Algebraic Computation, Numerical Computation and Verified Inclusions

Siegfried M. Rump
IBM Development and Research
Schoenaicher Strasse 220
D-7030 Boeblingen
West Germany

Abstract

The three different types of computation - the algebraic manipulation, the numerical computation and the computation of verified results - are aiming on different problems and deliver qualitatively different results, each method having its specific advantages for specific classes of problems. The following remarks give some thoughts on possible combinations of all three methods to obtain algorithms benefitting from the specific strength of either method.

Algebraic computation

Performing algebraic computations on the computer means computing without errors. When calculating in the ring of integers, the field of rational numbers or algebraic number fields the result of every single operation is computed exactly, not approximated by some (floating-point) number. No error occurs in the entire computation and the final result is the exact result to the given problem as well.

Terms like conversion errors, rounding errors or cancellation errors as one may associate with computer calculations do not exist in algebraic computations. In computer algebra we are definitely in the algebraic structure of, say, an algebraic number field with respect to the zero of a defining polynomial. The representation in algebraic computations on the computer is an <u>isomorphic</u> image of the mathematical structure (within the limits of the machine).

Algebraic computation and numerical computation deliver results of different quality and require different amounts of computing time for their tasks. Of course the advantage of performing every calculation exactly has to be payed. And it depends definitely on the problem, on what the user wants to get, whether the price of computing time should be payed or not. But it is inadequate to call one method better than the other: they are not comparable. Computing verified inclusions of a numerical problem is somewhere in between algebraic computation and purely numerical computation. We will come back to this point later.

Numerical computation

A numerical algorithm aims on an approximation of the exact result using an approximate computer arithmetic, usually floating-point arithmetic. The input data is often afflicted with a conversion error if either the data is obtained from a meter or, the data is given decimal whereas the computer has a binary or hexadecimal arithmetic. In this case the problem in the computer is different from the problem the user wants to solve.

Floating-point operations are afflicted with rounding errors, i.e. the result of a single floating-point operation is approximately equal to the exact result of the operation up to a certain error. The relative error of an operation may become very large if two numbers are subtracted which are almost equal. Then the relative error of the operation is small but due to the inaccuracy of the two operands the relative error of the result of the entire calculation may become very large.

Consider the following example on a 5-digit decimal computer. We choose a decimal computer to avoid conversion errors: every input data is exactly within the format of the computer. Let $a=115.4$ and $b=81.6$ and

$$z = a^2 - 2\cdot b^2 \quad .$$

Then $a^2=13317.16$ and $2\cdot b^2=13317.12$. Both intermediate results are to be rounded into the set of floating-point numbers on the computer. The floating-point numbers being immediate neighbours to the intermediate results are 13317.0 and 13318.0 on our 5-digit decimal computer. Obviously the best approximation in either case is 13317.0. The relative

error of the following subtraction 13317.0-13317.0 is in fact zero (assuming exact operands); the error of the final result 0.0 instead of 0.04 is large because the operands of the subtraction where afflicted with rounding errors and cancellation was caused.

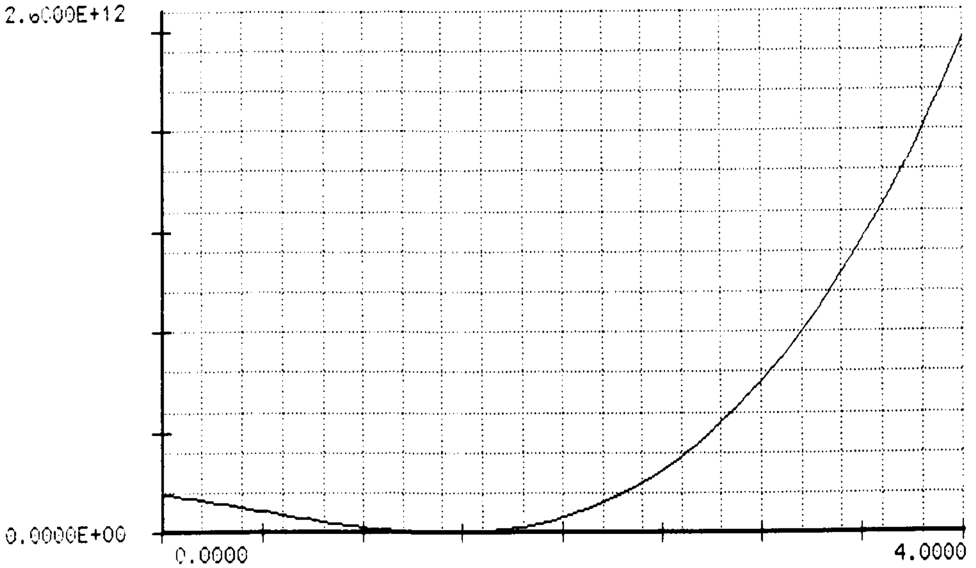
There are many examples of simple floating-point computations yielding approximations far away from the exact solution or even examples where approximations are calculated where in fact no solution exists. Consider the following polynomial:

$$p(x) = 67872320568\, x^3 - 95985956257\, x^2 - 135744641136\, x + 191971912515$$

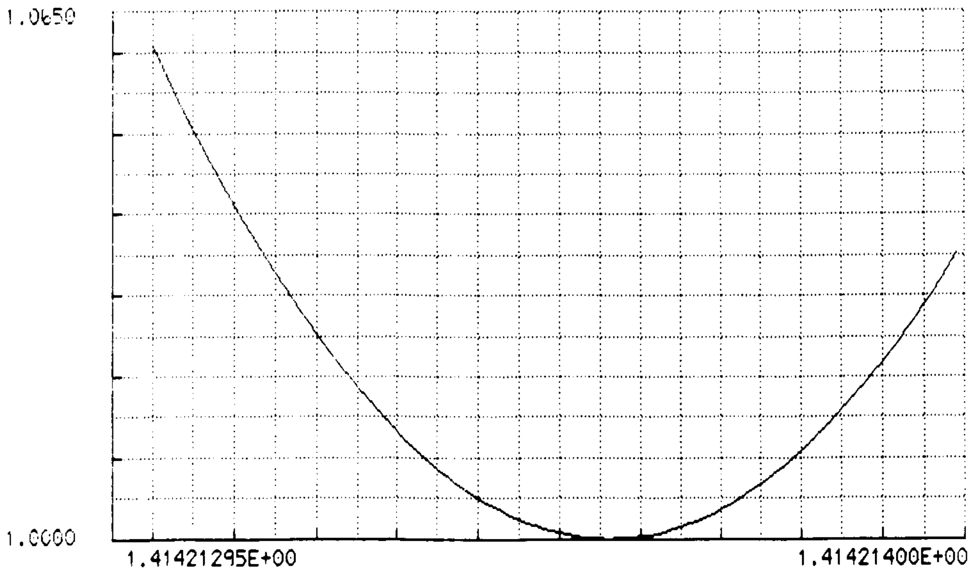
on a 12-digit decimal computer. We apply Newton's iteration with starting value $x^0 := 2.0$ and evaluate the polynomial and its derivative using Horner's scheme. Note that all coefficients of the polynomial and its derivative are exactly representable on the computer and an arithmetic with optimal rounding is used. The following values for the iteration are computed:

2.00000000000	
1.73024785661	0.269752143
1.57979152125	0.150456335
1.49923019011	0.080561331
1.45733317058	0.041897020
1.43593403289	0.021399138
1.42511502231	0.010819011
1.41967473598	0.005440286
1.41694677731	0.002727959
1.41558082832	0.001365949
1.41489735833	0.000683470
1.41455549913	0.000341859
1.41438453509	0.000170964
1.41429903606	0.000085499
1.41425628589	0.000042750
1.41423488841	0.000021397
1.41422414110	0.000010747
1.41421847839	0.000005663
1.41421582935	0.000002649
1.41421353154	0.000002298
1.41421353154	0.000000000
1.41421353154	0.000000000

In the first column the iterates are displayed, in the second column the difference between two adjacent iterates. Obviously the iteration "converges" monotonically with decreasing distance between adjacent iterates to the final value 1.41421353154. In fact there is <u>no</u> positive real zero of p. The graph of the polynomial looks like



and around 1.414213 the graph is



showing all values of p being above 0.

From a numerical point of view one immediately recognizes that the iteration does not show the expected quadratic behaviour. This is clear by visual inspection of the numbers. However, the iteration above might well pass a stopping criterion of a numerical algorithm.

Even the simplest conversion errors might cause an unexpected result of a computation. Only to mention the difference between 40/5 and 40•0.2, the first computation delivering the correct value 8.0 on (hopefully) every computer whereas the answer of the second on many machines is 7.999999 because 0.2 is not exactly representable in binary or hexadecimal format. When rounding the result to integers by chopping the result is 7 instead of 8.

As a last example we mention the following eigenvalue problem, communicated by A. Neumaier. Consider a lower triangular matrix with 30 rows and columns, 1's in the diagonal and identical elements 2 below the diagonal:

$$\begin{bmatrix} 100000000000000000000000000000 \\ 210000000000000000000000000000 \\ 221000000000000000000000000000 \\ 222100000000000000000000000000 \\ \cdots \\ 222222222222222222222222222210 \\ 222222222222222222222222222221 \end{bmatrix}$$

The matrix has a 30-fold eigenvalue 1, whereas one of the standard packages for eigenvalue computation calculates without error message the following approximations for the eigenvalues:

1.7963 - 0.0004i	0.9012 - 0.5693i
1.7578 + 0.2199i	0.7267 + 0.4432i
1.7573 - 0.2204i	0.8037 - 0.5107i
1.6501 + 0.4080i	0.6665 + 0.3719i
1.6495 - 0.4080i	0.6206 + 0.2988i
1.4967 + 0.5409i	0.7261 - 0.4436i
1.4964 - 0.5407i	0.5865 + 0.2251i
1.3269 + 0.6127i	0.6657 - 0.3720i
1.3270 - 0.6124i	0.5626 + 0.1508i
1.1641 + 0.6317i	0.5482 + 0.0758i

1.1643 - 0.6317i	0.6197 - 0.2986i
1.0210 + 0.6125i	0.5433 + 0.0004i
1.0212 - 0.6128i	0.5479 - 0.0750i
0.9013 + 0.5687i	0.5857 - 0.2246i
0.8040 + 0.5102i	0.5620 - 0.1501i

The arithmetic in use is equivalent to 17 decimal digits precision. The examples show that the various sources for errors in floating-point computations might accumulate to significant errors in the final result. The following considerations describe arithmetic requirements to minimize the error of every <u>single</u> floating-point operation. This will not change the results of the examples above but at least will leave the certainty that everything possible has been done on the lowest level.

Let F be a finite subset of the real numbers R considered to be machine numbers. Then a best possible rounding from the real numbers into F would be a mapping ρ satisfying for every real number r

$$| \rho(r) - r | \leq | f - r |$$

for all f in F. This property may be used as a definition for the rounding ρ with some extra rule for the case of two machine numbers f1, f2 having the same distance to the real number r to be rounded.

By definition such a rounding to nearest must satisfy

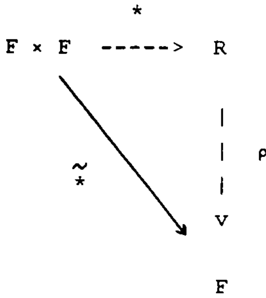
$$a \in F \Rightarrow \rho(a) = a$$

$$a, b \in R : a \leq b \Rightarrow \rho(a) \leq \rho(b) .$$

A floating-point operation $\tilde{*} \in \{+,-,\cdot,/\}$ is an approximation to the exact, real operation $*$. Considering the rounding defined above the best possible approximation to the exact real result, i.e. the result of the floating-point operation to be defined, would be the image of the exact result under the rounding ρ. Therefore we define for floating-point operations $\tilde{*}$ by

(1) $$a, b \in F : \quad a \mathbin{\tilde{*}} b := \rho (a * b) .$$

In other words the following diagram commutes:



Moreover one is interested in the largest floating-point number being less than or equal to the exact result and the smallest floating-point number being greater than or equal to the exact result. By changing the rounding modus, the arithmetical operations with these properties can be defined by (1) as well. All these operations are in fact implementable on computers. The definition agrees with the IEEE 754 floating-point standard; e.g. the Intel 8087 supports all these operations.

When defining the complex operations starting from the real floating-point operations we run into the following difficulty. Consider the complex multiplication

$$(a+bi)\cdot(c+di) = (ac-bd)+(ad+bc)i \quad .$$

Here 4 real multiplications and two real additions/subtractions occur. When replacing every such real operation by its corresponding floating-point operation we can't assume (1) to be satisfied. Consider as an example on our 5-digit decimal computer

$$\begin{aligned}
&(1 + 1.0001i) \,\tilde{\cdot}\, (1 + 0.99999i) && = \\
&(1 \,\tilde{-}\, 1.0001 \,\tilde{\cdot}\, 0.99999) + (0.99999 \,\tilde{+}\, 1.0001)i && = \\
&(1 \,\tilde{-}\, 1.000089999) + 2.00009i && \dashrightarrow \\
&(1 \,\tilde{-}\, 1.0001) + 2.0001i && = \\
&-0.0001 + 2.0001i &&
\end{aligned}$$

instead of the best possible rounded result in 5 decimal digits

$$-0.000089999 + 2.0001i \quad .$$

In order to satisfy (1), intermediate roundings have to be avoided, in fact, we may use (1) again as a definition for the complex floating-

point arithmetic. The same can be done for vector and matrix operations.

The main problem is obviously to calculate scalar products with sufficient accuracy. This is indeed possible, (1) can be used as a definition for a computer arithmetic (cf. [KuMi81] for more details).

From a mathematical point of view the structure of the space of machine numbers with floating-point operations is very poor. An isomorphism from real numbers into the finite set of floating-point numbers is, of course, not possible and it is easy to see that a homomorphism as well is impossible under simplest assumptions. But even associativity is not possible in todays floating-point systems. In fact the following can be proved (cf. [Ru86]): If for some adjacent floating-point numbers a and b the half differences (a-b)/2 and (b-a)/2 are again floating-point numbers, then the law of associativity for the addition is not satisfied. In todays floating-point systems this assumption is true for <u>all</u> floating-point numbers except in the underflow range. The assumption is not valid for fixed-point number systems where in fact addition is associative.

Verified inclusions

Even when using the best possible floating-point arithmetic (more precisely: a floating-point arithmetic performing a minimum error in every operation) the results of combined floating-point operations may still be afflicted with large errors.

If, for instance, the value of

$$z = x^4 - 4 \cdot y^2 - 4 \cdot y^4 \quad \text{for } x = 665857.0 \text{ and } y = 470832.0$$

is calculated on a main frame with equivalent to 17 decimals in the mantissa, the result is

$$z = -469762048.0 \quad .$$

A second computation with interchanged second and third term, i.e. the value of

$$z = x^4 - 4 \bullet y^4 - 4 \bullet y^2$$

with the same values for x and y yields

$$z = -474653696.0$$

indicating the exact value might be around -470 million. In fact the true value is

$$z = +1.0 \quad .$$

A common approach to get more information on the error of a floating-point computation is to evaluate in more than one precision and compare the results. However, this does not imply any guarantee of the correctness of coinciding figures. Consider the following example. Compute

$$f \;=\; 333.75\, b^6 + a^2 \,(\, 11a^2b^2 - b^6 - 121b^4 - 2 \,) + 5.5b^8 + a/(2b)$$

$$\text{for} \quad a = 77617.0 \quad \text{and} \quad b = 33096.0 \quad .$$

To calculate the value of the polynomial a FORTRAN program has been written, the computer in use is a S/370 main frame. All input data is exactly representable, the only errors occurring in the computation are rounding errors and mainly cancellation errors. In order to test the arithmetic rather than standard functions every exponentiation is replaced by successive multiplications. The program calculates the values for f in single, double and extended precision equivalent to approximately 6, 17 and 34 decimal digits precision. The obtained values are the following:

single precision	:	f = + 1.172603 ...
double precision	:	f = + 1.17260394005317847 ...
extended precision	:	f = + 1.17260394005317863185 ...

All three values agree in the first 7 figures, whereas the true value for f is

(2) exact value : $f = -\,0.827396059946821_{3}^{4}$

indicating that the first figures -0.827396... are guaranteed and the sixteenth figure after the decimal point is between 3 and 4. This re-

sult was obtained by an algorithm yielding verified inclusions. It is guaranteed to be correct.

Analyzing the expression above yields immediately the extreme sensitivity with respect to the input data. The 8th power of a 5-digit number yields a 40 digit result and a 1 figure (left of the decimal point) result on a 34-digit computer is by no means of any significance. On the other hand the polynomial need not to occur at once, the input data may be read from a file and the user can't analyze every operation in a million operation program.

It should be mentioned that in algebraic computation it wouldn't be difficult to calculate f. The immediate answer is

$$f = -54767 / 66192 \quad .$$

On the other hand when replacing the first multiplication sign by a division

$$g = 333.75/b^6 + a^2 (11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + a/(2b)$$

$$\text{for} \quad a = 77617.0 \quad \text{and} \quad b = 33096.0 \quad ,$$

it wouldn't be a problem to calculate the value of g even in single precision to full accuracy; the algebraic system would yield

$$g = -7685393445214614367375060703950561700028603401994300944 03$$

$$/ 1752232712494953955278716928 \quad ,$$

a rational number with 85 figures total. The example shows clearly how using numerical or algebraic computations may be of advantage or disadvantage depending on the problem and what the user wants to get.

In the following we will sketch how such verified inclusion can be calculated on a computer, details can be found in the literature. The exact value given in (2) shows a result of an algorithm calculating verified inclusions. The output is an approximation with a guaranteed error bound. All figures of the approximation before the error bound are verified to be correct.

The main principle is to verify that some set contains a solution of the given problem, where the verification has to be performed on the computer. There are well-known theorems stating the existence of a fixed point of a function in a set, namely Brouwer's Fixed Point Theorem: If a continuous function maps a nonempty, closed, bounded and convex subset X of a Banach space into itself, then this function has at least one fixed point within the set X. In order to obtain a zero of a function f (continuously differentiable) Brouwer's Fixed Point Theorem may be applied to the simplified Newton iteration

$$g(x) = x - R \cdot f(x) \quad ,$$

where R is an approximate inverse of the Jacobian of f at some point. A fixed point y of g yields $R \cdot f(y) = 0$ and if R is not singular $f(y) = 0$.

To apply these considerations in a computer program, either some matrix R has to be found which is nonsingular or, the nonsingularity of a given matrix R has to be verified. Furthermore the image g(X) has to be computed. The first problem is solved by the following lemma (here formulated for the real or complex number space).

<u>Lemma.</u> Let $z \in R^n$ (C^n), $B \in R^{n \times n}$ $(C^{n \times n})$ and $\emptyset \neq X \subseteq R^n$ (C^n) being compact. If

$$z + B \cdot X \subseteq \mathrm{int}(X) \quad ,$$

then $\rho(B) < 1$, there is one and only one $x \in R^n$ (C^n) with $z + B \cdot x = x$ and this x satisfies $(I-B)^{-1} \cdot z = x \in \mathrm{int}(X)$.

int(X) denotes the interior of X, I is the identity matrix and all operations in use are the power set operations.

Note that compared to Brouwer's Fixed Point Theorem our assumption is slightly stronger (inclusion in the interior of X is required), whereas, on the other hand, X need not to be convex.

With these considerations the following theorem can be proved for the inclusion of the solution of a system of nonlinear equations.

<u>Theorem.</u> Let $f : R^n \to R^n$ be a C^1 function, $S \in R^{n \times n}$, $x \in R^n$ and $\emptyset \neq X \subseteq R^n$ being compact. If

$$(3) \quad - S \cdot f(x) + \{I - R \cdot f'(x \quad X)\} \cdot X \subseteq \text{int}(X) \quad ,$$

then there is one and only one $y \in x+X$ with $f(y)=0$.

The crucial point is the verification of (3) on a computer. In order to check (3) for a given set X this set has to be represented on the computer, calculations involving X are to be performed and the inclusion in the interior of X has to be checked. For this purpose we use interval arithmetic. For the principles of interval arithmetic the reader is referred to [AlHe83]. One of the basic properties of interval arithmetic is

$$a \in A,\ b \in B \quad \Rightarrow \quad a * b \in A * B \qquad \text{for } * \in \{+,-,\cdot,/\}$$

for intervals A and B.

For the practical application of the theorem above an algorithm has to be designed with several further improvements compared to (3). For more details see [Ru83] and [Ru84]. These algorithms operate in single or double precision arithmetic with a speed comparable to standard numerical floating-point algorithms but delivering verified, guaranteed bounds for the result. Standard problems like linear equations (also over- and underdetermined), general nonlinear equations, polynomial zeros, optimization, eigenproblems and others are covered. These algorithms are available in the IBM program product ACRITH, part of them in the SIEMENS program product ACRITHMOS (cf. [IBM86] and [SIE86]).

An algorithm for calculating guaranteed inclusions for systems of nonlinear equations f(x)=0 works, in principle, as follows (an initial approximation has to be given):

1) Use a traditional floating-point algorithm to improve the given approximation yielding an approximate solution x

2) Define a small (relative diameter in the order of the relative rounding error) interval X with floating-point bounds containing x

3) Try to verify formula (3)

4) If (3) is not satisfied apply an iteration scheme to improve X

The goal of the algorithm is to proof the conjecture that the set X has the property to contain exactly one zero of the nonlinear system. We try to verify this conjecture by the sufficient criterion (3). If (3) is true, we have the verification, if not, an iteration scheme is applied (cf. [Ru83]). Conditions depending on X can be given, when (3) is satisfied and when the iteration finishes (cf. [Ru84]). They show that even extremely ill-conditioned problems can be handled by inclusion algorithms.

Another application of inclusion algorithms are problems with data afflicted with tolerances. All algorithms can be used almost unchanged to work with uncertain data. In this case the guaranteed results have the following property. Take any combination of real (complex) numbers out of the input tolerances and solve the problem with this data. Then it is guaranteed that all problems generated in this way do have a solution and that a solution to every single of these problems lies within the computed guaranteed bounds. This is a worst case analysis.

Compared to a numerical Monte Carlo like approach this method is even faster because the inclusion algorithms for data afflicted with tolerances need about the same computing time as for data without tolerances, whereas the computing time for the Monte Carlo approach is the time for one application of a standard numerical algorithm multiplied by the number of random sample problems generated.

It is guaranteed that a computed inclusion covers in fact all possible solutions to all problems with data within the input tolerances. Moreover bounds can be computed how sharp the computed inclusion is, i.e. a percentage can be computed with the property that the inclusion interval becomes wrong when narrowing the diameter by this percentage. All guarantees are correct from a mathematical point of view; the computed bounds are correct as long the machine performs correctly.

It is inadmissible to compare the results of the inclusion methods with the numerical Monte Carlo approach: The first method yields guaranteed error bounds with guaranteed estimations on the sensitivity of every parameter [Neu87], the other method gives numerical estimations of the

error bounds without guarantee. In fact it may happen that bounds produced by the Monte Carlo ansatz are orders of magnitude to small [KuRu87].

It should be mentioned that a Jacobian need neither to be calculated symbolically nor to be approximated numerically. The inclusion algorithm for systems of nonlinear equations requires an inclusion of the value of the Jacobian at a certain point. There is a very interesting algorithm for this task which has been found and forgotten several times. It calculates the value of the derivative (or simultaneously all partial derivatives) at a certain point together with the computation of the value of the function. It is very fast, accurate and allows the easy calculation of an inclusion of all derivatives and it can be programmed very easily. It may also be applied to compute coefficients of Taylor series. For details see [Ra81]. Following we give a simple example without formalizing the method.

Consider $f(x) = \{ x \cdot \sin(x) + \exp(1/x) \}^2$ at x=2. The following diagram shows how to calculate the value of f and f' simultaneously. In the leftmost column the commands are displayed

	value	derivative	
x	2	1	$x' = 1$
enter			
sin	0.909	-0.416	$(\sin u)' = u' \cos u$
.	1.819	0.077	$(uv)' = u'v + uv'$
x	2	1	$x' = 1$
1/	0.5	-0.25	$(1/u)' = -u'/u^2$
exp	1.649	-0.412	$(\exp u)' = u' \exp u$
+	3.467	-0.335	$(u+v)' = u' + v'$
$(\ )^2$	12.022	-2.324	$(u^2)' = 2uu'$

as they would have to be entered in an HP-Calculator for evaluating f at x=2. The next two columns show the intermediate values in the calculation of the function values and the derivative, the rightmost column the formulas used to calculate the values in the derivative column. Programming the algorithm using two stacks, one for the function values and one for the derivative, allows a very fast and easy implementation. The well-known laws for calculating the derivative of a sum, product, inverse, sin, exp etc. are used and applied to the values on the

stacks. Because all intermediate function values and derivative values are known (they are on the stack) programming the method is very easy.

The approach sketched above to calculate the inclusion of the zero of a system of nonlinear equations should be sharply distinguished from applying interval arithmetic in a naive manner. It is true that when replacing every operation in a numerical algorithm by its corresponding interval operation the final result contains the true solution of the given problem. Unfortunately the diameter of the intervals tend to grow very rapidly when trying this approach, finally yielding results of little significance.

Using formulas like (3) diminishes the overestimation by interval arithmetic and, most important, uses the original data (the function f) in the calculation. In the naive approach, every step depends only on its immediate predecessor resulting in the well-known effect of rapidly increasing diameters.

Some words should be added on the difference between operations in the power set over real or complex numbers, operations in the set of intervals over real or complex numbers and operations in the set of floating-point intervals over real or complex numbers (intervals with floating-point bounds). Let us consider power sets and intervals over complex numbers. We use rectangles parallel to the axis as intervals. With the induced order relation we write complex interval as

$$[e,f] = \{ z \in C \mid e \leq z \leq f \}$$

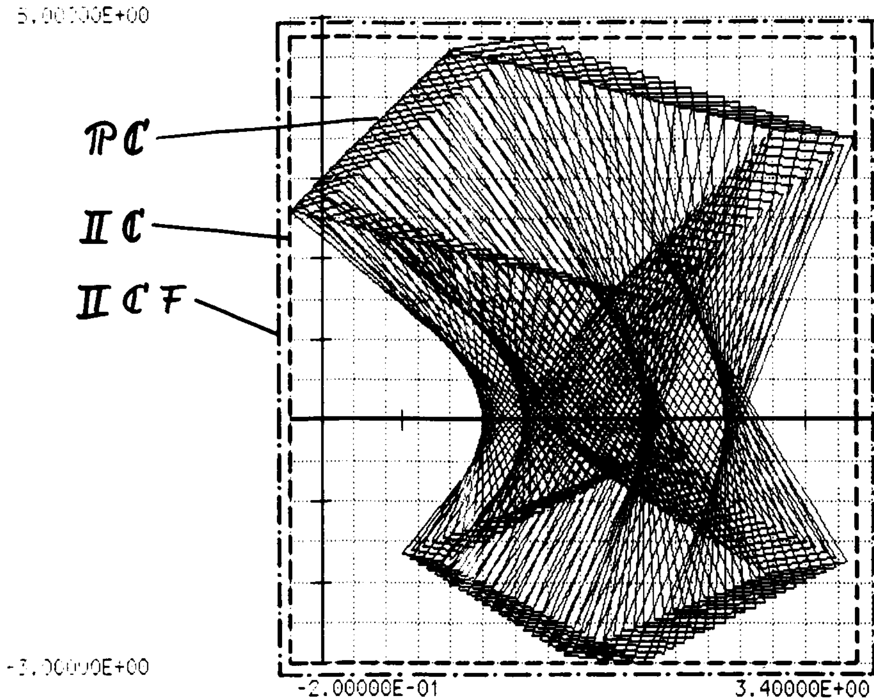
for complex numbers $e \leq f$. An image of the power set multiplication of two such intervals A and B may be obtained in the following way. Take any point on the boundary of A and multiply it by the boundary of B, yielding a rectangle. The convex hull of all these rectangles is then the result of the power set multiplication of A and B.

The interval multiplication of A and B yields the smallest interval (rectangle parallel to the axis) containing every product $a \cdot b$ for a in A, b in B. The floating-point interval multiplication of A and B is the smallest interval with floating-point bounds containing this set.

Consider an example. Let

$$A = [-0.2 - i , 0.3 - 0.5i] \quad , \quad B = [-4 + 2i , 2.5 + 2.5i] \quad .$$

Then the three products of A and B (power set product, interval product and floating-point interval product) look like the following.



The actual calculation of the power set product is performed by taking 20 sample points on each edge of the boundary of A and multiplying it by the boundary of B. The origin is contained in the interval products but not in the power set product.

It seems difficult to calculate the smallest rectangle parallel to the axes enclosing the power set product of A and B. In higher dimensions it seems to be impossible to perform this calculation on the computer. However, regarding the fact that all terms in the real and imaginary part in every component of a matrix can be treated separately leads to the solution of this problem. For details cf. [AlHe83] or [Mo79].

Combining the methods

The first and last mentioned method, algebraic computation and computing guaranteed bounds, have most in common of the three, where in fact even this is little. Both methods are aiming on true results, where the first delivers the exact answer, the latter delivers error bounds for the solution. At least if the problem is to find an answer to a question such as "is there a zero of the polynomial p between a and b" either of the two methods may be applied. Numerical computation hardly fits into this context. However, even numerical results (values without error bounds from a mathematical point of view) may help the other two methods to perform faster.

Let us start by listing some problems which may be solved by one method but hardly by any of the two others.

Any calculation of exact values such as arithmetic in an algebraic number field $Q(\alpha)$ could hardly be performed in numerical computation or using inclusion algorithms. The same holds true for e.g. calculating the Galoisgroup of a polynomial. These problems arising in mathematical spaces can hardly be transformed to fit in a finite set of floating-point numbers. But there are even numerical problems, where algebraic computation is superior to the other two. Consider the inversion of a large Hilbert matrix, say 50×50. Using a general numerical or inclusion algorithm for matrix inversion couldn't solve this problem in standard floating-point formats of todays computers. Using rational exact arithmetic eliminates rounding errors and cancellation errors and therefore the numerical difficulties with this particular problem.

Todays very large (sparse) linear systems arising in technical applications can't be treated with algebraic computation and (up to now) hardly with inclusion algorithms. The same is true for differential equations where just the definition fills dozens of pages. The reason is the tremendous speed of floating-point operations (compared to a software simulated exact arithmetic or interval operations), especially on todays vector and/or parallel computers. As soon as the numerical solution is at the time boundary of what can be performed any degradation in the computing time of only a small percentage is inacceptable. This situation may change for inclusion methods when more computers and programming languages do support directed roundings by hardware and appropriate operators.

On the other hand approximation delivered by numerical algorithms might be inaccurate. Here is the advantage of inclusion algorithms. Even for extremely ill-conditioned problems inclusions will be computed and, if the precision in use does not suffice, an appropriate message will be given rather than an inaccurate result. Another set of problems which can hardly be treated by numerical algorithms or algebraic computation as efficient as by inclusion algorithms are problems with data afflicted with tolerances. Here inclusion algorithms still yield guaranteed bounds for all possible solutions. With the estimation of the quality of the inclusion (how much can the diameter be narrowed without loosing the inclusion property) this yields an immediate sensitivity analysis.

There are few problems where the three methods, algebraic computation, numerical computation and inclusion methods, can compete. But there are areas where the methods may benefit from their respective specific advantages.

Consider the problem of calculating the sign of an algebraic number. Let Ψ $Q[x]$ be the defining polynomial for the algebraic number α, $[a,b]$ be an interval with rational endpoints a and b such that α is the only zero of Ψ in $[a,b]$ and let $\beta = P(\alpha)$ for some polynomial P with rational coefficients. The sign of β is the sign of the polynomial P at α, where α is usually not given numerically but as described above.

This is a typical problem where no exact value is asked for but simply the sign +, 0 or -. We may exclude 0 by calculating $\gcd(P,\Psi)$ or by assuming Ψ to be irreducible. An approach combining the advantages of either method could be the following:

1) Calculate α to floating-point precision with Newton iteration

2) Replace α by the interval I with floating-point bounds of smallest diameter

3) Calculate P(I) using inclusion methods. If $0 \notin P(I)$ then stop.

4) Eventually repeat steps 1 to 3 in higher precision; if up to a limit precision always $0 \in P(I)$ then apply algebraic methods (e.g. root isolation).

In the approach it is tried to map as much as possible into floating-point computations. The result is guaranteed to be correct because if $0 \notin P(I)$ then for all $x \in I$ holds $P(x)<0$ or $P(x)>0$, especially for $\alpha \in I$ and the sign is determined. There is a paper following a similar approach but using naive interval arithmetic instead of an inclusion algorithm. Even then the computing times drop drastically (cf. [Pi76]).

Whether the approach works or not depends on the sensitivity of evaluating P near α. However experience shows that in algebraic computations this problem is often not ill-conditioned; the sometimes high computing times of pure algebraic algorithms originate in the exact computation within the algebraic number field (cf. [Ru76]).

Another area where inclusion algorithms may help speeding up algebraic algorithms is real or complex root isolation for a polynomial P. As in the previous example it is not asked for a precise value but for real or complex intervals I_i each containing precisely one zero of P. A stepwise approach could be the following:

1) Replace the coefficients of P by the floating-point number nearest to each of them yielding a polynomial S

2) Use a traditional floating-point algorithm to compute floating-point approximations to the zeros of S

3) Replace the coefficients of P by the intervals with floating-point bounds of smallest diameter containing the original coefficients of P yielding a polynomial T

4) Use inclusion methods to calculate inclusions of the zeros of T based on the floating-point approximations computed in step 2

5) Eventually repeat steps 1 to 4 with higher precision or use purely algebraic methods.

Corresponding algorithms are on the way to be implemented.

There are also examples where algebraic computations may help in pure numerical computations. One of the problems in solving a system of nonlinear equations is the computation of an initial approximation to a solution. Here Groebner bases may help to find such approximations.

Conclusion

Either of the methods - algebraic computation, numerical computation and computing verified inclusions - has its specific advantages. There are many problems where the methods may be complementary to one another, some examples have been shown above.

In the future algorithms should be designed taking advantage of the strength of the specific approaches and combining them. For the implementation of such algorithms software shells have to be built allowing to work numerically without and with bounds and to work algebraically and symbolically in one programming environment. There is a big chance to start a new era of algorithms and many people are waiting for it.

Literature

[AlHe83] Alefeld, G. and Herzberger, J.: Introduction to Interval Computations, Academic Press, 1983. ACADEMIC PRESS, New York (1981).

[IBM86] ACRITH, High-Accuracy Arithmetic Subroutine Library, Program Description and User's Guide, IBM Publications, Document Number SC33-6164-3 (1986).

[KuMi81] Kulisch, U. and Miranker, W.L.: Computer Arithmetic in Theory and Practice, ACADEMIC PRESS, New York (1981).

[KuRu87] Kulisch, U. and Rump, S.M.: Rechnerarithmetik und die Behandlung algebraischer Probleme, in Buchberger/Feilmeier/ Kratz/Kulisch/Rump: Rechnerorientierte Verfahren, B.G. Teubner (1986).

[Mo79] Moore, R.E.: Methods and Applications of Interval Analysis, SIAM Studies in Applied Mathematics, (1979).

[Ra81] Rall, L.B.: Automatic Differentiation: Techniques and Applications, Springer Lecture Notes in Computer Science, 120 (1981).

[Neu87] Neumaier, A.: Private communication.

[Pi76] Pinkert, J.R.: Interval Arithmetic Applied to Polynomial Remainder Sequences, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, New York, 214-218, (1976).

[Ru76] Rump, S.M.: On the Sign of a Real Algebraic Number, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, New York, 238-241, (1976).

[Ru83] Rump, S.M.: Solving Algebraic Problems with High Accuracy, in "A New Approach to Scientific Computation", Edts. U.W. Kulisch and W.L. Miranker, ACADEMIC PRESS, p. 51-120 (1983).

[Ru84] Rump, S.M.: Solution of linear and nonlinear algebraic problems with sharp, guaranteed bounds, COMPUTING Supplementum 5, 23 Seiten, (1984).

[Ru85] Rump, S.M.: Properties of a Higher Order Computer Arithmetic, Proceedings of the 11th IMACS World Congress, Oslo, (1985); also in modified form in IMACS Transactions on Scientific Computation - 85, North Holland, (1986).

[SIE86] ARITHMOS, Benutzerhandbuch, Siemens AG, Bestellnummer U 2900-J-Z87-1, (1986).