High speed exception-free interval arithmetic, from closed and bounded real intervals to connected sets of real numbers

Ulrich W. Kulisch

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

Abstract. This paper gives a brief sketch of the development of interval arithmetic. Early books consider interval arithmetic for closed and bounded real intervals. It was then extended to unbounded real intervals. Considering $-\infty$ and $+\infty$ only as bounds but not as elements of unbounded real intervals leads to an exception-free calculus.
Formulas for computing the lower and the upper bound of the interval operations including the dot product are independent of each other. On the computer high speed can and should be obtained by computing both bounds in parallel and simultaneously. Another increase of speed and accuracy can be obtained by computing dot products exactly.
Arithmetic for closed real intervals even can be extended to open and half-open real intervals, to connected sets of real numbers. Also this leads to a calculus that is free of exceptions.

1 Remarks on the History of Interval Arithmetic

In early books on Interval Arithmetic by R. E. Moore, [30], G. Alefeld and J. Herzberger [7, 8], E. Hansen [15], and others interval arithmetic is defined and studied for closed and bounded real intervals. Frequent attempts to extend it to unbounded intervals [19, 20, 37] led to inconsistencies again and again. If $-\infty$ and $+\infty$ are considered as elements of a real interval, unsatisfactory operations like $\infty - \infty$, $0 \cdot \infty$, ∞/∞ occur and are to be dealt with.

The books [25, 27] eliminated these problems. Here interval arithmetic just deals with closed and connected sets of real numbers. Since $-\infty$ and $+\infty$ are not real numbers, they can not be elements of a real interval. They only serve as bounds for the description of real intervals. In real analysis a set of real numbers is called closed, if its complement is open. So intervals like $(-\infty, a]$ or $[b, +\infty)$ with real numbers a and b nevertheless are closed real intervals.

Formulas for the operations for unbounded real intervals can now be obtained from those for bounded real intervals by continuity considerations. Obscure operations as mentioned above do not occur in the operations for unbounded real intervals. For a proof of this assertion see section 4.10 in [27]. This result also remains valid for floating-point interval arithmetic. For details and proof see section 4.12 in [27]. Fortunately, this understanding of arithmetic for unbounded real and floating-point intervals was accepted by IEEE 1788 [31].

Early books on interval arithmetic as mentioned in the first paragraph just make use of the four basic arithmetic operations add, subtract, multiply, and divide $(+, -, \cdot, /)$ for real and floating-point intervals. The latter are provided with maximum accuracy. Later books [16–20, 25, 27] in addition provide and make use of an exact dot product.

2 High Speed Interval Arithmetic by Exact Evaluation of Dot Products

Since 1989 major scientific communities like GAMM and the IFIP Working Group on Numerical Software repeatedly required [1, 2, 5, 6] exact evaluation of dot products of two floating-point vectors on computers. The exact dot product (EDP) brings speed and accuracy to floating-point and interval arithmetic.

Solution of a system of linear equations is a central task of Numerical Analysis. A guaranteed solution can be obtained in two steps. The first step computes an approximate solution by some kind of Gaussian elimination in conventional floating-point arithmetic. A second step, the verification step, then computes a highly accurate enclosure of the solution.

By an early estimate of S. M. Rump [36] the verification step can be done with less than 6 times the number of elementary floating-point operations needed for computing an approximation in the first step.

The verification step just consists of dot products. For details see Section 9.5 on Verified Solution of Systems of Linear Equations, pp. 333-340 in [27]. Hardware implementations of the EDP at Karlsruhe in 1993 [9, 10] and at Berkeley in 2013 [11] show that it can be computed in about 1/6th of the time needed for computing a possibly wrong result in conventional floating-point arithmetic! So the EDP reduces the computing time needed for the verification step to the one needed for computing an approximate solution by Gaussian elimination. In other words: A guaranteed solution of a system of linear equations can be computed in twice the time needed for computing an approximation in conventional floating-point arithmetic.

The time needed for solving a system of linear equations can additionally be reduced if the EDP is already applied during Gaussian elimination in the first step. The inner loop here just consists of dot products. The EDP would reduce the computing time and additionally increase the accuracy of the approximate solution.

Using a software routine for a correctly rounded dot product as an alternative for a hardware implemented EDP leads to a comparatively slow process. A correctly rounded dot product is built upon a computation of the dot product in conventional floating-point arithmetic. This is already 5 to 6 times slower than an EDP. High accuracy then is obtained by clever and sophisticated mathematical considerations which all together make it slower than the EDP by more than one magnitude. High speed and accuracy, however, are essential for acceptance and success of interval arithmetic.

The simplest and fastest way computing a dot product is to compute it exactly. The unrounded products are accumulated into a modest fixed-point register on the arithmetic unit with no memory involvement. By pipelining this can be done in the time the processor needs to read the data, i.e., no other method can be faster, pp. 267-300 in [27], and [28]. Rounding the EDP, if necessary, is done only once at the very end of the accumulation.

A frequent argument against computing dot products exactly is that it needs an accumulator of about 4 thousand bits. This, however, is not well taken. The 4 thousand bits are a consequence of the huge exponent range of the IEEE 754 arithmetic standard. It aims for reducing the number of under- and overflows in a floating-point computation. There is no under- and overflow, however, in interval arithmetic. Interval arithmetic does not need an extreme exponent range of $10^{\pm 308}$ or $2^{\pm 1023}$. If in an interval computation a bound becomes $-\infty$ or $+\infty$ the other bound still is a finite floating-point number. In a following operation this interval can become finite again.

Floating-point and interval arithmetic are distinct calculi. Floating-point arithmetic as specified by IEEE 754 is full of complicated constructs, data and events like rounding to nearest, overflow, underflow, $+\infty$, $-\infty$, $+0$, -0 as numbers, or

operations like $\infty - \infty$, ∞/∞, $0 \cdot \infty$. All these constructs do not occur in interval arithmetic. In contrast to this, reasonably defined interval arithmetic leads to an exception-free calculus. It is thus only reasonable to keep the two calculi strictly separate.

Program packages for interval arithmetic for the IBM /370 architecture developed by different commercial companies like IBM, Siemens, Hitachi, and others in the 1980's provide and make use of an exact dot product [38–41]. See also [42].

3 From Closed Real Intervals to Connected Sets of Real Numbers

For about 40 years interval arithmetic was defined for the set of closed and bounded real intervals. The books [25, 27] extended it to unbounded real intervals. The book *The End of Error* by John Gustafson [14] finally shows that it can even be extended to just connected sets of real numbers. These can be closed, open, half open, bounded or unbounded. The book shows that arithmetic for this expanded set is closed under addition, subtraction, multiplication, division, also square root, powers, logarithm, exponential, and many other elementary functions needed for technical computing, i.e., arithmetic operations for connected sets of real numbers always lead to a connected set of real numbers. The calculus is free of exceptions. It remains free of exceptions if the bounds are restricted to a floating-point screen. John Gustafson shows in his book that this extension of interval arithmetic opens new areas of applications.

A detailed description and analysis of this expanded interval arithmetic for connected sets of real numbers including an exact dot product is given in [29].

In accordance with [14] we choose a floating-point number format as shown in Figure 1. It consists of a sign s, an exponent and a fraction part followed by a particular bit. We call this bit the ubit, u for short. A ubit $u = 0$ represents a closed and a ubit $u = 1$ an open interval bracket.

Figure 1 shows the format of a floating-point number.

Fig. 1. The floating-point number format.

So the ubit u allows to distinguish between open and closed interval bounds. A bound of the result of an interval operation can only be closed, if both operands are closed interval bounds. So in the majority of cases the bound in the result will be open.

4 Computing Dot Products Exactly

We now consider a general floating-point number system $F = F(b, f, emax, emin)$, with base b, f bits of the fraction, greatest and least exponent $emax$ and $emin$, respectively.

Let $a = (a_i)$, $b = (b_i)$ be two vectors with n components which are floating-point numbers $a_i, b_i \in F(b, f, emax, emin)$, for $i = 1(1)n$. We compute the sum
$s := \sum_{i=1}^{n} a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n, \quad a_i \cdot b_i \in F(b, 2f, 2emax, 2emin)$,

for $i = 1(1)n$, where all additions and multiplications are the operations for real numbers.

Then a register of

$$L = k + 2 \cdot emax + 2f + 2 \cdot |emin|$$

bits suffices for computing dot products exactly. Here k denotes a number of guard digits for counting intermediate overflows of the register. It is important to note that the size of this register only depends on the data format. In particular it is independent of the number n of components of the two vectors to be multiplied.

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

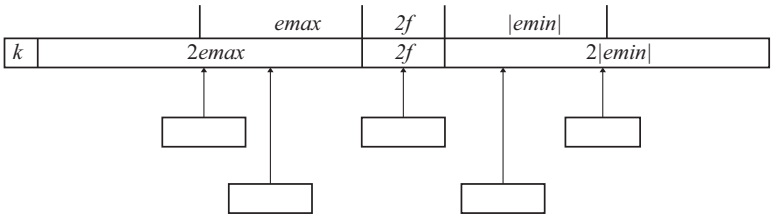

Fig. 2. Complete register for exact scalar product accumulation.

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 complete register will occur in the lifetime of the computer.

We now roughly analyze the number of bits for the register L for four different data formats. Since most computers nowadays use the IEEE 754 arithmetic standard we begin our discussion with this case **I**. However, we mention here already that for practical realizations the cases **II**, **III**, and **IV** are the more attractive.

I. A 64-bit floating-point arithmetic related to IEEE 754 double precision.
II. A 64-bit floating-point arithmetic with a binary exponent range of about ± 256.
III. A 64-bit floating-point arithmetic with a binary exponent range of about ± 128.
IV. A 32-bit floating-point arithmetic with a binary exponent range of about ± 128.

I. We begin with the IEEE 754 format double precision. One bit is needed for the representation of the ubit. So we shrink the exponent part from 11 to 10 bits only. Then we have $b = 2$; word length 64 bits; 1 bit sign; 10 bits for the exponent; $f = 53$ bits; $emin = -511$, $emax = 512$. The entire unit consists of
$L = k + 2 \cdot emax + 2 \cdot f + 2 \cdot |emin| = k + 1024 + 106 + 1022 = k + 2152$. With $k = 24$ we get $L = 2176$ bits. It can be represented by 34 words of 64 bits. L is independent of n.

Figure 3 informally describes the implementation of the EDP. The complete register (here represented as a chest of drawers) is organized in words of 64 bits. The exponent of the products consists of 11 bits. The leading 5 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 (or borrow in case of subtraction) is

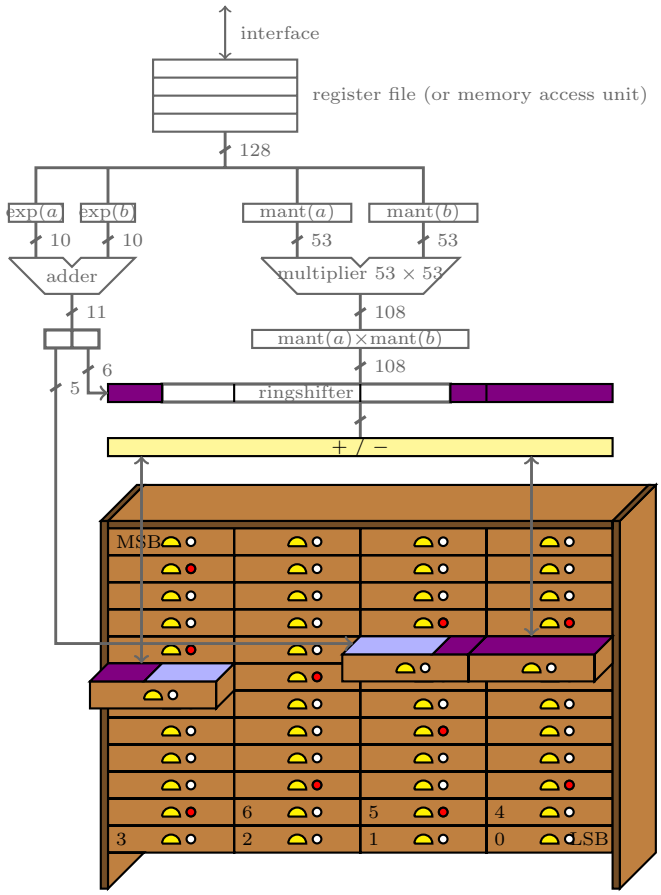

Fig. 3. Illustration for computing the exact dot product.

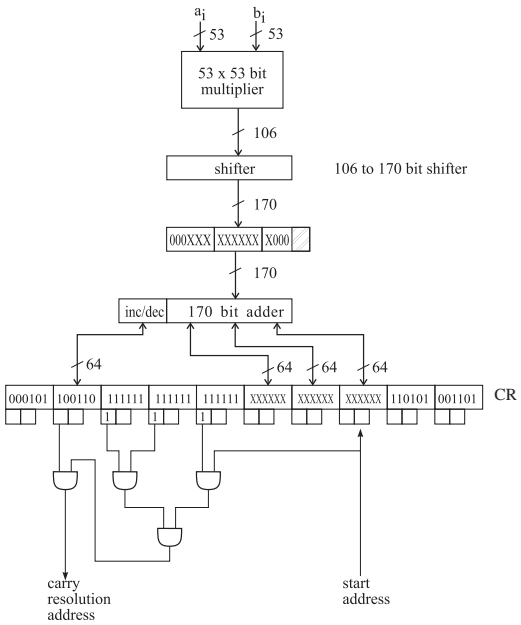

Fig. 4. Parallel accumulation of a product into the CR.

absorbed by the next more significant word in which not all bits are 1 (or 0 for subtraction). For fast detection of this word two flags are attached to each register word. One of these is set 1 (resp. 0) if all bits of the word are 1 (resp. 0). 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. Possible carries can be eliminated simultaneously with the addition. For more details see [25, 27].

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 of 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 [25] or [27]. Rounding the EDP into a correctly rounded dot product is done only once at the very end of the accumulation.

In [25] and [27] 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.

In a floating-point computation an overflow in general means a total breakdown of the accuracy, not so in interval arithmetic. Intervals bring the continuum on the computer. An overflow of the upper bound of an interval leads to an unbounded real interval but not to a total breakdown of the accuracy, since in any case the lower bound is a finite floating-point number. For the next operation the result can already be a finite interval again.

II. A 64-bit floating-point arithmetic with a binary exponent range of about ± 255. For interval arithmetic an exponent range between -77 and $+77$ in decimal seems to be more reasonable. This is a huge range of numbers.[1] It is 1/4th of the exponent range of the IEEE 754 floating-point format double precision. Then 9 bits suffice for the representation of the exponent. So in comparison with the 11 bits of the IEEE 754 number representation two bits are left for other purposes. We use one of these bits for extending the number of fraction bits by one from 53 to 54 and the other bit to indicate whether the interval bracket is open or closed. So in case of a 64-bit interval bound, one bit is used for the sign s, 9 bits are used for the exponent, 53 bits for the fraction and one bit for the ubit u. As usual the leading bit of the fraction of a normalized binary floating-point number is not stored, so the fraction actually consists of 54 bits. For the exponent *emin* subnormal numbers with a denormalized mantissa are permitted.

For this data format with $f = 54, emax = |emin| = 255$, and $k = 24$, we get for $L = 24 + 510 + 108 + 510 = 1152$ bits. This register can be represented by 18 words of 64 bits.

As justification for the exponent range of $emax = |emin| = 255$ we just mention that the data format *long* of the IBM /370 architecture covers a range of about 10^{-75} to 10^{75}. This architecture dominated the market for more than 25 years and most problems could conveniently be solved with machines of this architecture

[1] The number of atoms in the universe is less than 10^{80}.

within this range of numbers. We mention once more that there is no under- and overflow in interval arithmetic.

In Numerical analysis, in general, the dot product is a stable arithmetic operation with a modest exponent range. So assuming an excessive exponent range for implementing the EDP appears unappropriate. Reducing the exponent range simplifies the implementation of the EDP significantly. In case of an exponent overflow a corresponding software routine could be called.

III. A 64-bit floating-point arithmetic with a binary exponent range of about ± 127. A reduction of the exponent range to 8 bits with $emax = |emin| = 127$ would allow an extension of the fraction by one more bit to $f = 55$ bits. This leads to a register of $L = k + 2emax + 2f + 2|emin| = k + 254 + 110 + 254 = k + 618$ bits and with $k = 22$ to 10 words of 64 bits.

IV. A 32-bit floating-point arithmetic with a binary exponent range of about ± 127. For the data format single precision with a word length of 32 bits the size L of the register for computing dot products exactly even shrinks to 9 words of 64 bits: 1 bit is used for the sign, 8 bits are used for the exponent, 23 bits for the fraction, one bit is used for the ubit u, $emax = 127$, and $emin = -127$. So with $k = 22$ we get $L = k + 2emax + 2f + 2|emin| = 22 + 254 + 46 + 254 = 576$ bits. This register can be represented by 9 words of 64 bits.

The formulas for computing the lower and the upper bound of an interval operation are independent of each other. This also holds for the EDP. So on the computer the lower and the upper bound of the result of an interval operation can and should be computed simultaneously in parallel. This allows performing any interval operation at the speed of the corresponding floating-point operation. For details see Section 7.3 in [27], or [29]. High speed is essential for acceptance and success of interval arithmetic.

5 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 (1675). Also old commercial mechanic calculators added numbers and products of numbers into a wide fixed-point register, Figure 5. 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.

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 a recommended new data format *complete*, [27].

In summary it can be said: The technique of speeding up computing by accumulating numbers and 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 the conventional floating-point errors. So finally this technique was abolished.

Fig. 5. Mechanical computing devices equipped with the desired capability:
Burkhart Arithmometer, Glashütte, Germany, 1878;
Brunsviga, Braunschweig, Germany, 1917;
MADAS, Zürich, Switzerland, 1936; (**M**ult., **A**utomatic **D**ivision, **A**dd., **S**ubtr.)
MONROE, New Jersey, USA, 1956.

Conclusion: 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 of the unrounded products 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.

Acknowledgement: The author owes thanks to Goetz Alefeld and Gerd Bohlender 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, and in ISO/IEC/IEEE 60559:2011.
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 2.5 - IEEE 754R letter*, dated September 4, 2007.
6. The *IFIP WG 2.5 - IEEE P1788 letter*, dated September 9, 2009.
7. G. Alefeld and J. Herzberger, *Einführung in die Intervallrechnung*, Informatik 12, Bibliographisches Institut, Mannheim Wien Zürich, 1974.
8. G. Alefeld and J. Herzberger, *Introduction to Interval Computations*, Academic Press, New York, 1983.
9. 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 the 12th Symposium on Computer Arithmetic ARITH*, Bath, England, July 19–21, 1995, pp. 210–215, IEEE Computer Society Press, Piscataway, NJ, 1995.
10. Ch. Baumhof, *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaft-lichen Rechnens*, Dissertation, Universität Karlsruhe, 1996.
11. D. Biancolin and J. Koenig, *Hardware Accelerator for Exact Dot Product*, ASPIRE Laboratory, University of California, Berkeley, 2015.
12. G. Bohlender, *Floating-Point Computation of Functions with Maximum Accuracy*, IEEE Transactions on Computers, Vol. C-26, no. 7, July 1977.
13. G. Bohlender, *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen*, Dissertation, Universität Karlsruhe, 1978.
14. J. L. Gustafson *The End of Error.* CRC Press, Taylor and Francis Group, A Chapman and Hall Book, 2015.
15. E. R. Hansen, *Topics in Interval Analysis*, Clarendon Press, Oxford, 1969.
16. 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://www2.math.uni-wuppertal.de/xsc/ or http://www.xsc.de/.
17. 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 : //www2.math.uni − wuppertal.de/ ∼ xsc/* or http://www.xsc.de/.
    Russian translation MIR, Moscow, 1995, third edition 2006.
    See also *http : //www2.math.uni − wuppertal.de/ ∼ xsc/* or http://www.xsc.de/.
18. 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, Moscow, 2005.
19. 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://www2.math.uni-wuppertal.de/xsc/ or http://www.xsc.de/.
20. R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *C++ Toolbox for Verified Computing: Basic Numerical Problems.* Springer, Berlin Heidelberg New York, 1995.
21. R. Kirchner, U. Kulisch, *Hardware support for interval arithmetic.* Reliable Computing, 225–237, 2006.
22. 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.

23. 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.
24. U. Kulisch, *Grundlagen des Numerischen Rechnens - Mathematische Begründung der Rechnerarithmetik*, Bibliographisches Institut, Mannheim Wien Zürich, 1976.
25. U. Kulisch, *Advanced Arithmetic for the Digital Computer – Design of Arithmetic Units*, Springer, 2002.
26. U. Kulisch, *An Axiomatic Approach to Computer Arithmetic with an appendix on Interval Hardware*, LNCS 7204, Springer-Verlag, Heidelberg, pp. 484-495, 2012.
27. U. Kulisch, *Computer Arithmetic and Validity – Theory, Implementation, and Applications*, de Gruyter, Berlin, 2008, second edition 2013.
28. U. Kulisch and G. Bohlender, *High Speed Associative Accumulation of Floating-point Numbers and Floating-point Intervals*, Reliable Computing-23-pp-141-153, 2016.
29. U. Kulisch, *Up-to-date Interval Arithmetic: From Closed Intervals to Connected Sets of Real Numbers*, in R. Wyrzykowski (Ed.): PPAM 2015, Part II, LNCS 9574, pp. 413-434, 2016.
30. R. E. Moore, *Interval Analysis*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1966.
31. J. D. Pryce (Ed.), *P1788, IEEE Standard for Interval Arithmetic*, http://grouper.ieee.org/groups/1788/email/pdfOWdtH2mOd9.pdf.
32. 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.
33. M. Nehmeier, S. Siegel, J. Wolff von Gudenberg, *Specification of Hardware for Interval Arithmetic*, Computing, 2012.
34. 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.
35. S. M. Rump, *Kleine Fehlerschranken bei Matrixproblemen*, Dissertation, Universität Karlsruhe, 1980.
36. S. M. Rump and E. Kaucher, *Small bounds for the Solution of Systems of Linear Equations*, in Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis), G. Alefeld and R. D. Grigorieff (eds.), Computing Supplementum 2, Springer-Verlag Berlin Heidelberg Wien, 157–164, 1980.
37. Sun Microsystems, Interval Arithmetic Programming Reference, Fortran 95, Sun Microsystems Inc., Palo Alto (2000).
38. IBM, *IBM System/370 RPQ. High Accuracy Arithmetic*, SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1984.
39. 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.
40. 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.
41. 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.
42. INTEL, Intel Architecture Instruction Set Extensions Progamming Reference, 319433-017, December 2013, http://software.intel.com/en-us/file/319433-017pdf.