

## INTLAB - INTERVAL LABORATORY

SIEGFRIED M. RUMP\*

**Abstract.** INTLAB is a toolbox for Matlab supporting real and complex intervals, and vectors, full matrices and sparse matrices over those. It is designed to be very fast. In fact, it is not much slower than the fastest pure floating point algorithms using the fastest compilers available (the latter, of course, without verification of the result). Beside the basic arithmetical operations, rigorous input and output, rigorous standard functions, gradients, slopes and multiple precision arithmetic is included in INTLAB. Portability is assured by implementing *all* algorithms in Matlab itself with exception of exactly one routine for switching the rounding downwards, upwards and to nearest. Timing comparisons show that the used concept achieves the anticipated speed with identical code on a variety of computers, ranging from PC's to parallel computers. INTLAB is freeware and may be copied from our home page.

**1. Introduction.** The INTLAB concept splits into two parts. First, a new concept of a fast interval library is introduced. The main advantage (and difference to existing interval libraries) is that identical code can be used on a variety of computer architectures. Nevertheless, high speed is achieved for interval calculations. The key is extensive (and exclusive) use of BLAS routines [19], [7], [6].

Second, we aim at an interactive programming environment for easy use of interval operations. Our choice is to use Matlab. It allows to write verification algorithms in a way which is very near to pseudo-code used in scientific publications. The code is an *executable specification*. Here, the major difficulty is to overcome slowness of interpretation. This problem is also solved by exclusive use of high-order matrix operations.

The first concept, an interval library based on BLAS, is the basis of the INTLAB approach. The interval library based on BLAS is very simple to implement in any programming language. It is a fast way to start verification on many computers in a two-fold way: fast to implement and fast to execute.

There are a number of public domain and commercial interval libraries, among them [2], [5], [8], [9],[11], [12], [13], [14], [15], [18] and [27]. To our knowledge, INTLAB is the first interval library building upon BLAS.

The basic assumption for INTLAB is that the computer arithmetic satisfies the IEEE 754 arithmetic standard [10] and, that a permanent switch of the rounding mode is possible. Switching of the rounding mode is performed by `setround(i)` with  $i \in \{-1, 0, 1\}$  corresponding to rounding downwards, to nearest, and rounding upwards, respectively. We assume that, for example, after a call `setround(-1)` all subsequent arithmetical operations are using rounding downwards, until the next call of `setround`. This is true for almost all PC's, workstations and many mainframes.

Our goal is to design algorithms with result verification the execution time of which is of the same order of magnitude as the fastest known floating point algorithms using the fastest compiler available. To achieve this goal, it is not only necessary to design an appropriate arithmetic, but also to use appropriate verification algorithms being able to utilize the speed of the arithmetic and the computer. We demonstrate by means of examples the well known fact that a bare count of operations is not necessarily proportional to the actual computing time.

Arithmetical operations in INTLAB are rigorously verified to be correct, including input and output and standard functions. By that it is possible to replace every operation of a standard numerical algorithm by the corresponding interval operations. However, this frequently leads to dramatic overestimation and is not recommended. By the principle of definition, interval operations deliver a superset of the result, which is usually a slight overestimation. It is the goal of validated numerics to design verification algorithms in order

---

\*Inst. f. Informatik III, Technical University Hamburg-Harburg, Schwarzenbergstr. 95, 21071 Hamburg, Germany, (rump@tu-harburg.de).

to diminish this overestimation, and even to (rigorously) estimate the amount of overestimation. Therefore, for verified solution of numerical problems one should only use specifically designed verification algorithms. Some examples are given in Section 5.

Many of the presented ideas are known in one way or the other. However, the combination produces a very fruitful synergism. Development of validation algorithms is as easy as it could be using INTLAB.

The paper addresses the above mentioned two concepts and is organized as follows. In Section 2 the new interval library approach based on BLAS is presented together with timings for an implementation in C on scalar and parallel computers. Corresponding timings for the INTLAB implementation in the presence of interpretation overhead is presented in Section 3. The latter timings are especially compared with pure floating point. In Section 4 some details of programming in INTLAB are given and the question of correctness of programming is addressed. In the following Section 5 we present some algorithms written in INTLAB together with timing comparisons between verification algorithms and the corresponding pure floating point (Matlab) implementation. We finish the paper with concluding remarks.

**2. An interval library based on BLAS.** Starting in the late 70's an interface for Basic Linear Algebra Subroutines was defined in [19], [7], and [6]. It comprises linear algebra algorithms like scalar products, matrix-vector products, outer products, matrix multiplication and many more. The algorithms are collected in level 1, 2 and 3 BLAS.

The ingenious idea was to specify the interface, pass it to the manufacturers and leave the implementation to them. Since BLAS became a major part of various benchmarks, it is the manufacturers own interest to provide very fast implementations of BLAS for their specific computer. This includes all kinds of optimization techniques to speed up code such as blocked code, optimal use of cache, taking advantage of the specific architecture and much more.

In that way BLAS serves a mutual interest of the manufacturer and the user. No wonder, modern software packages like Lapack [4] extensively use BLAS. Beside speed, the major advantage is the fact that a code using BLAS is fast on a variety of computers, without change of the code.

This is the motivation to try to design an interval library exclusively using BLAS routines. At first sight this seems impossible because all known implementations use various case distinctions in the inner loops. In the following we will show how to do it. We assume the reader is familiar with basic definitions and facts of interval arithmetic (cf. [3], [21]).

In the following we discuss implementations in a programming language like C or Fortran; interpretation overhead, like in Matlab, is addressed in Section 3.

We start with the problem of matrix multiplication. Having fast algorithms for the multiplication of two matrices at hand will speed up many verification algorithms. We have to distinguish three cases. We first concentrate on real data, where a real interval  $A$  is stored by infimum  $A.\text{inf}$  and supremum  $A.\text{sup}$ . If  $A$  is a vector or matrix, so is  $A.\text{inf}$  and  $A.\text{sup}$ .

**2.1. Real Point matrix times real point matrix.** First, consider the multiplication of two point matrices (infimum and supremum coincide)  $A$  and  $B$ , both exactly representable within floating point. There is a straightforward solution to this by the following algorithm.

```
setround(-1)
C.inf = A * B;
setround(1)
C.sup = A * B;
```

ALGORITHM 2.1. *Point matrix times point matrix*

We use original INTLAB notation for two reasons. First, the sample code is *executable* code in INTLAB, and second it gives the reader a glimpse on readability of INTLAB notation (see the introduction for `setround`). Note that this is only *notation*, interpretation overhead is addressed in Section 3.

Note that Algorithm 2.1 assures that

$$(2.1) \quad \mathbf{C}.\text{inf} \leq \mathbf{A} * \mathbf{B} \leq \mathbf{C}.\text{sup}$$

including underflow and overflow (comparison of vectors and matrices is always to be understood entrywise); estimation (2.1) is valid under any circumstances. Moreover, an effective implementation of the floating point matrix multiplication  $\mathbf{A} * \mathbf{B}$ , like the one used in BLAS, will change the order of execution. This does not affect validity of (2.1).

**2.2. Real point matrix times real interval matrix.** Next, consider multiplication of a real point matrix  $\mathbf{A} \in \mathbf{M}_{m,n}(\mathbb{R})$  and a real interval matrix  $\text{intB} \in \mathbf{IM}_{n,k}(\mathbb{R})$ . The aim is to compute  $\mathbf{C} = \mathbf{A} * \text{intB}$  with bounds. Consider the following three algorithms to solve the problem.

```

C.inf = zeros(size(A,1),size(intB,2));
C.sup = C.inf;
setround(-1)
for i=1:size(A,2)
    C.inf = C.inf + min( A(:,i)*intB.inf(i,:) , A(:,i)*intB.sup(i,:) );
end
setround(1)
for i=1:size(A,2)
    C.sup = C.sup + max( A(:,i)*intB.inf(i,:) , A(:,i)*intB.sup(i,:) );
end

```

ALGORITHM 2.2. *Point matrix times interval matrix, first variant*

By Matlab convention, the minimum and maximum in rows 5 and 9 are entrywise, respectively, the result is a matrix. The above algorithm implements an outer product matrix multiplication.

```

Aneg = min( A , 0 );
Apos = max( A , 0 );
setround(-1)
C.inf = Apos * intB.inf + Aneg * intB.sup;
setround(1)
C.sup = Aneg * intB.inf + Apos * intB.sup;

```

ALGORITHM 2.3. *Point matrix times interval matrix, second variant*

This algorithm was proposed by A. Neumaier at the SCAN 98 conference in Budapest. Again, the minimum and maximum in lines 1 and 2 are entrywise, respectively, such that for example `Apos` is the matrix  $\mathbf{A}$  with negative entries replaced by zero. It is easy to see that both algorithms work correctly. The second algorithm is well suited for interpretation.

```

setround(1)
Bmid = intB.inf + 0.5*(intB.sup-intB.inf);
Brad = Bmid - intB.inf;
setround(-1)
C1 = A * Bmid;
setround(1)
C2 = A * Brad;
Cmid = C1 + 0.5*(C2-C1);

```

```

Crad = ( Cmid - C1 ) + abs(A) * Brad;
setround(-1)
C.inf = Cmid - Crad;
setround(1)
C.sup = Cmid + Crad;

```

ALGORITHM 2.4. *Point matrix times interval matrix, third variant*

This algorithm first converts the second factor into midpoint/radius representation and needs three point matrix multiplications. Recall that Algorithm 2.3 used four point matrix multiplications. This elegant way of converting infimum/supremum representation into midpoint/radius representation was proposed by S. Oishi during a visit of the author at Waseda university in fall 1998 [22].

**2.3. Comparison of results of Algorithms 2.3 and 2.4.** Without rounding errors, the results of all algorithms are identical because overestimation of midpoint/radius multiplication occurs only for two thick (non-point) intervals. Practically, i.e. in the presence of rounding errors, this remains true except for the interval factor being of very small diameter. Consider the following test. For  $A$  being

- I) the  $5 \times 5$  Hilbert matrix,
- II) the  $10 \times 10$  Hilbert matrix,
- III) a  $100 \times 100$  random matrix,
- IV) a  $100 \times 100$  ill-conditioned matrix ( $\text{cond}(A)=1e12$ )

we calculated an approximate inverse  $R=\text{inv}(A)$  followed by a preconditioning  $R*\text{midrad}(A,e)$  for different values of  $e$ . Here,  $\text{midrad}(A,e)$  is an interval matrix of smallest width such that  $A.\text{inf} \leq A-e \leq A+e \leq A.\text{sup}$ . Let  $C3$  and  $C4$  denote the result achieved by Algorithm 2.3 and 2.4, respectively. The following table displays the minimum, average and maximum of  $\text{rad}(C4) ./ \text{rad}(C3)$  for different values of the radius  $e$ .

e	case I)			case II)		
	1e-16	1e-15	1e-14	1e-16	1e-15	1e-14
minimum	0.74	1.02	1.00	0.78	0.99	1.00
average	0.85	1.04	1.01	0.90	1.02	1.00
maximum	0.94	1.10	1.02	0.98	1.06	1.01

  

e	case III)			case IV)		
	1e-16	1e-15	1e-14	1e-16	1e-15	1e-14
minimum	0.48	0.96	0.99	0.83	0.86	0.83
average	1.09	1.00	1.00	1.02	1.01	1.02
maximum	1.87	1.05	1.01	1.21	1.23	1.19

TABLE 2.5. *Ratio of radii of Algorithm 2.4 vs. Algorithm 2.3*

For very small radius of  $A$ , sometimes the one, sometimes the other algorithm delivers better results. However, in this case the radii of the final result are very small anyway (of the order of few ulps).

We performed the same test for the solution of systems of linear equations with dimensions up to 500. In this case, *all* computed results were identical in all components up to less than a hundredth of a per cent of the radius. Therefore we recommend to use the faster Algorithm 2.4.

The two cases discussed, point matrix times point matrix and point matrix times interval matrix, are already sufficient for many verification algorithms from linear to nonlinear systems solvers. Here, matrix multiplication usually occurs as preconditioning.

**2.4. Real interval matrix times real interval matrix.** Finally, consider two interval matrices  $\text{intA} \in \text{IIM}_{m,n}(\mathbb{R})$  and  $\text{intB} \in \text{IIM}_{n,k}(\mathbb{R})$ . The standard top-down algorithm uses a three-fold loop with an addition and a multiplication of two (scalar) intervals in the most inner loop. Multiplication of two scalar intervals, in turn, suffers from various case distinctions and switching of the rounding mode. This slows down computation terribly. An alternative may be the following modification of Algorithm 2.2.

```

C.inf = zeros(size(intA,1),size(intB,2));
C.sup = C.inf;
setround(-1)
for i=1:size(intA,2)
    C.inf = C.inf + min( intA.inf(:,i)*intB.inf(i,:) , ...
                        intA.inf(:,i)*intB.sup(i,:) , ...
                        intA.sup(:,i)*intB.inf(i,:) , ...
                        intA.sup(:,i)*intB.sup(i,:) );
setround(1)
for i=1:size(intA,2)
    C.sup = C.sup + max( intA.inf(:,i)*intB.inf(i,:) , ...
                        intA.inf(:,i)*intB.sup(i,:) , ...
                        intA.sup(:,i)*intB.inf(i,:) , ...
                        intA.sup(:,i)*intB.sup(i,:) );
end
end

```

ALGORITHM 2.6. *Interval matrix times interval matrix*

Correctness of the algorithm follows as before. For the multiplication of two  $n \times n$  interval matrices, Algorithm 2.6 needs a total of  $8n^3$  multiplications, additions and comparisons. At first sight this seems to be very inefficient. In contrast, the standard top-down algorithm with a three-fold loop and a scalar interval product in the most inner loop can hardly be optimized by the compiler. However, the timings to be given after the next algorithm tell a different story.

A fast algorithm is possible when allowing a certain overestimation of the result. Algorithm 2.6 computes the interval hull in the sense of infimum/supremum arithmetic. When computing the result in midpoint/radius arithmetic like in Algorithm 2.4, better use of BLAS is possible. Consider the following algorithm for the multiplication of two interval matrices  $\text{intA}$  and  $\text{intB}$ .

```

setround(1)
Amid = intA.inf + 0.5*(intA.sup - intA.inf);
Arad = Amid - intA.inf;
Bmid = intB.inf + 0.5*(intB.sup - intB.inf);
Brad = Bmid - intB.inf;
setround(-1)
C1 = Amid*Bmid;
setround(1)
C2 = Amid*Bmid;
Cmid = C1 + 0.5*(C2 - C1);
Crad = ( Cmid - C1 ) + ...
        Arad * ( abs(Bmid) + Brad ) + abs(Amid) * Brad ;
setround(-1)
C.inf = Cmid - Crad;
setround(1)
C.sup = Cmid + Crad;

```

ALGORITHM 2.7. *Interval matrix multiplication by midpoint/radius*

Unlike Algorithm 2.4, there may be an overestimation of the result. However, we want to stress that the overestimation is globally limited, independent of the dimension of the matrices. For narrow intervals the overestimation is very small (for quantification see [25]), for arbitrary intervals it is always globally limited by a factor 1.5. That means the radii of the intervals computed by Algorithm 2.7 are entrywise not larger than 1.5 times the radii of the result computed by power set operations. This has already been observed by Krier [17] in his Ph.D. thesis. For details see also [25]. The analysis holds for rounding-free arithmetic.

Note that `Amid`, `Arad`, `Bmid` and `Brad` are calculated in Algorithm 2.7 in such a way that

$$\begin{aligned} \text{Amid} - \text{Arad} &\leq \text{A.inf} \leq \text{A.sup} \leq \text{Amid} + \text{Arad} && \text{and} \\ \text{Bmid} - \text{Brad} &\leq \text{B.inf} \leq \text{B.sup} \leq \text{Bmid} + \text{Brad} \end{aligned}$$

is assured. It is easily seen that this is indeed always satisfied, also in the presence of underflow.

**2.5. Timing for real interval matrix multiplication.** An advantage of Algorithm 2.7 is that only  $4n^3$  multiplications and additions plus a number of  $O(n^2)$  operations are necessary. With respect to memory and cache problems those four real matrix multiplications are very fast. A disadvantage is the  $O(n^2)$  part of the algorithm with some memory operations for larger matrices.

Let  $A, B \in \mathbf{M}_n(\mathbb{R})$  and  $\text{int}A, \text{int}B \in \mathbf{IM}_n(\mathbb{R})$ , where  $\text{int}A$  and  $\text{int}B$  are assumed to be thick interval matrices. An operation count of additions and multiplications for the different cases of real matrix multiplication is as follows.

<code>A * B</code>	$n^3$	for floating point multiplication,
<code>A * B</code>	$2n^3$	for verified bounds, using Algorithm 2.1,
<code>A * intB</code>	$4n^3$	using Algorithm 2.3,
<code>A * intB</code>	$3n^3$	using Algorithm 2.4,
<code>intA * intB</code>	$8n^3$	using Algorithm 2.6,
<code>intA * intB</code>	$4n^3$	using Algorithm 2.7

TABLE 2.8. *Operation count for real matrix multiplication*

The following table shows computing times of C-routines for multiplication of two interval matrices for the standard top-down approach with scalar interval operations in the most inner loop compared to Algorithms 2.6 and 2.7. The timing is on our Convex SPP 2000 parallel computer. Computations have been performed by Jens Zemke.

dimension		100	200	500	1000
standard		0.30	2.40	72.3	613.6
Algorithm 2.6		0.42	8.9	245	1734
Alg. 2.6 parallel		1.14	10.7	216	1498
Algorithm 2.7		0.02	0.17	3.30	20.2
Alg. 2.7 parallel		0.02	0.11	0.93	5.9

TABLE 2.9. *Computing times interval matrix times interval matrix (in seconds)*

The data of the interval matrices were generated randomly such that all intervals entries were strictly positive. That means that no case distinction was necessary at all for the standard top-down approach. It follows that in the examples the standard algorithm needed only  $4n^3$  multiplications, additions and comparisons, but also additional  $4n^3$  switching of the rounding mode.

The timing clearly shows that the bare operation count and the actual computing time need not be proportional. Obviously, inefficient use of cache and memory slow down the outer product approach Algorithm 2.6 significantly. The lines 3 and 5 in Table 2.9 display the computing time using 4 processors instead of 1. We

stress that the same code was used. The only difference is that first the sequential BLAS, then the parallel BLAS was linked. For the standard top-down algorithm, use of 1 or 4 processors makes no difference.

For the other operations like addition, subtraction, reciprocal and division, may be used standard implementations combined with the above observations. We skip a detailed description and refer to the INTLAB code.

**2.6. Complex point matrix multiplication.** The next problem are complex operations. Again, we are mainly interested in matrix multiplication because this dominates the computing time of a number of verification algorithms.

Our design decision was to use midpoint/radius representation for complex intervals. For various reasons this is more appropriate for complex numbers than infimum/supremum representation using partial ordering of complex numbers. The definition of midpoint/radius arithmetic goes at least back to Sunaga [26].

In the multiplication of two point matrices A and B a new problem arises, namely that the error during midpoint calculation has to be treated separately. The following algorithm solves this problem.

```
setround(-1)
C1 = real(A)*real(B) + (-imag(A))*imag(B) + ...
    ( real(A)*imag(B) + imag(A)*real(B) ) * j ;
setround(1)
C2 = real(A)*real(B) + (-imag(A))*imag(B) + ...
    ( real(A)*imag(B) + imag(A)*real(B) ) * j ;
C.mid = C1 + 0.5 * (C2-C1);
C.rad = C.mid - C1;
```

ALGORITHM 2.10. *Complex point matrix times point matrix*

First, the implementation assures

$$C1 \leq A * B \leq C2$$

in the (entrywise) complex partial ordering. Second, the midpoint and radius are both calculated with rounding upwards. A calculation shows that indeed

$$C.mid - C.rad \leq A * B \leq C.mid + C.rad$$

is always satisfied, also in the presence of underflow.

**2.7. Complex interval matrix multiplication.** The multiplication of two complex interval matrices does not cause additional problems. The algorithm is as follows, again in original INTLAB code.

```
setround(-1)
C1 = real(intA.mid)*real(intB.mid) + (-imag(intA.mid))*imag(intB.mid) + ...
    ( real(intA.mid)*imag(intB.mid) + imag(intA.mid)*real(intB.mid) ) * j ;
setround(1)
C2 = real(intA.mid)*real(intB.mid) + (-imag(intA.mid))*imag(intB.mid) + ...
    ( real(intA.mid)*imag(intB.mid) + imag(intA.mid)*real(intB.mid) ) * j ;
C.mid = C1 + 0.5 * (C2-C1);
C.rad = abs( C.mid - C1 ) + ...
    intA.rad * ( abs(intB.mid) + intB.rad ) + abs(intA.mid) * intB.rad ;
```

ALGORITHM 2.11. *Complex interval matrix times interval matrix*

Note that the radius is calculated with only two additional point matrix multiplications. Also note that again midpoint and radius of the result are computed with rounding upwards. Correctness of the algorithm

is easily checked. The other complex operations are implemented like the above and do not cause additional difficulties.

We want to stress again that Algorithms 2.10 and 2.11 exclusively use BLAS routines. This assures fast execution times. Let  $A, B \in \mathbf{M}_n(\mathbf{C})$  and  $\text{int}A, \text{int}B \in \mathbf{IM}_n(\mathbf{C})$ , where  $\text{int}A$  and  $\text{int}B$  are assumed to be thick interval matrices. An operation count of additions and multiplications for the different cases of complex matrix multiplication is as follows.

$A * B$	$4n^3$	for floating point multiplication,
$A * B$	$8n^3$	for verified bounds,
$A * \text{int}B$	$9n^3$	for verified bounds,
$\text{int}A * \text{int}B$	$10n^3$	for verified bounds.

TABLE 2.12. *Operation count for complex matrix multiplication*

The factor between interval and pure floating point multiplication is smaller than in the real case; it is only 2, 2.25 and 2.5, respectively.

Finally we mention that, as in the real case, the overestimation of the result of interval matrix multiplication using the above midpoint/radius arithmetic cannot be worse than a factor 1.5 in radius compared to power set operations ([17], [25]), and is much better for narrow interval operands [25].

**3. The interval library and timings in INTLAB.** In the preceding section we discussed a general interval library - for implementation in a to-compile programming language like Fortran or C. When moving to an implementation in Matlab, the interpretation overhead becomes a big issue: The "interpretation count" becomes much more important than the operation count. That means algorithms should be designed loop-free using few computing intensive operators. This is true for most of the fast algorithms developed in the previous section. All following computing times are generated on a 120 Mhz Pentium I Laptop and using Matlab V5.1.

**3.1. Real matrix multiplication.** The first example is real matrix multiplication. In many applications this is a computing intensive part of the code. We compare the computing time of the built-in pure floating point routines with INTLAB times for verified computations. Both approaches use extensively BLAS routines. In Table 2.9 we listed the computing times for C-routines on a parallel computer for compiled code; following are INTLAB times including interpretation overhead on the 120 Mhz Laptop. The following code, for the example  $n = 200$ , is used for testing.

```
n=200; A=2*rand(n)-1; intA=midrad(A,1e-12); k=10;
tic; for i=1:k, A*A;          end, toc/k
tic; for i=1:k, intval(A)*A; end, toc/k
tic; for i=1:k, intA*A;      end, toc/k
tic; for i=1:k, intA*intA;   end, toc/k
```

The first line generates an  $n \times n$  matrix with randomly distributed entries in the interval  $[-1, 1]$ . INTLAB uses Algorithms 2.1, 2.4 and 2.7. The command `tic` starts the wall clock, `toc` is the elapsed time since the last call of `tic`. The computing times are as follows.

dimension	pure floating point	verified A*A	verified A*intA	verified intA*intA
100	0.11	0.22	0.35	0.48
200	0.77	1.60	2.84	3.33
500	15.1	29.1	44.7	60.4

TABLE 3.1. *Real matrix multiplication (computing time in seconds)*

Going from dimension 100 to 200 there is a theoretical factor of 8 in computing time. This is achieved in practice. From dimension 200 to 500 the factor should be  $2.5^3 = 15.6$ , but in practice the factor is between 18 and 19. This is due to cache misses and limited memory.

The INTLAB matrix multiplication algorithm checks whether input intervals are thin or thick. This explains the difference between the second and third column in Table 3.1. According to Algorithms 2.1, 2.4, 2.7 and Table 2.8, the theoretical factors between the first and following columns of computing times are approximately 2, 3 and 4, respectively. This corresponds very good to the practical measurements, also for  $n = 500$ . Note that the timing is performed by the Matlab wall clock; actual timings may vary by some 10 per cent.

For the third case, point matrix times interval matrix, alternatively Algorithms 2.3 and 2.4 may be used. The computing times are as follows.

dimension	Algorithm 2.3	Algorithm 2.4
100	0.45	0.35
200	3.79	2.85
500	59.2	44.7

TABLE 3.2. Algorithms 2.3 and 2.4 for point matrix times interval matrix (computing time in seconds)

The theoretical ratio 0.75 is achieved in the practical implementation, so we use Algorithm 2.4 in INTLAB.

**3.2. Complex matrix multiplication.** The computing times for complex matrix multiplication using Algorithms 2.10 and 2.11 are as follows.

dimension	pure floating point	verified A*A	verified A*intA	verified intA*intA
100	0.35	1.00	1.11	1.25
200	3.08	6.92	7.80	8.71
500	79.7	119.3	135.4	147.9

TABLE 3.3. Complex matrix multiplication (computing time in seconds)

For  $n = 500$ , again cache misses and limited memory in our Laptop slow down the computing time. According to Table 2.12, the theoretical factors between the first and following columns of computing times are approximately 2, 2.25 and 2.5, respectively. This corresponds approximately to the measurements. The timing depends (at least for our Laptop) on the fact that for  $A, B \in \mathbf{M}_n(\mathbf{C})$  the built-in multiplication  $A * B$  is faster than the multiplication used in Algorithm 2.10 for smaller dimensions, but slower for larger dimensions. Consider

```
A=rand(n)+j*rand(n); B=A; k=10;
tic; for i=1:k, A * B; end, toc/k
tic; for i=1:k, real(A)*real(B)+(-imag(A)*imag(B);
                real(A)*imag(B)+imag(A)*real(B); end, toc/k
```

for different dimensions  $n$ . The timing is as follows.

n	A * B	real/imaginary part separately
100	0.37	0.47
200	3.35	3.35
500	81.2	68.7

TABLE 3.4. *Different ways of complex matrix multiplication (computing time in seconds)*

**3.3. Interval matrix multiplication with and without overestimation.** Up to now we used the fast version of interval matrix times interval matrix. Needless to say that the standard approach with three loops and case distinctions in the inner loop produces a gigantic interpretation overhead. INTLAB allows to switch online between Algorithms 2.7 and 2.6. A timing is performed by the following statements.

```
n=100; A=midrad(2*rand(n)-1,1e-12);
IntvalInit('FastIVMult'); tic; A*A; toc
IntvalInit('SharpIVMult'); tic; A*A; toc
```

Computing times are

0.4 seconds	for fast multiplication according to Algorithm 2.7, and
23.0 seconds	for sharp multiplication according to Algorithm 2.6.

The difference is entirely due to the severe interpretation overhead. It is remarkable that, in contrast to compiled code, for the interpreted system now Algorithm 2.6 is apparently the fastest way to compute the product of two thick interval matrices without overestimation. However, we will see in Section 5 that the overestimation in Algorithm 2.7 due to conversion to midpoint/radius representation is negligible in many applications.

**4. The INTLAB toolbox.** Version 5 of Matlab allows the definition of new classes of variables together with overloading of operators for such variables. The use and implementation is as easy as one would expect from Matlab. However, quite some interpretation overhead has to be compensated. Remember that the goal is to implement everything in Matlab.

**4.1. The operator concept.** For example, a new type `intval` is introduced by defining a subdirectory `@intval` under a directory in the search path of Matlab. This subdirectory contains a constructor `intval`. The latter is a file `intval.m` containing a statement like

```
c = class(c,'intval');
```

This tells the Matlab interpreter that the variable `c` is of type `intval`, although no explicit type declaration is necessary. Henceforth, a standard operator concept is available with a long list of overloadable operators including `+`, `-`, `*`, `/`, `\`, `'`, `~`, `[]`, `{}` and many others.

It has been mentioned before that our design decision was to use infimum/supremum representation for real intervals, and midpoint/radius representation for complex intervals. From the mathematical point of view this seems reasonable. Therefore, our data type `intval` for intervals is a structure with five components:

<code>x.complex</code>	boolean, true if interval is complex,
<code>x.inf</code> , <code>x.sup</code>	infimum and supremum for real intervals (empty for complex intervals),
<code>x.mid</code> , <code>x.rad</code>	midpoint and radius for complex intervals (empty for real intervals).

The internal representation is not visible for the user. For example,

```
inf(x) or x.inf or get(x,'inf')
```

accesses the infimum of the interval variable `x` independent of `x` being real or complex. Operations check the type of inputs to be real or complex.

INTLAB offers three different routines to define an interval:

- by the constructor `intval`, e.g. `x = intval(3)`;
- by infimum and supremum, e.g. `x = infsup(4.3-2i,4.5-.1i)`;
- by midpoint and radius, e.g. `x = midrad(3.14159,1e-5)`;

The first is a bare constructor, the following define an interval by infimum and supremum or by midpoint and radius, respectively. The second example defines a complex interval using partial ordering.

**4.2. Rigorous input and output.** If the input data is not exactly representable in floating point, the (decimal) input data and the internally stored (binary) data do not coincide. This is a problem to all libraries written in a language like Fortran 90, C++ or Matlab. The well known example

$$(4.1) \quad \mathbf{x} = \mathbf{intval}(0.1)$$

converts 0.1 into the nearest floating number, and *then* assigns that value to `x.inf` and `x.sup`. However, 0.1 is not exactly representable in binary finite precision so that 1/10 will *not* be included in `x`.

This is the I/O problem of every interval library. Sometimes, the internal I/O routines do not even satisfy a quality criterion, or at least this is not specified and/or not guaranteed by the manufacturer (during implementation of INTLAB we found plenty of such examples). We had to solve this problem for rigorous input and to guarantee validity of displayed results.

We did *not* want to execute the assignment (4.1) in a way that a small interval is put around 0.1. One of the reasons not to do this is that `intval(-3)` would produce an interval of nonzero radius. Another reason is that the routine `intval` is also used just to change the type of a double variable, and the user would not want to get back an enlarged interval. The better way is to use

$$\mathbf{x} = \mathbf{intval}('0.1')$$

instead. In this statement the character string '0.1' is converted into an enclosing interval by means of the INTLAB conversion routine `str2intval with result verification`. For exactly representable input like `intval('1')` this does not cause a widening of the interval; this is also true, for example, for `intval('0.1e1-1.25i')`. Constants with tolerances may be entered directly by replacing uncertainties by '\_'. For example,

$$\mathbf{x} = \mathbf{intval}('3.14159_')$$

produces the interval [3.14158, 3.14160]. The underscore \_ in input and output is to be interpreted as follows: A correct inclusion is produced by subtracting 1 from and adding 1 to the last displayed figure.

We want to stay with our philosophy to use only Matlab code and not any C-code or assembly language code except the routine for switching the rounding mode. There are two problems in writing a conversion routine in INTLAB. First, the result should be as narrow as possible and second, there should not be too much interpretation overhead. Both problems are solved by a simple trick. We computed 2 two-dimensional arrays `power10inf` and `power10sup` of double floating point numbers with the property

$$(4.2) \quad \mathbf{power10inf}(m,e) \leq m * 10^e \leq \mathbf{power10sup}(m,e),$$

for all  $m \in \{1, 2, \dots, 9\}$  and  $e \in \{-340, -339, \dots, 308\}$ . These are one-digit mantissa floating point numbers with exponents corresponding to IEEE 754 double format. Those 11682 numbers were computed off-line and stored in a file. The numbers are sharp. The computation used a rudimentary long arithmetic, all written in INTLAB, and takes less than half a minute on a 120 Mhz Pentium I Laptop.

With these numbers available, conversion is straightforward. For example,

$$(4.3) \quad x = 0.m_1m_2\dots m_k \cdot 10^e \Rightarrow \sum_{i=1}^k \mathbf{power10inf}(m_i, e-i) \leq x \leq \sum_{i=1}^k \mathbf{power10sup}(m_i, e-i).$$

For sharp results, summation should be performed starting with smallest terms. This method using (4.3) together with (4.2) can also be vectorized. This allows fast conversion of long vectors of input strings. The sign is treated separately.

The output of intervals is also accomplished using (4.2) and (4.3). We convert a double number into a string, convert the string back into double with directed rounding and check whether the result is correct. If not, the string has to be corrected. This would be terribly slow if performed for every single component of an output vector or matrix. However, the code is vectorized such that the interpretation overhead is about the same for scalar or vector argument. The user hardly recognizes a difference in time of internal input/output and rigorous input/output. Summarizing, input and output is rigorously verified to be correct in INTLAB.

**4.3. Automatic differentiation, pure floating point and rigorous.** Most of the other features in INTLAB are more or less straightforward implementations of an interval library using the algorithms described in Section 2 and by using the operator concept in Matlab. It also allows, for example, the implementation of automatic differentiation. For some scalar, vector or matrix  $x$ ,

```
gx = initvar(x)
```

initializes  $gx$  to be the dependent variables with values as stored in  $x$ . The corresponding data type is `gradient`. An expression involving  $gx$  is computed in forward differentiation mode with access to value and gradient by substructures `.x` and `.dx`. Consider, for example, the evaluation of the function  $y = f(x) = \sin(0.1 \cdot \pi \cdot x)$ . The following m-file

```
(4.4)      function y = f(x)
           y = sin( 0.1*pi*x );
```

may be used to evaluate  $f(x)$  by  $y=f(x)$  for  $x$  being a real or complex scalar, vector or matrix, or a sparse matrix. By Matlab convention, evaluation of standard functions with vector or matrix argument is performed entrywise. Similarly,

```
gy = f(gx)
```

gives access to  $gy.x$ , the same as  $f(x)$ , and to the partial derivatives  $gy.dx$ . For  $[m,n]$  being the size of  $gx$ ,  $[m,n,k]$  is the size of  $gy.dx$ , where  $k$  is the number of dependent variables. This is true unless  $n=1$ , in which case the size of  $gy.dx$  is  $[m,k]$ . That means, the gradient information is put into an extra dimension with length equal to the number of dependent variables. For example, the input

```
X = initvar([3;-4]); Y = f(X)
```

produces the output

```
gradient value Y.x =
    0.8090
   -0.9511

gradient derivative(s) Y.dx =
    0.1847      0
      0      0.0971
```

The operator concept follows the usual rules how to choose the appropriate operator. For example, if one of the operands is of type interval, the computation will be carried out in interval arithmetic with verified result. If none of the operands are of type interval, the result is calculated in ordinary floating point arithmetic.

**4.4. A problem and its solution.** For example,

```
a = 0.1; x = intval(4); (a-a)*x
```

will produce a point interval consisting only of zero, although the multiplication is actually an interval multiplication. However, the left factor is exactly zero. In contrast,

```
a = intval('0.1'); x = intval(4); (a-a)*x
```

produces a small interval around zero. For a function defined in an m-file, like the one defined in (4.4), this might cause problems. The statement

```
X = intval([3;-4]); Y = f(X)
```

evaluates the first product  $0.1 \cdot \pi$  in floating point, and only the following multiplication and the `sin` is performed with directed roundings. In order to avoid this, additional knowledge about the type of the input variable `x` is needed: In case `x` is floating point, all operations are floating point, in case `x` is of type `intval`, only interval operations should be used. In fact, the problem is more complicated. In definition (4.4) we used the built-in variable `pi`, an approximation of the transcendental number  $\pi$ . For pure floating point calculations, this is legitimate, but not for interval calculations. Alternatively, we may use an interval constant like in the following algorithm:

```
(4.5) function y = f(x)
        cPi = midrad(pi,1e-15);
        y = sin( 0.1*cPi*x );
```

But this causes problems in the pure floating point case because the result would *always* be of type `intval`. Our solution to this is a type adjustment by the following algorithm.

```
(4.6) function y = f(x)
        cPi = typeadj( midrad(pi,1e-15) , typeof(x) );
        y = sin( 0.1*cPi*x );
```

The function `typeadj` adjusts the type of the left input argument, in this case `cPi`, to the right argument. If `x` is of type `intval`, nothing happens, `cPi` is equal to `midrad(pi,1e-15)`. If `x` is floating point, so will be `cPi` by taking the midpoint of `midrad(pi,1e-15)`, which is `pi` again. In the same way, the statements

```
X = initvar([3;-4]); Y = f(X)
X = initvar(intval([3;-4])); Y = f(X)
```

will work with definition (4.6). In the first case, a pure floating point gradient calculation is performed, and in the second case an interval gradient calculation. That means in the second case, `Y.x` and `Y.dx` are of type `intval` being inclusions of the function value and the true gradient.

Similar to the gradient toolbox, there is a slope toolbox in INTLAB for expanding a function with respect to an expansion "point" `xs` (which may, in fact, be an interval) within an expansion interval `x`. The above remarks for `typeof` and `typeadj` apply accordingly.

INTLAB makes use of the data types already offered in Matlab, like sparse matrices or complex data. All the functionality of Matlab concerning for example sparse matrix calculations are immediately available in INTLAB. For example, the algorithms listed in Section 3 are suitable for sparse matrices without change. This is the one of the big advantages of the extensive use of BLAS routines.

**4.5. Correctness of programming.** There are various sources of errors in numerical algorithms. To achieve rigorously verified results not only the verification ansatz need to be correct but also the implementation itself. Needless to say that all algorithms in INTLAB are examined by a large number of test routines. This improves confidence in the library but is no proof of correctness. However, we will give two arguments applying to INTLAB but not to other numerical algorithms which may imply some additional safety.

The first argument applies to every verification algorithm, namely the necessity of *correctness* of results. For example,  $p = 3.141592653589792$  is an excellent approximation for the transcendental number  $\pi$ ; however, the interval  $p \pm 10^{-15}$  is a *wrong* answer for the smallest positive zero of  $\sin(x)$ . It turns out, as a practical experience, that errors sometimes discover by such "inclusions" being only slightly off the true answer. Other examples are inclusions like  $[5, 5 + 10^{-16}]$  if the integer 5 is the true result. For many verification algorithms this is much likely pointing to an implementation error.

The second argument applies specifically to INTLAB. Due to the interpretative language algorithms are formulated in a very high level language. Therefore, the algorithms are more like *executable specifications*, and therefore very short, and correspondingly easy to check for correctness.

**5. Programming and applications in INTLAB.** In the following we give some sample programs to demonstrate the use of INTLAB. All examples are given in original and executable code. Also, we give execution times with comparisons to pure floating point algorithms (those are frequently near the fastest achievable for the computer in use). All of the following computing times are again generated on our 120 Mhz Pentium I Laptop using Matlab V5.1.

**5.1. Dense linear system.** The first example is the solution of a dense system of linear equations. The following code is used in INTLAB.

```
function X = denselss(A,b)           % linear system solver
    R = inv( mid(A) ) ;             % for dense matrices
    xs = R * mid(b) ;
    Z = R * (b-intval(A)*xs) ;      % residual correction
    C = speye(size(A)) - R*intval(A); % residual of preconditioner
    Y = Z;
    E = 0.1*rad(Y)*hull(-1,1) + midrad(0,10*realmin);
    k = 0; kmax = 15; ready = 0;
    while ~ready & k<kmax           % interval iteration
        k = k+1;
        X = Y + E;
        Y = Z + C * X;
        ready = in0(Y,X);          % check inclusion in interior
    end
    if ready                        % verified inclusion
        X = xs + Y;
    else
        disp('no inclusion achieved for \');
        X = NaN;
    end
end
```

ALGORITHM 5.1. *Solution of dense linear systems*

This is a well known algorithm for solving systems of linear equations based on the Krawczyk operator [16], [20]. The algorithm computes an inclusion of the difference of the true solution and the approximate solution  $xs$ . It is identical to the one presented in [23].

The 4th line  $Z = R*(b-intval(A)*xs)$  is a typical statement to ensure interval calculation of the residual. Note that a statement like, for example,  $Z = R*intval(b-A*xs)$  would not work correctly in general. The timing in seconds of Algorithm 5.1 for randomly generated real and complex linear systems of equations is as follows. The second column refers to both point matrix and right hand side, whereas for the third column both matrix and right hand side are thick.

dimension	pure floating point	verified point	verified interval
100	0.09	0.53	0.70
200	0.56	3.35	4.23
500	8.2	50.9	67.6

TABLE 5.2. *Solution of real linear systems (computing time in seconds)*

Algorithm 5.1 requires computation of an approximate inverse and subsequent multiplication of a point matrix times the system matrix. All other computing times are dominated by  $0(n^2)$ . Hence, the computing time in the real case is  $3n^3$  for point and  $4n^3$  for interval linear systems, respectively. Hence, the theoretical factor between the first column of computing times and following columns is 9 and 12, respectively.

The actual computing times are much better than this. The reason is that the  $n^3$  operations for matrix inversion and matrix multiplication take less than three times the computing time of Gaussian elimination. This is because code for matrix products can be better blocked and optimized. Consider

```
n=200; A=2*rand(n)-1; b=A*ones(n,1); k=10;
tic; for i=1:k, inv(A); end, toc/k
tic; for i=1:k, A*A; end, toc/k
tic; for i=1:k, A\b; end, toc/k
```

taking

```
1.24 seconds    for matrix inversion,
0.78 seconds    for matrix multiplication, and
0.56 seconds    for Gaussian elimination
```

on the 120 Mhz Laptop. The practical ratio of computing time for Gaussian elimination to inversion is 2.2 instead of a theoretical 3, and the ratio to matrix multiplication only 1.4 instead of 3.

The computing times for the solution of complex systems of linear equations are as follows. As before, **point** refers to both matrix and right hand side being thin, and **interval** refers to both being thick. For  $n = 500$ , again the effect of cache misses is visible.

dimension	pure floating point	verified point	verified interval
100	0.22	1.92	2.05
200	1.65	13.1	15.1
500	34.0	234	246

TABLE 5.3. *Solution of complex linear systems (computing time in seconds)*

As an example of ease of use of programming in INTLAB, we display the code for the verified solution of over- and underdetermined systems. It is done by solving a larger linear system with square matrix. Brackets are overloaded interval operators to define matrices by means of blocks. The statement **isnan** is true if verification of the constructed square linear system failed. Otherwise, the code is hopefully self-explaining. Note that the right hand side may be a matrix.

```
[m k] = size(A); [m1 n] = size(b);
if k~=m1
    error('linear system solver: inner matrix dimensions must agree')
end
if m==k % linear system with square matrix
    X = denselss(A,b);
else
    if m>k % least squares problem
        Y = denselss( [ A -eye(m) ; zeros(k) A' ] , [ b ; zeros(k,n) ] );
        if ~isnan(Y)
            X = Y(1:k,:);
        end
    else % minimal norm solution
```

```

    Y = denselss( [ A' -eye(k) ; zeros(m) A ] , [ zeros(k,n) ; b ] );
    if ~isnan(Y)
        X = Y(m+1:m+k,:);
    end
end
end
end

```

ALGORITHM 5.4. *Over- and underdetermined linear systems*

**5.2. Sparse linear systems.** As has been mentioned before, INTLAB supports interval sparse matrices. We use the algorithm for symmetric positive definite systems as presented in [24]. The INTLAB code is as follows.

```

function X = sparselss(A,b)          % linear system solver for sparse matrices
    [s,xs] = singmin(mid(A),mid(b));
    As = mid(A) - s*speye(size(A));
    C = chol(As);
    setround(-1)
    p1 = C'*C - As;
    setround(1)
    p2 = C'*C - As;
    r = norm(p2-p1,1);
    setround(-1)
    minsvd = s - r;
    if minsvd<0
        disp('matrix too ill-conditioned')
        X = NaN;
    else
        setround(1)
        X = midrad( xs , norm(A*xs-b)/minsvd ) ;
    end
end
setround(0)

```

ALGORITHM 5.5. *Sparse linear systems*

The algorithm uses the routine `singmin` for a guess of a lower bound of the smallest singular value of  $A$ . This is a straightforward implementation using an approximate Cholesky factorization of  $A$  and inverse power iteration.

For timing we use the Matlab functions `sprand` to generate a sparse matrix of dimension 1000 with random entries and random pattern. We add the identity matrix to ensure nonsingularity, and use  $A \cdot A'$  to ensure symmetric positive definiteness. The generation statement is as follows, the density is 0.9% on the average.

```
n=1000; A=sprand(n,n,2/n)+speye(n); A=A*A'; b=A*ones(n,1);
```

Algorithm 4.8 performs a factorization of the matrix. Therefore it is important to reduce fill-in, for example by minimum degree reordering. The Matlab code

```
p = symmmd(A); C = chol(A(p,p));
```

produces a Cholesky factor of the reordered matrix with density of about 11%, without reordering it is more than 50%. We solve the linear system in pure floating point and with verification using the following commands.

```
tic; A(p,p)\b(p); toc
tic; verifylss(A(p,p),b(p)); toc
```

The timing is

2.8 seconds for pure floating point solution, and  
 18.9 seconds with verification.

For larger dimensions the Matlab routine `symmmd` becomes very slow; the built-in solver and the verified solver work well for larger dimensions. The latter will be demonstrated for band matrices. The following statements produce a random band matrix with 21 nonzero elements per row and test the built-in linear system solver against our verification algorithm.

```
k=5; A=spdiags(2*rand(n,2*k+1)-1,-k:k,n,n)+speye(n); A=A*A'; b=A*ones(n,1);
tic, A\b; toc
tic, verifylss(A,b); toc
```

The computing time is slowed down by the fact that the matrix is *not* stored as a band matrix but as a sparse matrix. All matrix operations are general sparse matrix operations. The timing on the 120 Mhz Laptop is as follows.

n	floating point	verified
1000	0.12	1.10
2000	0.24	2.26
5000	0.60	5.71
10000	1.21	15.0
20000	4.1	183

TABLE 5.6. *Timing for banded linear systems (in seconds)*

Starting with dimension 10000 the timing is dominated by swapping, displaying mainly lack of memory on our Laptop and a slow disc. Note that due to the sparse matrix storage scheme, Matlab needs for dimension  $n = 20000$  already 14.6 MByte of storage for the matrix.

**5.3. Nonlinear systems.** Finally, consider the solution of systems of nonlinear equations. In the formulation of the function it is important to reduce interpretation overhead. That means, any possibility to use high order vector and matrix operations should be used. This is the drawback of interactive environments. This is in particular true when using overloaded operators. Consider, for example,

```
(5.1) tic; for i=1:1000, y=sqrt(exp(x)-x); end; toc
```

The computing time is

0.1 seconds for  $x=0.1$ , and  
 9.5 seconds for  $x=intval(0.1)$  .

In contrast, the following does the same in vector notation.

```
X=x*ones(1000,1); tic; Y=sqrt(exp(X)-X); toc
```

Now the computing time is

0.006 seconds for  $x=0.1$ , and  
 0.024 seconds for  $x=intval(0.1)$  .

We also mention that interpretation is particularly slow on the 120 Mhz Pentium I Laptop. The statement (5.1) executed on a 266 Mhz Pentium II PC needs

0.017 seconds for  $x=0.1$ , but  
 1.51 seconds for  $x=intval(0.1)$  ,

a speed-up factor 6 compared to the 120 Mhz Laptop. On the other hand, interpretation is at least partially compensated by the ease of notation. Consider the INTLAB code for the solution of a system of nonlinear equations. It is assumed that a call  $f(x)$  calculates the function value at  $x$ . An example for a function  $f$  is given below. The algorithm uses a modified Krawczyk operator [24].

```
function [ X , xs ] = verifynlss(f,xs)

% floating point Newton iteration
n = length(xs);
xsold = xs;
k = 0;
while ( norm(xs-xsold)>1e-10*norm(xs) & k<10 ) | k<1
    k = k+1;                % at most 10, at least 1 iteration performed
    xsold = xs;
    x = initvar(xs);        % initialization of gradient calculation
    y = feval(f,x);        % f(x) using gradient variable x
    xs = xs - y.dx\y.x;    % Newton correction
end

% interval iteration
R = inv(y.dx);            % approximate inverse of Jacobian
Z = - R * feval(f,intval(xs)); % initialization of interval iteration
X = Z;
E = 0.1*rad(X)*hull(-1,1) + midrad(0,realmin);
ready = 0; k = 0;
while ~ready & k<10
    k = k+1;
    Y = hull( X + E , 0 );    % epsilon inflation
    Yold = Y;
    x = initvar(xs+Y);        % interval gradient initialization
    y = feval(f,x);          % f(x) and Jacobian by
    C = eye(n) - R * y.dx;    % automatic differentiation
    i=0;
    while ~ready & i<2        % improved interval iteration
        i = i+1;
        X = Z + C * Y;
        ready = in0(X,Y);
        Y = intersect(X,Yold);
    end
end
if ready
    X = xs+Y;                % verified inclusion
else
    X = NaN;                 % inclusion failed
end
end
```

ALGORITHM 5.7. *Verified solution of nonlinear systems*

The first part is a pure floating point iteration, the second part is to calculate an inclusion. Note that the

floating point evaluation of  $f$  at  $x$  with gradient is performed by

```
x = initvar(xs);
y = feval(f,x);
```

in the first part, and the verified evaluation at  $xs+Y$ , again with gradient, is performed by

```
x = initvar(xs+Y);
y = feval(f,x);
```

As an example consider a problem given by Abbott and Brent in [1], the discretization of

$$3y''y + y'^2 = 0 \quad \text{with} \quad y(0) = 0 \quad \text{and} \quad y(1) = 20.$$

We solve the discretized system, not the continuous equation. In the paper, the initial approximation is a vector all entries of which are equal to 10, the true solution is  $20x^{0.75}$ . The discretized problem is specified by the following INTLAB function.

```
function y = f(x)
    y = x;
    n = length(x); v=2:n-1;
    y(1) = 3*x(1)*(x(2)-2*x(1)) + x(2)*x(2)/4;
    y(v) = 3*x(v).*(x(v+1)-2*x(v)+x(v-1)) + (x(v+1)-x(v-1)).^2/4;
    y(n) = 3*x(n).*(20-2*x(n)+x(n-1)) + (20-x(n-1)).^2/4;
```

Note the vectorized formulation of the function. The dimension of the nonlinear system is determined by the number of elements of the specified approximation. The timing for the solution of the nonlinear system by the following statement

```
tic; X = verifynlss('f',10*ones(n,1)); toc
```

is

```
5.4 seconds    for dimension n = 50,
8.5 seconds    for dimension n = 100, and
20.3 seconds   for dimension n = 200.
```

The first and last components of the inclusion for  $n = 200$  are

```
X(1:4)'
intval ans =
    0.346256418326_    0.6045521734322    0.8305219234696    1.0376691412984
X(197:200)'
intval ans =
    19.7005694833674    19.775568557350_    19.8504729393822    19.9252832242374
```

Finally, it may be interesting to analyze the computing intensive parts of the algorithm. This is possible by the Matlab profile function. The statements

```
profile verifynlss; n=200; verifynlss('f',10*ones(n,1));
profile report; profile done
```

produce the following output:

```
Total time in "c:\matlab\toolbox\intlal\intval\verifynlss.m": 21.41 seconds
```

```
100% of the total time was spent on lines:
[18 33 34 19 24 38 23 17 26 32]
```

```

16:    xsold = xs;
0.10s, 0% 17:    x = initvar(xs);
10.47s, 49% 18:    y = feval(f,x);
1.26s, 6% 19:    xs = xs - y.dx\y.x;
20:    end

22: % interval iteration
0.15s, 1% 23:    R = inv(y.dx);
0.42s, 2% 24:    Z = - R * feval(f,intval(xs));
25:    X = Z;
0.03s, 0% 26:    E = 0.1*rad(X)*hull(-1,1) + midrad(0,realmin);
27:    ready = 0; k = 0;

31:    Yold = Y;
0.02s, 0% 32:    x = initvar(xs+Y);
6.07s, 28% 33:    y = feval(f,x);           % f(x) and Jacobian by
2.68s, 13% 34:    C = eye(n) - R * y.dx;   % automatic differentiation
35:    i=0;

37:    i = i+1;
0.16s, 1% 38:    X = Z + C * Y;
39:    ready = in0(X,Y);

```

TABLE 5.8. *Profiling nonlinear system solver*

From the profile follows that more than half of the computing time is spent for the improvement of the poor initial approximation and, that most of the time is spent for the function evaluations in lines 18 and 33. Note that the matrix inversion of the  $200 \times 200$  Jacobian in line 23 takes only 1 % of the computing time or about 0.2 seconds. The reason seems that the Jacobian is tridiagonal and Matlab matrix inversion takes advantage of that.

**6. Future work and conclusions.** INTLAB is an interval toolbox for the interactive programming environment Matlab. It allows fast implementation of prototypes of verification algorithms. The goal has been met that the computing time for matrix operations or solution of linear systems is of the order of comparable pure floating point algorithms. For nonlinear systems, the computing time is usually dominated by interpretation overhead for the evaluation of the nonlinear function in use. This overhead is diminished proportional to the possibility of vectorization of the code for the nonlinear function.

Up to now we stay with our philosophy to have everything written in INTLAB, that is Matlab code, except for the routine for switching the rounding mode. This strategy implies greatest portability and high speed on a variety of architectures. Wherever Matlab and IEEE 754 arithmetic with the possibility of switching the rounding mode is available, INTLAB is easy to install and to use.

The (freeware) INTLAB toolbox may be freely copied from the INTLAB home page

<http://www.ti3.tu-harburg.de/~rump/intlab/index.html> .

**Acknowledgement.** The author is indebted to two anonymous referees for their thorough reading and for their fair and constructive remarks. Moreover, the author wishes to thank Shin'ishi Oishi and his group for many fruitful discussions during his stay at Waseda university, and Arnold Neumaier and Jens Zemke who contributed with their suggestions to the current status of INTLAB. The latter performed all test runs on our parallel computer.

## REFERENCES

- [1] J.P. Abbott and R.P. Brent. Fast Local Convergence with Single and Multistep Methods for Nonlinear Equations. *Austr. Math. Soc. 19 (Series B)*, pages 173–199, 1975.
- [2] ACRITH High-Accuracy Arithmetic Subroutine Library, Program Description and User’s Guide. IBM Publications, No. SC 33-6164-3, 1986.
- [3] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.
- [4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and Sorensen D.C. *LAPACK User’s Guide, Release 2.0*. SIAM Publications, Philadelphia, second edition, 1995.
- [5] *ARITHMOS, Benutzerhandbuch*, Siemens AG, Bibl.-Nr. U 2900-I-Z87-1 edition, 1986.
- [6] J.J. Dongarra, J.J. Du Croz, I.S. Duff, and S.J. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.
- [7] J.J. Dongarra, J.J. Du Croz, S.J. Hammarling, and R.J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Software*, 14(1):1–17, 1988.
- [8] D. Husung. ABACUS — Programmierwerkzeug mit hochgenauer Arithmetik für Algorithmen mit verifizierten Ergebnissen. Diplomarbeit, Universität Karlsruhe, 1988.
- [9] D. Husung. Precompiler for Scientific Computation (TPX). Technical Report 91.1, Inst. f. Informatik III, TU Hamburg-Harburg, 1989.
- [10] *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985.
- [11] R.B. Kearfott, M. Dawande, K. Du, and C. Hu. INTLIB: A portable Fortran-77 elementary function library. *Interval Comput.*, 3(5):96–105, 1992.
- [12] R.B. Kearfott, M. Dawande, and C. Hu. INTLIB: A portable Fortran-77 interval standard function library. *ACM Trans. Math. Software*, 20:447–459, 1994.
- [13] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and Ch. Ullrich. *PASCAL-XSC — Sprachbeschreibung mit Beispielen*. Springer, 1991.
- [14] O. Knüppel. PROFIL / BIAS — A Fast Interval Library. *Computing*, 53:277–287, 1994.
- [15] O. Knüppel. PROFIL/BIAS and extensions, Version 2.0. Technical report, Inst. f. Informatik III, Technische Universität Hamburg-Harburg, 1998.
- [16] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.
- [17] R. Krier. *Komplexe Kreisarithmetik*. PhD thesis, Universität Karlsruhe, 1973.
- [18] C. Lawo. C-XSC, a programming environment for verified scientific computing and numerical data processing. In E. Adams and U. Kulisch, editors, *Scientific computing with automatic result verification*, pages 71–86. Academic Press, Orlando, Fla., 1992.
- [19] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
- [20] R.E. Moore. A Test for Existence of Solutions for Non-Linear Systems. *SIAM J. Numer. Anal.* 4, pages 611–615, 1977.
- [21] A. Neumaier. *Interval Methods for Systems of Equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.
- [22] S. Oishi. private communication, 1998.
- [23] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Universität Karlsruhe, 1980.
- [24] S.M. Rump. Validated Solution of Large Linear Systems. In R. Albrecht, G. Alefeld, and H.J. Stetter, editors, *Computing Supplementum*, volume 9, pages 191–212. Springer, 1993.
- [25] S.M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):539–560, 1999.
- [26] T. Sunaga. Theory of an Interval Algebra and its Application to Numerical Analysis. *RAAG Memoirs*, 2:29–46, 1958.
- [27] J. Zemke. b4m - BIAS for Matlab. Technical report, Inst. f. Informatik III, Technische Universität Hamburg-Harburg, 1998.