slow software. No commercial processor provides them directly by hardware. In 2008 the author published a book entitled: *Computer Arithmetic and Validity - Theory, Implementation, and Applications* [23]. It puts considerable emphasis on speeding up interval arithmetic. It extends the accuracy requirements for floating-point arithmetic as given by the IEEE standard 754 to the usual product spaces of computation like complex numbers and vectors and matrices for real, complex, and interval data. The book shows and states already in its preface that interval arithmetic for all these spaces can efficiently be provided on the computer if two additional features are made available by fast hardware:

I. Fast and direct hardware support for double precision interval arithmetic and
II. a fast and exact multiply and accumulate operation or, an exact dot product (EDP).

Realization of I. and II. is discussed at detail in the book [23]. It is shown that I. and II. can be obtained at very little hardware cost. With I. interval arithmetic would be as fast as simple floating-point arithmetic. The simplest and fastest way for computing a dot product is to compute it exactly. To make II. conveniently available a new data format *complete* is used together with a few very restricted arithmetic operations. By pipelining the EDP can be computed in the time the processor needs to read the data, i.e., it comes with utmost speed. I. and II. would boost both the speed of a computation and the accuracy of the result. Fast hardware support for I. and II. must be added to conventional floating-point arithmetic. Both are necessary extensions. Computing the dot product exactly even can be faster than computing it conventionally in double or extended precision floating-point arithmetic.

In [23] a real interval is defined as a closed and connected set of real numbers. Such intervals can be bounded or unbounded. In the latter case $-\infty$ or $+\infty$ occur as bounds of the interval. They are, however, not elements of the interval. Six months after publication of the book [23] the IEEE Computer Society founded a standard committee P1788 for interval arithmetic in August 2008. Work

on the standard began immediately and is still in progress. Much emphasis is put on specifying details. The definition of a real interval as a closed and connected set of real numbers was accepted. But a majority of members in the standard committee views the intention of the standard in specifying the functionality of interval arithmetic only. Demanding particular hardware and language support for interval arithmetic is not seen as task of the standard.

This paper is intended to complement the current state of the development of the IEEE P1788 standard by clearly stating what hardware and language support is vital and must be made available for interval arithmetic to increase its acceptance in the scientific computing community and to move interval arithmetic more into the mainstream of scientific computing.

Computer technology has been dramatically improved since 1985. Arithmetic speed has gone from megaflops (10^6 flops) to gigaflops (10^9 flops) to teraflops (10^{12} flops), and it is already in the petaflops (10^{15} flops) 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 or petaflops computer conventional error analysis is no longer practical. An avalanche of numbers is produced when a petaflops computer runs. If the numbers processed in one hour were to be printed (500 on one page, 1000 on one sheet, 1000 sheets 10 cm high) they would need a pile of paper that reaches from the earth to the sun and back. Computing indeed has already reached astronomical dimensions!

Every floating-point operation is potentially in error. This brings to the fore the question of whether the computed result really solves the given problem. The only way to answer this question seriously is by using the computer itself. The capability of a computer should not be judged by the number of operations it can perform in a certain amount of time without asking whether the computed result is correct. It would be much more reasonable to ask how fast a computer can compute correctly to 3, 5, 10 or 15 decimal places for certain problems. If the question were asked that way, it would very soon lead to better computers. Mathematical methods that give an answer to this question are available for very many problems. Computers, however, are at present not designed in a way that allows these methods to be used effectively. The tremendous progress in computer technology since 1985 should be accompanied by improving the mathematical capacity of the computer. Old mechanical calculators (section 8) are already pointing the way what can be done.

Modern processor architecture is coming considerably close to what is requested in this paper. See [36], and in particular pp.1-1 to 1-3 and 2-5 to 2-6. These processors provide register space of 16 K bits. Only about 4 K bits suffice for a *complete register.*

Acknowledgement: The author owes thanks to Goetz Alefeld for useful comments on the paper.

References

1. IMACS and GAMM, IMACS-GAMM resolution on computer arithmetic, *Mathematics and Computers in Simulation* 31 (1989), 297–298, or in *Zeitschrift für Angewandte Mathematik und Mechanik* 70:4 (1990).
2. GAMM-IMACS proposal for accurate floating-point vector arithmetic, *GAMM Rundbrief* 2 (1993), 9–16, and Mathematics and Computers in Simulation, Vol. 35, IMACS, North Holland, 1993. News of IMACS, Vol. 35, No. 4, 375–382, Oct. 1993.
3. American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std. 754-1987, New York, 1985. (reprinted in SIGPLAN **22**, 2, pp. 9-25, 1987). Also adopted as IEC Standard 559:1989, Revised version 2008.
4. American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std. 854-1987, New York, 1987.
5. The *IFIP WG - IEEE 754R letter*, dated September 4, 2007.
6. The *IFIP WG - IEEE P1788 letter*, dated September 9, 2009.
7. G. Alefeld, *Intervallrechnung über den komplexen Zahlen und einige Anwendungen*, Dissertation, Universität Karlsruhe, 1968.
8. Ch. Baumhof, *A new VLSI vector arithmetic coprocessor for the PC*, in: Institute of Electrical and Electronics Engineers (IEEE), S. Knowles and W. H. McAllister (eds.), *Proceedings of 12th Symposium on Computer Arithmetic ARITH*, Bath, England, July 19–21, 1995, pp. 210–215, IEEE Computer Society Press, Piscataway, NJ, 1995.
9. Ch. Baumhof, *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaftlichen Rechnens*, Dissertation, Universität Karlsruhe, 1996.

10. G. Bohlender, *Floating-Point Computation of Functions with Maximum Accuracy*, IEEE Transactions on Computers, Vol. C-26, no. 7, July 1977.
11. G. Bohlender, *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen*, Dissertation, Universität Karlsruhe, 1978.
12. T. Teufel, *Ein optimaler Gleitkommaprozessor*, Dissertation, Universität Karlsruhe, 1984.
13. R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Sprachbeschreibung mit Beispielen*, Springer, Berlin Heidelberg New York, 1991.
    See also http://www.math.uni-wuppertal.de/ xsc/ or http://www.xsc.de/.
14. R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Language Reference with Examples*, Springer, Berlin Heidelberg New York, 1992.
    See also http://www.math.uni-wuppertal.de/ xsc/ or http://www.xsc.de/.
    Russian translation MIR, Moscow, 1995, third edition 2006.
    See also http://www.math.uni-wuppertal.de/ xsc/ or http://www.xsc.de/.
15. R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *Numerical Toolbox for Verified Computing I: Basic Numerical Problems (PASCAL-XSC)*, Springer, Berlin Heidelberg New York, 1993.
    Russian translation MIR, Moskau, 2005.
16. R. Klatte, U. Kulisch, C. Lawo, M. Rauch and A. Wiethoff, *C-XSC – A C++ Class Library for Extended Scientific Computing*, Springer, Berlin Heidelberg New York, 1993.
    See also http://www.math.uni-wuppertal.de/xsc/ or http://www.xsc.de/.
17. R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *C++ Toolbox for Verified Computing: Basic Numerical Problems*. Springer, Berlin Heidelberg New York, 1995.
18. R. Kirchner, U. Kulisch, *Hardware support for interval arithmetic*. Reliable Computing, 225–237, 2006.
19. U. Kulisch, *An axiomatic approach to rounded computations*, TS Report No. 1020, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, 1969, and *Numerische Mathematik* 19 (1971), 1–17.
20. 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.
21. U. Kulisch, *Grundlagen des Numerischen Rechnens - Mathematische Begründung der Rechnerarithmetik*, Bibliographisches Institut, Mannheim Wien Zürich, 1976.
22. U. Kulisch, *Complete interval arithmetic and its implementation on the computer*, in: A. Cuyt, et al. (eds.), *Numerical Validation in Current Hardware Architectures*, LNCS, Vol. 5492, pp. 7–26, Springer-Verlag, Heidelberg, 2008.
23. U. Kulisch, *Computer Arithmetic and Validity – Theory, Implementation, and Applications*, de Gruyter, Berlin, 2008, second edition 2013.
24. U. Kulisch, *Arithmetic Operations for Floating-Point Intervals*, as Motion 5 accepted by the IEEE Standards Committee P1788 as definition of the interval operations. See [27].
25. U. Kulisch, V. Snyder, *The Exact Dot Product as Basic Tool for Long Interval Arithmetic*.
26. U. Kulisch, *An Axiomatic Approach to Computer Arithmetic with an appendix on Interval Hardware*, LNCS, Springer-Verlag, Heidelberg, 484-495, 2012.
27. J. D. Pryce (Ed.), *P1788, IEEE Standard for Interval Arithmetic*,
    http://grouper.ieee.org/groups/1788/email/pdfOWdtH2mOd9.pdf.
28. 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.
29. M. Nehmeier, S. Siegel, J. Wolff von Gudenberg, *Specification of Hardware for Interval Arithmetic*, Computing, 2012.
30. S. Oishi, K. Tanabe, T. Ogita and S. M. Rump, *Convergence of Rump's method for inverting arbitrarily ill-conditioned matrices,* Journal of Computational and Applied Mathematics 205 (2007), 533–544.
31. S. M. Rump, *Kleine Fehlerschranken bei Matrixproblemen*, Dissertation, Universität Karlsruhe, 1980.
32. IBM, *IBM System/370 RPQ. High Accuracy Arithmetic*, SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1984.
33. IBM, *IBM High-Accuracy Arithmetic Subroutine Library (ACRITH)*, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), third edition, 1986.
    1. General Information Manual, GC 33-6163-02.
    2. Program Description and User's Guide, SC 33-6164-02.
    3. Reference Summary, GX 33-9009-02.
34. IBM, *ACRITH–XSC: IBM High Accuracy Arithmetic – Extended Scientific Computation. Version 1, Release 1*, IBM Deutschland GmbH (Department 3282, 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.

35. SIEMENS, *ARITHMOS (BS 2000) Unterprogrammbibliothek für Hochpräzisionsarithmetik. Kurzbeschreibung, Tabellenheft, Benutzerhandbuch*, SIEMENS AG, Bereich Datentechnik, Postfach 83 09 51, D-8000 München 83, Bestellnummer U2900-J-Z87-1, September 1986.
36. INTEL, Intel Architecture Instruction Set Extensions Progamming Reference, 319433-017, December 2013, http://software.intel.com/en-us/file/319433-017pdf.