Chapter 10
Computer-assisted Proofs and Self-validating Methods

Siegfried M. Rump

10.1 Introduction

In this chapter[41] we discuss the possibility of computing validated answers to mathematical problems. Among such approaches are so-called computer-assisted proofs, exact computations, methods from computer algebra, and self-validating methods. We will discuss common aspects and differences of these methods, as well as their potential reliability. We present in detail some concepts for self-validating methods, and their mathematical background, as well as implementation details.

Self-validating methods will be introduced using INTLAB, a MATLAB toolbox entirely written in MATLAB. Due to the operator concept it is a convenient way to get acquainted with self-validating methods. Many examples in this chapter are given in executable INTLAB code. This toolbox is freely available from our homepage for noncommercial use. To date we have an estimated number of 3500 users in more than 40 countries.

We stress that this exposition will cover only a small part of the subject, some basic concepts of self-validating methods. Nevertheless, we hope to give some impression of this interesting and promising area.

10.2 Proofs and Computers

Since the invention of digital computers the question has arisen (again) whether computers can perform mathematical proofs. Put into that form the answer, in my opinion, is a simple "No!" since computers don't do anything without some person ordering them to do so. So a more appropriate formulation may be whether proofs may be performed with the aid of digital computers.

[41] Extended workout of the special lecture presented at the Annual Conference of Japan SIAM at Kyushu University, Fukuoka, October 7–9, 2001; an excerpt has been published in Bull. JSIAM, 2004.

Figure 10.1. *IBM prime letterhead.*

There are a number of famous examples where people claimed to have proven a theorem with the aid of digital computers. And people became proud of this so that it even found its way into letterheads and postmarks. For example, when you received a letter from IBM Thomas J. Watson Research Center in Yorktown Heights in the 1970's, the letterhead appeared as pictured in Figure 10.1. At that time this was the largest known Mersenne [312] prime, with some 600 decimals. Primality was proven in 1971 [456] by the Lucas–Lehmer test, and this fact even found its way into the Guiness book of records. A new era in the run for the largest explicitly known prime began with the Great Internet Mersenne Prime Search (GIMPS). In this mutual international effort $2^{6972593} - 1$, a number with more than two million decimal digits, was found and earned the discoverer Nayn Hajratwala some 50,000 U.S. dollars. The latest finding in 2004 is $2^{24036583} - 1$ with more than seven million digits.

Figure 10.2. *University of Illinois postmark.*

When receiving a letter from the University of Illinois at Urbana-Champaign in the late 1970's or early 1980's, next to the postage stamp you would see what is pictured in Figure 10.2 (thanks to Bhama Srinivasan for providing the original picture), crediting the celebrated effort of Kenneth Appel and Wolfgang Haken. They checked thousands of sample graphs using some two thousand hours of computing time to attack the Four Color Theorem. At the time there was (and still is) quite some discussion about whether their efforts really "proved" the theorem. There are significant differences between the first and the second example.

The first example, proving primality of a Mersenne number, uses solely long integer arithmetic. There are numerous implementations of such packages for algebraic computations. And, the Lucas–Lehmer test for $n = 19937$ takes less than a minute on today's laptops. Moreover, the implementation is very clear and straightforward, and the arithmetic programs have been tested innumerable times. So it seems not "likely" that there are errors in such programs.

In contrast, the attack of the Four Color Theorem consists of very many individual programs, seeking coloring patterns for individual graphs. The programming is by no means straightforward, and comparatively few people have looked at the programs. Once I asked Wolfgang Haken what would be his answer to the obvious question of whether the proof is valid. He explained, "Well, put it this way. Every error in our programming found so far has been repaired within 24 hours." We tend to agree with Paul Erdös, the late ever-traveling mathematician, who said [208], "I'm not an expert on the four color problem, but I assume the proof is true. However, it's not beautiful. I'd prefer to see a proof that gives insight into why four colors are sufficient." In fact, the proof by Appel and Haken seems ill suited for *Proofs from THE BOOK* [9]. There are newer approaches to proving the Four Color Theorem; see, for example, [384].

Before going on we collect some more examples of possibilities of proofs by computers. Tarski [437] proved his famous theorem that the elementary theory of real closed fields is decidable. Later Collins [99] developed his Quantifier elimination method, which drastically reduces the necessary computational time to "prove" a corresponding theorem. With this quantifier elimination method, questions from geometry can be reduced to equivalent, quantifier-free formulae. The input formula is a theorem if and only if it reduces to "1" or "always true."

Another famous example is Risch's algorithm [379, 173] for integration in finite terms. This algorithm "decides" whether some function consisting of basic arithmetic operations, roots, powers, and elementary standard functions can be integrated in finite terms, and if so, it calculates a closed formula for the integral. It is a decision algorithm, and the maximum computing time can be estimated in terms of the length of the input. The algorithm is available, for example, in the computer algebra system Maple. For the input

$$\int x e^{x^2} dx = \frac{1}{2} e^{x^2} + C,$$

the integral can also be easily calculated by standard methods. But that

$$\int e^{x^2} dx \quad \text{is not solvable}$$

is a nontrivial result: It *proves* that no finite combination of arithmetic operations, powers, or elementary standard functions can be found such that the derivative is equal to e^{x^2}. In fact, the algorithm does more by calculating that

$$\int e^{x^2} dx = -\frac{1}{2}\sqrt{-\pi} \cdot \operatorname{erf}(\sqrt{-1} \cdot x) + C$$

involving the error function, which cannot be expressed in finite terms.

Another recent example is the Kepler conjecture that the face-centered cubic packing is the densest packing of equally sized balls. This almost 400-year-old conjecture was solved by Hales in 1998 [188]. More examples can be found in [155].

So what is common and what are differences in the above examples? In other words,

What is a proof?

We do not want (and cannot) give a final answer to that question, but we want to offer some relevant thoughts. The traditional mathematical proof can be followed line by line with

pencil and paper. And this is still the classical way mathematicians work. As mathematicians are human beings, they might draw false conclusions, and there are famous examples of that. Often such "false" approaches, originally intended to prove some specific problem, lead to other interesting and new insights. A typical and well-known example is Kummer's approach to prove Fermat's Last Theorem. Although it did not prove the theorem, it led to a deep and different understanding of the matter.

There are also examples of "theorems" that were accepted for years but proved later to be false. A nice example is the following.

NOT A THEOREM. *Let a C^1 function $f : \mathbb{R}^2 \to \mathbb{R}$ be given. For some given point x_0, assume that for every $y \in \mathbb{R}^2$ the projection $f_y : \mathbb{R} \to \mathbb{R}$ with $f_y(\lambda) := f(x_0 + \lambda y)$ has a minimum at $\lambda = 0$. Then f has a minimum at x_0.*

This was presented in beginners' courses in calculus some 100 years ago by saying the "proof" is obvious. With today's education we have seen too many counterexamples of similar "theorems" and would ask for a solid proof. In this case, Peano (cf. [430]) gave the nice counterexample $f(x_1, x_2) = (x_2^2 - 2px_1)(x_2^2 - 2qx_1)$ at the origin, where $p > q > 0$.

Moreover, a proof becomes a proof by human beings reading, understanding, and accepting the proof. If a proof becomes lengthy and difficult, naturally the number of mathematicians who really follow the ideas decreases. A famous example is the celebrated result by Feit and Thompson, who proved in 1963 that all nonabelian finite simple groups are of even order. They published this result in "Solvability of Groups of Odd Order," a 250-page paper that appeared in the *Pacific Journal of Mathematics*, 13 (1963), pp. 775–1029. Despite the importance of the paper several journals declined to publish it because of its length.

Of course, we do not question the correctness of the result. The point is that acceptance of a proof is also based on mutual trust: If trustworthy colleagues accept a proof, we also tend to accept it—a very reasonable point of view.

So what does it mean to "follow and to accept" a proof? Consider the following story. At the meeting of the American Mathematical Society in October 1903 in New York City, Frank N. Cole gave a quite unusual "talk." The story goes that when his talk was announced, Cole went to the blackboard and wrote the following two integers

$$761838257287 * 193707721$$

and multiplied them out manually. He then wrote

$$2^{67} - 1$$

on the board and worked it out manually. The results of the two computations were equal. Thunderous applause broke out. Cole sat down. He never spoke a word, and there were no questions. The account of this event appeared in a letter in the February 1983 issue of Scientific American. When asked later how long it took to find this factorization of the Mersenne number, Cole explained, "Three years, every Sunday."

Imagine if this happened today: how many of us would sit down and work out the multiplications by hand? I guess many would at least use a pocket calculator or use some computer algebra system—which can also be found on advanced pocket calculators today

(such as mupad and derive). I also guess that many of us would have no difficulty accepting the use of such electronic equipment as part of the proof.

However, there are different levels of trust. I think it is fair to say that for the purpose of supporting a mathematical proof,

- the use of pocket calculators is widely accepted,
- the use of computer algebra systems is more or less accepted,
- but the use of floating-point arithmetic seems questionable to many mathematicians.

But is this fair? Common sense is sometimes biased by what we want to believe. For example, statistics tells us that the chance is higher one would be killed by a bee than by a shark. Similarly, pocket calculators look simple enough to be reliable because—which is of course true in a certain sense—the probability of an error may be considered to be proportional to the complexity of the system. But does this mean that pocket calculators, on the market for decades, are error-free?

We know that this is not true. Severe errors due to the limited length of the internal accumulator occur in many pocket calculators (see Figure 10.3): the standard 8-digit plus-minus-times-divide-square root pocket calculator with decimal point but *without* exponent, the one you may have found as an ad in your mail for some years, are still widely in use today.

Figure 10.3. *Pocket calculators with 8 decimal digits, no exponent.*

Calculate with any calculator of this kind

$$1\,000\,000 - 999\,999.99$$

Both numbers have 8 significant digits, so both can be stored in the calculator without any error. If the units are Dollars, the result is obviously 1 cent, whereas the calculator will give the result as 10 cents—a typical effect of the too short internal 8-digit accumulator: First the numbers are adjusted according to their decimal point such that digits of equal value can be added.

$$\begin{array}{r|l} 1\,000\,000 & \\ 999\,999.9 & 9 \\ \hline 0.1 & \end{array}$$

This implies that the subtrahend 999 999.99 has to be shifted one to the right. However, the internal accumulator is also only 8 decimal digits long, so that the final figure 9 vanishes. The result is a kind of catastrophic cancellation, but due to an inappropriate implementation of the arithmetic.

So if such simple machines can produce results with a relative error of 900 %, we might ask whether personal computers or mainframes can produce any reliable result at all. Many would answer this question with a simple "no" meaning that "results obtained by floating-point operations are per se not reliable." Not long ago numerical mathematics was considered "dirty." And many people believe(d) that floating-point computations, where almost every operation is afflicted with some error, may produce some useful approximations, but they are not suited for any kind of verification or guarantee.

The above example occurs not only in pocket calculators. For example, mainframes such as the Univac 1108 in use until the 1980's calculated

$$16777216 - 16777215 = 2.$$

The reason is again the short accumulator of length 24 bits, the same as the working precision, and $16777216 = 2^{24}$. A similar error occurred in the early 1960's in IBM S/360 mainframes, and we can thank Vel Kahan that the company changed the architecture to cure this. Even until very recently similarly poor implementations could be found in Cray supercomputers. This is the reason why sometimes the statement

$$a = 2 * a - a$$

can be found in programs. It "simulates" one bit less precision so that an accumulator of working precision has one additional bit compared to this reduced precision.

10.3 Arithmetical Issues

To discuss whether it is at all possible to produce correct or validated results on a digital computer, consider the following example. Let a real matrix A be given, and for simplicity assume that all entries A_{ij} are floating-point numbers. In other words, the matrix is exactly representable in the computer. Can we "prove" that the matrix A is nonsingular?

A typical aspect of self-validating methods is the mutual benefit of theoretical analysis and practical implementation. A possible theoretical approach goes as follows. Let R be an

arbitrary real matrix, a preconditioner. If the spectral radius $\varrho(I - RA)$ is less than 1, then R and A are nonsingular because otherwise $C := I - RA$ would have an eigenvalue 1. By Perron–Frobenius theory we know that $\rho(C) \leq \rho(|C|)$, where $|C|$ denotes the nonnegative matrix consisting of entrywise absolute values $|C_{ij}|$. Moreover, a well-known theorem by Collatz [98] says that

$$|I - RA|x < x \tag{10.1}$$

for an arbitrary entrywise positive vector x implies $\rho(|I - RA|) < 1$, and therefore $\rho(I - RA) < 1$ and R and A are nonsingular. In other words, if we can prove (10.1) for any positive vector x, for example, the vector consisting of all 1's, then we have proved A to be nonsingular. The result remains true without any further requirements on the matrix R. A good choice is an approximate inverse of A.

Now we aim to verify (10.1) in floating-point arithmetic. To do that we need further information about properties of floating-point arithmetic or, as trivial as it sounds, we need to know

How is the floating-point arithmetic in use defined?

Until the 1980's not much information about this was available from computer manufacturers. This was one reason for a mutual initiative to define the first standardization of floating-point arithmetic, the IEEE 754 binary standard in [215]. Among other things, this standard defines the maximum relative error of all floating-point operations to be less than or equal to the unit roundoff $\mathbf{u}$. Moreover, the floating-point standard defines directed roundings, a rounding downward and a rounding upward.

Today the IEEE 754 standard of floating-point arithmetic is implemented in the large majority of all computers in use in scientific computation, from PCs to workstations to mainframes. Switching the rounding mode is frequently done by setting the processor into a specific rounding mode. For example, if the processor is switched to rounding downward, then *every* subsequent floating-point operation yields as the result the unique maximal floating-point number less than or equal to the exact result. This is quite a remarkable property, and it is a *mathematical* property we can build upon.

Denote by fl(*expression*) the value of an expression with all operations executed as floating-point operations. Moreover, let fl_∇(*expression*) and fl_Δ(*expression*) denote that the rounding mode has been switched to downward or upward, respectively (if the rounding symbol is omitted, we assume rounding to nearest). Then, we already obtain the mathematical statement

$$\text{for all } x, y \in \mathbb{F}: \quad \mathrm{fl}_\nabla(x + y) = \mathrm{fl}_\Delta(x + y) \quad \Leftrightarrow \quad x + y \in \mathbb{F}.$$

The (true, real) result of a floating-point operation is a (representable) floating-point number if and only if the floating-point results obtained by rounding downward and rounding upward are equal. This is also true for subtraction, multiplication, and division. It is also true for the square root:

$$\text{for all } x \in \mathbb{F}: \quad \mathrm{fl}_\nabla(\sqrt{x}) = \mathrm{fl}_\Delta(\sqrt{x}) \quad \Leftrightarrow \quad \sqrt{x} \in \mathbb{F}.$$

These results are also true in the presence of underflow; it is a *mathematically reliable statement*. At this point one may argue that this, as a mathematical statement, is true only as long as

- the implementation of floating-point arithmetic actually follows the definition by the IEEE 754 standard,
- the compiler, the operating system, and in fact all of the hardware and software involved are working correctly,
- no external distortions, such as very short power failures, radiation, and other nasty things occur.

This is absolutely true. In this sense we can never trust the result of any machine, and mathematical proofs with the aid of computers are not possible at all (however, human brains may also fail). On the other hand, the arithmetic of digital computers is tested very thoroughly every day, and a failure, at least due to one of the first two reasons, seems unlikely.

But there is more than a "likeliness argument," and we want to further explore this. Years ago I was working on a rigorous implementation of elementary standard functions, and I had an intensive discussion about the accuracy and reliability of the standard functions provided by some library. In a nutshell, I was asked why it is necessary at all to work on "reliable" standard functions when no example is known where the existing library produces an error greater than one unit in the last place.

But there is a fundamental difference between the accuracy of the floating-point arithmetic and the accuracy of standard functions in a library. The latter are implemented by very clever techniques such as continued fractions, table-based approaches, or Cordic algorithms. Everything is done to provide an accurate approximation of the result. But for most libraries there is no *rigorous* error estimate valid for *all* possible input arguments. And millions of nonfailing tests cannot *prove* that an algorithm never fails. In contrast, the implementation of floating-point algorithms according to the IEEE 754 standard is well understood, and the analysis shows that conclusions like (10.1) are valid for all possible input arguments.

So the mathematical specification of the algorithms for floating-point arithmetic can be correct and never-failing, but the implementation might not follow the specification. We assume in the following that the implementation of floating-point arithmetic follows its specification and therefore the IEEE 754 standard, and that the software and hardware in use are operating correctly. Then, we may ask again whether it is possible to validate results with the aid of digital computers.

For the above example, to validate (10.1) and therefore the nonsingularity of the matrix A this is simple. The only additional knowledge we need is that when x is a floating-point number, $|x|$ is a floating-point number as well. Then, for given R, A, and x, we calculate

$$\begin{aligned} C_2 &:= \mathrm{fl}_\nabla(R * A - I), \\ C_1 &:= \mathrm{fl}_\Delta(R * A - I), \\ C &:= \max(\mathrm{abs}(C_1), \mathrm{abs}(C_2)), \\ y &:= \mathrm{fl}_\Delta(C * x). \end{aligned}$$

By definition,

$$\mathrm{fl}_\nabla(r_{ik} \cdot a_{kj}) \le r_{ik} \cdot a_{kj} \le \mathrm{fl}_\Delta(r_{ik} \cdot a_{kj}) \quad \text{for all } i, j, k,$$

and therefore

$$C_1 \le RA - I \le C_2,$$

where the comparison is to be understood entrywise. This implies

$$|I - RA| \leq \max(|C_1|, |C_2|) = C,$$

where maximum, absolute value, and comparison are again to be understood componentwise. Combining this with (10.1) it follows that

$$y < x \quad \text{implies } A \text{ (and } R\text{) are nonsingular.}$$

Note that the inequalities need not to be true when replacing $RA-I$ by $I-RA$. Also note that this approach needs two switches of the rounding mode and otherwise only floating-point operations; in particular, no branches are required. This means that matrix multiplications can be performed by BLAS routines [285, 124, 120, 122] and thus are very fast. So in a certain way the rounded operations create speed with rigor. In the following we want to discuss this in more detail, particularly the domain of applicability and the distinction from computer algebra methods.

10.4 Computer Algebra Versus Self-validating Methods

A major objective of computer algebra methods is to deliver not only correct answers but also exact answers. For example, it is standard to calculate in algebraic number fields [173]. Then, for instance, $\sqrt[3]{17} - \sqrt[5]{5}$ is not calculated by some numerical approximation within hundreds of digits of accuracy but as an element of a suitable algebraic extension of the field of rational numbers. Thus the operation is *exact*. This allows for the development of so-called decision algorithms: Given an inquiry as a yes/no problem, a decision algorithm will definitely answer this question in a finite computing time, where the latter can be estimated in terms of the length of the input data.

We already mentioned famous examples of decision algorithms, such as quantifier elimination and Risch's integration in finite terms. Another famous and well-known example for the power of computer algebra algorithms is Gröbner bases. For a given system of multivariate polynomial equations it can be decided whether these are solvable, and the number of solutions can be calculated as well as inclusion intervals for the roots. A major breakthrough was the development of constructive and fast algorithms [58, 59].

Computer algebra methods generally always compute exactly. There are exceptions in the sense that error bounds may be calculated, if this is sufficient. But in that case there is usually a fall-back algorithm if error bounds are not of sufficient quality.

Exact computations may take quite some computing time. In contrast, self-validating methods calculate in floating-point arithmetic. This means they are fast, with the possibility of failure in the sense that no answer can be given. As we will see, they can *never fail* by giving a false answer.

This is the reason why self-validating methods are applicable only to well-posed problems. For example, we can verify that a matrix is nonsingular, but in general it is not possible to verify that it is singular. If the answer to a problem can change for arbitrarily small changes of the input data, then the problem is ill-posed and the slightest rounding error may change the answer. But self-validating methods *intentionally use floating-point arithmetic*, so such problems are beyond their scope.

Similarly, it is possible to calculate an inclusion of a multiple eigenvalue [390] or of a multiple root of a polynomial [393]. However, it is not possible to verify that the eigenvalue or root is indeed multiple, that is, that the multiplicity is greater than one. The computation of error bounds for a multiple root is well-posed, whereas the fact that the multiplicity is greater than one is ill-posed: The fact may change for arbitrarily small perturbations of the input data. So we may say that one reason for the speed of self-validating methods is the inaccuracy (the rounding) of floating-point operations.

One example of self-validating methods is that equation (10.1) implies nonsingularity of the matrices A and R. This is in fact a mainstream of self-validating methods, that is,

to verify the assumptions of mathematical theorems with the aid of digital computers.

This implies validity of the assertions, as in the simple example the nonsingularity of matrices. Obviously, the only requirement for the arithmetic in use to verify (10.1) is the availability of true error bounds. Using directed roundings is one possibility, using exact arithmetic would be another, and there are more possibilities. Another possibility that is widely used is interval arithmetic. Interval operations have the advantage of easy formulation and implementation. However, interval arithmetic does not have the best reputation. We will come to that in a moment.

10.5 Interval Arithmetic

Closed intervals are one possibility for representing sets of numbers. They were already described in a little recognized but nevertheless very comprehensive paper by Sunaga [432]; see also [318]. For the moment we ignore rounding errors and representation on the computer but define the set $\mathbb{IR}$ of real intervals as usual by

$$A \in \mathbb{IR} \ :\Leftrightarrow \ A \neq \emptyset \quad \text{and} \quad A = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} : \underline{a} \leq x \leq \overline{a}\}.$$

In the degenerate case $\underline{a} = \overline{a}$, the interval $[\underline{a}, \overline{a}]$ represents one real number $a = \underline{a} = \overline{a}$ and can be identified with it. This defines the natural embedding of $\mathbb{R}$ into $\mathbb{IR}$ using so-called *point intervals* $[a, a]$. The interval operations between real intervals are just the power set operations. One easily verifies for all $A = [\underline{a}, \overline{a}]$, $B = [\underline{b}, \overline{b}] \in \mathbb{IR}$ and $\circ \in \{+, -, \cdot, /\}$

$$\begin{aligned}\{a \circ b : \ a \in A, b \in B\} &= \bigcap \{C \in \mathbb{IR} : \ a \circ b \in C \text{ for all } a \in A, b \in B\} \\ &= [\,\min(a \circ b : \ a \in A, b \in B), \max(a \circ b : \ a \in A, b \in B)\,] \\ &= [\,\min(\underline{a} \circ \underline{b}, \underline{a} \circ \overline{b}, \overline{a} \circ \underline{b}, \overline{a} \circ \overline{b}), \max(\underline{a} \circ \underline{b}, \underline{a} \circ \overline{b}, \overline{a} \circ \underline{b}, \overline{a} \circ \overline{b})\,] \\ &=: A \circ B,\end{aligned}$$

where $0 \notin B$ in case of division. It is also evident that

$$\begin{aligned}A + B &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}], \\ A - B &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}].\end{aligned}$$

The quality of an interval may be measured by its diameter $d(A) = d([\underline{a}, \overline{a}]) = \overline{a} - \underline{a}$. One computes

$$d(A + B) = d(A) + d(B),$$

but also $d(A - B) = d([\underline{a} - \overline{b}, \overline{a} - \underline{b}]) = (\overline{a} - \underline{b}) - (\underline{a} - \overline{b}) = (\overline{a} - \underline{a}) + (\overline{b} - \underline{b})$, and therefore

$$d(A - B) = d(A) + d(B). \tag{10.2}$$

So the diameter of the sum *and of the difference* of two intervals is equal to the sum of the diameters. This is a major reason for overestimation, as we will discuss in more detail later.

Similarly, bounds for multiplication and division can be determined by some case distinctions depending on the operands being entirely positive, negative, or including zero, respectively. Interval vectors can be defined equivalently by

$$\begin{pmatrix} [\underline{x}_1, \overline{x}_1] \\ \dots \\ [\underline{x}_n, \overline{x}_n] \end{pmatrix} \in (\mathbb{IR})^n \quad \text{or} \quad X = [\underline{x}, \overline{x}] \in \mathbb{IR}^n,$$

where the latter uses the partial ordering of vectors by componentwise comparison. We proceed similarly for matrices and define operations among matrices, vectors, and scalars by replacing every operation in the usual definition by the corresponding interval operation. For example, $Y = AX$ for interval quantities $X, Y \in \mathbb{IR}^n$ and $A \in \mathbb{IR}^{n \times n}$ is defined by

$$Y := AX \quad \text{with} \qquad Y_i := \sum_{k=1}^{n} A_{ik} X_k \quad \text{for } 1 \le i \le n, \tag{10.3}$$

where additions and multiplications are the corresponding (scalar) interval operations. Obviously,

$$\widetilde{y} = \widetilde{A}\widetilde{x} \in Y \quad \text{for all } \widetilde{A} \in A, \widetilde{x} \in X.$$

This is one incarnation of the fundamental underlying principle of the inclusion isotonicity for all interval operations:

Inclusion isotonicity: For all interval quantities A, B and all operations $\circ \in \{+, -, \cdot, /\}$ such that $\widetilde{a} \circ \widetilde{b}$ is well defined for $\widetilde{a} \in A, \widetilde{b} \in B$ there holds

$$\widetilde{a} \circ \widetilde{b} \in A \circ B \quad \text{for all } \widetilde{a} \in A, \widetilde{b} \in B. \tag{10.4}$$

Inclusion isotonicity, the central property of interval arithmetic

We mention that the definition of interval operations is the *best possible* with respect to inclusion isotonicity: Moving any of the bounds "inward" will violate (10.4). However, there is nevertheless some overestimation. Consider

$$A \cdot X := \begin{pmatrix} -1 & 2 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} [1, 2] \\ [1, 3] \end{pmatrix} = \begin{pmatrix} [0, 5] \\ [3, 8] \end{pmatrix}.$$

This defines a linear transformation of a rectangle resulting in a parallelogram. The resulting interval vector is the best possible in the sense that both interval components [0, 5] and [3, 8] cannot be narrowed without violating inclusion isotonicity. It is easy to find (real) vectors out of X such that the bounds 0, 5, 3, and 8 of the resulting interval vector are met.

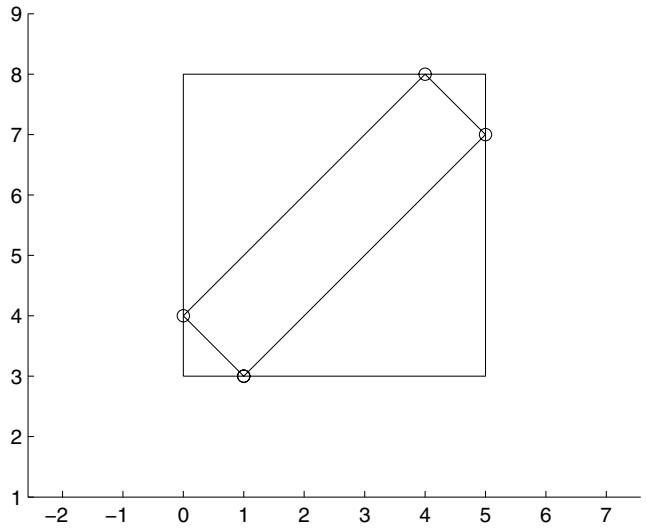

Figure 10.4. *Point matrix times interval vector.*

However, Figure 10.4 displays the true rectangle AX. Obviously no vector $x \in X$ can be found such that $A\widetilde{x} = (0, 3)^T$ for $\widetilde{x} \in X$. This illustrates a common pitfall of interval arithmetic: Given $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, and $X \in \mathbb{IR}^n$, and check $b \in AX$ to verify $A^{-1}b \in X$. This is an incorrect conclusion because

$$\text{for all } x \in X: \quad Ax \in AX$$

but not necessarily

$$y \in AX \quad \Rightarrow \quad \exists\, x \in X: \quad y = Ax.$$

These fundamentals of interval operations have been written down many times, and the interested reader is referred to [319], [340], or [11] and the literature cited therein for more details.

We briefly mention that complex interval arithmetic can be defined as well. In this case, for function theoretical reasons, it seems more appropriate to use discs, i.e., a midpoint-radius arithmetic (but rectangular arithmetic is also used). Define for $a \in \mathbb{C}$, $0 \le r \in \mathbb{R}$,

$$\langle a, r\rangle := \{z \in \mathbb{C}: \ |a - z| \le r\}. \tag{10.5}$$

Then interval operations satisfying inclusion isotonicity have been defined in [159], for example,

$$\langle a, r\rangle \cdot \langle b, s\rangle := \langle ab, \ |a|s + r|b| + rs\rangle. \tag{10.6}$$

Corresponding complex interval vector and matrix operations are defined similarly as in (10.3).

10.6 Directed Roundings

The definitions thus far are for real bounds and exactly performed operations. For implementation on digital computers, special care must be taken since the result of a floating-point

operation is, in general, not a floating-point number. These problems can be solved by using directed rounding. First, recall that the set $\mathbb{IF}$ of intervals with floating-point bounds is defined by

$$A \in \mathbb{IF} \ :\Leftrightarrow \ A \neq \emptyset \quad \text{and} \quad A = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} : \ \underline{a} \leq x \leq \overline{a}\} \text{ for } \underline{a}, \overline{a} \in \mathbb{F}.$$

Note that the two floating-point bounds $\underline{a}$ and $\overline{a}$ represent the set of all *real* numbers between them. In the degenerate case $\underline{a} = \overline{a}$ this is one single floating-point number, and every floating-point number can be identified with this point interval. In other words, $\mathbb{F} \subseteq \mathbb{IF}$ with the natural embedding. When computing with intervals having floating-point endpoints the same definitions as above can be used except that the lower bound is computed by rounding downward, and the upper bound is computed by rounding upward. For example,

$$[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\mathrm{fl}_\nabla(\underline{a} + \underline{b}), \mathrm{fl}_\Delta(\overline{a} + \overline{b})]$$

or

$$[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] = [\mathrm{fl}_\nabla(\underline{a} - \overline{b}), \mathrm{fl}_\Delta(\overline{a} - \underline{b})].$$

Other definitions follow the same lines. For example, definition (10.3) of $Y = AX$ does not change, but interval additions and multiplications are now performed with floating-point endpoints; i.e., the operations are executed with directed roundings. One may argue that this requires many switches of the rounding mode and may slow the computation significantly. This is indeed true, and we will discuss how to overcome this problem later.

Before continuing we mention that the interval arithmetical operations are implemented in a number of software packages. Most easy to use is INTLAB [389], a recent implementation as a MATLAB interval toolbox. A nice introduction to INTLAB and a tutorial can be found in [196]. INTLAB has a built-in new data type called `intval`, and due to operator overloading every operation between an `intval`-quantity and something else (for example, a real or complex number, or a real or complex interval) is recognized as an interval operation and executed with directed rounding. For example, an INTLAB implementation, i.e., executable code, of the above algorithm to verify nonsingularity of a matrix may be as in the program in Figure 10.5.

```
R = inv(A);
C = abss(eye(n)-R*intval(A));
x = ones(n,1);
setround(+1)
nonsingular = all( C*x < x )
```

Figure 10.5. *INTLAB program to check $|I - RA|x < x$ and therefore the nonsingularity of A.*

We want to stress that this is executable code in INTLAB. Given a square matrix `A` of dimension `n`, first an approximate inverse `R` is computed using pure floating-point arithmetic. Then, the multiplication `R*intval(A)` is performed in interval arithmetic since the type cast `intval(A)` transforms the matrix `A` to an interval matrix. The result is of type `intval`, so that the subtraction `eye(n)-R*intval(A)` is also an interval subtraction. For an interval matrix $\mathbf{C} \in \mathbb{IF}^{n \times n}$, the function `abss` is defined by

$$\mathrm{abss}(\mathbf{C}) := \max\{|\widetilde{C}| : \ \widetilde{C} \in \mathbf{C}\},$$

where the maximum is to be understood entrywise. So `abss(C)` $\in \mathbb{F}^{n\times n}$ satisfies $|\widetilde{C}| \leq$ `abss(C)` for all $\widetilde{C} \in \mathbf{C}$, and this is the entrywise smallest matrix with this property. This implies

$$|I - RA| \leq \mathtt{C},$$

where `C` denotes the computed (floating-point) result. The vector `x` is defined to be a vector of ones, and `setround(+1)` switches the rounding mode to upward. So the product `C*x` of the nonnegative quantities `C` and `x` is an upper bound of the true value of the product. The assertion follows.

Note that the value `nonsingular=1` proves `A` to be nonsingular, whereas the value `nonsingular=0` means the algorithm failed to prove nonsingularity. Rather than giving a false answer the result is to be interpreted as "don't know." So this self-validating program may fail to prove nonsingularity (for very ill-conditioned matrices), but it never gives a false answer. In contrast, a computer algebra algorithm could always decide nonsingularity or singularity when computing in sufficiently long arithmetic or infinite precision.

At this point we note that almost every self-validating algorithm for point data can be transformed into a corresponding one accepting interval data. In the above case let an interval matrix `A` $\in \mathbb{IF}^{n\times n}$ be given. It represents the set of all real matrices $\widetilde{A}$ within the interval entries. The question is whether every matrix $\widetilde{A}$ within `A` is nonsingular. Consider the algorithm in Figure 10.6.

```
R = inv(mid(A));
C = abss(eye(n)-R*intval(A));
x = ones(n,1);
setround(+1)
nonsingular = all( C*x < x )
```

Figure 10.6. *INTLAB program for checking nonsingularity of an interval matrix.*

The only difference between this and the program in Figure 10.5 is the first line. Here `mid(A)` denotes the mid-point matrix of `A`. We claim that the result `nonsingular=1` proves that every matrix $\widetilde{A} \in$ `A` is nonsingular. The *proof* of this fact is very simple and very typical for self-validating methods. Let $\widetilde{A} \in$ `A` be a fixed but arbitrary (real) matrix. Then, by the fundamental principle (10.4) of inclusion isotonicity,

$$I - R\widetilde{A} \in I - R\mathtt{A} \text{ and therefore } |I - R\widetilde{A}| \leq \mathtt{C},$$

and the assertion follows. The point is to take any fixed but arbitrary data out of the input data and apply inclusion isotonicity. What is true for interval data is true for every point data out of the interval data.

For interval rather than floating-point input data `intval` in the second line of the program in Figure 10.6 could be omitted. In the stated way the program works correctly for interval as well as for noninterval input matrices.

We note that the proof of nonsingularity of an interval matrix is nontrivial. In fact this is an NP-hard problem; see [365]. We also note that the present implementation is nothing else but to check $\|I - RA\|_\infty < 1$, which, of course, implies nonsingularity of `R` and `A`. If the vector `x=ones(n,1)` fails to prove nonsingularity, one may iterate `x`. This is a power iteration to approximate the Perron vector of `C`.

10.7 A Common Misconception About Interval Arithmetic

The arguments so far follow a simple but powerful pattern: If arithmetic operations are replaced by their corresponding interval operations, then the true result must be an element of the interval result. In our example we used this to prove $|I - RA|x < x$ and therefore the nonsingularity of the matrix A. This technique might be applied to any numerical algorithm, for example, to Gaussian elimination for a linear system $Ax = b$. After execution, the true result, the solution of the system of linear equations $Ax = b$ must be an element of the computed interval result, provided no pivot element contained zero, because otherwise the algorithm would come to a premature end.

This technique is called *naive interval arithmetic* and represents the most common *misuse* of interval arithmetic. For a general application this approach will most certainly fail. Note that the assertion, namely, that the final interval result contains the exact solution of the linear system, thereby also proving nonsingularity of the matrix in use, is true. However, this approach will most certainly come to a premature end due to a pivot interval containing zero. The reason is interval dependency, and we will explain the effects with a simple example.

Consider a linear system with lower triangular matrix, where all elements on and below the diagonal are equal to 1, and a right-hand side with all elements being equal to 1 except the first one, which is equal to 0.1. For $n = 4$ the linear system looks as follows:

$$\begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 1 & 1 & \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0.1 \\ 1 \\ 1 \\ 1 \end{pmatrix}. \tag{10.7}$$

The real number 0.1 is not exactly representable in binary floating point, so to treat the original linear system it has to be replaced by some interval $[\underline{\beta}, \overline{\beta}]$ containing 0.1, where $\underline{\beta}$ and $\overline{\beta}$ are adjacent binary floating-point numbers. This is an optimal inclusion, the best one can do in finite precision. For simplicity of the following we change this interval to midpoint-radius notation $[\underline{\beta}, \overline{\beta}] = \alpha \pm \mathbf{u}$, where α is the midpoint of $[\underline{\beta}, \overline{\beta}]$ and $\mathbf{u}$ is the radius. Now we perform a forward substitution using interval operations, and we will do this theoretically without any further rounding, i.e., using real interval operations. Obviously,

$$\begin{aligned} X_1 &= \alpha \pm \mathbf{u}, \\ X_2 &= 1 - X_1 = 1 - \alpha \pm \mathbf{u}, \\ X_3 &= 1 - X_1 - X_2 = X_2 - X_2 = \pm 2\mathbf{u}, \\ X_4 &= 1 - X_1 - X_2 - X_3 = X_3 - X_3 = \pm 4\mathbf{u}. \end{aligned}$$

The point is that the interval subtraction $X_2 - X_2$ does not cancel to zero but is an interval with midpoint zero and doubled diameter. As we saw in equation (10.2), this is always the case.

So, from X_2 on, the diameter of every X_i doubles and the overestimation grows exponentially! Unfortunately, this is the typical behavior for this kind of naive interval approach. For the same reason it is most likely, even for small dimensions, that a pivot interval will contain zero at an early stage of naive interval Gaussian elimination—unless the matrix has special properties such as being an M-matrix.

In Table 10.1 we show some statistics obtained for randomly generated matrices, where all matrix elements are uniformly distributed within $[-1, 1]$. For matrix dimensions $n = 40 \ldots 70$ we generate 100 samples each and monitor the average and maximum condition number, the number of failures of naive interval Gaussian elimination, i.e., the number of cases where a pivot element contains zero, and, for the nonfailing cases, the average and maximum of the radius of U_{nn}.

Table 10.1. *Exponential growth of radii of forward substitution in equation (10.7) in naive interval arithmetic.*

	cond(A)			rad(U_{nn})	
n	average	maximum	failed	average	maximum
40	$4.5280 \cdot 10^2$	$9.7333 \cdot 10^3$	0	$3.5633 \cdot 10^{-4}$	$4.6208 \cdot 10^{-3}$
50	$8.9590 \cdot 10^2$	$4.5984 \cdot 10^4$	1	$7.1158 \cdot 10^{-1}$	$3.8725 \cdot 10^1$
60	$1.6132 \cdot 10^3$	$9.1094 \cdot 10^4$	98	$1.5774 \cdot 10^0$	$1.5603 \cdot 10^2$
70	$1.1159 \cdot 10^3$	$4.3883 \cdot 10^4$	100	-	-

The failure is not a question of the condition number of the matrix but of the *number of operations*. The failure is due to the fact that certain computations depend on previous computations. The previously computed results are already perturbed by some error, and this amplifies in the next computation and so forth. This effect can also be read off the radius of the diagonal elements U_{ii}. Figure 10.7 shows typical behavior for a random matrix of dimension $n = 50$ on a semilogarithmic scale.

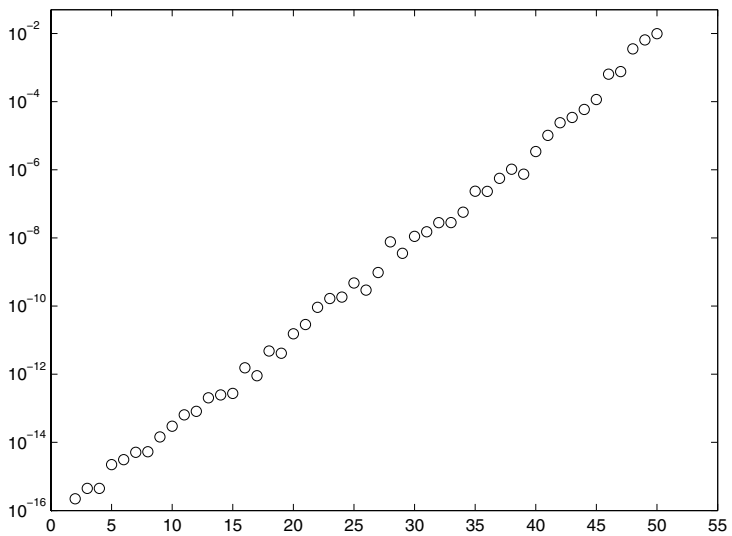

Figure 10.7. *Naive interval Gaussian elimination: Growth of* rad(U_{ii}).

These observations also give the key to the design of self-validating methods. First, using already computed data may cause amplification of interval diameters; therefore, one should try to use original data as much as possible. Second, the only way to decrease the radius of result intervals is to multiply them by a small factor (or divide by a large one). Adding and subtracting intervals add the radii. Therefore, one should try to have small factors multiplying interval quantities where possible.

Before continuing, two things should be mentioned. First, the application of interval methods is not restricted to linear problems and/or basic arithmetical operations. There are libraries for the standard elementary functions available [51, 267] for real and complex point and interval data. Also INTLAB provides rigorous interval standard functions for real and complex input data [391]. For any elementary standard function f and for any real or complex interval X, the computed result $f(X)$ is a superset of all $f(x)$ for $x \in X$. For monotone functions such as the exponential or logarithm this requires only rigorous standard functions for a point argument. For functions like sine some case distinctions are necessary. For the sine and other periodic functions an accurate argument reduction is also necessary [358] in order to maintain accuracy of the result for larger arguments. All this is implemented in INTLAB, which is fast and accurate even for very large arguments, following the methods presented in [391].

The second remark is that the naive interval approach may cause exponential overestimation, as in naive interval Gaussian elimination, but it need not. Consider the following example which describes exactly how it happened to us. In global optimization there are a number of well-known test functions, among them the following example by Griewank:

$$g(x) = (x^2 + y^2)/4000 + \cos(x)\cos(y)/\sqrt{2} + 1. \tag{10.8}$$

The minimum of this function in the domain

$$-60 \le x \le 60 \quad \text{and} \quad -60 \le y \le 60$$

is sought. In order to obtain a first overview we plotted this function. Figure 10.8 from MATLAB displays 20 meshpoints in each coordinate direction.

At first sight one may wonder why it should be difficult to find the minimum of this (apparently convex) function. Calculating the minimum of the function values at the meshpoints yields 1.7119. By nature this is in fact an *upper bound* for the minimum. Interval evaluation is also very simple. The following

```
G = inline('(x^2+y^2)/4000+cos(x)*cos(y)/sqrt{2}+1')

>> X = infsup(-60,60); Y = X; G(X,Y)
```

is executable INTLAB code and yields

```
intval ans =
[   0.2928,   3.5072]
```

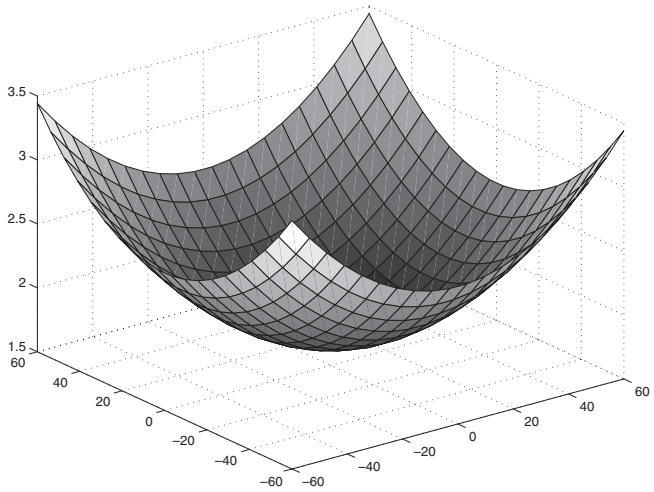

Figure 10.8. *Plot of the function in equation (10.8) with 20 meshpoints in x- and y-directions.*

Note that the left endpoint is a true lower bound of the true minimum of the function g within the domain under investigation. One may argue that some, though not much, overestimation took place. However, evaluating the same function with 50 meshpoints in each direction produces Figure 10.9.

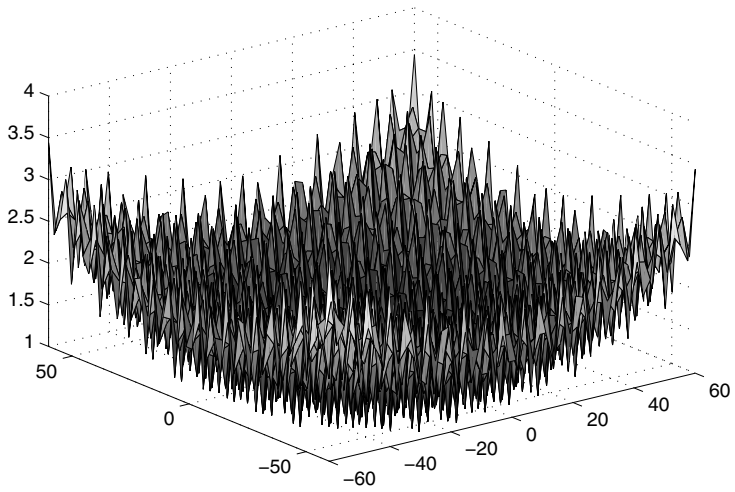

Figure 10.9. *Plot of the function in equation (10.8) with 50 meshpoints in x- and y-directions.*

This reveals more of the true nature of this nasty function and justifies its use as a test function for global optimization. The minimum of the 50^2 function values is 0.3901, again an *upper bound* for the true minimum. Evaluation at 2000 meshpoints in each direction, that is, a total of 4 million meshpoints, yields the upper bound 0.2957 for the true minimum, not too far from the valid lower bound 0.2928 obtained by naive interval evaluation.

Although sometimes affected by overestimation, naive interval evaluation provides a very powerful tool to estimate the range of a function over a domain. This estimation is valid and rigorous and is obtained in a very simple way without any further knowledge of the behavior of the function, a quite remarkable property.

This is why during the first steps of the development of verification algorithms people were a little overwhelmed by this powerful property. So sometimes interval methods were advocated to solve numerical problems per se in an easy and elegant way, with automatic verification of the correctness of the results. To be fair, we must say that up to that time, in the 1960's, computers and programming languages were in a fairly premature state. Compared to today's possibilities we readily have at hand, this is more like a horse pulled carriage compared to a sportscar. In particular, powerful interactive numerical programming environments like MATLAB were not widely available. So testing was not as easy as we have become accustomed to today at the push of the fingertip.

No wonder it took some time until others could try to verify the claims. Additionally, this was hindered by the fact that interval operations were available to few. And finally, computers were still slow. It could take a couple of minutes to solve a 100×100 linear system (cf. the enlightening Table 9.2 in [205]), not to mention how much additional time software simulation of interval operations would consume; and, of course, not everybody had access to the fastest computers.

But then people tried standard examples, such as naive interval Gaussian elimination, and had to assert that there is quite some overestimation. In fact, exponential overestimation is not only not rare but rather typical. No wonder interval advocates were measured by their own claims, and this remains the reason for divided reputation of interval analysis.

Naive interval arithmetic can also be regarded as an automated forward error analysis: The worst case of every single operation is estimated. For Gaussian elimination this has been shown in the famous paper [344]. The results were very pessimistic, as are those of naive interval arithmetic. This paper led for some time even to the conclusion that larger linear systems could not be solved reliably on digital computers because of accumulating rounding errors. Here, "large" could mean dimension 10; remember that, for example, in 1947 a 10×10 linear system was successfully solved on the Harvard Mark I in 45 minutes of computing time [37]. This was "fast" compared to the 4 hours for a problem of the same size on an IBM 602 in 1949 [466].

This kind of automated forward error analysis, i.e., naive interval analysis, estimates the worst case error of every simple operation, and it treats all operations as being independent. But rounding errors are not independent, and they are also by no means randomly distributed (cf. the enlightening talk by Kahan [249]). Fortunately we know today how to estimate the forward error very precisely, for example, in terms of the condition number.

However, interval analysis has developed since then, and the times of naive interval approaches are long over. In addition, despite overestimation, such an easy way to

obtain a valid inclusion of the range of a function can be useful. Moreover, the overestimation in naive interval Gaussian elimination for matrices with larger dimension stems from using computed results over and over again. This is not the case when using other algorithms.

Let us again consider an example, the computation of the determinant of a matrix. One method is, of course, to use Gaussian elimination and to compute the product of the diagonal elements of the factor U. This approach may work for small dimensions but definitely not for larger ones. Another approach is given in the program in Figure 10.10, again an executable INTLAB program.

```
[L U P] = lu(A);
Linv = inv(L);
Uinv = inv(U);
B = Linv*intval(P*A)*Uinv;
B0 = B;
B0(1:n+1:n*n) = 0;
G = diag(B)' + midrad(0,abss(sum(B0)));
d = prod(G)/prod(intval(diag(Uinv)))/det(P)
```

Figure 10.10. *INTLAB program for the inclusion of the determinant of a matrix.*

Here `Linv` and `Uinv` are approximate inverses of the approximate LU-factors of `A`. Then `B` is an inclusion of `Linv*P*A*Uinv`, so the determinant of `B` is equal to the quantity $\pm$ `det(A)prod(diag(Uinv))`, with the plus or minus depending on the sign of the permutation matrix `P`. The next statements compute an inclusion of the product of the Gershgorin circles of `B`, which is an inclusion of the product of the eigenvalues of `LinvPAUinv`, which in turn is equal to the determinant. The final line then computes an inclusion of the determinant of `A`.

The point is (i) to use original data where possible, (ii) to use floating-point approximations where possible, and (iii) to design the algorithm in a way that potential overestimation is minimized. This is the fundamental difference between the naive approach and a self-validating method. Figure 10.11 shows the accuracy by means of the relative error of the inclusion of the determinant calculation by naive interval Gaussian elimination (depicted by "$\times$") and by the above algorithm (i.e., Gershgorin applied to a preconditioned matrix, depicted by "$\circ$" for dimensions 10 to 500).

All matrices were generated randomly with entries uniformly distributed in $[-1, 1]$. Since the determinant of such random matrices of larger dimension are likely to be out of the floating-point range, matrices are scaled by powers of 2 to avoid this. For example, the determinant of a matrix of dimension 500 will be calculated in double precision by the above algorithm to about 8 correct figures. The naive approach cannot handle matrices of dimension greater than 60.

The lesson is that what works well for numerical computations need not be a good method for interval approaches. We need individual methods for each given problem. Standard numerical methods may help (and they do), but usually something more is needed.

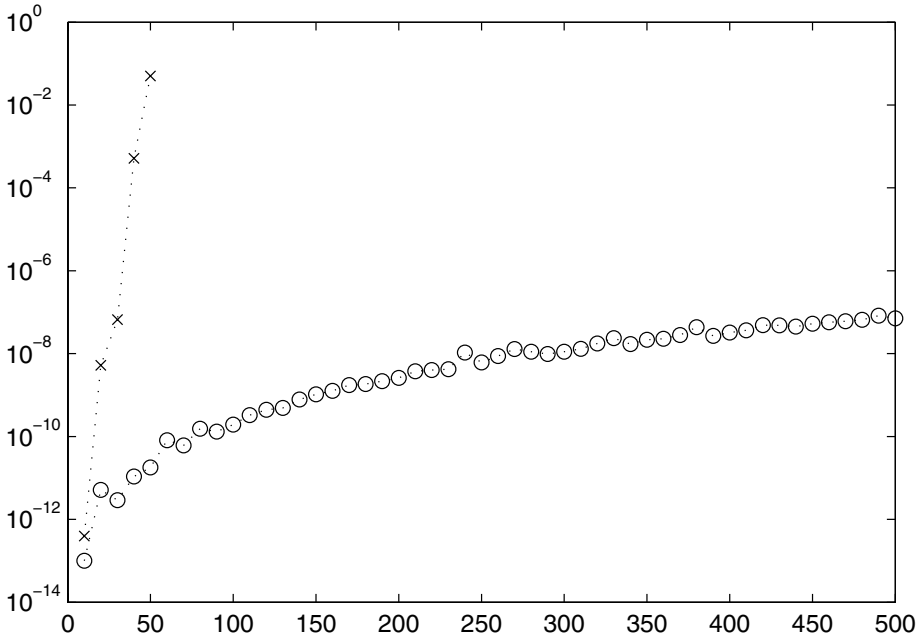

Figure 10.11. *Comparison of a naive interval algorithm and a self-validating method: Naive Gaussian elimination depicted by "×," Gershgorin for preconditioned matrix by "∘."*

10.8 Self-validating Methods and INTLAB

Consider a system of nonlinear equations given by a function $f : \mathbb{R}^n \to \mathbb{R}^n \in C^1$, for which we look for an inclusion of a zero $\widehat{x}$ near some approximation $\widetilde{x}$. A simplified Newton procedure transforms the function f into a function g with

$$g(x) := x - Rf(x)$$

for some preconditioner R. A standard choice for R is an approximate inverse of the Jacobian of f at $\widetilde{x}$. Suppose R is nonsingular and there is some interval vector $X \in \mathbb{IR}^n$ with

$$g(X) \subseteq X.$$

That means the continuous function g maps the nonempty, convex, and compact set X into itself, and Brouwer's fixed point theorem implies existence of a fixed point $\widehat{x}$ of g in X:

$$\exists\, \widehat{x} \in X : \; g(\widehat{x}) = \widehat{x}.$$

By definition this implies

$$R \cdot f(\widehat{x}) = 0 \quad \text{and therefore } f(\widehat{x}) = 0$$

because R was supposed to be nonsingular. In other words, the set X is proved to be an inclusion of a zero of f if we can verify that (i) R is nonsingular, and (ii) g maps X into itself. The first problem might be solved by the previously described method, and the second one by observing that

$$g(X) \subseteq X - R \cdot f(X).$$

This is a naive approach, and it definitely does not work. Although it is correct that $X - R \cdot f(X) \subseteq X$ implies $g(X) \subseteq X$ and therefore there exists a zero of f in X, the condition $X - R \cdot f(X) \subseteq X$ can be satisfied only if $R \cdot f(X) = 0$, which in turn is possible only if $f(X) \equiv 0$. The overall approach is good, but the point is that the condition $g(X) \subseteq X$ must be verified another way.

A convenient way to do this is by using a first-order Taylor expansion, i.e., the mean value theorem. To do this we need the differential of a function, and this can be conveniently calculated by automatic differentiation. We assume the reader is familiar with this technique; for an overview see [370] and [178]. The forward mode of automatic differentiation can easily be implemented in every programming language with an operator concept. In MATLAB version 5 and following versions there is a nice and easy-to-use operator concept available. Consider, for example, Broyden's test function [57]

$$\begin{aligned} f_1(x, y) &= 0.5 \sin(xy) - \frac{y}{4\pi} - \frac{x}{2}, \\ f_2(x, y) &= \left(1 - \frac{1}{4\pi}\right) \cdot (e^{2x} - e) + e\frac{y}{\pi} - 2ex. \end{aligned} \tag{10.9}$$

A MATLAB implementation of the function in equation (10.9) may use the inline concept:

```
>> f = inline('[ .5*sin(x*y) - y/(4*pi) - x/2 ; ...
       (1-1/(4*pi))*(exp(2*x)-exp(1)) + exp(1)*y/pi ...
        - 2*exp(1)*x ]')
```

Then the function value at $x = 0.5$ and $y = 3$ can be calculated by

```
>> f( [ .5 ; 3 ] )
ans =
    0.0100
   -0.1225
```

This is the screenshot from MATLAB. In INTLAB, we implemented automatic differentiation in forward mode. The corresponding call is

```
>> f( gradientinit( [ .5 ; 3 ] ) )
gradient value ans.x =
    0.0100
   -0.1225
gradient derivative(s) ans.dx =
   -0.3939   -0.0619
   -0.4326    0.8653
```

The computed result is an approximation of the Jacobian of f evaluated at $(x, y) = (0.5, 3)$. The remarkable property of interval arithmetic is the possibility of estimating the range of a function without any further knowledge of the function. In this example we may evaluate the range of Broyden's function for $x = 0.5$ and y varying within [2.9, 3.1]. The INTLAB call is

```
>> Z = [ .5 ; infsup(2.9,3.1) ]; f(Z)
intval ans =
   [   -0.0004,    0.0192]
   [   -0.2091,   -0.0359]
```

The result must be a superset of all function values $f(0.5, y)$ for $2.9 \le y \le 3.1$. In the same way we may calculate the range of function values and the range of partial derivatives:

```
>> f(gradientinit(Z))
intval gradient value ans.x =
   [   -0.0004,    0.0192]
   [   -0.2091,   -0.0359]
intval gradient derivative(s) ans.dx =
   [   -0.4699,   -0.3132] [   -0.0744,   -0.0494]
   [   -0.4327,   -0.4326] [    0.8652,    0.8653]
```

The mathematical assertion is that for every (x, y) within Z the values of the partial derivatives of f are included in the intervals shown. However, this correct mathematical assertion refers to the given function f and the given interval Z. Here things are a little tricky because of possible conversion errors, and we want to bring attention to this. When the user inputs the interval `Z = infsup(2.9,3.1)`, the set of real numbers between the *real* numbers 2.9 and 3.1 is anticipated. Neither 2.9 nor 3.1 is exactly representable in floating point. So, to be on the safe side, the smallest interval with floating-point endpoints containing this real interval is the best choice. But the function `infsup`, which creates an interval of the given endpoints, requires floating-point input parameters. And the input data 2.9 and 3.1 are already converted to binary floating point when arriving at the function `infsup`.

This is a common problem in every programming language. In order to round the input data correctly, seemingly the only way is to specify it as a *string* and to perform the conversion in the correct way. The INTLAB input may be

```
Y = intval('[2.9,3.1]'); Z = [ .5 ; Y ];
```

Accidentally (or luckily?) the intervals Y and `infsup(2.9,3.1)` coincide. This solves the first problem that the input data are not exactly representable.

The second problem is that the original Broyden function contains the constant π. The MATLAB constant `pi` is the best double precision floating-point approximation to the transcendental number π, but, of course, is not equal to π. For floating-point evaluation of the function or the gradient value this is no problem; in fact, it is the best one can do. But for a valid estimation of the range, it is not enough.

The problem is a little delicate because for floating-point evaluation, i.e., floating-point arguments, we wish to use the floating-point approximation `pi`, whereas for range

estimation we wish to use a valid inclusion of the transcendental number π. So the constant in the function depends on the type of the input argument. This is solved in INTLAB as in the program in Figure 10.12.

```
function y = f(x)
  y = x;
   cpi = typeadj( intval('3.14159265358979323') , typeof(x) );
   c1 = typeadj( 1 , typeof(x) );
   y(1) = .5*sin(x(1)*x(2)) - x(2)/(4*cpi) - x(1)/2;
   y(2) = (1-1/(4*cpi))*(exp(2*x(1))-exp(c1)) ...
           + exp(c1)*x(2)/cpi - 2*exp(c1)*x(1);
```

Figure 10.12. *INTLAB program of the function in equation (10.9) with special treatment of* π.

The function `typeof` determines whether `x` is of type `intval`. Then the function `typeadj` adjusts the output to the same type as the second input argument. If both arguments of `typeadj` are of type interval or not of type interval, nothing happens. Otherwise, the output either is the midpoint of the first argument or it is the point interval equal to the first argument. In that way either an approximation of π or an inclusion is used, depending on the type of the input argument `x`.

Note that the same is applied to the constant `1`. Of course, `1` is exactly representable in floating point. But in the function, `exp(1)` is calculated. The operator concept adheres strictly to the type concept, which means that `exp(1)` computes a floating-point approximation to the transcendental number e, whereas `exp(intval(1))` calculates an inclusion.

Let us now turn to the calculation of bounds for the solution of systems of nonlinear equations. We mention only some characteristic points, as there are a number of well-written surveys (see, for example, [11, 392]). We need an estimation of the range $g(X)$. So first the function g is expanded at some $\widetilde{x}$, an approximate zero of f. Denote the convex union by $\underline{\cup}$. Then for every x we have

$$g(x) = g(\widetilde{x}) + M \cdot (x - \widetilde{x}), \tag{10.10}$$

where $M_i = \frac{\partial g}{\partial x}(\zeta_i)$, $\zeta_i \in x \,\underline{\cup}\, \widetilde{x}$. Note that the points ζ_i are determined individually within $x \,\underline{\cup}\, \widetilde{x}$. Then, for every $x \in X$,

$$\begin{aligned} g(x) &= g(\widetilde{x}) + M \cdot (x - \widetilde{x}) \\ &= \widetilde{x} - R \cdot f(\widetilde{x}) + \{I - R \cdot M\}(x - \widetilde{x}) \\ &\subseteq \widetilde{x} - R \cdot f(\widetilde{x}) + \{I - R \cdot [M]\}(X - \widetilde{x}) \\ &=: K_f(\widetilde{x}, X) \qquad \text{Krawczyk operator.} \end{aligned} \tag{10.11}$$

In the proof most calculations are performed over $\mathbb{R}^n$; only the last step changes to intervals. The reason is that for interval calculations simple laws like distributivity or $X - X = 0$ do not hold. Moreover, an inclusion $[M]$ of the set of matrices M for which equation (10.10) is true is easily calculated by interval automatic differentiation evaluation of g at X. To complete the approach we need the nonsingularity of a preconditioning matrix R. This is established by Theorem 10.1.

Theorem 10.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^n \in C^1$, $\widetilde{x} \in \mathbb{R}^n$, and $X \in \mathbb{IR}^n$ be given. Suppose $\widetilde{x} \in X$ and $K_f(\widetilde{x}, X) \subseteq \mathrm{int}(X)$, where the operator K_f is defined in equation (10.11). Then there exists one and only one $\widehat{x} \in X$ with $f(\widehat{x}) = 0$.*

Here $\mathrm{int}(X)$ denotes the interior of X; the check whether the left-hand side is enclosed in the interior of the right-hand side means just replacing the "less than or equal" comparison by "less than." This is the principle of a self-validating method for systems of nonlinear equations [386]. Of course we need to discuss many details. For example, how to construct a suitable starting interval X and what to do if the inclusion condition $K_f(\widetilde{x}, X) \subseteq \mathrm{int}(X)$ is not satisfied. Such details are worked out in the routine `verifynlss` (cf. [387]) which is part of INTLAB. Figure 10.13 shows the executable code with input and output.

```
>> xs = [ .5 ; 3 ];  verifynlss(f,xs)
intval ans =
[    0.4999, 0.5001]
[    3.1415, 3.1416]

>> xs = [ 0 ; 0 ];  X = verifynlss(f,xs)
intval X =
[ -0.2606, -0.2605]
[  0.6225,  0.6226]

>> format long; X
intval X =
[ -0.26059929002248, -0.26059929002247]
[  0.62253089661391,  0.62253089661392]
```

Figure 10.13. *INTLAB results of* `verifynlss` *for Broyden's function (10.9).*

The number of displayed digits depends on the output format in MATLAB chosen by the user. The first inclusions look rather crude, but this is the best that can be done in short output format. Changing the format to long reveals the actual accuracy of the bounds.

We wrote specially designed routines for the output in INTLAB to ensure that the displayed answer is indeed correct. For example, the real number 0.1 is not exactly representable in binary floating point, and `X=intval('0.1')` calculates a valid inclusion of this real number. Displaying `X` in short format yields

```
>> X=intval('0.1')
intval X =
[    0.0999,    0.1001]
```

because this is the *best possible* answer using 5 decimal digits: The interval `X` must contain the real number 0.1, and because this is not a floating-point number, 0.1 *cannot* be a bound for `X`. Sometimes it may be tedious to count coinciding digits. For example,

```
>> X = midrad(2.718281828459045,1e-12)
intval X =
[   2.71828182845804,   2.71828182846005]
```

declares an interval of radius 10^{-12}, which is not immediately recognizable from the output. In that case one may switch to another output by

```
>> intvalinit('display_'); X
intval X =
   2.71828182846___
```

The rule is that subtracting one unit from the last displayed digit and adding one unit produces a correct inclusion of the result. In this way the accuracy is easily determined by looking at the number of displayed digits. Sometimes the format can describe vividly the convergence rate of an iteration. Consider, for example, the interval Newton iteration, displayed in Figure 10.14 with executable code and input and output.

```
>> f=inline('sqr(x)-2'), X=infsup(1,1.5);
   for i=1:4, y=f(gradientinit(X)); X=X.mid-y.x.mid/y.dx, end
f =
     Inline function:
     f(x) = sqr(x)-2
intval X =
   1.4_____________
intval X =
   1.414___________
intval X =
   1.4142136_______
intval X =
   1.41421356237309
```

Figure 10.14. *INTLAB example of quadratic convergence.*

10.9 Implementation of Interval Arithmetic

The implementation of interval arithmetic may be performed along the lines of the definition of interval arithmetic for scalars, vectors, and matrices as in (10.3), and this is the way it was done for a long time. However, this may result in poor performance. When floating-point multiplication took significantly more computing time than floating-point addition, it was reasonable to compare the speed of algorithms by comparing the count of floating-point multiplications (the time for additions was comparatively negligible). Also, it was not too important whether an algorithm contained some branches.

This situation has changed completely. Today, many elementary operations, such as addition, subtraction, multiplication, division, and even a multiplication and an addition (multiply and add) require one cycle. This is also the reason why today multiplications and additions are counted equally. Both statements that Gaussian elimination requires $\frac{1}{3}n^3$ or $\frac{2}{3}n^3$ operations are correct, the first one 20 years ago, the other today.

By far more important, the actual execution time of today's algorithms depends on how the data are available: in registers, in cache, in memory, or possibly on external devices. Depending on data location the computing time may vary tremendously.

The various optimizing options in a compiler try to improve a given code as much as possible in that respect. But sometimes it is advisable to "help" the compiler to do a

better job. A well-known example is the use of unrolled loops. The standard loop for the computation of a scalar product $x^T y$ is given in Figure 10.15.

```
s = 0;
for i=1:n
  s = s + x(i)*y(i);
end
```

Figure 10.15. *Dot product.*

An unrolled loop is given in Figure 10.16.

```
r = mod(n,4);
s = 0;
nmax = n-r-3;
for i=1:4:nmax
  s = s + x(i)*y(i) + x(i+1)*y(i+1) + x(i+2)*y(i+2) + x(i+3)*y(i+3);
end
for i=n-r+1:n
  s = s + x(i)*y(i);
end
```

Figure 10.16. *Dot product with unrolled loop.*

Here we use MATLAB notation to describe the approaches; of course, MATLAB execution time in this case would be mainly determined by the interpretation overhead. But executing the two versions in a compiled programming language reveals the difference. Here we used the C programming language and Gnu compiler [165]. The performance for non-optimized code without unrolled loops is normed to 1; i.e., a larger number indicates better performance. The difference is significant—more than a factor 4 from plain code to optimized unrolled code.

Table 10.2. *Relative performance without and with unrolled loops.*

	Without unrolled loop	With unrolled loop
opt. 0	1.0	1.6
opt. 2	3.0	4.4

This ansatz is well known and implemented in today's packages, such as the BLAS [285, 124, 122] and Atlas [482]. For higher operations the differences are even more significant. Consider, for example, matrix multiplication. A standard loop for multiplying two $n \times n$ matrices A, B is given in the program in Figure 10.17, where the elements of C are assumed to be initialized with zero.

Again, we use MATLAB notation to show the point. It is well known and often exploited that the loops can be interchanged. Depending on the sequence of execution of the i-, j-, and k-loops, the access to matrix elements is rowwise or columnwise in the inner loop, and the data in memory may or may not be available in cache or registers.

```
for i=1:n
  for j=1:n
    for k=1:n
      C(i,j) = C(i,j) + A(i,k)*B(k,j);
    end
  end
end
```

Figure 10.17. *ijk-loop for matrix multiplication.*

Table 10.3 shows computing times in MFLOPS for the different possibilities, again for the GNU C-compiler. The data are for 500×500 matrices.

Table 10.3. *Relative performance for different methods for matrix multiplication.*

	jki	*kji*	*ikj*	*kij*	*ijk*	*jik*
opt. 0	1.0	1.0	1.4	1.4	1.4	1.4
opt. 2	2.8	2.6	12.9	11.9	29.5	29.5
BLAS 3			47.6			

The BLAS3 [122] routines use, in addition, blocked routines again optimizing memory access. So from worst to best there is a factor of almost 50 in computing time.

So far all code was rather simple. Next, we will demonstrate the disastrous effects of branches in professionally written codes. Consider first matrix multiplication of two real $n \times n$ matrices, second LU decomposition of a real $n \times n$ matrix with partial pivoting, and third LU decomposition with complete pivoting. Counting operations $+$, $-$, $\cdot$, $/$ and if-statements as one flop, these routines require $2n^3 + \mathcal{O}(n^2)$, $\frac{2}{3}n^3 + \mathcal{O}(n^2)$, and $\frac{4}{3}n^3 + \mathcal{O}(n^2)$ operations, respectively. In fact, each of the mentioned operations is executed in one cycle; the if-statement, however, implies great consequences with respect to optimization of the code.

The three tasks are accomplished by the LAPACK routines DGEMM, DGETRF and DGETC2, respectively.[42] The first test is on a 933 MHz Mobile Pentium 3 CPU and executed using MKL, the Intel Math Kernel Library. Table 10.4 shows the performance in MFLOPS of the matrix multiplication routine DGEMM, of LU decomposition with partial pivoting DGETRF, and of LU decomposition with complete pivoting DGETC2, for different dimensions n.

Table 10.4 shows that matrix multiplication is executed at close to the peak performance, that matrix multiplication executes about 10% faster than Gaussian elimination with partial pivoting, and that complete pivoting decreases the performance by about a factor 10.

We executed the same example on a 2.53 GHz Pentium 4 CPU using the ATLAS routines. The results are shown in Table 10.5.

Here matrix multiplication is beyond peak performance due to multiply-and-add instructions, and is faster by 60% to 20% compared with Gaussian elimination with partial

[42] Thanks to Dr. Ogita from Waseda University, Tokyo, for performing the following computations.

Table 10.4. *Performance of LAPACK routines using Intel MKL.*

	Performance [MFLOPS]		
n	DGEMM	DGETRF	DGETC2
500	708	656	73
1000	746	672	68
2000	757	688	66
3000	757	707	64

Table 10.5. *Performance of ATLAS routines.*

	Performance [MFLOPS]		
n	DGEMM	DGETRF	DGETC2
500	2778	1725	215
1000	2970	2121	186
2000	3232	2525	151
3000	3249	2663	101

pivoting. The seeming decrease of relative performance of DGEMM is rather a more rapid increase of performance of DGETRF. This makes it even worse for Gaussian elimination with complete pivoting: In this case if-statements slow computation up to *a factor 30*.

For interval operations things are even worse. Consider an implementation of Theorem 10.1 for the inclusion of the set of solutions

$$\sum(\mathbf{A}, \mathbf{b}) := \{x \in \mathbb{R}^n : \exists\, A \in \mathbf{A}\ \exists b \in \mathbf{b} \quad \text{with } Ax = b\}$$

of a linear system with interval data $\mathbf{A} \in \mathbb{IR}^{n\times n}$ and $\mathbf{b} \in \mathbb{IR}^n$. The computation of the product $\mathbf{C} = R\mathbf{A}$, a point matrix times an interval matrix, takes the major part of the computing time. A standard loop is given in Figure 10.18.

```
for i=1:n
   for j=1:n
      for k=1:n
         C(i,j) = C(i,j) + R(i,k)*A(k,j);
      end
   end
end
```

Figure 10.18. *Top-down approach for point matrix times interval matrix.*

For better readability we typed the interval quantities in boldface. Note that both the addition and the multiplication in the inner loop are interval operations.

The multiplication of a point matrix and an interval matrix is implemented this way in many libraries. However, the inner loop is very costly in terms of computing time. Even in an optimal implementation it requires two changes of the rounding mode, and at least one if-statement concerning the sign of `R(i,k)`. The if-statement, however, frequently empties the computation pipe and the contents of registers become unknown, so it jeopardizes any attempt to optimize this code.

In 1993, Knüppel found a simple way to improve this computation; it is published as BIAS [263, 264]. He interchanged the j- and the k-loops such that the element in the `R(i,k)` inner loop remains constant. Then the sign can be determined in the j-loop reducing the number of changes of rounding mode and the number of if-statements from n^3 to n^2. The BIAS code in Figure 10.19 illustrates his idea.

```
for i=1:n
   for k=1:n
      if R(i,k)>=0
         setround(-1)
         for j=1:n
            Cinf(i,j) = Cinf(i,j) + R(i,k)*Ainf(k,j);
         end
         setround(+1)
         for j=1:n
            Csup(i,j) = Csup(i,j) + R(i,k)*Asup(k,j);
         end
      else
         setround(-1)
         for j=1:n
            Cinf(i,j) = Cinf(i,j) + R(i,k)*Asup(k,j);
         end
         setround(+1)
         for j=1:n
            Csup(i,j) = Csup(i,j) + R(i,k)*Ainf(k,j);
         end
      end
   end
end
```

Figure 10.19. *Improved code for point matrix times interval matrix.*

Note that in the inner loops there is no changing of rounding mode and no if-statement. Computing times on a Convex SPP 200 for different matrix dimensions are given in Table 10.6. The decreasing performance for higher dimensions is due to cache misses since the BIAS routine was implemented as above, i.e., not blocked.

For the comparison of different interval libraries, Corliss [102] developed a test suite for basic arithmetic operations, vector and matrix operations, nonlinear problems, and others. Comparing BIAS (which is implemented in the C++ library PROFIL [265]) with other

Table 10.6. *Performance of algorithms in Figures 10.18 and 10.19 for point matrix times interval matrix.*

[MFLOPS]	$n = 100$	$n = 200$	$n = 500$	$n = 1000$
Traditional	6.4	6.4	3.5	3.5
BIAS	51	49	19	19

libraries shows a performance gain of a factor 2 to 30, depending on the application. This is basically due to better optimizable code.

The approach has some drawbacks. It does not carry over to the multiplication of two interval matrices, and it cannot be used for a MATLAB implementation. This is because the interpretation overhead would be tremendous. Table 10.7 shows the performance for a matrix product A*B implemented in MATLAB with the traditional 3 loops, with using a scalar product in the most inner loop, with outer products, and finally with the MATLAB command $A * B$ (using BLAS3 routines).

Table 10.7. *Relative performance of* MATLAB *matrix multiplication with interpretation overhead.*

Loops	3	2	1	$A * B$
MFLOPS	0.05	1.9	36	44

So the BIAS implementation would, besides other overheads, still be slower by more than an order of magnitude, only because of the interpretation overhead due to the use of loops.

Fortunately we developed another method for overcoming those problems. It uses solely BLAS3 routines, needs no branches, is applicable to the multiplication of two interval matrices, and is parallelizable by using parallel BLAS. The key is the midpoint-radius representation of intervals; see (10.5). An interval matrix $\mathbf{A} \in \mathbb{IR}^{n\times n}$ may be represented by

$$\mathbf{A} = [\underline{A}, \overline{A}] \quad \text{or} \quad \mathbf{A} = mA \pm rA \quad \text{with} \quad \underline{A}, \overline{A}, mA, rA \in \mathbb{R}^{n\times n}.$$

Here the $\pm$ is understood to be independent for each individual component. The radius matrix rA is nonnegative. In any case

$$\mathbf{A} = \{A \in \mathbb{R}^{n\times n} : \ \underline{A} \le A \le \overline{A}\} = \{A \in \mathbb{R}^{n\times n} : \ mA - rA \le A \le mA + rA\},$$

with comparison understood to be componentwise. Then, it is not difficult to see (cf. (10.6)) that

$$R \cdot \mathbf{A} = R \cdot mA \pm |R| \cdot rA = \{C \in R^{n\times n} : \ R \cdot mA - |R| \cdot rA \le C \le R \cdot mA + |R| \cdot rA\},$$

with the absolute value understood to be componentwise. The result is the best possible in the sense that none of the bounds on the result can be improved without violating inclusion isotonicity. This result, however, doesn't take rounding errors into account. In the world

of floating-point operations we face two additional problems. First, the input matrix $\mathbf{A} = [\underline{A}, \overline{A}]$ has to be transformed into midpoint-radius form $mA \pm rA$, with floating-point matrices mA and rA such that $[\underline{A}, \overline{A}] \subseteq mA \pm rA$, and, second, the product $R \cdot mA$ is, in general, not exactly representable in floating point.

The first problem can be solved by an ingenious trick due to Oishi [353]. Consider the INTLAB code in Figure 10.20.

```
setround(+1)
mA = (Ainf+Asup)/2;
rA = mA-Ainf;
```

Figure 10.20. *INTLAB program to convert inf-sup to mid-rad representation.*

Given $\mathbf{A} = [\texttt{Ainf}, \texttt{Asup}]$, this code computes `mA` and `rA` such that

$$\mathbf{A} = [\texttt{Ainf}, \texttt{Asup}] \subseteq \{A \in \mathbb{R}^{n \times n} : \texttt{mA} - \texttt{rA} \leq A \leq \texttt{mA} + \texttt{rA}\}.$$

The second problem can be solved by computing the product $R * mA$ both in rounding downward and rounding upward. So the entire INTLAB code for point matrix times interval matrix is as in Figure 10.21.

```
setround(+1)
mA = (Ainf+Asup)/2;
rA = mA-Ainf;
rC = abs(R)*rA;
Csup = R*mA + rC;
setround(-1)
Cinf = R*mA - rC;
```

Figure 10.21. *INTLAB program for point matrix times interval matrix.*

This calculates an interval matrix $\mathbf{C} = [\texttt{Cinf}, \texttt{Csup}]$ with inclusion isotonicity

$$RA \in \mathbf{C} \quad \text{for all} \quad A \in \mathbf{A}.$$

The computational effort is basically three floating-point matrix multiplications, and those can be executed in BLAS or Atlas. The interpretation overhead is negligible. So the factor in computing time compared to one floating-point multiplication of matrices of the same size is almost exactly three, and that is what is observed in MATLAB/INTLAB. Note that the result is an interval matrix, so we cannot expect a factor less than two. Therefore, the result is almost optimal. Again, note that we are talking about the *actual computing time*, not about an operation count.

The concept is easily applied to the multiplication of two interval matrices. The program in Figure 10.22 is executable INTLAB code for $A * B$.

Note that only two switches of the rounding mode are necessary, and that in total only four matrix multiplications are needed. Only computing the mutual products of the bounds of A and B would require the same computing time—although this would not help much for the computation of $A * B$.

```
setround(+1)
mA = (Ainf+Asup)/2;
rA = mA-Ainf;
mB = (Binf+Bsup)/2;
rB = mB-Binf;
rC = abs(mA)*rB + rA*(abs(mB)+rB);
Csup = mA*mB + rC;
setround(-1)
Cinf = mA*mB - rC;
```

Figure 10.22. *INTLAB program for interval matrix times interval matrix.*

In MATLAB/INTLAB we observe exactly this factor four. In a compiled language, in this case C, the results on the Convex SPP 200 for the multiplication of two interval matrices are given in Table 10.8. Here, we can also use the parallel processors just by linking the parallel BLAS routines to our MidRad approach. The other approaches might be parallelized, but this would require a lot of programming. Note that there is almost no difference between the traditional and the BIAS approaches. The lower performance of the MidRad approach for $n = 500$ may be due to suboptimal utilization of the cache.

Table 10.8. *Performance of interval matrix times interval matrix.*

[MFLOPS]	$n = 100$	$n = 200$	$n = 500$	$n = 1000$
Traditional	4.7	4.6	2.8	2.8
BIAS	4.6	4.5	2.9	2.8
MidRad	91	94	76	99
MidRad 4 proc.	95	145	269	334

The implementation of complex interval arithmetic follows the same lines. Note that midpoint-radius arithmetic causes some overestimation compared to the infimum-supremum arithmetic. An example is

$$[3.14, 3.15] * [2.71, 2.72] = [8.5094, 8.5680],$$
$$(3.145 \pm 0.005) * (2.715 \pm 0.005) = 8.538675 \pm 0.029325 = [8.50935, 8.56800].$$

However, one can show that the overestimation is limited to a factor 1.5 in the worst case, and it is in general much smaller, as in the example above, depending on the relative accuracy of the input intervals. This is true for scalar, vector, and matrix operations, real and complex, and independent of the size of the numbers. Those results can be found with various other details in [387].

Sometimes it may be useful to calculate inner products with higher precision (and then to round the result to working precision). For example, in the residual iteration

$$x^{k+1} = x^k - A\backslash(Ax^k - b),$$

the residual $Ax^k - b$ is exposed to severe cancellation (we use MATLAB notation $A\backslash y$ to indicate the solution of a linear systems with matrix A and right-hand side y). It has

been shown by Skeel [418] that a residual iteration with the residual calculated in *working precision* suffices to produce a *backward stable* result. However, for a small *forward error*, higher precision accumulation of dot products is necessary. This may be interesting in case of exactly given and exactly representable input data; otherwise the input error is amplified by the condition number and a highly accurate forward error does not make much sense. There are various algorithms available [40, 275, 289] as well as a discussion of hardware implementations [274]. A practical implementation by means of a coprocessor board is presented in [441, 442].

Recently, new and very fast summation and dot product routines have been developed in [351]. They use only double precision floating point addition, subtraction, and multiplication. Nevertheless they compute a result of an accuracy *as if* computed in quadruple or even higher precision. The reason is the extensive use of so-called error-free transformations. The algorithms do not contain a single branch so that the compiled code can be highly optimized. Therefore they are not only fast in terms of flop count but, more importantly, in terms of *measured computing time*. Apparently, these are the fastest dot product algorithms currently available, some 40 % faster than the corresponding XBLAS routines [289], while sharing similar error estimates.

Other design and implementation details of INTLAB can be found in [389]. For a tutorial and applications see [196]. INTLAB is comprised of

- real and complex interval scalars, vectors, and matrices,
- dense and sparse interval matrices,
- real and complex (interval) elementary functions,
- automatic differentiation (gradients and Hessians),
- automatic slopes,
- univariate and multivariate (interval) polynomials,
- a rudimentary (interval) long arithmetic.

Finally we mention that INTLAB is entirely written in MATLAB. All algorithms are designed to minimize interpretation overhead. Under Windows operating systems, they do not depend on assembly language routines since the switch of the rounding mode is provided by MATLAB (thanks to Cleve Moler). For other operating systems such a routine is provided, and this is the only non-MATLAB routine. For details see our home page [389].

10.10 Performance and Accuracy

Measuring computing time in MATLAB is a little unfair because of the interpretation overhead. However, with the fast implementation of interval arithmetic as described in the previous section there is, at least when solving linear systems, not much overhead. Notice that a pure flop count for a self-validating algorithm for linear systems along the lines of Theorem 10.1 needs $6n^3 + O(n^2)$ operations, with additions and multiplications counted

separately. These are $2n^3$ operations to calculate the approximate inverse R and $4n^3$ operations to calculate the lower and upper bound of the inclusion of RA. So we may expect a factor 9 compared to the $\frac{2}{3}n^3 + O(n^2)$ floating-point operations for Gaussian elimination. Applying Theorem 10.2 to linear systems yields the computing times in Table 10.9, measured in INTLAB.

Table 10.9. *Measured computing time for linear system solver.*

INTLAB computing time ($n = 500$, 300 MHz Laptop)	
5 sec	for $A\backslash b$ (built-in solver)
27 sec	with verification as in Theorem 10.2
16 sec	verification by Oishi's method [353]

The first line in Table 10.9 refers to the built-in linear system solver in MATLAB 6.5, about the fastest we can get. The second line is the total computing time for the verification routine `verifylss` available in INTLAB, which is based on the methods described above. The last line is an improvement described in [353].

So the measured computing times of our self-validating methods are significantly better than can be expected. This is because the $2n^3$ floating-point operations for matrix multiplication do not take 6 times the time for $\frac{2}{3}n^3$ floating-point operations of Gaussian elimination due to better optimized code.

There is no question that one has to pay for an inclusion. However, the final result is verified to be correct, under any circumstances. If the matrix of the linear system is too ill-conditioned to be solved in the precision available, a corresponding error message is given rather than a false result.

It may happen that pure floating-point routines deliver false answers, i.e., severely wrong approximations are computed *without a corresponding error message*. A first, though not really fair, example is the following. Foster [146] describes a linear system arising from the integration of

$$\dot{x} = x - 1 \quad \text{with} \quad x(0) = x(T). \tag{10.12}$$

His example is in fact a little more general. He uses the trapezoidal rule to obtain a linear system, which he solves by MATLAB. This is an example derived from the work of [493] to show that there are other than constructed examples where partial pivoting may result in exponential growth of the pivot elements. Foster's paper contains the MATLAB code for the resulting linear system $Ax = b$. For $n = 70$ and $T = 40$, for example, the MATLAB solver $A\backslash b$ delivers the results in Table 10.10, where only the first five and last ten components of x are displayed.

But the true solution not only of the defining equation (10.12) and also of the generated linear system is $x_i \equiv 1$ for all components! This "approximation" is given *without any warning or error message*. This is *not* due to ill-conditioning of the system matrix but solely due to the exponential increase of the growth factor for partial pivoting. In fact, the condition number of the matrix in this example is about 30. The self-validating method `verifylss` of INTLAB computes an inclusion of 1 with relative error less than $2.2 \cdot 10^{-16}$ in all components.

Table 10.10. MATLAB *solution of a linear system derived from (10.12) for* $n = 70$ *and* $T = 40$.

```
>> x([1:5 61:70])
ans =
   1.00000000000000
   1.00000000000000
   1.00000000000000
   1.00000000000000
   1.00000000000000
          ...
   1.40816326530612
   2.81632653061224
   4.22448979591837
   5.63265306122449
  11.26530612244898
  11.26530612244898
  22.53061224489796
                  0
                  0
   1.00000000000000
```

As mentioned before the comparison is not really fair. This is because obviously the MATLAB routine does not perform a single residual iteration. As has been shown by Skeel [418] this would yield a backward stable result, even if the residual iteration is performed in working precision. The verification does perform iteration steps before starting the inclusion step. In that sense the comparison is not quite fair. Nevertheless the result computed by MATLAB is grossly wrong and delivered without any warning.

When solving systems of nonlinear equations it is much easier to find examples where the computed approximation is afflicted with severe errors but is delivered without an error message. We will return to such examples later.

10.11 Uncertain Parameters

Next we discuss nonlinear systems with uncertain parameters. This is at the heart of interval extensions and self-validating methods. Consider a function

$$f : \mathbb{R}^{k\times n} \to \mathbb{R}^n, \qquad f(\widetilde{p}, \widetilde{x}) \approx 0,$$

and assume the parameter p varies within a certain interval P. We wish to calculate an inclusion X such that *for all* $p \in P$ there exists some $x \in X$ with $f(p, x) = 0$. This is surprisingly simple by the following theorem.

Suppose an interval extension $F(P, X)$, $F : \mathbb{IR}^{k\times n} \to \mathbb{IR}^n$ of f is given such that for $P \in \mathbb{IR}^k$, $X \in \mathbb{IR}^n$ and for all $p \in P, x \in X$ it holds that $f(p, x) \in F(P, X)$. Furthermore, define

$$L_F(P, \widetilde{x}, X) := -R \cdot F(P, \widetilde{x}) + \{I - RM\}X,$$

where M is computed by automatic differentiation applied to F at P and X. Then the following is true [386].

Theorem 10.2. *Let $f : \mathbb{R}^{k\times n} \to \mathbb{R}^n \in C^1$, $P \in \mathbb{IR}^n$, and $\widetilde{x} \in \mathbb{R}^n$ be given. Suppose $\widetilde{x} \in X$ and $L_F(P, \widetilde{x}, X) \subseteq \mathrm{int}(X)$. Then for all $\widehat{p} \in P$ there exists one and only one $\widehat{x} = \widehat{x}(\widehat{p}) \in \widetilde{x} + X$ with $f(\widehat{p}, \widehat{x}) = 0$.*

The proof is trivial: just consider *some fixed but arbitrary* $p \in P$, and apply Theorem 10.1 to $f(p, \widetilde{x} + x)$. Using this approach for given values of a parameter also gives a clue to the sensitivity of the zeros with respect to this parameter. Replacing, for example, the constant `pi` in Broyden's function by

```
>> Pi = midrad( 3.141592653589793, 1e-15 )
```

yields the result in Table 10.11.

Table 10.11. *INTLAB example of* `verifynlss` *for Broyden's function (10.9) with uncertain parameters.*

```
>> xs = [ .6 ; 3 ];  verifynlss(f,xs)
intval ans =
[ 0.49999999999740, 0.50000000000260]
[ 3.14159265357934, 3.14159265360025]

>> xs = [ 0 ; 0 ];  y = verifynlss(f,xs)
intval ans =
[ -0.26059929002903, -0.26059929001592]
[  0.62253089659741,  0.62253089663041]
```

Obviously, both zeros do not react very sensitively to changes in that parameter. Note that we used a formulation of the inclusion theorem, where the interval X includes the difference of the approximation $\widetilde{x}$ rather than the solution itself. This turns out [387] to produce inclusions of better quality than the original operator K_f in Theorem 10.1.

So far the application has been restricted to continuously differentiable functions. This allows us to use automatic differentiation and also ensures the uniqueness of the computed zero within the computed bounds. It also implies that it is impossible to calculate an inclusion of a multiple zero. For a real root of even multiplicity of a univariate function this is clear because an arbitrarily small perturbation can move the roots into the complex plane. But also for odd multiplicities greater than one this approach does not allow us to calculate an inclusion. We mention that inclusion of multiple zeros of continuous but not necessarily differentiable functions is possible using slopes (see [269] and improvements described in [388]).

We return to parameterized problems with data varying within some tolerances. The aim is to investigate whether the amount of overestimation can be estimated. Consider the

following simple model problem.

$$A = \begin{pmatrix} [-0.5796, -0.5771] & [\ 0.2469,\ \ 0.2581] \\ [\ 0.2469,\ 0.2581] & [-0.4370, -0.4365] \end{pmatrix}, \qquad b = \begin{pmatrix} 0.5731 \\ -0.4910 \end{pmatrix}. \tag{10.13}$$

An inclusion of the set of all solutions

$$\sum(A, b) := \{x \in \mathbb{R}^2 :\ \widetilde{A}x = \widetilde{b},\ \widetilde{A} \in A, \widetilde{b} \in b\} \tag{10.14}$$

may be computed by Theorem 10.3, much along the lines of the previous discussion.

Theorem 10.3. *Let $A \in \mathbb{I}M_n(\mathbb{R})$, $b \in \mathbb{IR}^n$, $R \in M_n(\mathbb{R})$, $X \in \mathbb{IR}^n$ be given and suppose*

$$R(b - A\widetilde{x}) + (I - RA)X \subseteq \mathrm{int}(X).$$

Then for all $\widetilde{A} \in A$ and for all $\widetilde{b} \in b$, $\widetilde{A}$ and R are nonsingular and $\widetilde{A}^{-1}\widetilde{b} \in \widetilde{x} + X$.

Theorem 10.3 is a direct consequence of Theorem 10.1 applied to $Ax = A\widetilde{x} - b$. The definition and solution of our model problem in INTLAB is Table 10.12. A plot of the inclusion interval `X` is given in Figure 10.23.

Table 10.12. *INTLAB example of* `verifylss` *for a linear system with uncertain data.*

```
>> A = infsup( [-0.5796 0.2469 ; 0.2469 -0.4370 ] , ...
               [-0.5771 0.2581 ; 0.2581 -0.4365 ] );
          b =  [ 0.5731 ; -0.4910 ];
>> X = verifylss(A,b)
 intval X =
[   -0.6862,   -0.6517]
[    0.7182,    0.7567]
```

First remember that this result includes the proof that all matrices $\widetilde{A} \in A$ are nonsingular, and this is, in general, an NP-hard problem. But, we may ask how much the interval X overestimates the true solution set $\sum(A, b)$. The latter is known to be convex in every orthant.[43] Indeed a so-called *inner* inclusion can be calculated by Theorem 10.4 [388], which is based on ideas developed in [339].

Theorem 10.4. *Let $A \in \mathbb{IR}^{n\times n}$, $b \in \mathbb{IR}^n$, $\widetilde{x} \in \mathbb{R}^n$, $R \in \mathbb{R}^{n\times n}$, $X \in \mathbb{IR}^n$ be given and define*

$$Z := R(b - A\widetilde{x}) \quad \textit{and} \quad \Delta := \{I - RA\} \cdot X.$$

[43] Orthant is the generalization of quadrant (2D) and octant (3D) to higher dimensions.

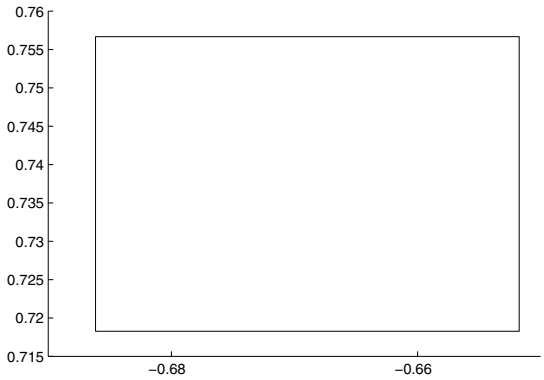

Figure 10.23. *Inclusion as computed by* `verifylss` *for the example in Table 10.12.*

Let the solution set $\sum(A, b)$ *be defined as in (10.14) and assume*

$$Z + \Delta \subseteq \mathrm{int}(X).$$

Then

$$\sum(A, b) \subseteq \widetilde{x} + Z + \Delta,$$

or, in coordinate notation, for all $x \in \sum(A, b)$ *and all* $i \in \{1, \ldots, n\}$,

$$\widetilde{x}_i + \inf Z_i + \inf \Delta_i \leq x_i \leq \widetilde{x}_i + \sup Z_i + \sup \Delta_i.$$

Furthermore, for all $i \in \{1, \ldots, n\}$ *there exist* $\underline{x}, \overline{x} \in \sum(A, b)$, *with*

$$\underline{x}_i \leq \widetilde{x}_i + \inf Z_i + \sup \Delta_i \quad \textit{and} \quad \widetilde{x}_i + \sup Z_i + \inf \Delta_i \leq \overline{x}_i.$$

Theorem 10.4 estimates every component of the solution set from the outside *and the inside*. For our model problem, the inner rectangle has the property that the projection to every coordinate is an inner inclusion of the corresponding projection of the true solution set $\sum(A, b)$. The outer and inner inclusions together with the true solution set (the parallelogram) are displayed in Figure 10.24.

Of course, this approach has its limits. When widening the interval components of the linear system, the inner inclusion becomes smaller and smaller, and finally vanishes (which means nothing can be said about the quality of the outer inclusion), and, when further widening the input intervals, the inclusion will fail at a certain point. This happens, in general, *before* a singular matrix enters the input intervals. This is to be expected because the self-validating algorithm for an $n \times n$ linear system requires some $O(n^3)$ operations, whereas, as has been mentioned before, the problem of determining whether an interval matrix contains a singular matrix is NP-hard [365].

The model problem above looks pretty simple. However, a nontrivial problem has been solved: the determination of the maximum variation of the individual components of

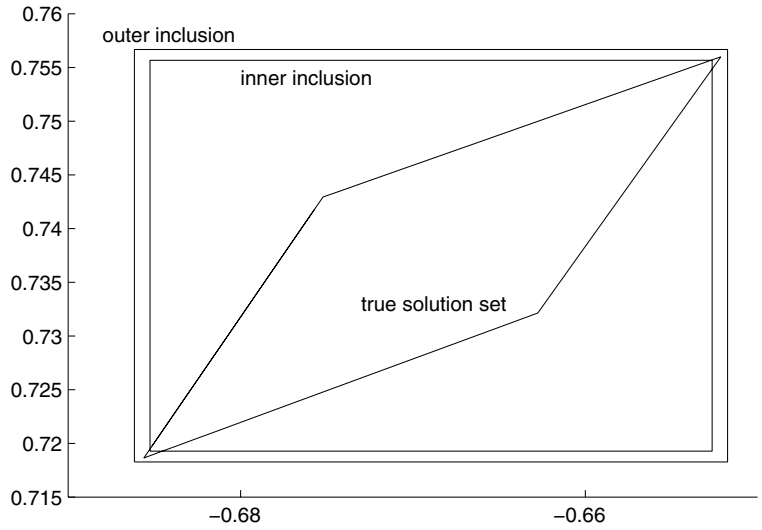

Figure 10.24. *Inner and outer inclusions and true solution set for the linear system with tolerances in Table 10.12.*

a linear system when varying the input data within a certain range, and the computation of an upper bound *and a lower bound* for this variation. Using traditional numerical methods such as Monte Carlo to determine similar information may be difficult.

In order to demonstrate this, consider an interval matrix A and interval right-hand side b, for which we wish to estimate $\sum(A, b)$. A Monte Carlo approach may be like in the program of Figure 10.25.

```
for   i = 1 : K
      choose Ã ∈ A
      choose b̃ ∈ b
      compute x̃ = Ã\b̃
      x_min = min(x_min, x̃)
      x_max = max(x_max, x̃)
end
```

Figure 10.25. *Monte Carlo approach to estimate the solution set of a parameterized linear system.*

Consider a linear system with randomly generated matrix $\widetilde{A}$ and right-hand side $\widetilde{b}$, both with entries within $[-1, 1]$. Then an interval matrix A and interval right-hand side b are defined by perturbing each component of $\widetilde{A}$ and $\widetilde{b}$ with a relative error 10^{-4}, respectively. The Monte Carlo approach is improved by choosing $\widetilde{A}$ and $\widetilde{b}$ only on the boundary of A and b, respectively, in order to achieve a best possible (i.e., as wide as possible) result. For

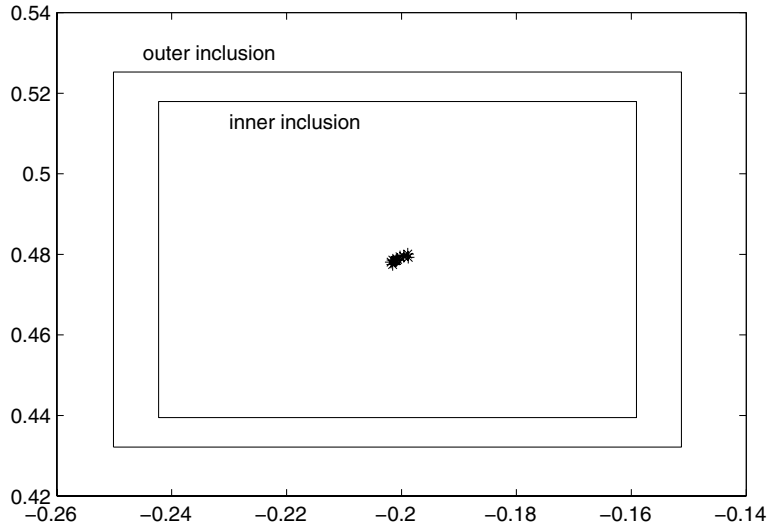

Figure 10.26. *Result of Monte Carlo approach as in the program in Figure 10.25 for* 100×100 *random linear system with tolerances, projection of 1st and 2nd component of solution.*

$K = 100$ samples the projection of the first and second components of the result is shown in Figure 10.26.

Figure 10.26 shows the variation achieved by the Monte Carlo method (the black cloud of asterisks) and the outer and inner inclusions computed by the self-validating method based on Theorem 10.4. The computing times on our 300 MHz laptop are as follows.

Self-validating method	0.39 sec,
Monte Carlo method	11.9 sec.

Note that 100 samples have been used for a 100×100 linear system. These results are typical. Although the computed inclusions are almost sharp, they are usually achieved only for a very few choices of input data.

This good news is shadowed by the bad news that the self-validating method assumes the input data vary *independently* within the tolerances. This assumption is frequently not fulfilled. For example, the restriction of variation to symmetric matrices may shrink the size of the solution set significantly. However, this and other linear dependencies between the input data can be handled by self-validating methods. The following shows a plot of example (10.13) with symmetric matrices; see [237, 387].

The plot in Figure 10.27 shows the previous unsymmetric solution set (big parallelogram) as well as the true symmetric solution set (in bold). The latter can be calculated exactly using methods described in [10]. Moreover, the outer and inner inclusions for the symmetric solution set are displayed. In the same way, Toeplitz, Hankel, tridiagonal symmetric, and other linear structures can be handled [387]. However, the Monte Carlo

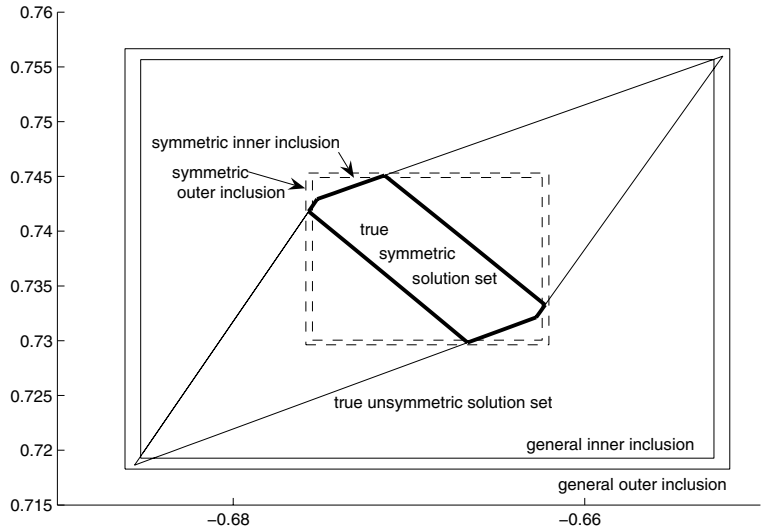

Figure 10.27. *Inner and outer inclusions of symmetric solution set for the example in Table 10.12.*

approach easily allows almost arbitrary dependencies between the input data, for which no self-validating method is known.

We mention that there are self-validating algorithms for sparse systems of linear and nonlinear equations [387]. Of course, they do not use an approximate inverse as in Theorem 10.1 but are based on a lower bound of the smallest singular value of the system matrix or the Jacobian matrix, respectively. Figure 10.28 shows a matrix from the Harwell/Boeing test suite [128]. It has 3948 unknowns and comes from a static analysis in structural engineering of some offshore platform. Computing times for this example for the MATLAB built-in sparse linear system solver and INTLAB routine `verifylss` are given in Table 10.13.

Table 10.13. *Computing time without and with verification.*

	Time [sec]
MATLAB	2.5
`verifylss`	6.3

So verification is still slower by a factor of 3, which is partly due to interpretation overhead in MATLAB. However, verified linear system solvers are still based on factorizations. For matrices with large fill-in these methods are not applicable. In particular, practically no self-validating methods based on iterative approaches are known.

As stated earlier, a numerical method may deliver a poor approximate answer without warning. Problems may occur when using this information for further computations. Consider, for example, the matrix in Table 10.14.

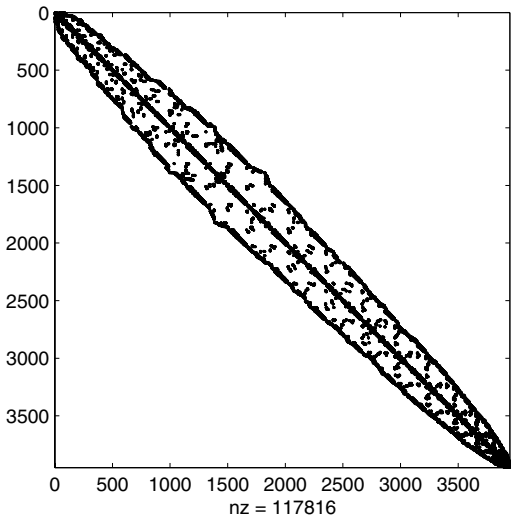

Figure 10.28. *Nonzero elements of the matrix from Harwell/Boeing BCSSTK15.*

Table 10.14. *Matrix with ill-conditioned eigenvalues.*

```
A =
   170   122   -52  -317  -247   265    86
   -38   -28    13    71    56   -59   -21
   -90   -64    27   167   130  -140   -46
    61    42   -16  -111   -85    94    31
    -7    -4     1    11     8   -10    -5
   -26   -19     9    49    39   -41   -14
   -51   -37    16    96    75   -80   -25
```

The question is "What are the eigenvalues of that matrix?" When entering this matrix into MATLAB, the approximations in Figure 10.29 depicted by the plus signs are computed without warning or error message. When computing the eigenvalues of A^T, the approximate "eigenvalues" depicted by the circles are computed, again without warning or error message. Notice the scale of the plot: the approximations of eigenvalues of A and A^T differ by about 0.005. The reason for the inaccuracy is that this matrix has a multiple (7-fold) eigenvalue zero of geometric multiplicity 1. This implies a sensitivity of $\sqrt[7]{\epsilon_M} \approx 0.006$ for changes of the input data of order ϵ_M, the machine precision $2^{-52} \approx 2.2 \cdot 10^{-16}$.

The circle is a verified inclusion of all eigenvalues of the matrix. It has been computed by the methods described in [390] for the inclusion of multiple eigenvalues and the corresponding invariant subspace. In view of the previous discussion the circle gives reasonable information about the sensitivity of the eigenvalues. It is a little pessimistic, but it is correct information. Taking only the approximations (depicted by the plus signs or the circles) one

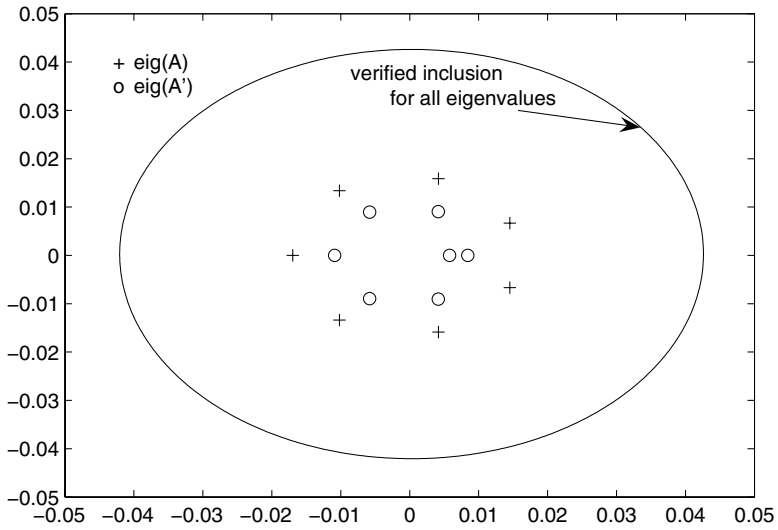

Figure 10.29. *Approximation of eigenvalues of the matrix in Table 10.14 computed by* MATLAB.

might be led to the conclusion that these are accurate approximations to the eigenvalues of the matrix—neglecting their sensitivity.

Another interesting point is *testing*. In our experience, testing self-validating algorithms is different from testing traditional numerical algorithms. Consider an algorithm for solving systems of linear equations. A standard test suite would randomly generate some matrices of different condition numbers. Then, for given solution vector $\widehat{x}$, one would calculate $b := A\widehat{x}$ and test for the difference of the computed approximation $\widetilde{x}$ and $\widehat{x}$.

However, in general the product $A\widehat{x}$ would be perturbed by rounding errors, such that $\widehat{x}$ would *not* be the solution of the linear system $Ax = b$. Accordingly, the test suite could only check for

$$\|\widehat{x} - \widetilde{x}\|/\|\widetilde{x}\| \leq \varphi \cdot \text{eps} \cdot \text{cond}(A) \tag{10.15}$$

with a moderate factor φ. Such a test suite would not be suitable for testing a self-validating algorithm because the solution of the given linear system $Ax = b$ is not known. In contrast, one would make sure that the product $A\widehat{x} = b$ is exactly representable in floating point (and is computed exactly). One way to achieve this is to round A (and $\widehat{x}$) such that only a few (leading) digits of all entries are nonzero. Then, for a computed inclusion X, one can test for $\widetilde{x} \in X$—a more stringent test than (10.15).

Another, even more stringent, test for robustness and accuracy is the following. Suppose we have a model problem, and the exact solution is π. Moreover, suppose a self-validating method produces the result

$$[3.141592653589792, \quad 3.141592653589793].$$

Each of the bounds would be an *approximation* of superb quality, and any numerical algorithm delivering one of the two bounds as final result would pass any test with flying colors.

But a self-validating algorithm delivering these bounds *would fail*, because the transcendental number π is not enclosed in the above interval! Such tests are powerful for detecting erroneous implementations of self-validating methods.

10.12 Conclusion

We have tried to show that self-validating methods may be used to compute true and verified bounds for the solution of certain problems. There would be much more to say to that, and many more examples could be given. Because of the limited space we could in particular show only some small examples and applications. But self-validating methods have been applied to a variety of larger problems with tens of thousands of unknowns, particularly in connection with computer-assisted proofs.

Examples of those include

- verification of the existence of the Lorenz attractor [455],
- the verification of the existence of chaos [343],
- the double-bubble conjecture [199],
- verification of the instability for the Orr–Sommerfeld equations with a Blasius profile [278],
- dynamics of the Jouanolou foliation [63],
- solution of the heat convection problem [335],
- verified bounds for the Feigenbaum constant [133],
- existence of an eigenvalue below the essential spectrum of the Sturm–Liouville problem [56],
- eigenfrequencies of a turbine (Kulisch et al., unpublished),
- SPICE program for circuit analysis (Rump, unpublished),
- extreme currents in Lake Constance (Rump, unpublished),
- forest planning [236].

A more detailed description of some of these can be found in [155]. Also, self-validating methods have been designed for various other areas of numerical mathematics. They include

- global optimization [52, 86, 171, 179, 182, 238, 239, 240, 241, 265, 320, 341, 374, 375, 461],
- *all* the zeros of a nonlinear system in a box [182, 265, 375, 396, 461, 498],
- least squares problems [268, 302, 386],

- sparse linear and nonlinear problems [387],
- ordinary differential equation initial value and boundary value problems [294, 337],
- partial differential equations [25, 334, 335, 362, 363, 364].

Most recently, efforts were started to design self-validating methods which require in total *the same* computing time as a traditional numerical algorithm; see [353].

We started with the question "What is a proof?" As mentioned earlier, we did not intend to (and cannot) give an answer to that. However, we have shown why using self-validating methods, executed in floating-point arithmetic, can be considered as a serious way to ascertain the validity of mathematical assertions. Self-validating methods are by no means intended to replace a mathematical proof, but they may assist. A lot has been done, but there is much more to do; see, for example, [342].

As a final remark we want to stress that self-validating methods are also not designed to replace traditional numerical methods. This fact is quite obvious as (i) most self-validating methods rely on a good numerical approximation to start a verification, and (ii) in direct comparison an interval computation must include the floating-point result. This applies to the direct solution of numerical problems such as systems of nonlinear equations or partial differential equations. However, self-validating methods do more: They verify existence and possibly uniqueness of a solution within computed bounds. This is outside the scope of traditional numerical algorithms.

A different regime is global optimization methods, where sophisticated inclusions of the range of functions may be used to discard certain boxes by proving that they cannot contain a global minimum. This is a very promising area with recent and quite interesting results.

The MATLAB toolbox INTLAB is a simple way to get acquainted with self-validating methods and to solve problems. It is freely available from our homepage for noncommercial use. Every routine comes with a header explaining input, output, and behavior of the routines. To date we have an estimated number of 3500 users in more than 40 countries.

In conclusion, we want to say that self-validating methods are an option; they are a possibility to verify the validity of a result, when necessary. In that sense they deserve their space in the realm of computational mathematics.