A LUCID INTERVAL

Brian Hayes

Give a digital computer a problem in arithmetic, and it will grind away methodically, tirelessly, at gigahertz speed, until ultimately it produces the wrong answer. The cause of this sorry situation is not that software is full of bugs—although that is very likely true as well—nor is it that hardware is unreliable. The problem is simply that computers are discrete and finite machines, and they cannot cope with some of the continuous and infinite aspects of mathematics. Even an innocent-looking number like ⅒ can cause no end of trouble: In most cases, the computer cannot even read it in or print it out exactly, much less perform exact calculations with it.

Errors caused by these limitations of digital machines are usually small and inconsequential, but sometimes every bit counts. On February 25, 1991, a Patriot missile battery assigned to protect a military installation at Dahrahn, Saudi Arabia, failed to intercept a Scud missile, and the malfunction was blamed on an error in computer arithmetic. The Patriot's control system kept track of time by counting tenths of a second; to convert the count into full seconds, the computer multiplied by ⅒. Mathematically, the procedure is unassailable, but computationally it was disastrous. Because the decimal fraction ⅒ has no exact finite representation in binary notation, the computer had to approximate. Apparently, the conversion constant stored in the program was the 24-bit binary fraction 0.000110011001100110011001100, which is too small by a factor of about one ten-millionth. The discrepancy sounds tiny, but over four days it built up to about a third of a second. In combination with other peculiarities of the control software, the inaccuracy caused a miscalculation of almost 700 meters in the predicted position of the incoming missile. Twenty-eight soldiers died.

Of course it is not to be taken for granted that better arithmetic would have saved those 28 lives. (Many other Patriots failed for unrelated reasons; some analysts doubt whether *any* Scuds were stopped by Patriots.) And surely the underlying problem was not the slight drift in the clock but a design vulnerable to such minor timing glitches. Nevertheless, the error in computer multiplication is mildly disconcerting. We would like to believe that the mathematical machines that control so much of our lives could at least do elementary arithmetic correctly.

One approach to dealing with such numerical errors is a technique called interval arithmetic. It does nothing directly to improve the accuracy of calculations, but for every number it provides a certificate of accuracy—or the lack of it. The result of an ordinary (non-interval) computation is a single number, a point on the real number line, which lies at some unknown distance from the true answer. An interval computation yields a pair of numbers, an upper and a lower bound, which are guaranteed to enclose the exact answer. Maybe you still don't know the truth, but at least you know how much you don't know.

Measuring Ignorance

Suppose you are surveying a rectangular field. No matter how carefully you read the measuring tape, you can never be certain of the exact dimensions, but you can probably state with confidence that the correct figures lie within certain bounds. Perhaps you are sure the length is no less than 68 meters and no more than 72, while the width is between 49 and 51 meters. Then you can state with equal confidence that the area of the field lies somewhere between 3,332 square meters and 3,672 square meters. This is interval arithmetic in action: $[68, 72] \times [49, 51] = [3{,}332, 3{,}672]$. (The bracketed pair $[a, b]$ signifies the interval from a to b inclusive, with $a \le b$. Another useful notation is $[\underline{x}, \overline{x}]$, where the underscore indicates a lower limit and the overscore an upper limit.)

Doing basic arithmetic with intervals is not much more difficult than with ordinary ("point-like") numbers. Indeed, a single formula extends the definition of the four standard arithmetic operations to intervals. If $\circ$ represents any of the operations $+$, $-$, $\times$ or $\div$, then the corresponding interval operation is defined as:

$$[\underline{u}, \overline{u}] \circ [\underline{v}, \overline{v}] = [\, min(\underline{u} \circ \underline{v}, \underline{u} \circ \overline{v}, \overline{u} \circ \underline{v}, \overline{u} \circ \overline{v}),\ max(\underline{u} \circ \underline{v}, \underline{u} \circ \overline{v}, \overline{u} \circ \underline{v}, \overline{u} \circ \overline{v})\,].$$

In other words, compute the four possible combinations of the upper and lower bounds, then

Brian Hayes is Senior Writer for American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701; bhayes@amsci.org*

choose whichever of these results is the smallest as the new lower bound, and likewise take the largest as the new upper bound. Every possible combination $u \circ v$ must lie within these limits.

The *min-max* formula is a convenient definition of interval operations, but it is not always the best implementation. For example, in the case of addition it's obvious that $\underline{u}+\underline{v}$ will always be the smallest sum, and $\overline{u}+\overline{v}$ the largest, so that interval addition is simply $[\underline{u},\overline{u}]+[\underline{v},\overline{v}] = [\underline{u}+\underline{v},\overline{u}+\overline{v}]$. By similar reasoning, subtraction is just $[\underline{u},\overline{u}]-[\underline{v},\overline{v}] = [\underline{u}-\overline{v},\overline{u}-\underline{v}]$. Multiplication is not quite as well-behaved. Although shortcuts are sometimes possible (depending on the signs of the operands), in the worst case there is no choice but to compute all four of the combinations and select the extrema.

Division is much like multiplication, but with a further annoyance—the possibility of a zero divisor. With pointlike numbers, if you try to carry out an operation such as $2 \div 0$, the error is obvious, and the system software will prevent you from doing the impossible. An interval division such as $[2,4] \div [-1,1]$ has the same problem, but in disguise. You could very well perform the necessary calculations on the end points of the intervals without raising any alarms and without noticing that the divisor interval includes the value zero. But the answer you would arrive at in this way, $[-4,4]$, is utterly wrong. It's not just wrong in the formal sense that it might be tainted by an illegal operation. It's also wrong because the interval $[-4,4]$ does not enclose all possible quotients, even if the zero point itself is excluded from the divisor. A reliable system for interval arithmetic needs protection against this hazard. Usually, division by an interval that includes zero is simply forbidden, although there are also other ways of coping with the problem.

Apart from the rules for manipulating intervals arithmetically, there remains the question of what an interval really is and how we ought to think about it. In the context of dealing with errors and uncertainties of computation, we may see $[\underline{x},\overline{x}]$ as standing for some definite but unknown value x such that $\underline{x} \le x \le \overline{x}$. But $[\underline{x},\overline{x}]$ could also be interpreted as the set of *all* real numbers between $\underline{x}$ and $\overline{x}$—in other words, as a closed segment of the real number line. Or the interval $[\underline{x},\overline{x}]$ could be taken as denoting a new kind of number, in much the same way that two real numbers x and y combine to specify the complex number $x+iy$ (where i represents the square root of –1). This last view is the most ambitious. It suggests the goal of a computing system where intervals are just another data type, interchangeable with other kinds of numbers. Wherever a pointlike number can appear in a program, an interval can be substituted. Conversely, exact pointlike numbers can be represented by "degenerate" intervals of zero width; the number 2 could be written $[2,2]$.

Perils of Precision

Why should we have to put up with errors and approximations in computing? Why can't the computer just give the right answer?

Sometimes it can. Calculations done entirely with integers yield exact results as long as the numbers are not too big for the space allotted. Often the allotted space is quite scanty—as little as 16 bits—but this is an artificial constraint; in principle a computer can store the exact value of any integer that will fit in its memory.

Integers have the pleasant property that they form a closed set under addition, subtraction and multiplication; in other words, when you add, subtract or multiply any two integers, you always get another integer. Absent from this list of operations is division, because the quotient of two integers is not always an integer. If we allow numbers to be divided, we must go beyond the integers to the rational numbers, such as ⅔ or ⅗. But rational values can also be represented exactly in the computer; all that's needed is to keep track of a *pair* of integers, which are interpreted as the numerator and the denominator. Thus the constant ⅒, which caused such havoc in the Patriot software, could have been encoded in the two binary integers 1 and 1010. A few programming languages—notably Lisp and its offspring—provide integers of unlimited size ("bignums") and exact rationals as built-in data types. Similar facilities can be added to other languages.

If we can have exact numerical computation, why would anyone choose approximate arith-

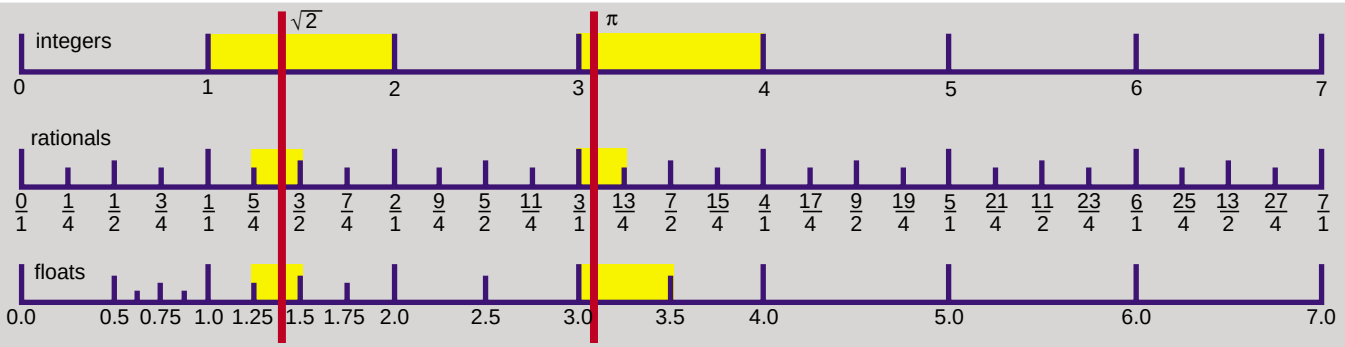

Figure 1. Computer numbering systems include integers, rationals and floating-point numbers, but none of these schemes can represent all possible quantities. For irrational values such as π and the square root of 2, the best you can do is choose the nearest representable number. Intervals (shown here in yellow) bracket an unrepresentable number and thereby put bounds on the error of approximation.

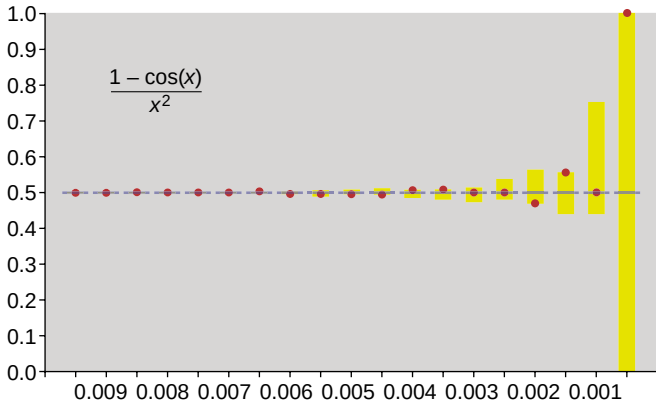

Figure 2. Numerically unstable formulas can cause a catastrophic loss of accuracy; although interval arithmetic cannot prevent such failures, it can issue a warning. Here the blue lines represent the correct value of $1-\cos(x)/x^2$ for values of x near 0. The red dots are the results of a floating-point calculation done with 20 bits of precision; for values of x below about 0.004 these results become unreliable. An interval calculation *(yellow)* shows the progressive loss of certainty. The example was first discussed by Michael J. Schulte and Earl E. Swartzlander, Jr., of the University of Texas at Austin.

metic? One reason is that there are numbers beyond the rationals: No ratio of finite integers gives the exact value of $\sqrt{2}$ or π or $\log_2(3)$. Perhaps more important, exact computations tend to become hopelessly unwieldy. Consider the series $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$. If you sum a thousand terms, the result is vanishingly close to 2, but the exact rational representation fills two thousand binary digits. Doing arithmetic with such obese numbers is slow and cumbersome. And outside the realm of pure mathematics the cost of maintaining exactness is seldom justified. Nothing in the physical world can be measured with such precision anyway.

The usual alternative to exact rational computations is floating-point arithmetic, a scheme that resembles scientific notation. A number takes the form $D \times \beta^E$, where D is called the significand, E is the exponent, and β is the base (which in modern computer systems is always 2). For example, the decimal number 6.0 can be expressed as 0.75×2^3, with significand 0.75 and exponent 3. In this case the representation is exact, in binary as well as in decimal (the binary significand is 0.11). Other numbers are not so lucky. As noted above, no finite significand corresponds exactly to the decimal fraction $\frac{1}{10}$. Furthermore, it's obvious that *some* numbers must be missing from the system simply because it has only a finite capacity. In one common floating-point format, the total space available for storing the significand and the exponent is 32 bits, and so the system cannot possibly hold more than 2^{32} distinct numbers, or about 4 billion of them. If you need a number that is not a member of this finite set, the best you can do is choose the nearest member as an approximation. The difference between the true value and the approximation is the roundoff error.

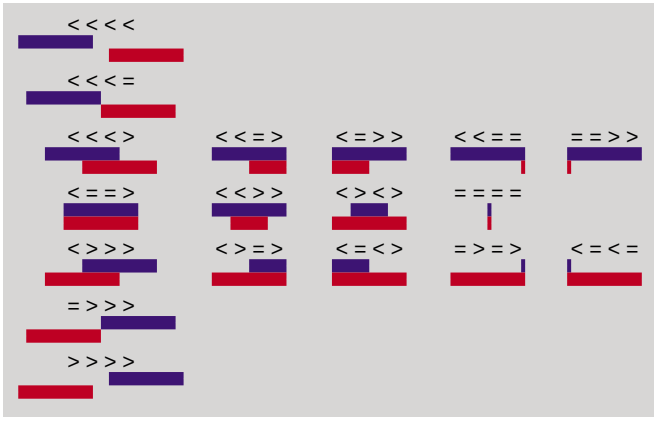

Figure 3. Comparisons between intervals are more complicated than those of pointlike numbers. There are 18 meaningful relations between intervals (including cases where one or both intervals are in fact pointlike). It's unclear even how to name all these comparisons; one encoding lists the relations of the four pairs of end points in a fixed sequence.

Interval arithmetic cannot eliminate roundoff error, but it can fence it in. When a result x falls between two floating-point values, those nearest representable numbers become the lower and upper bounds of the interval $[\underline{x}, \overline{x}]$. But this is not the end of the story. Subsequent computations could yield a new interval for which $\underline{x}$ and $\overline{x}$ are themselves numbers that have no exact floating-point representation. In this situation, where even the interval has to be approximated, rounding must be done with care. To preserve the guarantee that the true value always lies within the interval, the end points of the interval must be rounded "outward": $\underline{x}$ is rounded down and $\overline{x}$ is rounded up.

Historical Intervals

Interval arithmetic is not a new idea. Invented and reinvented several times, it has never quite made it into the mainstream of numerical computing, and yet it has never been abandoned or forgotten either.

In 1931 Rosalind Cicely Young, a recent Cambridge Ph.D., published an "algebra of many-valued quantities" that gives rules for calculating with intervals and other sets of real numbers. Of course Young and others writing in that era did not see intervals as an aid to improving the reliability of machine computation. By 1951, however, in a textbook on linear algebra, Paul S. Dwyer of the University of Michigan was describing arithmetic with intervals (he called them "range numbers") in a way that is clearly directed to the needs of computation with digital devices.

A few years later, the essential ideas of interval arithmetic were set forth independently and almost simultaneously by three mathematicians—Mieczyslaw Warmus in Poland, Teruo Sunaga in Japan and Ramon E. Moore in the United States. Moore's version has been the most influential, in part because he emphasized solutions to problems of machine computation but also because he has continued for more than four decades to publish on interval methods and to promote their use.

Today the interval-methods community includes active research groups at a few dozen uni-

versities. A web site at the University of Texas at El Paso (www.cs.utep.edu/interval-comp) provides links to these groups as well as a useful archive of historical documents. The journal *Reliable Computing* (formerly *Interval Computations*) is the main publication for the field; there are also mailing lists and annual conferences. Implementations of interval arithmetic are available both as specialized programming languages and as libraries that can be linked to a program written in a standard language. There are even interval spreadsheet programs and interval calculators.

One thing the interval community has been ardently seeking—so far without success—is support for interval algorithms in standard computer hardware. Most modern processor chips come equipped with circuitry for floating-point arithmetic, which reduces the process of manipulating significands and exponents to a single machine-language instruction. In this way floating-point calculations become part of the infrastructure, available to everyone as a common resource. Analogous built-in facilities for interval computations are technologically feasible, but manufacturers have not chosen to provide them. A 1996 article by G. William Walster of Sun Microsystems asks why. Uncertainty of demand is surely one reason; chipmakers are wary of devoting resources to facilities no one might use. But Walster cites other factors as well. Hardware support for floating-point arithmetic came only after the IEEE published a standard for the format. There have been drafts of standards for interval arithmetic (the latest written by Dmitri Chiriaev and Walster in 1998), but none of the drafts has been adopted by any standards-setting body.

Gotchas

Although the principles of interval computing may seem obvious or even trivial, getting the algorithms right is not easy. There are subtleties. There are gotchas. The pitfalls of division by an interval that includes zero have already been mentioned. Here are a few more trouble spots.

In doing arithmetic, we often rely on mathematical laws or truths such as $x + -x = 0$ and $(a+b)x = ax + bx$. With intervals, some of these rules fail to hold. In general, an interval has no additive inverse; that is, given a nondegenerate interval $[\underline{u}, \overline{u}]$, there is no interval $[\underline{v}, \overline{v}]$ for which $[\underline{u}, \overline{u}] + [\underline{v}, \overline{v}] = [0,0]$. There is no multiplicative inverse either—no pair of nondegenerate intervals for which $[\underline{u}, \overline{u}] \times [\underline{v}, \overline{v}] = [1,1]$. The reason is clear and fundamental: No valid operation can ever diminish the width of an interval, and [0,0] and [1,1] are intervals of zero width.

The distributive law also fails for intervals. In an expression such as $[1,2] \times ([-3,-2] + [3,4])$, it makes a difference whether you do the addition first and then multiply, or do two multiplications and then add. One sequence of operations gives the result [0,4], the other [–3,6]. Strictly speaking, either of these results is correct—both of them bound any true value of the original expression—but the narrower interval is surely preferable.

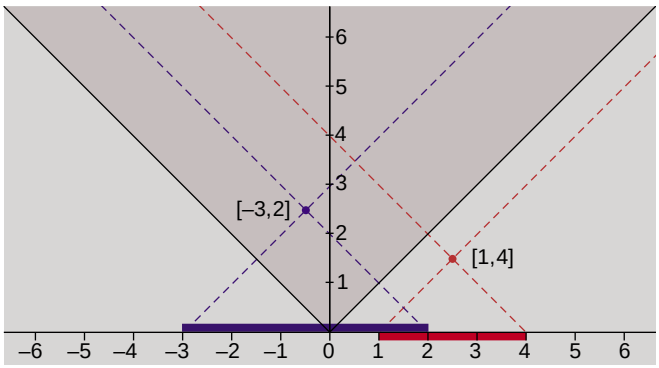

Figure 4. Diagrammatic scheme introduced by Zenon Kulpa of the Polish Academy of Sciences represents an interval as a point on a plane, somewhat like the representation of a complex number. Position of the point along the horizontal axis gives the value of the midpoint of the interval; height on the vertical axis encodes the radius (or half the width) of the interval. Diagonals extended to the horizontal axis reveal the interval itself. Any point within the shaded region represents an interval that includes the value zero.

Another example: squaring an interval. The obvious definition $[\underline{x}, \overline{x}]^2 = [\underline{x}, \overline{x}] \times [\underline{x}, \overline{x}]$ seems to work in some cases, such as $[1,2]^2 = [1,4]$. But what about $[-2,2]^2 = [-4,4]$? Whoops! The square of a real number cannot be negative. The source of the error is treating the two appearances of $[\underline{x}, \overline{x}]$ in the right-hand side of the formula as if they were independent variables; in fact, whatever value x assumes in one instance, it must be the same in the other. The same phenomenon can arise in expressions such as $2x/x$. Suppose x is the interval [2,4]; then naïvely following the rules of interval arithmetic yields the answer [1,4]. But of course the correct value is 2 (or [2,2]) for *any* nonzero value of x.

Comparisons are yet another murky area. Computer programs rely heavily on conditional expressions such as "if $(x < y)$ then...." When x and y are intervals, the comparison gets tricky. Is [1,3] less than [2,4], or not? Whereas there are just three elementary comparisons for pointlike numbers (<, = and >), there are as many as 18 well-defined relations for intervals. It's not always obvious which one to choose, or even how to name them. (Chiriaev and Walster refer to "certainly relations" and "possibly relations.")

Finally, look at what happens if a naïve implementation of the sine function is given an interval argument. Sometimes there is no problem: sin([30°,60°]) yields the correct interval [0.5,0.866]. But sin([30°,150°]) returns [0.5,0.5], which is an error; the right answer is [0.5,1.0]. What leads us astray is the assumption that interval calculations can be based on end points alone, which is true only for monotonic functions (those that never "change direction"). For other functions it is necessary to examine the interior of an interval for minima and maxima.

In fairness, it should be noted that many cherished mathematical truths fail even in ordinary

(noninterval) floating-point arithmetic. An identity such as $x = \sqrt{x^2}$ is not to be trusted in floating point. And there are remedies for all the interval gotchas mentioned above—or at least strategies for coping with them. M. H. van Emden has shown that by building on the existing IEEE floating-point standard (including its facilities for representing infinity), it would be possible to create a system of interval arithmetic that would never fall into an error state, not even as a result of division by zero. (Of course the system would sometimes return results such as $[-\infty, +\infty]$, which may be of questionable utility.)

Intervals at Work

The interval community can point to a number of success stories. In 1995 Joel Hass, Michael Hutchings and Roger Schlafly proved part of the "double-bubble conjecture" by a method that entailed extensive numerical calculations; they used interval methods to establish rigorous bounds on computational errors. The conjecture concerns soap films enclosing a pair of volumes, and states that the common configuration of two conjoined quasi-spherical bubbles has the smallest surface-to-volume ratio. Hass, Hutchings and Schlafly proved the conjecture for the case of two equal volumes, essentially by calculating the best possible ratio for all configurations. The calculations did not have to be exact, but any errors had to be smaller than the differences between the various ratios. Interval methods provided this guarantee. (The general case of the double-bubble conjecture was proved a few years later by Hutchings, Frank Morgan, Manuel Ritoré and Antonio Ros—without interval arithmetic and indeed without computers, using "only ideas, pencil, and paper.")

A quite different application of interval methods was reported in 1996 by Oliver Holzmann, Bruno Lang and Holger Schütt of the University of Wuppertal. Instead of trying to control the errors of a calculation, they were estimating the magnitude of errors in a physical experiment. The experiment was a measurement of Newton's gravitational constant G, done with two pendulums attracted to large brass weights. The interval analysis assessed various contributions to the uncertainty of the final result, and discovered a few surprises. An elaborate scheme had been devised for measuring the distance between the swinging pendulums, and as a result this source of error was quite small; but uncertainties in the height of the brass weights were found to be an important factor limiting the overall accuracy.

Would we be better off if intervals were used for *all* computations? Maybe, but imagine the plight of the soldier in the field: A missile is to be fired if and only if a target comes within a range of 5 kilometers, and the interval-equipped computer reports that the distance is $[4,6]$ kilometers. This is rather like the weather forecast that promises a 50-percent chance of rain. Such statements may accurately reflect our true state of knowledge, but they're not much help when you have to decide whether to light the fuse or take the umbrella. But this is a psychological problem more than a mathematical one. Perhaps the solution is to compute with intervals, but at the end let the machine report a definite, pointlike answer, chosen at random from within the final interval.

Bibliography

Chiriaev, Dmitri, and G. William Walster. 1998. Interval arithmetic specification. http://www.mscs.mu.edu/~globsol/Papers/spec.ps

Dwyer, Paul S. 1951. *Linear Computations*. New York: John Wiley and Sons.

Hass, Joel, Michael Hutchings and Roger Schlafly. 1995. The double bubble conjecture. *Electronic Research Announcements of the AMS* 1:95–102.

Holzmann, Oliver, Bruno Lang and Holger Schütt. 1996. Newton's constant of gravitation and verified numerical quadrature. *Journal of Reliable Computing* 2(3):229–239.

Hyvönen, Eero, and Stefano De Pascale. 1996. Interval computations on the spreadsheet. In *Applications of Interval Computations,* R. Baker Kearfott and Vladik Kreinovich (eds.), pp. 169–209; Dordrecht, Boston: Kluwer Academic.

Kearfott, R. Baker. 1996. *Rigorous Global Search: Continuous Problems*. Dordrecht, Boston: Kluwer Academic.

Kulpa, Zenon. 2003. Diagrammatic analysis of interval linear equations. Part 1: Basic notions and the one-dimensional case. *Reliable Computing* 9:1–20.

Markov, Svetoslav, and Kohshi Okumura. 1999. The contribution of T. Sunaga to interval analysis and reliable computing. In *Developments in Reliable Computing*, Tibor Csendes (ed.), pp. 167–188. Dordrecht, Boston: Kluwer.

Moore, Ramon E. 1966. *Interval Analysis*. Englewood Cliffs, N.J.: Prentice-Hall.

Moore, Ramon E. 1979. *Methods and Applications of Interval Analysis*. Philadelphia: Society for Industrial and Applied Mathematics.

Schulte, Michael J., and Earl E. Swartzlander, Jr. 1996. Software and hardware techniques for accurate, self-validating arithmetic. In *Applications of Interval Computations,* R. Baker Kearfott and Vladik Kreinovich (eds.),pp. 381–404; Dordrecht, Boston: Kluwer Academic.

Semenov, Alexander L. 1996. Solving optimization problems with help of the Unicalc solver. In *Applications of Interval Computations,* R. Baker Kearfott and Vladik Kreinovich (eds.), pp. 211–225; Dordrecht, Boston: Kluwer Academic.

Skeel, Robert. 1992. Roundoff error and the Patriot missile. *SIAM News* 25(4):11.

Sunaga, Teruo. 1958. Theory of interval algebra and its application to numerical analysis. In *RAAG Memoirs, Gqujutsu Bunken Fukuy-kai*. Tokyo, Vol. 2, pp. 29–46. Also at http://www.cs.utep.edu/interval-comp/sunaga.pdf

van Emden, M. H. 2002. New developments in interval arithmetic and their implications for floating-point standardization. http://arXiv.org/abs/cs.NA/0210015

Walster, G. William. 1996. Stimulating hardware and software support for interval arithmetic. In *Applications of Interval Computations,* R. Baker Kearfott and Vladik Kreinovich (eds.), pp. 405–416; Dordrecht, Boston: Kluwer Academic.

Warmus, M. 1956. Calculus of approximations. *Bulletin de l'Academie Polonaise des Sciences* 4(5):253–257. Also at http://www.cs.utep.edu/interval-comp/warmus.pdf

United States General Accounting Office. 1992. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. Washington: General Accounting Office.

Young, Rosalind Cecily. 1931. The algebra of many-valued quantities. *Mathematische Annalen* 104:260–290. Also at http://www.cs.utep.edu/interval-comp/young.pdf